Virasztó Tamás

Titkosítás és adatrejtés

Biztonságos kommunikáció és algoritmikus adatvédelem

NetAcademia Oktatóközpont

Virasztó Tamás

Titkosítás és adatrejtés

Biztonságos kommunikáció és algoritmikus adatvédelem

NetAcademia Kft.
2004

Első kiadás, 2004. január

A könyv írása során a szerző és a kiadó a legnagyobb gondossággal és körültekintéssel igyekezett eljárni. Ennek ellenére előfordulhat, hogy némely információ nem pontos vagy teljes, esetleg elavulttá vált.

Az algoritmusokat és módszereket mindenki csak saját felelősségére alkalmazza. Felhasználás előtt próbálja ki és döntse el saját maga, hogy megfelel-e a céljainak. A könyvben foglalt információk felhasználásából fakadó esetleges károkért sem a szerző, sem a kiadó nem vonható felelősségre.

A cégekkel, termékekkel, honlapokkal kapcsolatos listák, hibák és példák kizárólag oktatási jelleggel kerülnek bemutatásra, kedvező vagy kedvezőtlen következtetések nélkül.

ISBN: 963 214 253 5

Lektor: Fóti Marcell

Kiadó: NetAcademia Kft.
http://www.netacademia.net

Felelős kiadó: a NetAcademia Kft. ügyvezetője

Borító: Webenium Bt.
http://www.webenium.hu

Nyomdai munkák: Aduprint Kft.

A biztonság nem egy eszköz vagy termék, hanem egy folyamat. A védelem kialakítása során felmérésre, elemzésre kerülnek a kockázati tényezők, lehetséges támadások és azok várható károkozása is. A következő lépés ezek módszeres megszüntetése. Az összes kockázat megszüntethető, így gyakorlatilag végtelen biztonság is elérhető, de ennek végtelen ára van. Csakhogy a felhasználók nem költenek többet rendszerük védelmére, mint az valójában ér. A gyakorlati megvalósítás során elért és a tökéletes biztonság közötti rést maradványkockázatnak nevezzük, amelynek létezését és esetleges következményeit a rendszer tulajdonosának és üzemeltetőjének tudomásul kell vennie.

Köszönetnyilvánítás

Ez a könyv négy év munkájának gyümölcse. Igaz ugyan, hogy e munka oroszlánrésze nekem jutott, de van egy maroknyi ember, akik nélkül nem jutottam volna el idáig. Nekik szeretnék most köszönetet mondani, és egyúttal nekik ajánlom a könyvem is.

Köszönettel tartozom Édesanyámnak, akitől jónéhány hétvégi hazalátogatást loptam el, hogy a könyvön dolgozhassak.

Köszönet illeti Tiszai Tamást, aki lelkesedésével biztatta a kezdeteket. Köszönet Bartha Tibornak, Deé Juditnak, Kálló Kamillnak, Szűcs Ferencnek (egykori és jelenlegi főnökeimnek), hogy hallgatólagos beleegyezésükkel (esetenként anélkül) néha még munkaidőben is a könyvön a dolgozhattam.

Köszönet Czapár Kornélnak, aki az első olvasóm volt, így tőle kaptam az első, viszont meglehetősen letaglózó kritikát is. Korántsem az volt, amit akkor hallani akartam, de nagy lendületet adott a későbbi kritikus időszakokban.

Végül – de mint mondani szokták, nem utolsó sorban – köszönet Fóti Marcellnek, a NetAcademia Kft. vezetőjének, aki a könyv lektorálása során fáradhatatlanul és könyörtelenül gyomlálta ki a felesleges képleteket, miközben mindenbe belekötött, ami egy kicsit is nem volt érthető. Volt néhány pillanat, amikor nagyon nem szerettem ezért, de biztos vagyok benne, hogy az ő lelkesedése, fáradozása és tudása megduplázta a könyv értékét.

Virasztó Tamás

2004. január

"... A rejtjelző gépek elterjedése akkor várható,

ha nem lesznek nagyobbak és sokkal drágábbak, mint egy narancs."

A hírszerzés és a kémkedés története, 1936

1. Bevezetés

Talán úgy is kezdhetném, hogy már az ókoriak is.... Nem lenne túlzás, hiszen már akkor is voltak olyan helyzetek, amikor olyan üzenet megalkotása volt a cél, aminek értelmét csak a beavatottak tudták megfejteni. Egyszóval titkosításra volt szükség. Vajon hány háború, cselszövés és kivégzés múlt a titkos üzenetek célba jutásán, célba nem jutásán vagy azon, ha olyan valaki fejtette azt meg, akinek nem kellett volna? Napjainkban sem más a helyzet, azonban jóval több feladatunk van, igaz, a rendelkezésre álló eszközök is többet tudnak, és jóval biztonságosabbak.

Az emberek továbbra is tengernyi információt cserélnek egymással, akár magánügyben, akár üzleti érdekből. Régen erre a célra rajzokat, füstjeleket vagy dobot használtak, azután jött az írás, a levél, majd a technikai fejlődésével a különböző elektronikus eszközök: távíró, rádió és televízió, telefon és végül(?) az email. A felsorolás korántsem teljes, de lesz-e a sornak vége? Reméljük nem, mert az másnak a végét is jelentené. Egy felmérés szerint 1998-ban csak az USA területén 107 milliárd levelet kézbesítettek. Ez a szám nagysága ellenére szinte eltörpül az emailek 4 trilliós becsült száma mellett (szintén az USA-ra vonatkoztatva) [32].

Kinek van szüksége védelemre?

Mindenkinek. Az információ és a kommunikáció korszakát éljük, az információ és a tudás ma már igen komoly érték lehet. Sokan áldoznak arra, hogy a birtokukban lévő információ ott is maradjon, és sokan áldoznak legalább ugyanannyit arra, hogy ezt az információt megszerezzék. Ma már a számítógép sokkal több egy okos írógépnél, rengeteg helyen, rengeteg feladat ellátására használják. A magánemberek általában játszanak rajta, leveleznek, Interneteznek vele. Van, aki hazaviszi a munkáját és otthon munkaeszközként használja a számítógépét. Közszolgálati intézmények, szervezetek általában adatbázist üzemeltetnek, információt nyújtanak. Van, ahol nyilvántartást vezetnek, adatokat gyűjtenek, mint például a kórházak, könyvtárak, kormányzati és államigazgatási szervek, szolgáltató vállalatok és még sorolhatnánk.

Egyes felhasználók „aktívan” kezelik az adatokat: kereskedelmi tevékenység, vezetői döntéstámogatás esetén az adatok feldolgozása befolyásolhatja a piacpolitikát. A tudomány egyes területein számításokat végeznek a számítógépekkel, elméleti kísérleteket folytatnak, vélt vagy valós helyzeteket szimulálnak és a kutatások eredményeit szintén számítógépen tárolják. Egy katonai vagy nemzetvédelmi összetett alkalmazás pedig minden eddig felsorolt tevékenységre használhatja a számítógépet.

Az iménti felhasználók két nagy csoportban sorolhatók:

- Azoknak, akik nyilvános adatbázist üzemeltetnek nem érdekük a hozzáférők körét korlátozni, ők csak abban érdekeltek, hogy *adataikat illetéktelen ne módosíthassa*.

- Azoknak, akik olyan adatokat tárolnak vagy dolgoznak fel, melyek törvényi védelemben részesülnek (személyes adatok, különleges személyi adatok, nemzetvédelmi adatok) vagy stratégiai fontosságúak (üzleti, katonai célok) már fontos az adatok olyan védelme, amely lehetővé teszi a *hozzáférések szabályozását* és bizonyos *adatok titokban tartását* is.

De ne mindig csak a számítógépről beszéljünk! Gondoljunk arra, hogy egy korszerű GSM kapcsolatnál a továbbított csomagok a személyiségi és magánélethez való jogok (és persze a szolgáltató) védelmében éppúgy titkosításra kerülnek (egy A5/1 nevű algoritmussal), mint egy bankkártya – bankautomata – bankközpont tranzakció minden lépése. Hasonlóan védett sok kereskedelmi TV csatorna jele is: a műhold és a földi állomás között titkosítottan közlekednek a jelek, védve magát a fizetős szolgáltatást és az előfizetők adatait egyaránt. Manapság az elektronikus kommunikáció minden eddiginél nagyobb mértékű lett. Ezen nemcsak az Interneten történő levelezést és adatátvitelt kell érteni, hanem a telefonvonalakon bonyolított egyéb kommunikációt is: faxüzenetek, telebankszolgáltatások igénybevétele, vagy egy egyszerűnek tűnő pizza- vagy mozijegy-rendelés. Ha egy-egy ilyen kapcsolat alkalmával valaki a személyes adatait is használja (cím, email cím, bankkártyaszám, stb.), tovább fokozódik a veszély. A telefonvonalak és központok „digitalizálódásával" egyre könnyebb egy kapcsolatot (annak tényét és tartalmát) rögzíteni és tárolni – esetleg később feldolgozni.

Korábban elképzelhetetlen feladatok és célok ma már megvalósíthatók. Ezzel egy időben a tárolt információk felértékelődtek és gyakori célpontjai lettek az ipari kémkedésnek és a „csakazértis" stílusú *hacker* valamint a jóval veszélyesebb *cracker* támadásoknak, de egyes direktmarketinges megoldásoknak is jól jön egy-egy karbantartott, biztos címlista.

Az adatok illetéktelenekhez kerülése vagy elvesztése sok bosszúságot, de igen jelentős anyagi és erkölcsi károkat is okozhat. Ezért ha valakivel biztonságosan akarunk kommunikálni, vagy adatainkat biztonságban akarjuk tudni, előbb-utóbb valamilyen védelem után kell néznünk. Szerencsére számtalan módszer áll rendelkezésünkre: olyan titkosítási algoritmusokat ismerünk, amelyek igen bonyolultak, rendkívül nehezen fejthetők meg (ha egyáltalán meg lehet ezt tenni), viszont számítógéppel könnyen megvalósíthatóak. De vajon titkos-e az az üzenet, amely az eredetihez képest mindenféle összevissza jeleket tartalmaz, így a beavatatlan emberi szem vagy mikroprocesszor számára értelmetlen? Kétféle válasz adható erre a kérdésre:

1. *Igen*, titkos, mert a megfelelő kiegészítő ismeret nélkül nem lehet elolvasni, értelmezni.
2. *Nem*, nem titkos, mert tudjuk, hogy az egy valódi üzenet és az olvashatatlanság miatt „ordít" róla, hogy „ÉN TITKOS VAGYOK!". Az egy más dolog, hogy miként lehet feltörni, de ez már nem titok, „csak" probléma.

Cracker vs. Hacker

Gyakran felmerülő kérdés, hogy ki a hacker és ki a cracker. Egyesek szerint csak hitvita az egész és a két magatartásforma különbsége nem definiálható egyértelműen. Ezen persze főleg a „hacker"-ek sértődnek meg és mindig találhatunk olyan véleményeket, amelyek újabb és újabb definíciót adnak közre. Az egyik leglogikusabbnak tűnő besorolás a következő:

- ***Hacker***: A kihívást keresi, általában nem akar kárt okozni. Nagy tudású, jól felkészült, tudja mit miért csinál: ismeri a védelmi rendszereket, a protokollokat, az operációs rendszereket és mindezek gyenge pontjait is. Eszközeit általában saját maga fejleszti, esetleg más – de megválogatott forrásból – szerzi be. Behatolásának nyomait eltünteti, esetleg külön felhívja a figyelmet tettére. A hackerek egy részének ez a „szakmája", ők a professzionális hackerek, akik tudásukból élnek, mint biztonságtechnikai tanácsadók vagy mint adatrablók[1].
- ***Cracker***: Hasonló a hackerhez, de óriási különbség a kettő között, hogy a cracker alapvetően „anarchista", tevékenységének fő szándéka a rombolás.
- ***Script Kiddie*** – valaki ezt egyszerűen „hülyegyerekek"-nek fordította – akik az Internetről letöltött mindenféle programot eresztenek a kiszemelt rendszerre. Tudásuk általában igen csekély, ezért gyakran nem is tudják, mivel játszanak valójában. (Bár ez egyáltalán nem gátolja meg őket abban, hogy magukat hackernek vagy crackernek nevezzék. Az igyekvő script-kiddie-t, aki igazi hacker akar lenni, de „nincs rá ideje", meg „nincs hozzá türelme", szokás „wannabe hacker"-nek is nevezni... Ez lényegében semmi különbséget nem jelent, inkább gúnyosabb hangvételű jelző.)

Bruce Schenier a *Titkok és Hazugságok* című könyvében nem ért egyet ezzel a kissé önigazolásnak tűnő osztályozással. Ha a támadó cracker („bad hacker"), szinte garantált a közvetlen vagy közvetett károkozás. Ha viszont hacker („good hacker"), a fenti jellemzés alapján az áldozat szerencsésnek érezhetné magát. Csakhogy az áldozat (és a rendszer biztonsága) szempontjából bizonyos mértékig mindegy, hogy az őt megtámadó személy melyik kategóriába esik, hiszen egyik sem élvezi a bizalmát. (Senki sem szeretné, hogy egy betörő „csak" körülnézne a lakásában, bár nem vinne el semmit.) Ne felejtsük el, hogy hackert vékony határ választja el a crackertől, és ez a határ nem más, mint a szándék [44].

A fenti „definíció" egyébként korántsem egységes vagy széles körben elfogadott. Egy fórumban olvastam egy olyan megkülönböztetést, amely a szándékra helyezi a hangsúlyt: „...hacker az, aki a 'nem működik' állapotból a 'működik' állapotba akar eljutni (még akkor is, ha ez valakinek a szerzői jogait sérti), a cracker pedig, aki mások rendszerét akarja a 'működik'-ből a 'nem működik'-be juttatni. Például aki a csak binárisan elérhető program [vagy egy biztonsági rendszer] hibáit kijavítja, hogy az működjön, vagy leszedi a másolásvédelmet, [hogy a program használható legyen], az hacker. Aki tönkrevágja a számítógépedet, csak azért, hogy megmutassa, az cracker." (Igaz ugyan, hogy korábban ugyanebben a topicban egészen más szempontból egész más jellemzés is napvilágot látott. Például: a cracker: programvédelmekkel játszik, míg a hacker: hálózati védelmekkel szórakozik. Ez részben ellentmond az előzőnek.)

Eric S. Raymond egyszerű módon tesz különbséget a két fogalom között: „Az alapvető különbség: a hackerek építenek, míg a crackerek rombolnak." [URL37]

[1] A „hacker mitológia" szerint a hackerek az információ birtoklásával, ellenőrzésével és korlátozásával szemben az információ megosztásáért és terjesztéséért harcolnak. Alapelvük, a „tudni akarom" szemben áll az információt birtokló „ennyit tudhatsz" előírásával.

Rejtjelezés = elektronikus boríték?

Kis kitérő után kanyarodjunk vissza az eredeti témánkhoz. A kérdés az volt, hogy szükség van-e titkosításra? Ha igen, mikor? Próbáljuk megválaszolni ezt a kérdést egyszerű következtetéssel [33]:

- Mit kell védeni? *Az információt.*
- Melyik információt kell védeni? *Az értékeset.*
- Mi az értékes információ? *Amit annak tartunk.*
- Hol van az értékes információ? *Adathordozón vagy átviteli csatornán.*
- Mitől kell védeni az értékes információt? *Megsemmisüléstől, eltulajdonítástól*[2].

Ha egy jól menő üzlet adatait, szervereit vagy leveleit kell védeni, valószínűleg mindenki elfogadja a védelem igényét, és nem teszi fel a kérdést, hogy „szükség van-e rá". De ha megkérdezzük ugyanezeket az embereket, hogy a magánlevelezéseikhez – ha az elektronikus úton történik – használnak-e titkosítást, valószínűleg nemleges választ kapunk és (némi habozás után) magyarázatként pedig azt, hogy nincsenek titkaik, vagy törvénytelen üzelmeik. Azonban ugyanezek a személyek a papíralapú levelezéseikhez minden bizonnyal nem levelező- vagy képeslapot használnak, hanem zárt borítékot, jóllehet „nincsenek titkaik, vagy törvénytelen üzelmeik" egyszerűen csak a *magánélethez való jogukat* gyakorolják. De miért nem teszik ezt akkor is, amikor e-mailt küldenek? Talán nincsenek tisztában azzal, hogy az elküldött e-mail védelem nélkül vándorol egyik szerverről a másikra, miközben tucatnyi telefontársaság és adatátviteli szolgáltatást nyújtó cég eszközein halad át? Ez nem ugyanolyan, mintha boríték helyett képeslapot használnának? Remélhetően idővel megváltozik a helyzet és mindenki belátja, hogy az „*információs társadalomban a magánélethez való jog csak erős kriptográfia használatával őrizhető meg".* (*Philip Zimmermann, a PGP atyja*)

Mi is a kriptográfia? A ***kriptográfia*** azon elvek és gyakorlati technikák tanulmányozásával foglakozik, melyek lehetővé teszik az üzenetek, adatok olyan módon történő továbbítását és tárolását, hogy ahhoz már csak a jogosult fél fér hozzá. Ez lehetetlen feladat azok számára, akik nem birtokolják a megfejtéshez szükséges kulcsot, vagy legalábbis időben lehetetlen feladat. A ***kriptoanalízis*** területe viszont pont azzal foglalkozik, hogy az egyes titkosított üzenetekből miként fejthető vissza az üzenet a kulcs ismerete nélkül. A két terület összességét ***kriptológiának*** hívjuk, beleértve az olyan határos területeket is, melyek nem sorolhatók egyértelműen az egyik vagy a másik csoportba. A ***szteganográfia*** kínál egy érdekes alternatívát: olyan üzenetet kell készíteni, amely elrejti az eredeti – esetleg külön titkosított – üzenetet, és a kívülállónak egészen mást mutat, mint annak, aki tudja, mit keressen. Beavatatlanok számára ez jelenthet egy képet, amit meg lehet nézni, egy zenét, amit meg lehet hallgatni és így tovább. Az adatrejtés egyik vitathatatlan előnye, hogy az elrejtett üzenet nem „provokálja" a feltörést, hiszen ha a támadó nem tud az üzenet létezéséről, értelemszerűen nem is akarja azt feltörni. Hasonló volt már az ókorban is, gondoljunk csak a trójai fa paci esetére…

[2] Érdekes, hogy ez a két veszélyforrás a lehetséges védekezések tekintetében ellentétes. Ugyanis ha sok-sok másolatot készítünk az értékes adatról, csökken az elvesztés esélye (természeti vagy egyéb katasztrófa, adattároló hibája vagy emberi gondatlanság miatt), viszont nagyobb valószínűséggel kerül illetéktelen kezekbe. Ha pedig a lehető legkevesebb példányunk van belőle (tipikusan egy), csökken ugyan az eltulajdonítás veszélye, viszont véglegesen elbúcsúzhatunk tőle, ha az adathordozó megsérül.

Honnan indult ez a könyv?

Jelen könyvvel olyan bevezető szintű írás elkészítése volt a célom, amely nélkülözi a sokak számára rémisztő matematikai háttér – szükségesnél nagyobb mértékű – ismertetését. Olvastam egy-két egyetemi jegyzetet, melyeknek címe általában *Rejtjelezés*, vagy valami hasonló volt. Bár legtöbbjük mindössze két tucat oldalból állt, tele voltak olyan képletekkel, halmazelméleti jelölésekkel és fogalmakkal, melyek megértéséhez mindenképpen egyetemi – vagy legalábbis főiskolai – szintű matematika ismeretek szükségesek. Elismerem, hogy titkosító algoritmusok fejlesztéséhez ezek az eszközök és fogalmak nélkülözhetetlenek, de az én célom nem az, hogy megalkossam a XXI. század titkosító algoritmusát, hanem csak az, hogy alapvető fogalmakkal, fogásokkal tisztában legyen az olvasó, és egyfajta rálátása legyen a témára. Természetesen, aki ezt a könyvet elolvassa, nem lesz kriptográfus. De néhány olyan dolgot megérthet, ami eddig fehér folt volt az ismereteiben, és áttekintő képet kaphat a titkosításról, a kriptorendszerekről, azok működéséről, fejlődéséről és a főbb ötletekből. Utólag elolvasva azokat a bizonyos jegyzeteket, már nem is olyan rémisztőek, de meglepően egyszerű módon is el lehetett volna magyarázni mindazt, ami azokban van. Ezt azonban ne értse félre senki! A következő oldalakon csak a nagyon alapvető ismereteket fogom ismertetni, és mindez csak a jéghegy csúcsa! Aki komolyan kriptográfiával akar foglalkozni és kriptográfus szeretne lenni, ennél sokkal több és főleg mélyebb ismeretre lesz szüksége.

Sajnálatos módon a magyar nyelvű dokumentáció ebben a témában (kriptográfiában és nem általánosan az informatikai biztonságban) elég szegényes, és talán még ez a jelző is túlzás. Az angol nyelvű irodalom viszont bőséges, egy kis kutatómunkával szinte mindent meg lehet találni. Jelen könyv ilyen kutatómunka eredménye: többarasznyi dokumentumot, szabványt, RFC-t, leírást, ajánlást, RSA hírújságot, könyvet, konferenciaösszefoglalót, összehasonlító tanulmányt, programkódot, cikket, „technical report”-ot dolgoztam fel és tartalmukat igyekeztem magyar nyelven logikusan egységes egészbe összefoglalni.

Vannak részek, melyek – gyakran nyilvánosan is elérhető – angol nyelvű dokumentumok tartalom szerinti fordításai[3], mások – például a DES-ről, az RSA-ról, elvekről és módszerekről szóló információk – tucatnyi forrás feldolgozásának eredményei. Az olyan helyzetekben, ahol az egyes források egymásnak ellentmondó adatokat közöltek, ott a hivatalos szabványok publikációit vagy *Menezes, Oorshot, Vanstone: Handbook of Applied Cryptography* című könyvét [11] és *Bruce Schneier: Applied Cryptography, Protocols, Algorithms and Source Code in C* című könyvét [24] tekintettem döntő irodalomnak, és sok nem bizonyított állítás bizonyítása is ezekben található meg[4]. Sok gondolatot vettem át Bruce Schneier: *Secrets and Lies – Digital security in a networked world* című könyvéből is. Érdekes, hogy még olyan forrás is, mint az RSA Inc. *Cryptobytes* című időszakos kiadványa is keveredik ellentmondásba – néha még a saját irodalomjegyzékével szemben is.

[3] Például az „6. Titkos vs. nyilvános kulcsú módszerek – és néhány záró gondolat” című fejezet első része szinte teljesen [11]-ből való.
[4] A szabványok meglehetősen kusza kapcsolatában igyekszik eligazítani a Függelék egyik alfejezete...

> Cryptobytes, 1997 Ősz, *Preneel – Bosselaers – Dobbertin: The Cryptograhic Hash Function RIPEMD-160* című cikkben, a 10. oldalon a „*Hash function cryptanalysis*" bekezdésben a szerzők 1992. január 31.-re dátumozzák az SHA publikálását és a [16]-os irodalomra hivatkoznak. A [16]-os irodalom azonban – ami a hivatalos publikáció – 1993. május 11.-én került kiadásra. Hasonlóan az SHA-1 publikálását a cikkben 1994. július 11.-re dátumozzák, holott a hivatalos – és a cikkben [17]-ként is hivatkozott – publikáció 1995. április 17.-én jelent meg. (Lásd még: *9.4. RIPEMD* fejezetet hasonló, bár szembeszökőbb következetlenségét.) Egyébként az ilyen dátumokkal vigyázni kell, mert a nyilvánosságra hozatal nem mindig egyenlő a hivatalos publikálással.

A legtöbb ellentmondás egyébként az évszámok, pénzben kifejezett értékek és egyéb történelmi adatok valamint a javasolt vagy mért értékek körében található, főként abban az esetben, ha a cikk szerzője saját algoritmusát hasonlítja össze más algoritmusokkal. Szerencsére elvi, vagy egyéb megértést és megvalósítást befolyásoló ellentmondásokkal nagyon ritkán találkoztam (csak az RC4 és A5/1 algoritmusok esetében, amelyek egyébként sem publikusak).

Kinek szól?

A fentiek értelmében a könyv a következő olvasóknak szól:

- Minden, a téma iránt érdeklődő ember számára, aki az új ismeretei birtokában eldöntheti, hogy kíván-e további részletekkel megismerkedni vagy sem. Ha úgy dönt, hogy neki ennyi is elég, akkor is tiszta képet kaphat az alapvető fogalmakról, módszerekről.
- Programozóknak, hogy programjaikban elkerülhessék a hamis biztonság csapdáját, és bátran használják a különböző *cryptoAPI*-kat úgy, hogy lehetőségeikhez mérten még válogatni is tudjanak közöttük.
- Olyan érdeklődőknek, akik hozzám hasonlóan nem férnek (fértek) hozzá egyszerű, világos, alapot teremtő információkhoz.
- Rendszergazdáknak, hogy ne halljak még egyszer olyat, hogy „Te hiszel ezekben a módszerekben?" (Amúgy a válaszom, igen hiszek!)
- És végül olyan informatikai szakembereknek, akiknek nem az informatikai vagy információbiztonság a szakterületük, de érdeklődnek a téma iránt.

További lehetőségek

A könyv jelen formájában nem teljes, valószínűleg soha nem is lesz az. A terület a számítási kapacitások fejlődése valamint az újabb matematikai eredmények miatt állandóan változik. Hiányzik belőle a gyakorlatban használt biztonsági protokollok ismertetése, mint például a *SET*, az *IPSec* vagy akár az *IPv6* kriptográfiai támogatása[5]. Remélem, hogy a hiány nem érezhető, mert a felsorolt területek a könyvben tárgyalt témák alkalmazásáról szólnak, így nem

[5] Sok helyen a SET-et valamilyen titkosításnak, titkosítási algoritmusnak tekintik. Ez azonban helytelen! A SET egy protokoll-leírás, amely úgy szabályozza az elektronikus fizetési és vásárlási folyamatot, hogy mindenki csak a számára szükséges információkhoz jusson hozzá. (A kereskedő hozzájut a rendelési adatokhoz, de nem jut hozzá a fizetés részleteihez, a bank intézi a fizetést, de nem tudja meg, hogy a vevő mit vett stb.) Ehhez a szabványos algoritmusokat használja. Tehát a SET protokoll és nem titkosítási algoritmus!

az elvi kriptográfia témakörébe tartoznak, hanem inkább egy alkalmazott kriptográfiáról szóló könyvben lenne a helyük. És had említsem meg azt is, amiről itt biztosan nem lesz szó:

- általános üzemeltetési és hozzáférési biztonságtechnika
- vírusvédelem, tűzfalak
- biztonsági rések
- hackelési és crackelési technikák, ismeretek,
- biztonsági rések és azok kihasználását lehetővé tevő eszközök.

Amiről pedig a továbbiak olvashatunk

Ennek a könyvnek tehát nem célja a teljes információbiztonság áttekintése, csak ennek egy részterületébe, az *algoritmikus információvédelembe, a kriptográfiába* nyújt betekintést. Igyekszik nem a legmélyebb dolgokkal kezdeni, bár lesz benne olyan is, amely egy kicsit nehezen emészthető. De arra biztatok mindenkit, akit egy kicsit is érdekel a téma, próbálja meg elolvasni, nem fog tőle gyomorrontást kapni. Természetesen vannak olyan könyvek és leírások, amelyek ennél sokkal pontosabb, részletesebb és alaposabb megközelítését kínálják a témának, de sajnos ezek legtöbbje csak azok számára érthető, akik tisztában vannak a teljes matematikai háttérrel és angoltudásuk sem elhanyagolható. Nem állítom, hogy a matematikai háttér nélkül is keresztül lehet vágni a rejtjelezés hegyein és völgyein, de véleményem szerint azt nem arra kell használni, hogy elriasszunk minden érdeklődőt, és egyfajta szűrőként használjuk az olvasók előtt. Remélem senkinek sem lesz elpocsékolt idő az a néhány óra, amit e könyv elolvasására szán.

E fejezet hátralévő részében egy kis fogalommagyarázatot találunk, majd egy kis történelmi fűszerezésű áttekintés olvasható. Ebben a legegyszerűbb és/vagy a legrégibb módszerekkel, ötletekkel találkozhatunk. A történeti háttér iránt érdeklődők számára *David Kahn: Codebreakers* című könyvét ajánlom, ami igen alapos történeti áttekintést ad a 4000 éves Egyiptomtól a közelmúltig. A meglehetősen vaskos, mintegy 1200 oldalas könyvet először 1967-ben adták ki, utoljára pedig 1996-ban. Szerencsére van egy közelebbi, sőt magyar nyelvű forrás is: *Simon Singh: Kódkönyv - A rejtjelezés és a rejtjelfejtés története (Park Könyvkiadó, 2001).*

Az ezt követő fejezetekben megismerkedhetünk napjaink szimmetrikus és a nyilvános kulcsú módszereivel (DES, AES, RSA, ECC), illetve ezek működési módjaival valamint a blokktitkosítók és folyamtitkosítók közötti legfőbb különbségekkel. Megismerkedhetünk az adatintegritás védelmét szolgáló és a digitális aláírásokban is gyakran használt üzenetpecsétekkel. Röviden áttekintjük a digitális aláírás logikáját, megvalósítását és a magyar jogrendszerben történő szabályozását is. Olyan témákról is lesz szó, melyek nem kapcsolódnak szigorúan a titkosításhoz, de jól kiegészítik azt: az egyik az adatrejtés – görög eredetű szóval: szteganográfia – a másik a titokmegosztás, illetve a titokszétvágás. Mindkettő alkalmas arra, hogy a különböző biztonsági protokollokban ellenőrző szerepet töltsön be, illetve segítsen a titkot titokban tartani.

Ha valakinek megjegyzése, javaslata, észrevétele vagy kérdése van, szívesen veszem a `cryptox@message.hu` e-mail címen illetve az [URL00]-án elérhető fórumon.

1.1. ALAPVETŐ FOGALMAK

Egyszerűen fogalmazva a titkosítás célja az, hogy az információt úgy juttassunk el a címzetthez, hogy annak tartalmához csak ő férhessen hozzá. Vagyis az üzenetek tartalmához való hozzáférés és azok megváltoztatása nem lehetséges azok számára, akik nem jogosultak rá. Ugyanakkor fel kell tételezni, hogy a **címzetten** (*receiver*, az információ fogadóján) és a **feladón** (*sender*, az információ forrásán) kívül létezik legalább egy harmadik résztvevő is: a **támadó** *(intruder, adviser, attacker)*.

> Bruce Schneier vezette be a beszédes, szerepkörhöz kötődő névhasználatot, amely azóta az angol szakirodalomban szinte de facto szabvánnyá vált:
>
> - **Alice, Bob, Carol, Dave** (a beszélgető felek: általában Alice a feladó, Bob a címzett);
> - **Trudy** (*intruder*) támadó általános szándékkal;
> - **Eve** (*eye, eaves dropper*) passzív támadó;
> - **Mallory** (*malicious active attacker*) az aktív támadó;
> - **Trent** (*trusted arbitrator*) a döntőbíró,
> - **Peggy** (*prover*), a bizonyító
> - **Walter** (*warden*), Alicet és Bobot felügyeli az egyes protokollok végrehajtása során
> - **Victor** (*verifier*) az ellenőrző.

A támadó alapfeltétele a titkosításnak, mert ha nem létezne, nem lenne szükség a titkosításra sem. Mint látni fogjuk, sok fejtörést okoz, mert amíg Alice és Bob mindent elkövet, hogy „beszélgetésük" titkos maradjon, addig Eve, Trudy és Mallory azon vannak, hogy törekvésük kudarcba fulladjon, vagyis:

- Megpróbálnak illetéktelenül *hozzáférni* az üzenet tartalmához.
- Alice nevében *hamis üzenetet* próbálnak küldeni *Bob* számára.

Bobnak ezért egyaránt képesnek kell lennie az üzenet olvasására és Alice személyazonosságának ellenőrzésére is. Mint látni fogjuk, ez utóbbi nem is olyan egyszerű feladat...

1.1.1. Terminológia

A továbbiakban

- azt az üzenetet, adatot, amit Alice el akar küldeni (és nincs szükség semmi extra műveletre annak értelmezéséhez), **nyílt szövegnek** (*plaintext, cleartext*) nevezzük.
- azt a műveletet, amely a nyílt szöveget, annak értelmét vagy más jellemző tulajdonságait elrejti, **titkosításnak** nevezzük (*enciphering, encryption*). Eközben valamilyen kriptográf **algoritmust** (*cipher*) használunk[6].

[6] Sok irodalomban – különösen a magyar fordításokban, magyar irodalmakban – helytelenül használják a kódolás, dekódolás fogalmát (lásd például „DES kódolás", „kódolt adás", stb), bár legalább ennyi helyen felhívják erre a figyelmet. Anélkül, hogy egzakt definíciót adnék, a következő a különbség a kódolás és a titkosítás (rejtjelzés) között: a kódolás során egy olyan táblázatot vagy megfeleltető algoritmust használunk, amelyhez nincs szükség kulcsra, a táblázat e nélkül is egyértelmű kapcsolatot teremt a jelek között. Például az „a" betű az ASCII (*American Standard Code for Information Interchange*) táblában 97, a „b" betű 98 stb. Hasonlóan kódolás, ha a „nyolcas" számot felírjuk a különböző számrendszerekben: 8_{10}, 8_{16}, 100_2, 10_8 stb. De az „a" betűt írhatom így is: 1100001_2, ami nem más, mint az „a" betű ASCII kódjának bináris kódolása. A lényeg, hogy a kódolás során az egyik szimbólumot kölcsönösen egyértelműen meg lehet feleltetni a másiknak. A rejtjelzés vagy titkosítás során viszont egy szimbólumhoz („a" betű) nagyon sok másik szimbólum párosítható, a „helyes" párost a kulcs jelöli ki. Ilyen értelemben a titkosítás egyfajta paraméterezett kódolásnak tekinthető.

- a létrejövő értelmezhetetlen adathalmazt **titkosított vagy kriptoszövegnek** (*ciphertext*) nevezzük.
- a titkosított szöveg nyílt szöveggé való jogosult visszaalakítását **megfejtésnek** (*deciphering, decryption*) nevezzük.
- a titkosított szöveg nyílt szöveggé való jogosulatlan (értsd: kulcs nélküli) megfejtését **visszafejtésnek** vagy **feltörésnek** nevezzük.
- és mindehhez kell a **kulcs** (*key*). A titkosító módszerekkel szemben alapvető elvárás, hogy egy adott információból úgy készítsen másikat, hogy ez utóbbiból *csak egy kiegészítő adat ismeretében* lehessen megismerni az eredetit. Ezt a kiegészítő adatot nevezzük *kulcs*nak, ami egy lehetőleg hosszú, véletlenszerű jelsorozat. Ajánlott a kulcsok *gyakori cseréje*: ha a támadó megfejt egy üzenetváltáshoz használt kulcsot, el fogja tudni olvasni az összes korábbi, e kulccsal titkosított üzenetet, de a későbbieket csak akkor, ha a kulcs továbbra is változatlan marad. Ha viszont a kulcsot gyakran – a feltételezett visszafejtési időn belül – cseréljük, a támadót passzív tevékenységre kényszerítjük, mert idejének jelentős részét az aktuálisan használt kulcs keresése teszi ki. Az lehetséges kulcsok halmazát **kulcstérnek** nevezzük.

1.1.2. Kerckhoffs követelmények

A titkosító rendszerek általános követelményeit 1883-ban *Auguste Kerckhoffs von Nieuwenhof* holland nyelvész fogalmazta meg *La criptographie militaire* című művében. Ma már újabb elvárások is vannak a tárgyalt rendszerekkel szemben, de *Kerckhoffs* gondolatai továbbra is érvényben vannak és röviden a következők:

1. ***Ha egy rendszer elméletileg nem feltörhetetlen, akkor a gyakorlatban legyen az.*** Az elmélet gyakorlatba juttatását valamilyen módon meg kell akadályozni. A legtöbb titkosító módszer olyan algoritmusokat használ, melyek feltörhetőek ugyan, de a támadásnak nem kivitelezhető idő- és/vagy tárigénye van.
   - Egy rendszer elméletileg biztonságos, ha a feltörésének valószínűsége független a támadó számítási kapacitásától vagy a támadásra szánt időtől.
   - Gyakorlatilag biztonságos, ha a feltöréshez ismert mennyiségű lépést kell végrehajtania, de ennek lehetetlen idő- vagy társzükséglete van.
   - Nem biztonságos, megfejthető, ha a feltöréshez használt módszer tárigénye kielégíthető és időszükséglete egy bizonyos reális korláton belüli.
2. ***A rendszer részleteinek kompromittálódása ne okozza a rendszer egészének kompromittálódását.*** Ha a támadó részinformációkat szerez egy rendszerről, ne veszélyeztesse a rendszer egészét. Ez egyfelől azt jelenti, hogy az egyes biztonsági szinteken megszerzett információk a támadót ne segítsék a további szintek áttörésében. *Másfelől a támadónak a biztonsági rendszer teljes ismerete sem jelenthet segítséget.* Vagyis a biztonság kizárólag a kulcs ismeretének függvénye. **Ez a Kerckhoffs-elv.**

> „A titkosítási rendszer megbízhatósága nem függhet a titkosítás algoritmusától, azt csak a kulcs titkának megőrzése garantálja.”

A ma használt algoritmusok és protokollok nagy része teljesen nyílt, illetve ismert. Az egyik legelterjedtebb civil felhasználású rendszer, a PGP forráskódja az Internetről letölthető. Hasonlóan nyilvános a *DES*, a *Rijndael*, az *RC5, az RSA* (és még sorolhatnánk sokáig) specifikációja is.

3. ***Az alkalmazott kulcsnak – feljegyzések nélkül is – könnyen megjegyezhetőnek és könnyen megváltoztathatónak kell lennie***. Egy rendszer általában kötött méretű kulcsokat használ (legalábbis a rendszeren belül), a felhasználók kulcsainak ehhez a mérethez kell igazodniuk, vagy ehhez kell azokat igazítani. Ezt az alkalmazkodási kényszert meg lehet szüntetni a hashfüggvények alkalmazásával, amelyek egy tetszőleges karakterláncból rögzített hosszúságú bitsorozatot generálnak. Így a felhasználók valóban szabadon választhatnak számukra könnyen megjegyezhető jelszót vagy akár jelmondatot is, a rendszer ennek *hashértékét* használja kulcsként. A kulcs cseréjére két esetben lehet szükség:
   - Ha egy 8 karakteres jelszót használunk, amit az összes lehetőség kipróbálásával 1 hónap alatt ki lehet találni, célszerű a jelszót 2-3 hetente vagy gyakrabban cserélni.
   - Ha felmerül a gyanúja annak, hogy jelszavunkat valaki más is ismeri. Itt jegyzem meg, hogy a támadó a legtöbbször nem veri sikerét nagydobra, hiszen azzal a kulcsok azonnali lecserélését váltaná ki.
4. ***A rejtjeles szöveg táviratban is továbbítható legyen***. Ennek a feltételnek ma már nincs nagy jelentősége, hiszen a digitális számítógépek bitjei gond nélkül átalakíthatók ASCII jelekké – vagy bármi mássá, végső esetben hexadecimális karaktersorozattá. Ez a követelmény igazából azt jelenti, hogy a rejtjeles szöveget a nyílt szöveggel megegyezően kell tudni továbbítani, nem igényelhet semmilyen különleges bánásmódot, kódolást vagy speciális – a nyílt szövegétől eltérő – átviteli közeget az átvitel során.
5. ***A titkosító rendszer legyen hordozható és egy személy által is üzemeltethető***. A szoftvereszközök ideálisan teljesítik ezt a feltételt a legtöbb elektronikus, elektromechanikus és mechanikus eszközhöz hasonlóan.
6. ***A rendszer legyen egyszerű, könnyen kezelhető és ne igényelje listányi szabályok betartását***. A jól elkészített eszközök biztosítják ezt a feltételt, mert az esetleges szabályok figyelését átvállalják a felhasználótól. Gyakran rejtett módon, ritkábban a felhasználó felügyelete mellett teszik ezt. Ez a követelmény azért fontos, mert ha egy rendszer biztonsága függ a betartandó szabályoktól, akkor kérdéses a rendszer biztonsága, ha valaki – rossz- vagy jóhiszeműen – elfelejti betartani a sok szabály egyikét. Működőképes marad egyáltalán a rendszer?

1.1.3. A kriptográfia önmagában nem védelem

Egy teljes kriptográfiai rendszer a következő fontosabb komponensekből épül fel, melyek gyakran nem határolhatók el élesen egymástól:

- algoritmikus rendszer, az egyes szolgáltatások matematikai háttere,
- kulcselosztás, -tárolás, -továbbítás (kulcsmenedzsment),
- kiegészítő védelmi rendszer, amely a teljes rendszer (ön)védelmét látja el: a támadások és kezelői hibák károkozását igyekszik csökkenteni, lehetőleg kiküszöbölni.

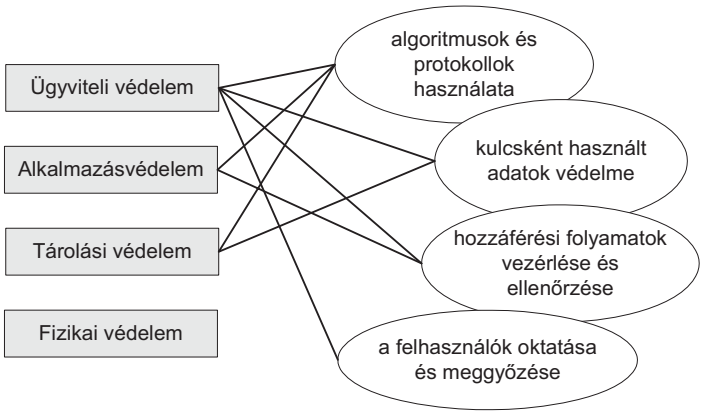
Az információvédelem megvalósítási módszereire és az algoritmusok felhasználására a *kriptográfiai protokollok* adnak útmutatást. Ezek mondják meg, hogy mit-mivel-mikor kell titkosítani vagy megfejteni és mit-hova-mikor kell küldeni, valamint egyéb utasításokat, ellenőrzési pontokat tartalmazhatnak. Tehát az algoritmusok a protokolloknak csak eszközei. *Ezt azért fontos tudomásul venni, mert a kriptográfiai algoritmusok önmagukban nem nyújtanak megfelelő védelmet, tehát a kriptográfia önmagában nem védelem!*

> Kriptográfiai protokollnak nevezzük azt a protokollt, amely a szabályok betartatását és a csalók leleplezését kriptográfia eszközökkel (értsd algoritmusokkal) valósítja meg.

A megfelelő információvédelemhez hozzátartozik a megfelelő ügyvitel kialakítása is: a kulcsok cseréje, tárolása, a titkosított és nyílt szövegek kezelésének szabályai. Például ha egy program lehetővé teszi, hogy egy titkosítva küldött-kapott levelet, állományt titkosítás nélkül mentsünk el – esetleg a titkos szöveg mellé –, de ennek veszélyeire még csak nem is figyelmeztet, megkérdőjelezhető a program helyes védelmi elvi működése, hiszen komoly biztonsági rést hagy rejtve a felhasználó előtt. Más kérdés, hogy gyakran előforduló hiba az alkalmazások egy részénél az is, hogy a titkosítva érkezett adatok a megtekintés vagy feldolgozás ideje alatt plaintext formában tárolódnak a memóriában (mégpedig lapozható memóriában, ahonnan a virtuális memória lapozásával lemezre kerülhetnek) vagy – és az a durvább hiba – eleve ideiglenes fájlban. Ezek ellen a mezei felhasználó nem sokat tehet.

Hasonló problémát jelentenek az olyan kényelmi szolgáltatások, amelyek egyes jelszavak elmentését ajánlják fel. (E szolgáltatás egyik következménye a jelszó elfelejtése is. A felhasználó csak nézi a csillagokat és nyom egy „OK”-ot. Aztán egyszer újratelepíti a gépét és a beviteli mező üres lesz, a felhasználó pedig mérges…) Bárki kipróbálhatja: végy egy főiskolai számítógéptermet. Ülj le valamelyik géphez és nézd meg, milyen FTP programok, Commanderek vannak a gépen. Ha nem találsz semmit, ülj át egy másikhoz. Gyorsan fogsz olyan gépet találni, ahol nyugodtan mászkálhatsz egyes – egyébként jelszóvédelemmel ellátott – FTP szervereken anélkül, hogy egyetlen jelszót is megtudnál. Tipikusan ilyen például a Windows Commander, mert egy szöveges fájlban tárolja a titkosított jelszavakat. Ha elmásoljuk az állományokat egy másik gépre, a megfelelő könyvtárakba, ott is ugyanúgy használhatóak. Sőt, ez az egyszerű módszer működhet Linuxon is, ha az a feladat, hogy egy másik gépre kell „klónozni” a már létező felhasználókat és jelszavaikat: a password/shadow fájl lecserélése 100%-os eredményt garantál. (Kipróbáltam…) De nem kell ennyire lemenni rendszerszintre: a Windows telefonos kapcsolatok párbeszédpanelén is bepipálható a „Jelszó mentése”, így – az egyéb védelemmel el nem látott – számítógépünket bekapcsolva bárki kapcsolódhat a mi nevünkben (és a mi számlánkra) valamilyen identifikációt igénylő kiszolgálóhoz.

Másik elvi probléma, hogy egyes operációs rendszerek illetve alkalmazások a megadott jelszót a memóriában eltárolják – gyakran plaintext formában – és ha legközelebb szükség van rá, nem kérik újra a felhasználótól (vagy egy autentikáló szervertől), hanem a memóriából keresik elő azt. A baj azonban az, hogy gyakran nemcsak az operációs rendszer tudja elővenni a kért adatot, hanem egy bejuttatott szimatoló (sniffer) program is. A jelszó (egyirányúan) kódolt tárolása sem mindig megoldás, mert gyakran a jelszót kódolva kell elküldeni, így a kódolt jelszó valamilyen szempont szerint egyenértékű a kódolatlannal. (Csak a kódolt az hottentottául van…)



1. ábra A hatékony védelem alapeszközei

A jelszó egyébként is legtöbbször az emberi meggondolatlanság és kényelem miatt kerül veszélybe, hiába támogatja a rendszert egyébként erős algoritmikus háttér. Általában is igaz, hogy ha egy rendszerbe valaki be akar jutni, annak érdemes előbb a „humán" oldalról megközelíteni a rendszert. Néhány felhasználó a monitorára vagy a billentyűzetére írva tárolja jelszavát. Egyes titkárnők tudják a főnökük jelszavát és gyakran szó nélkül megmondják a telefonon bejelentkező szervizesnek, vagy éppen egy „áltitkárnő" telefonálhat be a rendszergazdához saját vagy inkább a főnök elfelejtett jelszava ügyében vagy más probléma kapcsán[7]. Tapasztalatom szerint a módszer használható arra is, hogy egy titkos telefonszám tulajdonosának adatait a tudakozóból megszerezzük. Nem könnyű feladat – bár nem lehetetlen – az ügyfélszolgálatosokat meggyőzni arról, hogy szegje meg a munkaadó szabályzatát, így elég kicsi a siker aránya is. Mindez lényegében azt jelenti, hogy a támadó az emberek manipulálásával kerüli meg a védelmi rendszert. Ezt a módszert hívják nemes egyszerűséggel „*social engineering*"-nek. (~társadalommérnökség, de nekem legjobban a *pszicho-hack* kifejezés tetszik.) Egy összetett védelem esetleges gyenge pontjai így nemcsak magából a rendszerből, hanem a kezelői hibákból, emberi mulasztásokból is adódhatnak. Minden rendszer leggyengébb láncszeme az ember.

[7] Valahogy úgy, mint az „*Adatrablók*" (*Hackers*) című film elején…, bár nem volt egészen világos, hogy egy informatikai problémához mi köze a rendészetnek, de lényegében erről volt szó. Sokkal több, de főként árnyaltabb példát láthatunk a „*A rendszer ellensége*" (*Takedown*) című filmben, amely Kevin Mitnick elfogásának történetét próbálja elmesélni több, de inkább kevesebb sikerrel...

Ha már a humán oldal vizsgálatánál tartunk, mik is az adatlopás, betörés fő kiváltó okai? Erre a következő hét emberi tulajdonság és cselekedet adja meg a leggyakoribb választ, melyek egyaránt lehetnek motiváló tényezők vagy módszerek (***7E***):

Hiúság	***E***go
Sikkasztás	***E***mbezzlement
Lehallgatás	***E***avesdropping
Ellenségeskedés	***E***nmity
Kémkedés	***E***spionage
Zsarolás	***E***xtortion
Hibás döntés	***E***rror

Egy rendszer védelme naplózással tovább javítható. Fel kell jegyezni minden eseményt és tevékenységet: ki mit csinált és mikor tette azt. Sajnos a legtöbb esetben csak utólagos ellenőrzésre van mód, amikor már „*baj van*”, de a naplók rendszeres ellenőrzése fényt deríthet sikertelen betörési kísérletekre vagy más rendellenes működésre, felhasználói viselkedésre is.

És egy pár szó erejéig meg kell említeni azt a védelmi megoldást, ami relatíve a legolcsóbb, de egyben a leghatékonyabb is, bár csak a támadások egy része ellen nyújt védelmet. Ez a *fizikai védelem*. Sajnos nagyon sok – szakszerűtlenül előkészített – védelmi megoldásnál nem fektetnek kellő hangsúlyt rá. A legtöbb problémát tehát nem maga a rejtjelezés alkalmazása, hanem a kialakított védelmi rendszer egészének ellenálló képessége, *egyenszilárdságának* biztosítása okozza. Egy védelmi rendszer akkor egyenszilárdságú, ha bármely pontján is támadjuk meg, ugyanakkora erőforrást kell befektetnünk a sikerért. Nincs olyan pontja, amely gyengébb védelmet nyújtana, mint egy másik.

A Függelék jelen fejezethez kapcsolódó alfejezetei

14.1. A titkosítás értékelése és alapvető feladatai

14.8. Szabványok összefoglaló táblázata

További kiegészítések folyamatosan bővülő helye a http://www.netacademia.net/konyv webcím.

History has taught us: never underestimate the amount of money, time, and effort someone will expend to thwart a security system. It's always better to assume the worst. Assume your adversaries are better than thay are. Assume science and technology will soon be able to do thins they cannot yet.

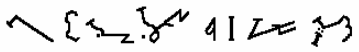
A történelem megtanított minket: soha ne becsüljük alá azt a mennyiségű pénzt, időt és erőfeszítést, melyet valaki egy biztonsági rendszer hatástalanítására szán. Mindig a legrosszabb esetre kell felkészülni. El kell fogadni, hogy az ellenfél jobb, mint a rendszer. El kell fogadni, hogy a tudomány és a technológia fejlődése előbb vagy utóbb lehetővé teszi azt, ami ma még nem lehetséges.

Bruce Schneier
Counterpane Inc.

2. A TITKOSÍTÓ MÓDSZEREK TÖRTÉNELMI ÁTTEKINTÉSE

Ez a fejezet történeti áttekintést ad a titkosító módszerekről, példaként ismertetve néhány klasszikusnak számító módszert. Megismerhetjük a titkosítás célját, a szimmetrikus kulcsú titkosító módszerek alapjait. A mai kriptográfiának (*görög kryptos = titkos, graphos = írás*) matematikai eszközökkel kell biztosítani azt, hogy stratégiai fontosságú információk, üzleti adatok, dokumentációk vagy személyiségi jogokat érintő adatok csak az azok felhasználására kijelölt körben legyenek elérhetők, ne juthassanak illetéktelenek birtokába. Korábban ugyan mások voltak az eszközök, de a cél nem változott.

A bevezetőben említettem, hogy ma már szinte megfejthetetlen algoritmusok állnak rendelkezésünkre. Egykor azonban nem voltak ilyen „top”-módszerek, a legtöbb alkalmazott titkosítás alapja inkább az ötletesség, a találékonyság volt *(helyettesítéses ábécék, sablonok),* vagy bizonyos ismeretek tudására vagy nem tudására alapoztak *(gyorsírás, rovásírás)*. Bizonyára most felvetődik a kérdés, hogy miért hozom szóba egyáltalán ezeket a régi eljárásokat? A válasz rendkívül egyszerű: napjaink algoritmusai is ugyanezeken az elveken nyugszanak! A későbbi fejezetek elolvasása után mindenki számára belátható lesz, hogy a legkorszerűbb titkosítások építőkockái ugyanazok az elvek és „elemi algoritmusok”, melyekről most lesz szó. A fejezet során bemutatom az egy- és többábécés helyettesítő titkosítókat, a keverő titkosítókat valamint minden kriptográfus (rém)álmát, az egyszer használt kulcsok algoritmusát. Így jutunk majd el a valóban ősi megoldásoktól a XVI-XVIII. századi módszereken át a II. Világháborúban alkalmazott elektromechanikus titkosítási eszközökhöz. A végső (de nem utolsó) állomás többek között a DES, a matematikai eszközökkel létrehozott RSA és az AES lesz. Mindezek fényében higgyék el a kételkedők, ez nem időpazarlás...



2. ábra A Tiro - féle gyorsírás néhány jele

2.1. HELYETTESÍTŐ TITKOSÍTÁSOK – S-BOXOK

A ***helyettesítéses titkosítók*** *(*substitution *ciphers, S-Box, S-dobozok)* megvalósítására számtalan példát lehetne mutatni, de ezeknek a lényege mindig ugyanaz: a titkosítandó üzenet egyes betűit, jeleit vagy jelcsoportjait egy másik betűvel, jellel vagy jelcsoporttal helyettesítjük. Általánosítva: *az üzenet egy elemének csak az alakja változik meg, az üzenetben elfoglalt helye nem*. Minél bonyolultabbak a jelek, vagy minél több betűből állnak a helyettesítő betűcsoportok, annál több jelentés nélküli elem szúrható be a titkosított üzenetbe, egyre nehezítve így a kulcs nélküli feltörést.

2.1.1. Caesar módszer

A legegyszerűbb esetben egy betűt egy betűvel helyettesíthetünk, de a helyettesítő ábécé betűit az eredetihez képest valamennyi pozícióval eltolva kapjuk meg. Az általánosított változatban *k* betűnyi eltolást használunk, de eredetileg *k=3* volt, vagyis A-ból D lett, B-ből E és így tovább. Hogy miért pont hárombetűnyi volt az eltolás, azt már nem tudjuk Julius Caesartól megkérdezni, de biztosan jó oka volt rá... Fontos megjegyezni, hogy a betűk sorrendje – szavakban lévő pozíciójuk, illetve egymáshoz viszonyított helyzetük – nem változik meg, csak a képük, alakjuk lesz más.

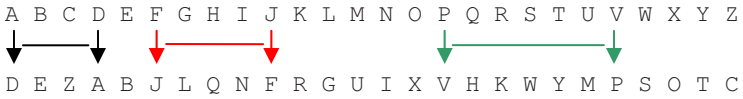
```
A nyílt szöveg betűi:
               A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A titkos szöveg betűi:
               V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
```

3. ábra A Caesar módszer ábécéje 5 pozíciós eltolásnál

Ha az eredeti szó „titkos”, akkor a titkosított párja : „odofjn”

A Caesar titkosító feltörésének legegyszerűbb – de nem egyetlen – módszere, ha az első néhány szónyi karakter alapján kipróbáljuk mind a 26 variációt. Ha a próbálgatáshoz felhasználunk egy kész Vigenere táblát (*8. ábra*), a feltörés mindössze 10-15 percnyi munka. (Aki nem hiszi, próbálja ki!) Másik hátrány, hogy egy betű helyes megállapítása az ábécé rendezettsége miatt, egyúttal az összes többi betű helyes megállapítását eredményezi.

Jobb a titkosítás eredményessége, ha a második sorban a betűket nem eltoljuk, hanem összekeverjük. Ekkor az eredeti 26 helyett $26!=4\times10^{26}$ lehetséges kulcs lesz (ennyiféle módon lehet a második sort felírni). Ha ezt valaki mind kipróbálja 1 millió próbálkozás/sec sebességgel, bizony 10^{13} évig fog próbálgatni. Célszerű továbbá a szóközök és az írásjelek kihagyása, mert egyes jellemző hosszúságú szavak (névelők, kötőszavak) segítségül szolgálhatnak a megfejtőnek. A kevert ábécék között van néhány speciális, mint az alábbi is:



4. ábra Speciális Caesar ábécé: önmaga inverze

Ez a nyílt szöveg – titkos szöveg összerendelés szimmetrikus titkosítást és megfejtést tesz lehetővé: mindkét művelethez ugyanazt a táblázatot, ugyanúgy kell használni. Ha a felső sorban megkeressük az A B C betűket, az alsó sorban rendre a D E Z betűket kapjuk eredményül. Megfejtéskor a D E Z betűket szintén a felső sorban megkeresve kapjuk az A B C megoldást. Hasonló szimmetriát látunk majd az Enigma működésében is: ha egy adott kezdeti beállítással állítottak elő egy rejtjeles üzenetet, azt ugyanolyan kezdeti beállítás mellett begépelve a nyílt szöveget adta a gép. A titkosításhoz és megfejtéshez pontosan ugyanabban a sorrendben ugyanazokra a műveletekre van szükség: az ilyen algoritmusok önmaguk inverzei. Fontos megjegyezni, hogy csak a betűk képe, alakja változik meg, de a szavakban lévő pozíciójuk illetve, egymáshoz viszonyított helyzetük nem változik meg.

Csoportos helyettesítés

Hasonló módszeren alapul a betűcsoporttal való helyettesítés is, itt azonban egy betűt nem egy karakterrel helyettesítünk, hanem többel. A módszer egyik hátránya, hogy a titkosított üzenet hossza annyiszorosára nő, ahány betű van a helyettesítő betűcsoportban. Az *5. ábra* táblázata egy három jelen alapuló ábécét tartalmaz. Ha az egyes betűket a táblázatba máshova írjuk, értelemszerűen a rejtjelezett üzenet is másként fog kinézni, de a megfejtéshez mindig ugyanarra a táblázatra van szükség, amellyel a titkosítást végeztük, vagyis a táblázat a megfejtés kulcsa. Az X-szel jelölt jelentés nélküli helyettesítő kódok (BCC, CCC) tetszőleges helyen beszúrhatók, vagy felhasználhatók szóközként. A módszer igazi gyöngéje azonban az, hogy a kriptoszövegből látszik, hány különböző jelet használunk, ami elárulhatja azt, hogy egy betűt hány betűvel helyettesítünk, így első lépésként meghatározható a betűk valószínű határa, és a betűk visszafejtésének máris neki lehet állni valamilyen módszerrel.

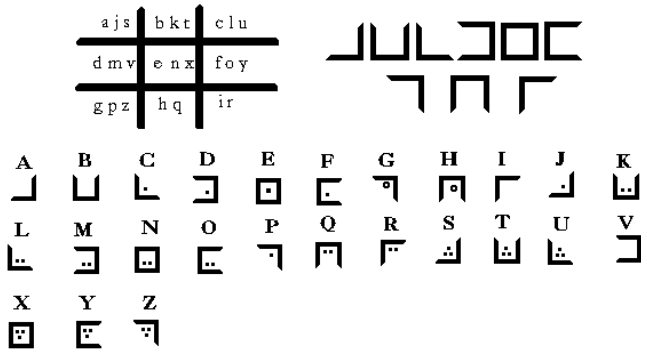
	AA	AB	AC	BA	BB	BC	CA	CB	CC
A	f	u	p	z	t	o	e	r	c
B	v	b	i	d	x	h	k	n	X
C	s	j	m	a	l	q	y	g	X

5. ábra A három jelen alapuló ABC

Ha az eredeti szó: „titkos”, akkor a titkosított szó: „abbbacabbbcaabccaa”

A jellel, jelcsoporttal való helyettesítés alapjaiban megegyezik a fentebb vázolt helyettesítéses módszerrel, viszont végletekig bonyolítható, a használt jelek közé rengeteg jelentés nélküli jel beszúrható. Ezeket a jelentés nélküli jeleket nem szabad lebecsülni, hiszen nagyjából ugyanannyi erőforrást igényel kideríteni egy jelről, hogy az égvilágon semmit sem jelent, mint meghatározni a valódi jelentését. Ezeket a jeleket egyébként *nullitásnak* hívjuk. Ha az *5. ábra* táblázatába az X-ek helyére mondjuk egy A betűt írunk be, a titkosításnál több lehetőségünk van azt eldönteni, hogy mivel jelöljük az A betűt: ugyanaz a szöveg és ugyanaz a kódtáblázat más eredményt adhat. Ez nem baj sőt, ha véletlenszerűen hol a CCC, hol a BCC, hol pedig a CBA betűhármast használjuk az A betű helyettesítésére, nehezíthetjük a visszafejtéssel próbálkozók dolgát. (Egyúttal a következő bekezdésben bemutatott gyakorisági elemzés alapját, a jellemző betűeloszlást is kiegyenlíthetjük). Gyakran nem baj, sőt előny, ha egy módszer rögzített feltételek – adott nyílt szöveg és adott kulcs – mellett több rejtjeles üzenetet is adhat (ezek a titkosított szövegre nézve nem determinisztikus módszerek), de **egy titkosított üzenetnek – egy adott kulcsra nézve – csak egy megfejtése lehet**, különben a megoldás nem egyértelmű! Ha nem szöveget, hanem számokat vagy bináris állományt titkosítunk ilyen módszerrel, ez végzetes lehet az adott üzenetre nézve.

> Már itt is szóba került és majd a brute-force támadásnál is előbukkan egy kifejezés: „értelmezhető”. Mit jelent ez? Attól függ, hogy ki értelmezi az adatot, illetve milyen adatról van szó. Ha a feldolgozás kimenete emberi értelmezésre alkalmas adat (például szöveg) és egy ember vizsgálja azt, valószínűleg nem jelent problémát a kisebb nagyobb elírások, karaktertévesztések felismerése és javítása. Ha egy gép teszi ugyanezt, a felismerés pontossága a szoftver algoritmusának hatékonyságán múlik. Amennyiben az adat egy bináris állomány vagy „1”-esek és „0”-ák kusza folyama, rendkívül nehéz meghatározni az értelmes megfejtés fogalmát.



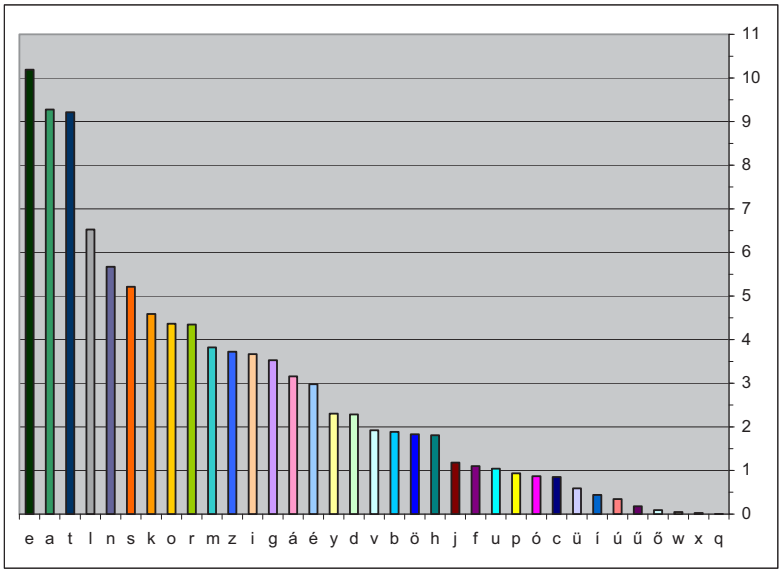
6. ábra Az Achiram - féle titkosírás ábécéje

Rengeteg jelentés nélküli jel szúrható a titkosított szövegbe. A jelkészlet egy másik, hasonló változata Karám jelábécé néven volt ismert.

A helyettesítő titkosítók feltörése

A helyettesítéses ábécék feltörése általában tapasztalati eszközökön alapszik. Ha ezzel a módszerrel elsősorban szövegeket titkosítunk, és ugyanazt a betűt mindig ugyanazzal a betűvel helyettesítjük, feltörésnél ki lehet használni a nyelv sajátosságát: az egyes betűk nem egyforma gyakorisággal fordulnak elő a szövegben. De nemcsak az egyes betűkre lehet jellemző gyakoriságot megállapítani, hanem a betűkettősökre (*digram*), sőt betűhármasokra (*trigram*) is. Ha a kódtörő tudja, hogy az adott nyelvben például az *e* betű a leggyakoribb, érdemes a rejtjeles szöveg leggyakoribb betűit ezzel helyettesítenie feltöréskor. A második leggyakrabban előforduló betűt a másodikkal. Így már kaphat olyan betűkettősöket és -hármasokat, amik alapján egy-egy rövidebb szót már ki tud találni, és lassan egyre több és több betű jelentését fejti meg. Minél hosszabb a megfejtendő szöveg, statisztikája annál jobban hasonlít a „referencia" statisztikához: vagyis egy hosszú szöveg visszafejtése valószínűsíthetően könnyebb, mint egy rövidé. A gyakoriságvizsgálat miatt a módszert szokás frekvenciaanalízisnek is nevezni. Az egyes jeleloszlások kiegyenlíthetők, ha egy betű helyettesítésére nem mindig ugyanazt a betűt használjuk.

Az alábbi grafikonon a magyar nyelvre jellemző betűeloszlást láthatjuk. Az ábra 11 magyar nyelvű regény és novella mintegy 4 500 000 karaktere alapján készült, a számszerű adatok a Függelékben megtalálhatóak egy másik diagram kíséretében (58. ábra), ahol az oszlopok nem gyakoriság szerinti, hanem ábécé rendben vannak. Ezek a hisztogramok egyébként meglepően jellemzőek egy-egy nyelvre, de különösen az ábécé-rendes ábrának van gyakorlati haszna… Az ilyen statisztikák használata előtt rendkívül fontos, hogy a fejtendő szöveg nyelvét és nyelvezetét helyesen becsüljük meg.



7. ábra A betűk eloszlása magyar szövegben (%)

Egy más megközelítésben ha tudjuk, hogy a szöveg milyen környezetből származik, vagy hova tart, lehet elképzelése a kódtörőnek arra nézve, hogy milyen szó vagy szavak fordulhatnak elő a rejtjeles szövegben. Például egy pénzügyi vagy üzleti üzenetben valószínűleg szerepel (előbb vagy utóbb) a „millió” szó. A szó egyik speciális tulajdonságát felhasználva - az I és az L egyaránt kétszer szerepel, és ráadásul egymást közrefogják -, olyan betűcsoportokat kell keresni az üzenetben, ami illeszkedik az ABBA mintára. Ezeket a feltételezett szavakat, mintákat *támpontoknak* hívjuk. Korábban szóltam róla, hogy ajánlott a szóközök eltávolítása a titkosítás előtt. Miért? Mert a tapasztalat szerint a szóköz általában kétszer gyakrabban fordul elő, mint a leggyakoribb betű, így rendkívül könnyű azonosítani. A szóhatárok megállapítása viszont a rövid, gyakori és ismert betűképű szavakra tereli a figyelmet. Belátható, hogy Caesar-titkosításhoz hasonló egyszerű helyettesítést végző eszközök nem biztosítanak megfelelő védelmet a gyakoriságelemzést vagy más nyelvi sajátosságot kihasználó támadás ellen. Az eddig leírt helyettesítéses ábécék gyengéje abban rejlik, hogy minden betűt mindig ugyanazzal a betűvel (jellel) helyettesítenek, és ez jó kiindulópont lehet a feltöréssel próbálkozó számára.

2.1.2. Vigenere titkosítás

Ebből adódik az egyik lehetséges megoldás is: a betűket más és más betűkkel kell helyettesíteni például attól függően, hogy hányadik helyen állnak az eredeti szövegben, vagy egy kulcsszó függvényében. Így lényegében nem egy ábécét használunk az egész folyamat során (*monoalphabetic substitution*), hanem többet (*polyalphabetic substitution*). Az első ilyen módszert *Blaise de Vigenere*, francia diplomata írta le az 1560-as években. Az elgondolás nem volt egészen a sajátja, de ő gyúrta először használható módszerré.

Numerikus módszer

Alakítsuk a betűket számokká: A…Z = 0…25. A kulcsszót ismétlődően írjuk a titkosítandó szöveg fölé! Ezután a nyílt szöveg betűinek és a kulcs betűinek megfelelő számokat adjuk össze *mod* 26. A kapott számsort ismét betűkké alakítva megkapjuk a rejtjelezett szöveget.

Kulcsszó:	**V**	**A**	**D**	**K**	**A**	**C**	**S**	**A**	**V**	**A**	**D**	**K**	**A**	**C**
Érték:	21	0	3	10	0	2	19	0	21	0	3	10	0	2
Nyílt szöveg:	**K**	**E**	**T**	**H**	**E**	**T**	**M**	**U**	**L**	**V**	**A**	**J**	**O**	**N**
Érték:	10	4	19	7	4	19	12	20	11	21	0	9	14	13
Összeg:	31	4	22	17	4	21	31	20	32	21	3	19	14	15
mod 26	5	4	22	17	4	21	5	20	6	21	3	19	14	15
Rejtjelezett:	**F**	**E**	**W**	**R**	**E**	**V**	**F**	**U**	**G**	**V**	**D**	**T**	**O**	**P**

Mint látható, az egyes betűket a titkosítás során a kulcsszó függvényében más és más betű helyettesíti: a 3. helyen álló T-ből W, a 6. helyen álló T-ből viszont V lett, így a betűeloszlások sajátos jellemzői megváltoztak. Minden pozícióban más ábécét használunk, azt pedig, hogy mikor melyiket, a kulcsszó dönti el. Ha a kulcsszó betűje az adott pozícióban A, akkor $k=0$, ha B, akkor $k=1$, ha C, akkor $k=2$ paraméterű Caesar-féle titkosítást használunk az adott pozícióban. Egy titkosított üzenet megfejtéséhez a kapott szöveget előbb számokká kell alakítani, ezután betűről-betűre a kulcsszó betűinek értékét ki kell belőlük vonni *mod* 26. Az eredményt betűkké alakítva megkapjuk az eredeti nyílt üzenetet.

Táblázatos módszer

Egy 26×26-os betűmátrix szemléletesebben (bár jóval tárigényesebben) valósítja meg a célt. A táblázat első sorába beírjuk az eredeti ABC betűit. A másodikba az egy karakterrel eltoltat, a harmadikba a két karakterrel eltoltat és így tovább. Már ránézésre is látszik az, amit eddig nem mondtunk ki: itt 26 ábécét fogunk használni. A tábla működése a numerikus módszeren alapszik, de a papír-ceruza feladatoknál gyakran praktikusabb egy ilyen segédeszköz. A táblát a következő útmutatás alapján használhatjuk:

- **Titkosításkor** a kulcsszó aktuális betűjét megkeressük az első oszlopban, a titkosítandó betűt az első sorban. Az oszlop és a sor kereszteződésében lévő betű adja a helyettesítést.
- **Megfejtéskor** megkeressük a kulcsszó betűjét az első oszlopban, majd az így kijelölt sorban megkeressük a titkosított szöveg aktuális betűjét. Ennek az oszlopnak a tetején van a nyílt szöveg megfelelő betűje.

Vigenere eljárásáról azonban majd' két évszázadig elfeledkezett a világ: megszületésekor kissé nehézkesnek tűnt, később egyszerűen elfelejtették. Miután újra felfedezték és egyre szélesebb körben használták, erősödni kezdett a gondolat: ez az első olyan eljárás, ami feltörhetetlennek tűnik, és a gyakorlat sokáig igazolta is ezt a feltevést. A módszer gyengéje elsőre nem nyilvánvaló, de belátható *Charles Babbage* törésének helyessége [34]. A következő oldalakon ezt mutatom be: a felismerést, majd a módszer alkalmazását egy példán keresztül.

	→ nyílt szöveg betűi →																								
→ kulcsszöveg betűi →	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

8. ábra Vigenere tábla

A kiemelt sorok a HUMOR kulcsszó által meghatározott ábécéket jelölik.

Vigenere titkosítás feltörése – Kasiski módszer

Ha visszaemlékszünk, a Caesar típusú rejtjelezők gyengéje az volt, hogy az egyes betűket mindig elárulta a gyakoriságelemzés, mindegy milyen alakot vettek fel. Ezzel szemben a Vigenere titkosítás ereje abban rejlik, hogy nem egy, hanem több ábécé-t használ, így ugyanazt a betűt mindig más és más betű mögé rejti, lehetetlenné téve így a sikeres elemzést. A használt ábécék számát a kulcsszó hossza határozza meg: ha az négy- vagy ötbetűs, akkor a Vigenere titkosítás négy vagy öt ábécét használ. A *8. ábra* táblázatán megfigyelhetjük, hogy az HUMOR kulcsszó használata esetén az E betű rendre a L, Y, Q, S és V betűk valamelyikére képződhet le. Ez a szabályszerűség azonban nemcsak a betűkre igaz, hanem a betűkből alkotott szavakra is, például a T E V E szó szintén csak ötféleképp titkosítható, attól függően, hogy a kulcsszó hányadik betűje van a T betű felett:

HUMORHUMO TEVE	UMORHUMOR TEVE	MORHUMORH TEVE	ORHUMORHU TEVE	RHUMORHUM TEVE
AYHS	NQJV	FSML	HVCY	KLPQ

Egy szót csak annyi különböző alakra tud a Vigenere-rejtjelező titkosítani, ahány betűs kulcsszót használunk. Ha egy szó sokszor fordul elő a titkosított szövegben, nagy valószínűséggel egy vagy több rejtjeles alakja ismétlődni fog. Példaként álljon itt egy rövid titkosítás, amelyben „A teve fohásza” első két sorát titkosítottam a H U M O R kulcsszóval:

```
humorhumorhumorhumorhumorhumorhumorhumorhumorhumorhumorhu
TeTevelTeveveEngemEleveTeveledNemErFelTevefejTetovaVeleje
AYFSMLFFSMLPQSENYYSCLPQHVCYXSUUYYSIMYXHVCYRSAAYFCMHPQZVQY
```

A nyílt szövegben a T E V E szó többször is felbukkan, ezért azt várjuk, hogy a kódszövegben a lehetséges öt változat közül több is előfordul, lehetőség szerint még ismétlődik is valamelyik alak. Szerencsénk van, mert két alakot is találhatunk és mindkettő ismétlődik[8]. Az első két esetben a T E V E szót F S M L, a második két esetben H V C Y jelöli. Az első ismétlődésének oka, hogy öt betűnyi távolságra vannak (a kulcsszó egyszer fordult körbe), a második esetben 15 betű a távolság (a kulcsszó háromszor fordult körbe). A kulcsszó körülfordulása után az ismétlődő párok felett a kulcsszó ugyanabban a pozícióban áll, következésképpen a T E V E szó titkosított párja is ugyanaz lesz. A kulcsszóhossz egész számú többszöröse által meghatározott távolságban lévő azonos nyílt szövegek azonos kódszöveget adnak. *Babbage* felismerte, hogy az ismétlődések jó támpontot jelentenek a feltörhetetlennek hitt algoritmus visszafejtéséhez. Az alábbi táblázat is ilyen ismétlődéseket mutat egy olyan rejtjeles szövegben, amelyről semmi mást nem tudunk, csak azt, hogy Vigenere titkosítással készült, és angol szövegről van szó. Nem ismerjük sem a kulcsot, sem az eredeti szöveget vagy részletet.

```
          0123456789 0123456789 0123456789 0123456789
  0: JZQIHFQJRQ ITIYJKSHGS NGISCWSJGJ RXFOLGKFTT
 40: AGWERDYAGP VWIBAWCEBT YTGTKQDFSG PVUXEZWZGC
 80: CWHGVYUPKH SUVAJKWLAJ WHMSEKKQPE KSAJZTMHWT
120: KMPLGMOVRW XKGTJRPLGM OVWDNFSFMC GLOJKVQPKS
160: TGKRTKHZCE RWXMSTVYHL IEGJAXXBUG RLSMSUJEMA
200: HUQYZJALCG PSCPUZWVFB DMVAPXQIAS QERLCHHQGK
```

Ismétlődő betűfüzérek	Ismétlődés száma	Előfordulási pozíciók	Pozíciók távolsága	Távolságok faktorai
GPV	2	48, 69	21	1, 3,7, 21
PLG	2	122,136	14	1,2, 7,14
LGM	2	123,137	14	1,2, 7,14
GMO	2	124,138	14	1,2, 7,14
MOV	2	125,139	14	1,2, 7,14
RWX	2	128,170	42	1,2,3,7,14
PLGM	2	122,136	14	1,2, 7,14
LGMO	2	123,137	14	1,2, 7,14
GMOV	2	124,138	14	1,2, 7,14
PLGMO	2	122,136	14	1,2, 7,14
LGMOV	2	123,137	14	1,2, 7,14
PLGMOV	2	122,136	14	1,2, 7,14

Például a P L G M karaktersor kétszer található meg, a 122. és a 136. pozícióban, ennek távolsága 14 betűhely. Az ismétlődés oka lehet az, hogy:

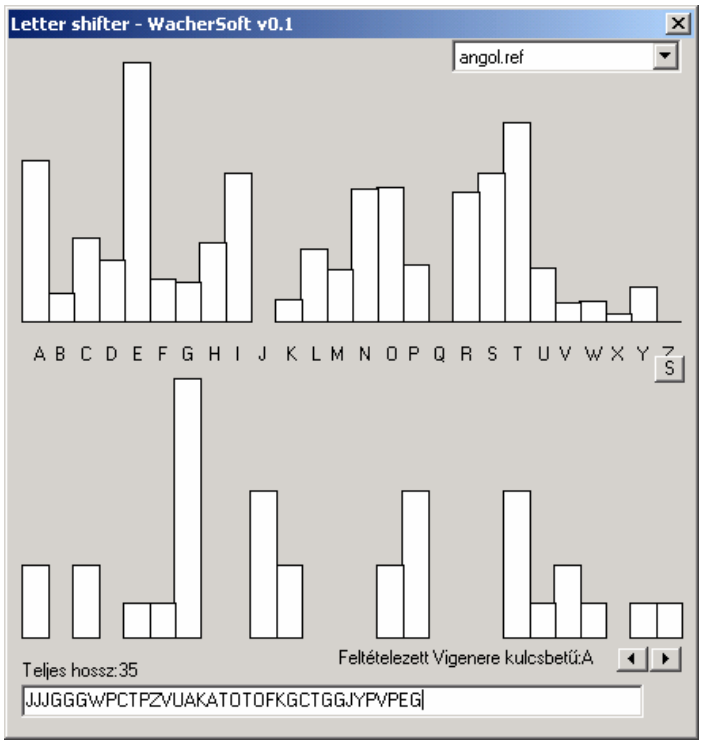
- a jelszó 1 betű hosszú és 14-szer fordult körbe
- a jelszó 2 betű hosszú, 7-szer fordult körbe
- a jelszó 7 betű hosszú és 2-szer fordult körbe
- a jelszó 14 betű hosszú és 1-szer fordult körbe.

[8] Természetesen el kell ismerni, hogy a példakódolás speciális, mert a mindennapi életben ritkán van ennyi ismétlődő szó egyetlen mondatban.

A többi ismétlődés távolságának faktorait is megvizsgálva azt látjuk, hogy a jelszó a legtöbb esetben 2, 7 vagy 14 betűs lehet. A kettő betűs jelszó használata elég valószerűtlen, ezért egyelőre vessük el. A 14 és a 7 betűs jelszó egyaránt jónak tűnik. Figyelembe véve, hogy az átlagos jelszóhossz 5-6 karakter körül van [22], tételezzük fel először azt, hogy a jelszó 7 betűs. Ha nem sikerül ezt igazolni, majd megpróbáljuk a fejtést 14 betűs jelszóval[9]. Egy hétkarakteres jelszó használata azt jelenti, hogy minden hetedik karakter ugyanakkora eltolású Caesar-kóddal van titkosítva, ezért emeljük ki először az 1., 8., 15., 22., 29., 36. stb. karaktereket. Ekkor a következő 35 karakteres betűfüzért kapjuk:

J J J G G G W P C T P Z V U A K A T O T O F K G C T G G J Y P V P E G

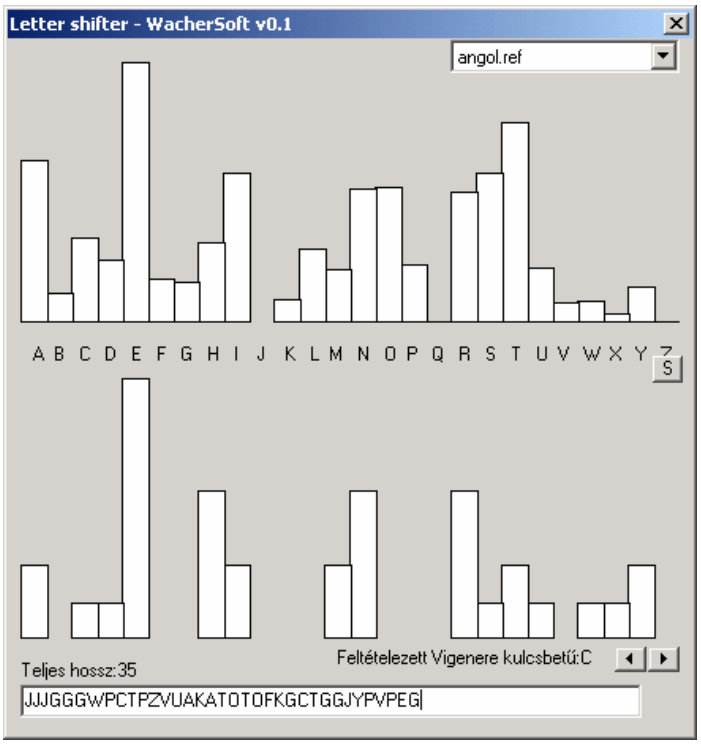
E karakterek előfordulási gyakorisága a karakterláncban a következő (az ábra alsó része):



Az angol nyelvben a leggyakoribb az E betű (lásd az ábra felső részét)[10]. Ezért próbaképpen tételezzük fel, hogy az alsó statisztika legkiugróbb betűje, a G nem más, mint az álruhás E betű. A referencia hisztogramon láthatunk még néhány jellegzetességet, de sajnos a mi hisztogramunknak nem nagyon van más olyan jellemzője, amit érdemben össze lehetne hasonlítani vele. (Talán csak az, hogy az I-O-T betűknél látható csúcsok többé-kevésbé egybeesnek

[9] Becsülettel bevallom, hogy a jelszóhossz nem szokott ennyire egyértelműen kijönni, a példa szembeszökősége a szerencsének köszönhető.
[10] Ez egyébként a magyar nyelvre is igaz, de egy latin betűkkel írt thaiföldi szövegben rengeteg „a" betűre számítsunk …

a mi hisztogramunk „csúcsaival”.) Léptessük balra kettő hellyel az alsó részt, és ellenőrizzük feltevésünket:



Ez még akár jó is lehet, így ha elfogadjuk, hogy a feltételezett eltolás E → G, akkor *k=2*, vagyis a kulcsszó első betűje C. Ekkor az eredeti karaktersorozat így néz ki a megfejtés után (8. ábra alapján):

J J J G G G W P C T P Z V U A K A T O T O F K G C T G G J Y P V P E G
h h h e e e e u n a r n x t s y i y r m r m d i e a r e e h w n t n c e

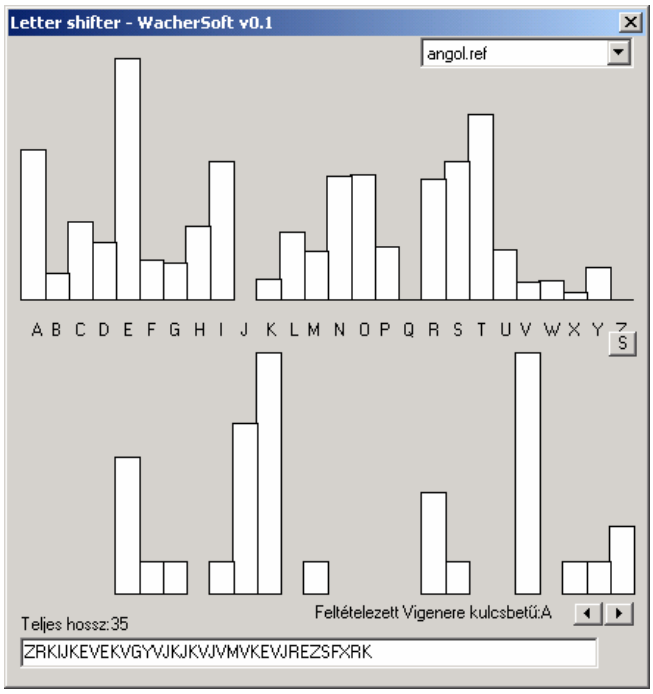
Eddigi eredményünk:

```
C??????
h......h......h......e......e......e......u......n
......a......r......n......x......t......s......y.
.....i......y......r......m......r......m......d..
....i......e......a......r......e......e......h...
...w......n......t......n......c......e.
```

A kulcsszó második betűjének meghatározásához emeljük ki a 2., a 9., 16., stb. betűket, melyek a következők:

Z R K I J K E V E K V G Y V J K J K V J V M V K E V J R E Z S F X R K

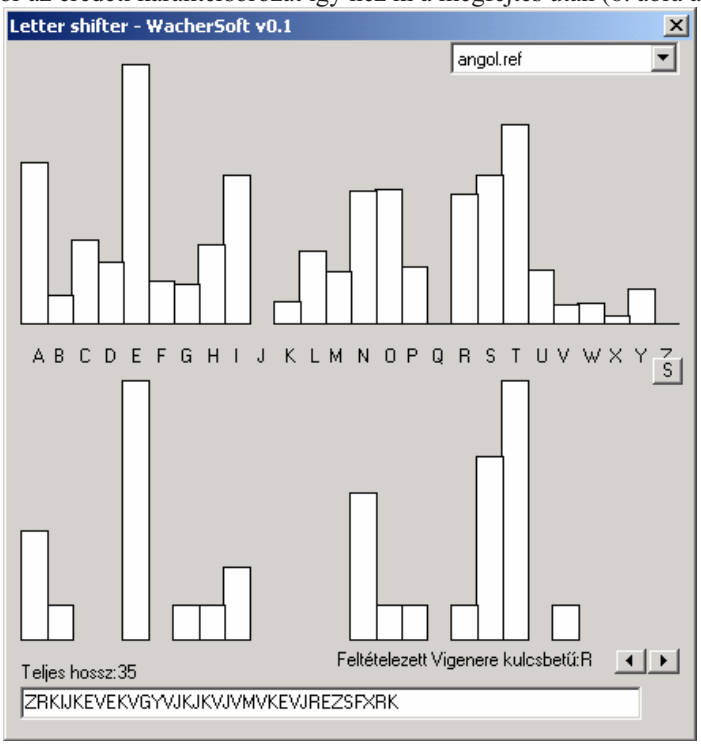
E karakterek előfordulási gyakorisága az ábra alsó részén látható:



Hát ez jóval szerényebb lehetőségeket kínál, mint az előbbi. Némi „szemmel verés” után gondolhatunk arra, hogy a jelenlegi I J K hármasnál lévő emelkedőnek van valami köze a referencia R S T emelkedéshez. Ezt a gondolatot alátámasztja az is, hogy ha az I J K-ból R S T lesz, akkor a jelenleg V-nél lévő kiugrás pontosan a referencia E alá kerül. Ehhez mindkét esetben 17 hellyel kell a hisztogramot balra forgatni. (Lehetne kilenccel jobbra is, de az első betűnél is balra forgattunk, maradjunk következetesek, még a végén eltévesztjük a kulcsbetű kiválasztását!) Biztassuk magunkat tovább azzal, hogy a jelenlegi V – vagyis a feltételezett E – környezete hasonlít a referencia E környezetére: tőle balra négy hellyel egy nagyobbacska kiugrás van, ami a referencia A-ra, a referencia E-től balra négy hellyel lévő oszlopra emlékeztet. A V-től jobbra lévő „néhány kis oszlop után egy nagyobb” (W X Y Z) hasonlít a referencia F G H I-re.

Miután teljesen meggyőztük magunkat a feltételezés helyességéről, próbáljuk ki azt, vagyis léptessük balra 17 hellyel az alsó hisztogramot, és ellenőrizzük feltevésünket!

Az előbb a kulcsszó C betűjéhez kettőt forgattunk, most pedig 17-et, így a kulcsszó második betűje: R Ekkor az eredeti karaktersorozat így néz ki a megfejtés után (8. ábra alapján):



Z R K I J K E V E K V G Y V J K J K V J V M V K E V J R E Z S F X R K
i a t r s t n e n t e p h e s t s t e s e v e t n e s a n i b o g a t

Eddigi eredményünk:

```
CR?????
hi.....ha.....ht.....er.....es.....et.....un.....n
e.....an.....rt.....ne.....xp.....th.....se.....ys
.....it.....ys.....rt.....me.....rs.....me.....dv.
....ie.....et.....an.....re.....es.....ea.....hn..
...wi.....nb.....to.....ng.....ca.....et
```

A további betűk megfejtését már nem részletezem, az eddigiek mintájára kell azokat is megkeresni. Nekem sajnos a harmadik betűt nem sikerült a hisztogram alapján beazonosítanom, ezért a többivel folytattam a munkát. A negyedik azonosítása nem volt nehéz: P betű. (CR?P???) Az ötödik betű nehézsége a másodikéhoz hasonlítható, egyébként értéke: T. (CR?PT??) A hatodik betű nagyon könnyű volt, az E és T betűk csúcsa alapján könnyen

meg lehetett határozni az eltolás mértékét. A hatodik betű: O (CR?PTO?) A hetedik betűt nem sikerült kitalálni. A megtalált kulcsbetűk alapján az alábbi részmegoldáshoz jutunk:

Harmadik betűcsoport (3., 10., 17., 24., stb. betűk) – kulcs: nem ismert

Q Q S S R F R W B Q U C U A W Q Z M R R W C Q R R Y A L M J C B Q L
? ?

Negyedik betűcsoport (4., 11., 18., 25., stb. betűk) – kulcs: P

I I H C X T D I T D X C P J H P T P W P D G P T W H X S A A P D I C
t t s n i e o t e o i n a u s a e a h a o r a e h s i d l l a o t n

Ötödik betűcsoport (5., 12., 19., 26., stb. betűk) – kulcs: T

H T G W F T Y B Y F E W K K M E M L X L N L K K X L X M H L U M A H
o a n d m a f i f m l d r r t l t s e s u s r r e s e t o s b t h o

Hatodik betűcsoport (6., 13., 20., 27., stb. betűk) – kulcs: O

F I S S O A A A T S Z H H W S K H G K G F O S H M I B S U C Z V S H
r u e e a m m m f e l t t i e w t s w s r a e t y u n e g o l h e t

Hetedik betűcsoport (7., 14., 21., 28., stb. betűk) – kulcs: nem ismert

Q Y N J L G G W G G W G S L E S W M G M S J T Z S E U U Q G W A Q Q
? ?

Az eddigi eredmény tehát a következő:

CR?PTO?

hi.tor.ha.tau.ht.sne.er.nde.es.ima.et.eam.un.ofm.n
e.tim.an.eff.rt.ome.ne.ill.xp.ndt.th.art.se.uri.ys
.ste.it.alw.ys.ett.rt.ass.me.hew.rs.ass.me.our.dv.
rsa.ie.are.et.ert.an.hey.re.ssu.es.ien.ea.dte.hn.l
og.wi.lso.nb.abl.to.oth.ng.the.ca.not.et

Nem lenne meglepő, ha a kulcsszó harmadik betűje Y lenne. Tegyük fel, hogy az! A titkosított szöveg harmadik pozícióján Q van, ehhez pedig Y kulcs mellett S nyíltszöveg tartozik. (**histor?**) Ez értelmes szónak tűnik, főként, ha a hiányzó hetedik nyílt betű az Y , amihez szintén Q titkosított betű tartozik. (**history**). Ha ez igaz, akkor az Y nyílt és a Q titkosított betűhöz az S kulcsbetű tartozik. (A 8. ábra Vigenere tábláján az Y oszlopban lévő Q betű sora S-el kezdődik.) Így a teljes kulcsszó: CRYPTOS a szöveg pedig nem más, mint a jelen fejezet mottója. Most egyesek talán úgy gondolják, hogy a végét nagyon elnagyoltam, és biztosan meg se csináltam. Higgyék el, a megfejtés a részeredmények felbukkanásával egyre egyszerűbb és egyszerűbb lesz, ezt fejezi ki a „*9. Üzenetpecsétek*” fejezet mottója is. Az egész olyan, mint egy láncreakció. Nem befejezni nehéz, hanem elkezdeni, és jó barátságban kell lenni a 8. ábrával.

A figyelmes olvasónak feltűnhet, hogy a fejezet címében *Kasiski*-módszer áll és nem *Babbage*-módszer. *Babbage* valószínűleg 1854-ben fejtette meg a Vigenere-titkosítást, de felfedezését egyszerűen nem publikálta, arra csak a XX. században derült fény, amikor a *Babbage* hagyatékot átvizsgálták. Időközben, 1863-ban rátalált a módszerre – és publikálta is azt – *Friedrich Wilhelm Kasiski* is, a módszer azóta az ő nevét viseli (*Die Geheimschriften und die Dechiffrir-Kunst – A titkosírás és a desifrírozás művészete, 1863*).

2.2. KEVERŐ TITKOSÍTÓK – P-BOXOK

A helyettesítéses módszerek a betűk helyét nem változtatják meg, csak alakjuk lesz más. Ezzel szemben a ***keverő titkosítók*** *(permutation ciphers, P-Box, P-doboz)* a betűk megjelenését hagyják változatlanul, viszont sorrendjüket a kulcs függvényében megváltoztatják: például egy adott oszlopszámú táblázatot használva az üzenetet a sorokba balról-jobbra írjuk és a rejtjeles szöveget az oszlopokból, fentről-lefelé kapjuk meg. Itt a táblázat oszlopszáma a kulcs.

A	N	A	G	Y
T	A	N	Á	C
S	D	Ö	N	T
É	S	E	N	Y
O	M	Á	N	X

Eredeti szöveg:
A NAGYTANÁCS DÖNTÉSE NYOMÁN …

A rejtjelezett üzenet (szóközök nélkül):
ATSÉONADSMANÖEÁGÁNNNYCTYX …

Az oszlopokat megkeverhetjük egy kulcsszó felhasználásával is. A táblázat fölé írjuk egy olyan kulcsszót, melyben nincs két egyforma betű! A kulcs szerepe az oszlopok megszámozása lesz olyan módon, hogy az első oszlopot az a kulcskarakter fogja jelölni, amelyik az ábécében legelőször szerepel. A többi oszlop sorrendje hasonlóan dől el. A nyílt szöveget beírjuk a sorokba, a rejtjelest pedig az oszlopokból olvassuk ki, a megállapított sorrend figyelembe vételével. Valahogy így:

M	**O**	**U**	**S**	**E**	← A kulcsszó, amely EMOSU sorrendet eredményez.
A	N	A	G	Y	
T	A	N	Á	C	Eredeti szöveg: A NAGYTANÁCS DÖNTÉSE NYOMÁN …
S	D	Ö	N	T	
É	S	E	N	Y	A rejtjelezett üzenet (szóközök nélkül): YCTYXATSÉONADSMGÁNNNANÖEÁ …
O	M	Á	N	X	

2.3. PRODUKCIÓS TITKOSÍTÓK

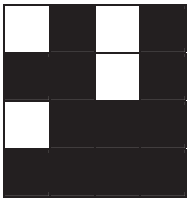
Produkciós titkosítónak (*production ciphers*) az olyan módszereket nevezzük, melyek kettő vagy több ***eltérő elvű*** művelet kombinálásával szolgáltatják eredményüket. A produkciós titkosítók nagyobb biztonságot kínálnak, mint a benne szereplő műveletek külön-külön. Ha azonos elvű algoritmusokat kötünk sorba, előfordulhat, hogy azok egymás hatását kioltják vagy a biztonságot nem növelik számottevően, csak a feldolgozási időt. Emiatt elfogadott az a tervezési módszer, hogy a produkció részműveletei egymástól eltérő elven működjenek. Egyik speciális eset a helyettesítő-keverő hálózat (*substitution - permutation network*, *SP network*), amely helyettesítéseket (*S-doboz*) és keveréseket (*P-doboz*) végez egymás után.

2.4. NÉHÁNY EGYSZERŰBB PÉLDA A TÖRTÉNELEMBŐL

Természetesen a régi módszerek sem csak a fentebb vázolt ábécékben merültek ki. Az emberi leleményességnek köszönhetően sok - sok módszer volt az üzenetek rejtjelezésére, de ezek felsorolására és ismertetésére most nem vállalkozom. Csak a teljesség igénye nélkül, csupán érdekességképpen röviden lássunk néhányat:

Ha egy papírcsíkot feltekerünk egy adott vastagságú rúdra úgy, hogy minden egyes fordulat után a csíkok egymás mellé illeszkedjenek, egy olyan hengerszerű felületet kapunk, ami összefüggő és lehet rá írni. Erre felírjuk az üzenetet, majd a papírcsíkot legöngyöljük róla. Az üzenet csak egy ugyanekkora méretű rúd segítségével olvasható el, így most a rúd mérete a *kulcs.* Érdemes vigyázni arra, hogy az egyes betűk mindig a csíkon maradjanak, ne forduljon elő olyan, hogy a betű egyik fele a csík egyik részén van, a másik a másik részén, mert ily módon a feltörésnél a betűk összeillesztése kiinduló pont lehet hiányzó átmérő meghatározásához. Egyébként ez egy keverő titkosítás, ami „meglepő" módon szintén az ókorig nyúlik vissza. A bal oldali ábrán egy „*Skitali*", egy spártai vezéri pálca látható, amit a fenti módszer szerint titkosításra is használtak. Az *American Cryptogram Association* hivatalos szimbóluma. (A pálcán olvasható felirat: „Intellegere est praevalere" – szabad fordításban „a tudás naggyá tesz")

Aki egy sablonnal titkosított üzenetet próbál megfejteni a sablon ismerete nélkül, igen kellemes időtöltésnek néz elébe. A művelethez két módszer is szóba jöhet: az egyik, hogy a titkosítandó szöveg szavait fogalmazzuk bele egy gyanúmentes szövegbe és a sablont a szövegre helyezve csak az eredeti üzenetet látjuk (ez viszont nem más, mint szteganográfia). A másik módszer az, hogy készítünk egy olyan négyzet alakú sablont, amin csak a betűk helye van kivágva. Ennek a sablonnak illik teljesítenie azt a feltételt, hogy elforgatással minden mezőt egyszer, és csak egyszer fed fel, ezekbe lehet beírni az eredeti üzenetet. Sablon nélkül csak egy négyzethálót látunk, amibe egy csomó betű van írva. A betűk képe nem változik meg, csak a sorrendje, ezért ez – a Skitalihoz hasonlóan – keverő titkosító.



H	A	O	P
H	A	L	S
N	Z	V	I
Z	E	A	M

9. ábra Egy sablon és a vele titkosított üzenet

A II. Világháborúban fogtak el egy férfit, akinél a következő feljegyzést találták[6]:

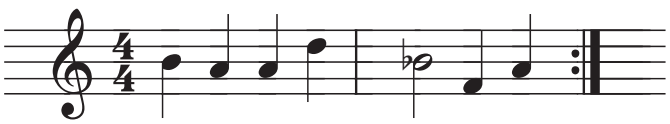
Helység	*Lakosság*
Urvados	34276
Décolemavas	92165
Nelomi	52610
Pogézos	23475
Trabkano	358716
Redakamin	74398

Állítása szerint a városok népességének fejlődését vizsgálta, azonban a felsorolt helységek Dél-Amerika jelentéktelen községei voltak. Mivel a feltüntetett lélekszámok valótlanok voltak, elkezdték vizsgálni a feljegyzést. Végül rájöttek, ha az „Urvados" szó harmadik, negyedik, második, hetedik és hatodik (a lakosság számának jegyei) betűjét összeolvassák, a „varso" szó jön ki. A többi helységnéven is végigmenve ezzel a módszerrel megkapjuk az igazi üzenetet: „Varsó védelme inog, északon támadni". A módszer a keverő titkosításhoz hasonló.

A következő üzenetre 1945-ben lettek figyelmesek Budapesten. A látszatra ártatlan levél valójában egy kém jelentése volt, és így szólt:

„Kedves Sándor!

Elküldöm Neked annak az indulónak első sorát, amelyet komponáltam. A szovjet hadsereg, mint tudod, bennünket már felszabadított. Átkelt a nevetséges csapdákon, sőt a Dunán is. Remélem, nemsokára vége lesz a háborúnak, és az idén ismét a Balaton mellett nyaralhatunk. Tegnap este felé egyedül voltam itthon, néztem az utcát és éreztem, hogy szívembe áramlik a hála és a jókedv, akkor vetettem papírra ezt az egyszerű dalt. Íme a kotta:



Fogadd szívesen, szeretettel küldöm. A viszontlátásig ölel barátod,

Péter"

A levél azért vált gyanússá, mert a kotta zeneileg eléggé „furcsa". Végül sikerült megfejteni a titkot: a kotta hangjai sorrendben: *h, a, a, d, b, f, a*. A levélben az első h-val kezdődő szó a *hadsereg*. Ezután a következő a-val (á-val) kezdődő szó az *átkelt*. Ha ezt az eljárást folytatjuk, megkapjuk a valódi üzenetet: „Hadsereg átkelt a Dunán, Balaton felé áramlik". A módszer a sablonos titkosításhoz hasonló, de ha figyelembe vesszük azt, hogy a „fontos" szavak egy csomó „nem fontos" közé vannak rejtve, a később ismertetett szteganográfiának is lehet hozzá köze…

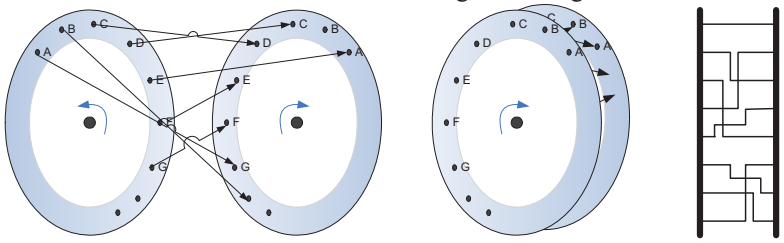
10. ábra Székely-magyar rovásírás jelei

Az iménti felsorolás korántsem teljes, de ízelítőt adhat abból, hogy gyakorlatilag bármilyen módszert lehet titkosításra használni, ha az kellően bonyolult(nak néz ki), vagy olyan kiegészítő információt használunk fel, amely ismerete nélkül a titkosított szöveg nem fejthető meg. A korábban említett Tiro-féle titkosírásnál a kulcs maga a tudás, a jelek ismerete. A helyettesítéses ábécében, az Achiram-féle jelkészletnél (**6**. ábra) maga a táblázat a kulcs, ami a titkosítást meghatározza. A Vigenere-módszer esetében a kulcsszó határozza meg, hogy melyik pozícióban melyik ábécét kell használni. Azonban az, hogy egy-egy ilyen módszer menynyire áll ellen a támadásoknak, már más, és korántsem könnyű kérdés.

Tegyünk egy nagyobbacska ugrást az időben, és ismerkedjünk meg egy olyan eszközzel, amelynek működése a korábbi sémákon alapul. Elve talán a Vigenere módszerhez hasonlít leginkább, bár attól eltérő, de van benne többábécés helyettesítés és keverés is. Az eszközt, amely az első elektromechanikus eszközök egyike, a II. Világháború során használták, és feltörése a Szövetségesek egyik legfontosabb eredménye volt a háború során.

2.5. ENIGMA

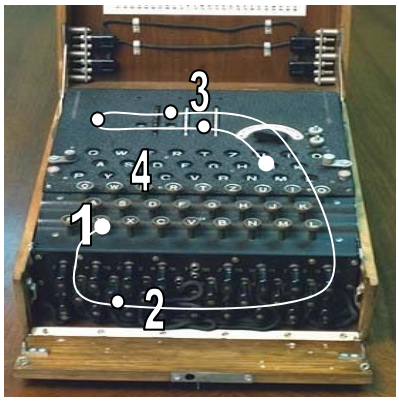
A kriptográfia kései történelmének egyik legfontosabb és legmeghatározóbb titkosító eszközei a rotor-alapú készülékek voltak. Számos pneumatikus és optikai változatuk létezett, de a legelterjedtebbek az elektronikus megoldások lettek. Egy rotort úgy kell elképzelni, mint két korongot, melyek egymáshoz vannak rögzítve, együtt forognak, rajtuk egyenként 26 elektromos érintkező van. A szemben lévő érintkezőit tetszőlegesen kötögessük össze valahogy így:



Ha egy elektromos jel a „szendvics” bal oldalán az A érintkezőre érkezik, akkor a „szendvics” másik oldalán a G betűnél fog távozni, vagyis ez egy *egyábécés* helyettesítés. Most fogjunk kettő ilyen rotort, és tegyük őket egymás mellé. Az elsőn A→G, a másodikon G→F lesz a jel útja. Majd bonyolítsuk egy kicsit a helyzetet: minden egyes jel után az egyik rotor lépjen egyet! Ha körbefordult, lépjen egyet a következő rotor is!

Az Enigma a II. Világháborúban a németek által használt titkosítógép volt, működésének alapelve az előbbi rotor-elv. 1918 körül tervezte *Arthur Scherbius* Németországban, és mintegy tíz évvel később kezdték általánosan használni a hadseregben a légi- és tengeri erőknél, valamint néhány kormányzati szervnél, illetve az üzleti életben. Maga a gép nem katonai fejlesztés eredménye, sőt megjelenése jóval megelőzte a II. Világháborút is. Szakkiállításokon is szerepelt, de a hadsereg felfigyelt rá és kivonta a forgalomból. A neve sem egyetlen gépet takar, hanem egy gépcsaládot, amelynek a „G”-vel jelölt szériáját használta a német titkosszolgálat (a képen is egy ilyen G-312-es látható) [11]. A rádiózás elterjedése és fejlődése új problémát jelentett az információközlésben: a rádió nagy előnye, hogy segítségével roppant gyorsan lehet üzenetet továbbítani, hátránya viszont, hogy bárki lehallgathatja, ha van rádióvevője. Ezért úgy tűnt, nem lehet használni a katonai és diplomáciai információcserében. Megoldást kellett találni a rádión továbbított üzenetek védelmére és e törekvés egyik eszköze lett az Enigma gépcsalád. Sokáig úgy tartották, hogy a gép titkosítása feltörhetetlen, azonban a szövetséges csapatok titkosszolgálatai a lengyelek kutatásai alapján már a háború korai szakaszában megfejtették azt, és végül folyékonyan olvasták a rejtjelezett üzeneteket. A lengyel titkosszolgálat ugyanis nem sokkal a kereskedelmi forgalomban való megjelenés után beszerzett egy (kereskedelmi) modellt, sőt 1928-ban hozzájutott – igaz, csak néhány napig – egy katonai változathoz is, amiről másolatokat is készítettek. A feltörés sikerességét két további tényező segítette: egyrészt a német felső vezetés gyakran kisebb hatásfokú titkosító gépek használatával küldte el a következő napi beállításokat tartalmazó parancsokat, másrészt a tevékenységet hírszerző munka is kiegészítette. A francia és az angol titkosszolgálatok és kódfejtőik megfejthetetlennek tartották az Enigmát, ezért gyakorlatilag nem is próbálkoztak vele. Az első előrelépés *Hans-Thilo Schmidt*, egy hazájától elfordult német férfi árulásának volt köszönhető. Schmidt a *Chiffrierstelle* munkatársaként hozzájutott néhány Enigmával kapcsolatos irathoz, melyet a franciáknak adott át (1931). Sem a franciák, sem az angolok nem tudtak mit kezdeni ezekkel. Francia- és Lengyelország között azonban létezett egy katonai együttműködési szerződés, és ennek keretében a franciák egyszerűen átadták a lengyeleknek a Schmidt-féle dokumentumokat. Az Enigma feltörésének reménytelen feladata így a lengyelekre maradt.

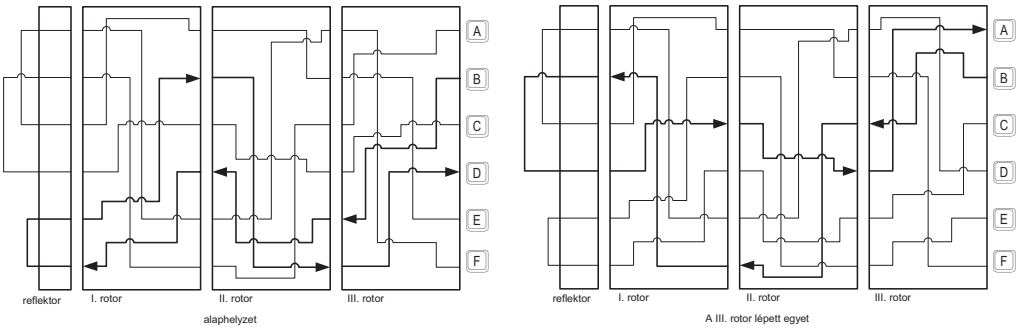
[11] A G312-es sorozat egyetlen megmaradt példányát (melynek eredetileg sem volt stecker-e) 1998-ban ajándékozta az angol titkosszolgálat a Bletchley Park múzeummá alakított egykori kódtörő központjának. A 150 000 font eszmei értékű gépet 2000. április 1-én – korántsem áprilisi tréfaként – ellopták a múzeumból. Fél évvel később postán visszaküldték a gépet. Előzőleg 25000 font „váltságdíjat” kértek érte – amit a múzeum hajlandó lett volna kifizetni – azonban a pénzért végül senki sem jelentkezett.



A gép a következő elemekből állt:

- egy 26 latin betűs billentyűzet (1)
- kapcsolótábla (*Stecker*) – a kezdeti keverés biztosítéka (2)
- három forgó tárcsa (*rotor*) + reflektor (3)
- egy 26 lámpás kijelző, ami a titkosítás és a megfejtés eredményét mutatta (4)

Az elrendezési mód gyakorlatilag három helyettesítő eszköz soros összekötésének felel meg. Tételezzük fel, hogy az alábbi ábrán a billentyűzet a III. rotorhoz csatlakozik, és egy B betűt nyomtak meg (a kapcsolótábla keverő hatását most nem vesszük figyelembe). A III. rotoron a B-ből E lesz, a II. rotoron a E-ből D, végül az I. rotoron a D-ből F. Ezután az I. rotor F betűjét az úgynevezett „reflektoron" keresztül visszavezetjük erre a „rotorszendvicsre". Azt, hogy a kilépő F betű miként fog visszatérni a rotorokhoz, szintén kézzel lehetett huzalozni, vagyis a reflektor is szabadon konfigurálható volt.



11. ábra Az Enigma három rotorjának egy lehetséges huzalozása

Lépjen vissza mondjuk E betűként! Ekkor rendre az E→B B→F F→D átalakítás fog megtörténni és végül D betűként lép ki. A titkosítás szimmetrikus, tehát ha D betűt nyomtak le, akkor B betű lépett ki, így ugyanaz a gép ugyanabban a konfigurációban egyaránt alkalmas volt titkosításra és megfejtésre. Ez a reflektornak volt köszönhető, amely 13 elektromos csatlakozópárral zárta a „kört" és vezette vissza a kilépő kódot egy ismételt helyettesítésre. A reflektoron nem lehetett olyan beállítást elérni, hogy egy betű önmagaként lépjen vissza, így a művelet során egy betű soha nem képződött le önmagára. Ez a speciális tulajdonság azonban igen sok lehetséges kombinációt kizárt, valamelyest könnyítve a kódfejtők dolgát is. További szabály volt, hogy a kezdeti keverést biztosító kapcsolótábla sohasem cserélt meg két szomszédos betűt, valamint nem az egész ábécét cserélgette meg, hanem csak hat betűpárt. Ha a gép csak ennyit tett volna, a betűk gyakoriságára vonatkozó vizsgálatok hamar feltörtek volna

minden üzenetet, azonban a már megismert rotorelv szerint forogtak: a III. (*fast*) rotor minden egyes gombnyomás után lépett egyet, a II. (*medium*) rotor akkor, ha a III. körbefordult és az I. (*slow*), ha a II. körbefordult. A rotorok és a reflektor huzalozása valamint a kezdő pozíciók minden nap változtak.

12. ábra Egy négyrotoros Enigma-variáns - közelebbről

1938-ban a németek további két tárcsát vezettek be, így a mindenkori három tárcsát már öt tárcsa közül választhatták ki. Az Enigma lehetséges kulcsainak számát minden eddig megismert alkatrésze az alábbiak szerint befolyásolta:

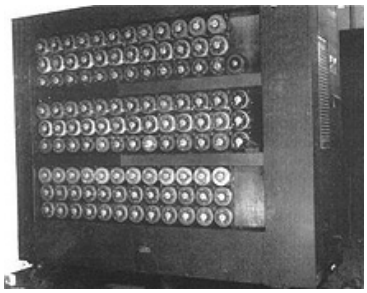
Rotorok
– három rotor, egyenként 26 lehetséges pozícióval: 26×26×26 = ***17576***
– három rotor a következő 6 sorrendben rakható egymás mellé: ***6***
(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1)

Stecker - kapcsolótábla
– a 26 betűs ábécé hat betűpárja rengeteg módon cserélhető meg:
$(26\times25 \times 24\times23 \times 22\times21 \times 20\times19 \times 18\times17 \times 16\times15) / (6! \times 2^6) =$
100 391 791 500

$100391791500 \times 6 \times 17576 =$ ***10 586 916 764 424 000***

Öt tárcsa esetén, amiből tetszőleges hármat választhatunk, ez a szám pontosan tízszeresére emelkedik...

Az Enigmával foglalkozó lengyel kriptográfusok *Marian Rejewski* vezetésével rájöttek arra, hogy az üzenetek első három betűje a tárcsák kezdőbetűit (kezdő pozícióit) azonosította, sőt ez a három betű a biztonság kedvéért még egyszer megismétlésre került az üzenet elején. Ezt a hat betűt az adott napi parancs szerint titkosították, de magát az üzenetet már ezzel a – viszonykulcsként, üzenetkulcsként alkalmazott – rotorbeállítással. Ezután a forgótárcsák lehetséges sorrendjét (permutációit) kitaláló gépet építettek, melyet „*Bombe*" (*Bomba,* 1940) névre kereszteltek.



13. ábra Egy Turing bomba

Az eredeti lehetséges 3!=6 korongsorrend a bevezetett két új korong miatt 60-ra nőtt, amely már meghaladta az eredeti lengyel „bomba" kapacitását. Ezért 1939-ben Varsóban megosztották eredményeiket a megdöbbent angol és a francia kollégákkal. *Alain Turing* továbbfejlesztette az eredeti lengyel gépet. *Turing* fő feladata az volt, hogy az Enigma feltörésének elvét készítse fel arra, hogy a németek esetleg változtatnak az üzenetkulcs küldésének módszerén. Ez meg is történt, de *Turing* módszerének köszönhetően a titkosított üzenetek feltörése továbbra is sikeresen folyt. [34]

Az Enigma használata könnyű volt, maga a berendezés pedig hordozható (a maga közel 50kg-os tömegével), valamint az általa elérhető titkosítás is relatíve biztonságosnak volt mondható. Ezért sokáig használtak olyan gépeket még a háború után is, amelyeket az Enigma alapján, elvén készítettek. Még az 1982-es Falklandi Háborúban is használtak egy Enigma-variánst, amely a KL7 kódnevet viselte. Ennek a gépnek hét rotorja volt, beállításait minden nap változtatták. A rotorok szabadon huzalozhatóak voltak, ami igen gyors változtatást tett lehetővé. A KL7 nemcsak a betűket ismerte, hanem a számokat valamint a központozáshoz használt jeleket is, jóllehet ez utóbbiakat a gyakorlatban nem használták, hanem inkább „kimondták" az írásjelet. A számítógépek fejlődésével azonban az Enigma rendszerű gépek elvesztették versenyképességüket, és végül a gépek használatának célja már nem is a titkosítás volt, csak az időnyerés, amit az a késleltetés jelentett, amíg az ellenség megfejtette az üzenetet.

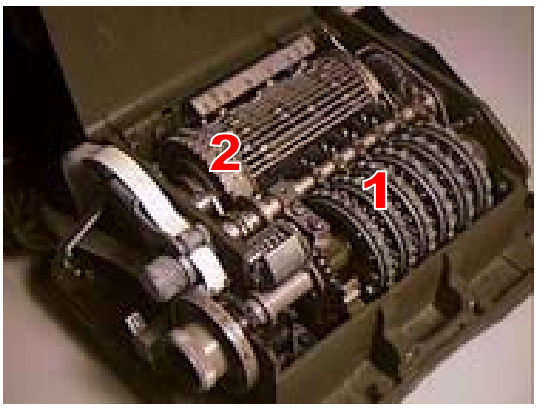
14. ábra Az Enigma működés közben

Bár az Enigma kódját már a világháború alatt megfejtették, a szövetségesek zárolták az Enigmáról szóló összes információt, többek között a feltörés tényét is[12]. A háború után eladták a németektől zsákmányolt Enigma másolatait és változatait a harmadik világ fejlődő országainak, azonban azt „elfelejtették" közölni a vevőkkel, hogy a géppel védett üzenetek feltörése csupán rutinfeladatot jelent. Az Enigma feltöréséről bővebbet – más történelmi vonatkozású

[12] *Marian Rejewski* még a háború idején Franciaországba, majd Angliába emigrált. Nem világos, hogy miért, de nem hívták meg a Bletchey Parkba sem. Gyakorlatilag eltűnt a „süllyesztőben". *Alain Turing* 1952-ben öngyilkos lett.

adatok mellett – a [34]-ben olvashatunk: megismerhetjük *Rejewski* és *Turing* elgondolásait, ötleteit valamint eredményeit is[13].

2.5.1. Hagelin M-209



15. ábra Hagelin M209-B

A felnyitott tetejű szerkezet jobb oldalán a kódoló mechanika látható. A bal oldalon lévő fehér csík egy papírszalag, amelyre a gép a kódolt/dekódolt üzenetet nyomtatta.

Természetesen az amerikai hadsereg sem védelem nélkül kommunikált, az általuk használt gép, a *Hagelin M-209* volt. A gépből többféle verzió is készült (a US Navy által használt verzió például a „CSP-1500" névre hallgatott), de ezek alapelve egyező volt.[14] A gép 1942 novemberétől, Afrika inváziójától állt szolgálatba, de a koreai háborúban is használták. A II. Világháború során több mint 140 000 darabot állítottak elő belőle. A nevét egy svéd mérnökről, *Boris Hagelin*ről kapta, aki az 1930-as évek végén tervezett és készített egy hasonló gépet. 1940-ben a US Army több gépet felvásárolt, majd azok „egyesítésével" és egyszerűsítésével született meg a CSP-885 (M-94), amit később a CSP-1500 (M-209) váltott le. Az M-209 egy hordozható, kézi működtetésű papírszalagos mechanikus eszköz volt, amelyet a taktikai üzenetek titkosítására használtak. *David Kahn* rövid jellemzése szerint „*a kriptográfia történelmének legtalálékonyabb mechanikai eszköze*". Robosztus felépítésével, mégis kompakt méretével (kb. 18x7x13 cm és 2,7kg – emlékeztetőül: az Enigma ~50 kg volt!) a hadsereg egyik kedvenc eszköze lett, jóllehet sebessége sem volt túl nagy, mindössze 30betű/perc. Kezelése egyszerű volt, a felhasználót néhány órai oktatással útnak lehetett indítani. Magát a gépet sem kellett különösebben védeni vagy titkolni, főként, hogy az általa nyújtott biztonság sem volt éppen elsőrangú. De nem is ez volt a célja, mert a taktikai utasítások védelméhez elég volt néhány óra időnyerés is.

[13] A 14. ábra képe az U-571 című filmből való, melynek kriptográfiai konzultánsa dr. David Kahn volt. Sokféle géptípus szoftveres szimulációját megtalálhatjuk az [URL65] címen.
[14] Igazság szerint a CSP-1500 és az M209 két különböző gép volt. A CSP-1500-at a US Navy, az M209-t a US Army használta. Két okból mosódik össze a két típus: (1) ugyanaz volt a működési elv és (2) ugyanaz volt a kulcsgenerálás elve, szinte csereszabatosak is lehettek volna.

A biztonságos kommunikációhoz szükséges hosszú kulcsot hat rövidebb kulcsból állították elő. Ezek hossza rendre 26, 25, 23, 21, 19 és 17 volt - ennyi foga volt az egyes fogaskerekeknek. A képzett hosszú kulcs így 101 405 850 bit hosszú lett, vagyis a kezdeti fogaskerék beállítás csak ennyi karakter továbbítása után állt be ismét (lásd *2.6.1. Vernam titkosítás*). Ezt egy fogaskerék sorozattal (1) valósították meg, a meghajtó fogaskerék aktív fogainak száma és elhelyezkedése az előre egyeztetett kulcsbeállításnak megfelelően változott. A fogaskerekek mögött volt egy „mókuskerék" szerű henger (2), amelynek 27 rúdján két-két mozgatható bütyök volt. A bütykök aktuális állásától függően kialakult egy változó fogszámú fogaskerék. Két ábécé volt, az egyik egy normál A B C D E… a másik egy Z Y X W V… fordított. Végül is az volt a lényeg, hogy ha a fogaskeréknek az adott pozícióban volt foga, akkor A→Z, B→Y stb. leképzés valósult meg, ha nem volt foga, akkor az A→Y, B→Z, stb. szabály lépett életbe.

2.6. AZ EGYSZER HASZNÁLT BITMINTA

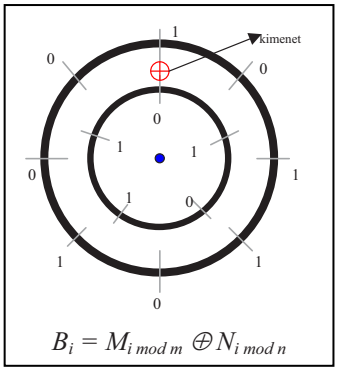
Egy másik problémája a titkos kulcsú rejtjelezési algoritmusoknak, hogy az ismert és gyakorlatban alkalmazott algoritmusok döntő többségénél a kulcs kitalálható megfelelően hosszú vagy megfelelő számú üzenet alapján. Emiatt a titkos kulcsokat időközönként cserélni kell. Ezt figyelembe véve feltörhetetlen kódot készíteni nem is olyan nehéz: minden kódoláshoz használjunk más-más kulcsot, ekkor egy kulcs esetleges visszafejtése nem veszélyezteti az egész kommunikációt, csak azt az egy üzenetet, amihez felhasználtuk.

> *Shannon* elméleti definíciója alapján akkor *tökéletesen biztonságos* (*unconditionally secure*) egy algoritmus, ha egy passzív támadónak végtelen nagy számítási kapacitása van, de a nyílt szöveg hosszán kívül semmi más információhoz nem jut a rejtjeles szövegek tanulmányozásával. Ez csak abban az esetben teljesülhet (szükséges, de nem elégséges feltétel), ha a kulcs hossza legalább olyan hosszú, mint az üzenet maga és soha nem használjuk fel újra. (Valamint a kulcstér entrópiája nagyobb vagy egyenlő, mint az üzenettér entrópiája.) Minden más kriptorendszer megtörhető a lehetséges kulcsok próbálgatásával, és a keletkező nyílt szövegek vizsgálatával (értelmes részek keresésével, ha vannak). Ezek az algoritmusok olyan hosszú kulcsokat használnak, amelyek végigpróbálgatása sok időbe telik, így csak *számítási* vagy *erős biztonságot* (*computationally secure, strong security*) adnak.

2.6.1. Vernam titkosítás

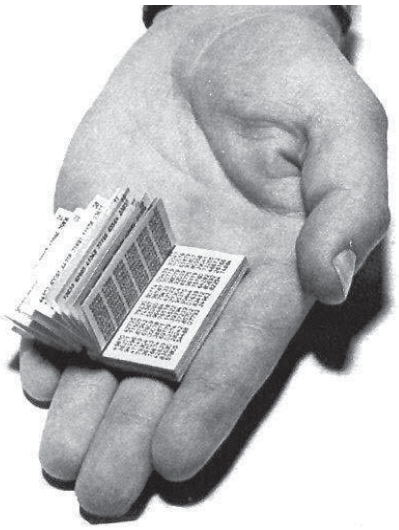
Gilbert S. Vernam, az AT&T mérnöke 1918-ban javasolt egy módszert a telegráfüzenetek titkosítására. Akkoriban az üzeneteket Baudot kódolással továbbították, mely binárisan kódolta az átvitt betűket, számokat és egyéb szimbólumokat, hasonlóan a Morse kódhoz. Vernam szerint a korábbi Vigenere-féle titkosítók legfőbb gyengesége – a ciklikusság – kiküszöbölhető, ha a továbbítandó üzenethez valamilyen szabály szerint véletlenszerű egyeseket és nullákat adunk. A javasolt szabály az volt, hogy a továbbított jel legyen „1", ha a véletlen jel és a továbbítandó jel egyezik, és legyen „0" ha különbözik. (Ez egyébként ez pont a XOR művelet negáltja!) Vernam ötletének sajnos van néhány gyakorlati hiányossága. A rendszernek

mindegy egyes átvitt üzenetszimbólumhoz egy kulcsszimbólumra van szüksége, ami a kommunikációs feleket arra kényszeríti, hogy az üzenetcsere előtt nagy mennyiségű kulcsinformációt cseréljenek. Vernam javasolt egy alternatív megoldást is a kulcsok problémájára: legyen két rövidebb kulcs, egy *m* és egy *n* hosszú, ahol *m* és *n* relatív prím, tehát nincs közös valódi osztójuk. Ebből a két rövidebb bitsorozatból egy *mn* hosszúságú hosszabb bitsorozat képezhető.



Az ilyen módon számolt bitsorozat soha nem ismétlődik *mn* lépésen belül. Ezt a kiegészített Vernam rendszert használta a U.S. Army is egy ideig. *Joseph O. Mauborgne* nem sokkal később megmutatta, hogy azok a titkosító eszközök, melyek kulcsa kettő vagy több rövid kulcsból származik nem biztonságosak, mert a fellépő ciklikusság miatt a kulcs egy idő múlva ismétlődik, ami egyenértékű azzal, mintha egy hosszú kulcsot újra és újra felhasználnának. *Mauborgne* és *William F. Friedman* vonta le a végső konklúziót: csak akkor biztonságos a Vernam-modell, ha a kulcsot véletlenszerűen választjuk, és soha nem használjuk fel újra. Ha ezt a szabályt betartjuk, egy feltétel nélkül biztonságos, tökéletes titkosítóeljárást kapunk eredményül. Vernam elképzelésének eme speciális esetét **egyszer használt bitmintának** (*OTP, One Time Pad*) hívják. *Vernam, Mauborgne* és *Friedman* ötletétének helyességét 30 évvel később *Claude Shannon* bizonyította be.

A II. Világháború során számtalan módszert használtak, amelyek alapja az OTP volt. A véletlenszerű „1”-esek és „0”-ák számtalan sorozatát a gyakorlatban az ABC betűivel helyettesítették, és adott szó vagy karakter kulcsát a betűfolyam megfelelő pozíciója jelölte ki. Ezek a kódkönyvek a betűk rendszertelen halmazát tartalmazták, és sajnos már korántsem nyújtottak olyan biztonságot, mint az őket szülő elmélet. Ennek pedig viszonylag egyszerű oka volt: a kódkönyveket általában úgy készítették, hogy a gépírók rendszertelenül, össze-vissza kalimpáltak a billentyűzeten. Azonban előbb vagy utóbb felvették azt a ritmust, hogy a billentyűzet jobb oldalán lévő karakter után egy bal oldalit üssenek és fordítva. Gyakran személy-, helység- vagy állatneveket kezdtek ismételgetni. Mindezzel jelentősen rontották az elérhető hatékonyságot. Elméletileg annak a valószínűsége, hogy egyetlen üzenet alapján valaki megfejtse a kulcsot, igen kicsi, sőt zérus. Ráadásul nem tudná felhasználni, hiszen minden elfogott új üzenetet újra és újra fel kell törnie a támadónak, mert mindegyikhez más kulcs tartozik. Egyetlen üzenetet nem is lehet megfejteni, mert például egy ötbetűs szó visszafejtésének eredményeként az összes (értelmes) ötbetűs szó számításba jöhet – persze más-más kulcsokkal. Az olyan titkosított szöveg, amelyből bármilyen nyílt szöveg *egyforma valószínűséggel* fejthető vissza, az igazából nem is fejthető vissza.



Sajnos más kedvezőtlen tulajdonsága is van ennek a módszernek:

1. Mivel a kulcs nem jegyezhető meg, mindkét félnek van másolata róla. Ha bármelyik példányt a támadó megszerzi, a kulcsfolyam további használata nem biztonságos. Így újabb kulcsgenerálásra, és -cserére van szükség.
2. A „kulcstár" véges mérete korlátozza az üzenetek hosszát vagy mennyiségét, előbb vagy utóbb elfogy a kulcs. Ez vagy újabb kulcsgenerálást és –cserét jelent, vagy a kulcsfolyam újbóli felhasználására ösztönöz.
3. A módszer igen érzékeny a jelek kiesésére vagy beékelődésére: ha az adó és a vevő egyszer elvesztette a szinkront, soha többé nem találják meg egymást, az üzenet hátralévő része értelmetlen lesz. Ha az átvitt jelek értéke változik meg, az üzenet sérült része értelmetlen lesz, de a következő helyes jelek megfejtése jó lesz, így ez a hiba nem jelent végzetes hibát.

2.6.2. Véletlen bitsorozatok

A kriptográfiai alkalmazásokban sokszor használunk véletlen bitsorozatokat egyszeri kulcsként (mint az iménti egyszer használt bitminta esetében is), vagy kiegészítő, ellenőrző információként. Általában maga az algoritmus nem használ véletlen értékeket a működése során (hiszen nem lenne determinisztikus), csak kulcsot, vagy más inicializáló értéket választunk véletlenszerűen. Például az RSA titkosítás kulcsait is véletlen prímek alapján állítjuk elő. Vagy ha egy kommunikációt DES titkosítással védünk, és a DES-kulcsokat minden egyes kommunikációs folyamat előtt valódi véletlen számként generáljuk, a támadónak minden folyamatnál átlagosan 2^{55} kulcsot kell kipróbálnia, mire eredményre jut. A blokktitkosítók különböző működési módokban is használnak véletlen kezdőértéket (kivéve ECB, lásd: *7. A blokkos rejtjelezők működési módjai, és a folyamtitkosítók* fejezetet). Ha ezeket a véletlen sorozatokat valamilyen algoritmikus módszerrel generáljuk, fennáll a veszélye annak, hogy a támadó megismeri az algoritmust, és reprodukálja a sorozatot. A legtöbb programozási nyelv rendelkezik valamilyen véletlenszám-generátorral, azonban ezek aritmetikai módszerekkel periodikus sorozatokat állítanak elő. Minél hosszabb a periódus, annál inkább „véletlennek" tűnik a számsor, és a legtöbb statisztikai vizsgálaton is átmegy. Erről Neumann Jánosnak a következő volt a véleménye: „*Ha valaki azt állítja, hogy aritmetikai módszerekkel állított elő véletlen bitsorozatot, az természetesen a bűn oldalán áll. (1951)*" Kriptográfiai szempontból (legalább) három kategóriát szokás megkülönböztetni.

Álvéletlen sorozat

A programozási nyelvek *RANDOM()* függvényei ilyen sorozatot állítanak elő. Olyan determinisztikus és periodikus sorozat, amely nem tűnik sem determinisztikusnak, sem periodikusnak, mert periódusa igen hosszú. A statisztikai tesztek is általában véletlennek találják (*pseudo random generators*). Az egyetlen gond az, hogy a sorozat az aritmetikai módszer is-

meretében reprodukálható és a következő *i+1.* sorszámú bit az előző *i* darab bit ismeretében kiszámolható.

```
S0 = seed
Si = Fstate(Si-1)
output = Foutput(Si), ahol S a generátor állapota
```

A sorozatot egy kezdőértékkel (*seed*) indítjuk. Ez a kezdőérték meghatározza az egész sorozatot, és ha legközelebb ugyanerről az értékről indítunk, ugyanezt a sorozatot kapjuk. A kezdőértéknek azonban legalább olyan hosszúnak kell lennie, mint a generátor belső állapotát leíró *S* hossza! Hiába 64 bites a belső állapot, ha a kezdővektor csak 16 bites, a legjobb esetben is csak 2^{16} egymástól eltérő sorozatot kapunk. Ennyiféleképpen tud elindulni a generátor... *Vigyázat! S* nem feltétlenül jelenti a kimenetet, csupán a belső állapotot képviseli. Az egyik állapotból másikba való áttérést jelképezi az $F_{state}()$ függvény, a kimenetet pedig az $F_{output}()$ állítja elő.

Egy jól megtervezett generátor előbb vagy utóbb minden állapotot felvesz, hiszen az állapotokat képviselő *S* véges. Ebből az is következik, hogy van egy olyan állapot, amely egy korábbiba – talán az indulóba – visz vissza. Csakhogy abban a pillanatban, amikor egy *S* érték egyenlő lesz valamelyik korábbi értékkel, a sorozat ismételni fogja önmagát (hiszen determinisztikus), így a sorozat periodikus lesz. Ha ennek a periódusnak a hossza *L* darab állapot, a generátort legalább *K<L* állapotváltás után újra kell inicializálni. Tekintsük példaként a következő sorozatot:

```
Paraméterek:   Z=2^k ;      A=4*r+1;       B=2*m+1;    // ahol r és m tetszőleges
               X0 = seed, ahol 0 ≤ X0 ≤ (Z-1)
               Xn = (A*Xn-1 + B) mod Z
               output = Xn
```

A sorozat $X_0 ... X_{z-1}$ elemei a $\{0..Z-1\}$ halmaz minden elemét egyszer és csak egyszer veszik fel – látszólag véletlenszerűen. Ha $Z=16$, $A=5$, $B=1$ és $X_0=3$, akkor $X_{0..15}$={3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10} és innentől a sorozat ismétli önmagát. Az algoritmusnak az a tulajdonsága, hogy minden elemet egyszer és csak egyszer generál, néha előnyös, de kriptográfiai szempontból gyakran hátrányos. Az *A, B* és *Z* paraméterek hiányában ugyan nem tudjuk megmondani, mi lesz a következő elem[15], de azt biztosan tudjuk mi nem lesz: az előző, az azt megelőző és így tovább: $X_{i+1} \notin \{X_i, X_{i-1}, X_{i-2}, ...X_0\}$

Biztonságos álvéletlen sorozat

Vegyünk egy álvéletlen sorozatot generáló eszközt, amelynek generátora valódi véletlen kezdőértékkel indul, illetve szabálytalan időközönként újra és újra inicializáljuk. A kimeneten megjelenő sorozat közelíteni fog a jó véletlen sorozat felé: hosszútávon a sorozat nem lesz reprodukálható, illetve a meglévő *i* bit ismeretében a következő *i+1.* sorszámú bit nem számolható ki. Ne feledjük azonban, ettől ez még nem igazi véletlen sorozat- illetve forrás, hiszen az algoritmus ismeretében a közbülső részsorozatok reprodukálhatók.

[15] Mi nem, de magasabb fokú algebrai módszerekkel a sorozat néhány eleméből igen jó találati arányú jóslás adható.

ANSI X9.17 pszeudorandom bit generátor

Bemenet:
- s – egy véletlen 64 bites inicializáló (*seed*) érték
- m – integer, mint paraméter
- E_k – *TripleDES*(CDC) k kulccsal.

Kimenet:
- M darab álvéletlen 64 bites bitsorozat $x_1, x_2, \ldots x_m$

1. *Inicializáló vektor számítása*: $I=E_k(D)$, ahol D 64 bites reprezentációja a pillanatnyi dátum-idő párosnak, olyan finom felbontásban, ahogy csak lehetséges.
2. *ciklus i=1-től m-ig*
   - $x_i = E_k(I \oplus s)$ [16]
   - $s = E_k(x_i \oplus I)$
3. *ciklus i vége*
4. eredmény vissza: $x_1, x_2, \ldots x_m$

Valódi véletlen sorozat

Az ilyen típusú sorozatnak nemcsak a kezdőértéke véletlen, hanem minden tagja valódi véletlen forrásból készül, elemei egyenletes eloszlást mutatnak. Sajnos a gyakorlati megvalósítás legtöbbször nehézkes, gyakran lehetetlen. Véletlen forrásként használhatjuk:

1. Hardvermegvalósításban valamilyen fizikai jelenség mérését[17]
   - radioaktív sugárzás
   - félvezető elemek termikus zaja
   - szabadonfutó (gerjedő) oszcillátor
   - mikrofon által összegyűjtött zaj
   - valamilyen turbulens jelenség
2. Szoftvermegvalósításban a befoglaló környezet jellemzőit
   - billentyűleütések és egérmozgások között eltelt idő (Ezt a módszert használja a PGP és a SCRAMDISK is.)
   - felhasználó által gépelt adat
   - valamilyen I/O puffer tartalma
   - operációs rendszer statisztikai adatai: szabad vagy foglalt hely mérete, fájlok száma, hálózati forgalom adatai stb.

Az ilyen módon előállított véletlen sorozat semmilyen módon nem reprodukálható és a következő bitértékek sem jósolhatók meg semmilyen módszerrel.

[16] A $\oplus$ jel XOR műveletet jelent.
[17] Mivel ezek főként külső egységek, külön fizikai védelemmel kell ellátni őket.

2.7. AZ INFORMÁCIÓ MÉRETE – EGY KIS KITÉRŐ

Vegyünk egy sorozatot, és válogassuk ki belőle az előforduló elemeket. Legyenek ezek az elemek $x_1, x_2, x_3, \ldots x_n$, tehát a sorozat n-féle számból (elemből, szimbólumból) áll. Például egy angol szöveget (mint sorozatot) vizsgálva – a szöveg hosszától függetlenül – az X sorozat elemei rendre {a, b, c, d, e, … x, y, z} lesznek és $n=26$. Számoljuk ki mindegyik elemre, mekkora gyakorisággal fordul elő az eredeti sorozatban (p_i). Ezután számoljuk ki a következő kifejezést:

$$H(X) = -\sum_{i=1}^{n} p_i * \log_2 p_i$$

Ez a kifejezés az entrópia (bizonytalanság, *entropy, uncertainty*) definíciója és egyúttal az X sorozat információtartalmának matematikai mérésének eszköze. Például az $X=\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ sorozat entrópiája, mivel minden számjegy 10% valószínűséggel fordul elő:

$$H(X) = -10 * 0{,}1 * \log_2 0{,}1 = 3{,}321 \text{ bit},$$

ami összhangban van azzal, hogy 3 bit kevés a sorozat leírásához, 4 pedig már sok, mert 4 biten nem a lehetséges 15-ig, hanem csak 10-ig számolunk el. Ez azt is jelenti, hogy a fenti kifejezés a sorozat – általában az adat – információtartalmát adja meg bitekben mérve és a megfelelő kódolást feltételezve a minimálisan szükséges tárolókapacitást is megadja. (Ha tudnánk 3,321 biten ábrázolni, nem kellene 4 bitet „elpazarolni" egy BCD számábrázolás esetében…) Más szempontból a $2^{H(X)}$ megadja a sorozat generátorának belső állapotainak számát. Lássunk még néhány sorozatot és a hozzá tartozó értéket:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15	p(0) = p(1) = p(2) = … = p(15) = 1/16 H(X) = -16*1/16*$\log_2$ (1/16)=4 bit	
0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0	p(0) = 9/20 = 0,45 p(1) = 11/20 = 0,55 H(X) = 0,99277445 bit	
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0	p(0) = 20/20 = 1 H(X) = -(1*$\log_2$1) = 0 bit	
a, a, a, a, a, a, a, a, a, a, g, g, g, g, g, g, g, g, g, g	p(a) = 10/20 = 0,5 p(g) = 10/20 = 0,5 H(X) = -(2*0,5*$\log_2$ 0,5) = 1 bit	
Magyar nyelv	4,56 bit	23,58 karakter
Angol nyelv	4,21 bit	18,50 karakter

Az entrópia néhány tulajdonsága (*Vigyázat*! Az n a sorozat elemkészletének számát jelöli, nem pedig a sorozat elemeinek számát!):

- $0 \le H(X) \le \log_2 n$
- $H(X)$ csak akkor 0, ha egyféle elemből áll a sorozat.
- $H(X)$ csak akkor $\log_2 n$, ha minden elem egyforma valószínűséggel fordul elő.

Mint az a fentiekből is látható, a számításához szükségünk van az egyes értékek előfordulási valószínűségére. Tehát mintákra van szükségünk. Ez viszont azt jelenti, hogy valójában soha-

sem ismerjük az egyes elemek valódi gyakoriságát, csupán a már ismert minták alapján számított gyakoriságot. A valódi gyakoriság időről időre változik, így az entrópia sem számolható ki pontosan. Ugyanez a helyzet a véletlenszerűséget vizsgáló statisztikai tesztekkel is: csak a meglévő mintákat veszik figyelembe, ami alapján a sorozatok következő tagjaira valamilyen következtetést vonhatunk le. A fenti kifejezés értéke semmit nem árul el a sorozat véletlenszerűségéről: a táblázat második és negyedik sorában megfigyelhető, hogy az eredmény közel azonos, jóllehet a negyedik sorban lévő sorozatot senki sem mondaná véletlenszerűnek. (Pedig az lehetne, csak meg kellene cserélgetni a tagokat egy kicsit…) Viszont azt meg tudja mutatni, ha egy sorozat nem igazán véletlen: ha egy „0”-ból és „1”-ből álló sorozatra $H(X) \neq 1$, akkor vagy a „0” vagy az „1” többször fordul elő, mint a másik elem. Egy 8 bites adatábrázolásban az elérhető legnagyobb információtartalom 8 bit. Ha hosszútávon ehelyett csupán 3,88 bitet mérünk, az két dolgot jelenthet:

- Minden elem előfordul a lehetséges 256-ból, csak sok az ismétlődés: egyesek gyakrabban jelennek meg, mások ritkábban.
- Nem fordul elő mind a lehetséges 256 elem.

Ha az információtartalom 1 bit, kétféle elem száguldozik a kommunikációban egyforma valószínűséggel. Ha ez a kommunikáció során változatlan, hiába küldözgetünk 8 bites kódokat, nem teszünk mást, mint feleslegesen foglaljuk a sávszélességet.

Ami a bitekben kifejezett elvi adatméret és a valódi kódolás adatmérete között van, az a *redundancia*. Ez a különbözet eltüntethető, de ez már más területre vezet, ez a tömörítés feladata. A tömörítés kriptográfiai szempontból is fontos lehet, mert nemcsak a transzmissziós időt csökkenti, hanem a titkosítás biztonságát is jelentősen növeli, például nehezíti a mintakeresést, növeli az entrópiát.

2.7.1. A titkosítás hatása az entrópiára

A titkosítás is jelentősen megváltoztatja a szóban forgó adat entrópiáját, történetesen növeli. Olyannyira, hogy a titkosított adat jószerével nem is tömöríthető, ezért ha csökkenteni szeretnénk a továbbítási időt és ezért tömöríteni szeretnénk – a titkosítás előtt tegyük meg! Ez támpontot adhat egy passzív megfigyelő számára is: ha tömöríthetetlen adatfolyamot lát, talán éppen titkosított adatot vizsgál. Az alábbi táblázat néhány összehasonlító adatot mutat, a fájl neve pedig az alkalmazott algoritmusra és működési módra utal:

Adatfájl	*Entrópia*	*Eredeti méret*	*Zippelt méret*
plaintext.data	4.43722 bit	243 712 bájt	45 852 bájt
BLOWFISH_CBC.data	7.99924 bit	243 712 bájt	243 919 bájt
BLOWFISH_CFB.data	7.99932 bit	243 712 bájt	243 919 bájt
BLOWFISH_ECB.data	7.88020 bit	243 712 bájt	106 469 bájt
DES_CBC.data	7.99928 bit	243 712 bájt	243 909 bájt
DES_CFB.data	7.99914 bit	243 712 bájt	243 909 bájt
DES_ECB.data	7.88373 bit	243 712 bájt	106 472 bájt
IDEA_CBC.data	7.99924 bit	243 712 bájt	243 911 bájt
IDEA_CFB.data	7.99929 bit	243 712 bájt	243 911 bájt
IDEA_ECB.data	7.87501 bit	243 712 bájt	106 452 bájt

(Láthatóan az ECB módok kicsit lemaradtak a versenyben…)

2.8. TITKOSÍTÁSI MÓDSZEREK GENERÁCIÓI

A fejezetben számtalan módszert láthattunk a régmúlt és a közelmúlt titkosításra használt eljárásaiból. A kriptográfia eredetileg írott üzenetek titkosítására alakult ki, de az elv kiválóan alkalmazható a mai világ változó követelményeihez is. Az eljárások működése, bonyolultsága, elterjedtsége alapján a következő négy generációt jelölhetjük ki összefoglalásul:

Első generáció: Az írott történelem kezdeteitől a XVI – XVII. századig elsősorban **egyábécés helyettesítő módszereket** alkalmaztak (*Caesar ábécé*).

Második generáció: A XVI – XVII. századtól kezdve bonyolultabb**, többábécés helyettesítő módszereket** alkalmaztak (A XVIII. századtól főként Vigenere).

Harmadik generáció: A XX. század elejétől a technikai fejlődés lehetővé tette a **mechanikus és elektromechanikus eszközök** fejlesztését és használatát (*Enigma, Hagelin, Sigaba*). Ezek általában még mindig többábécés helyettesítést használtak, de már nem egyszerűen „több", hanem „gigantikusan sok" ábécével. Megjelennek a produkciós eszközök is.

Negyedik generáció: Egyre több produkciós kódoló jelenik meg, és ezzel párhuzamosan gigantikusra nő a használt kulcstér mérete is. A XX. század második felétől az elektronikus számítási teljesítmények robbanásszerűen fejlődnek. Ez a fejlődés napjainkban is tart és lehetővé teszi olyan algoritmusok fejlesztését és használatát, melyek kivitelezése korábban lehetetlen lett volna. A DES algoritmusát is lehet „papíron, ceruzával" alkalmazni, bár mint látni fogjuk, nem lenne túl egyszerű feladat, viszont egy számítógépnek ez már csak rutinmunka. Ugyanakkor ez a rohamos fejlődés folyamatosan megkérdőjelezi azokat az algoritmusokat, amelyek kifejlesztését maga tette lehetővé. Jó példa a DES, mert összehasonlítva a korábbi generációk algoritmusaival, sokkal bonyolultabb és biztonságosabb ugyan, más elveken nyugszik, de ma már a számítási kapacitások növekedése miatt az „alap DES" nem nyújt megfelelő védelmet, csak a továbbfejlesztett változatok, mint például a *TripleDES* a maga 168 bit hosszú kulcsával. (Ilyen tekintetben a nyilvános kulcsú algoritmusok a negyedik és ötödik generáció közé sorolhatjuk, mert megfelelő számítási kapacitással feltörhetőek, de – a titkosító algoritmus architektúrájának megtartása mellett – nagyobb kulcs választásával a biztonság szinte a végtelenségig fokozható.) A negyedik generáció másik fontos ismérve, hogy a titkosítás – a történelem során talán első alkalommal – kilépett a katonai, diplomáciai területről, és a polgári életben is szinte mindenki számára elérhetővé vált.

Ötödik generáció: Kvantumelvű titkosítások, amelyek elvei a kvantumfizika törvényeire épülnek. Gyakorlati alkalmazásuk ma még futurisztikus ötletnek tűnhet.

A Függelék jelen fejezethez kapcsolódó alfejezetei

14.2. Betűeloszlás egy magyar szövegben

14.3. Enigma-, és Hagelin-múzeum

További kiegészítések folyamatosan bővülő helye a http://www.netacademia.net/konyv webcím.

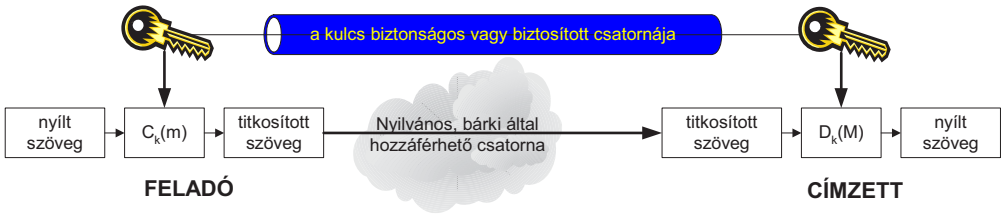
Aki azt hiszi, hogy feltörhetetlen titkosítási módszert talált ki, az vagy hihetetlenül ritka zseni vagy naiv és tapasztalatlan.

Phil Zimmermann

3. SZIMMETRIKUS KULCSÚ MÓDSZEREK

Egy közös jellemzője van az eddig megismert eljárásoknak: a titkosító és a megfejtő eljárás alapvetően mindkét oldalon ugyanaz, valamint mind Alicenak, mind Bobnak ismernie kell a művelethez használt kulcsot, ezt pedig *titkos, vagy szimmetrikus kulcsú titkosításnak* nevezzük. Ezeknek a módszereknek igen jelentős és tapasztalatokban gazdag múltja van illetve az elmúlt évszázadok eszközei is mind ezen alapultak, ezért gyakran hagyományos titkosításnak (*conventional cryptography*) is nevezik.

A titkosítást megelőzően Alicenak és Bobnak egy biztonságos, mások által le nem hallgatható módon meg kell osztaniuk a kulcsot, és azt titokban kell tartaniuk. A titkos kulcsú rejtjelezési módszerek tehát azon a feltételezésen alapulnak, hogy Alice és Bob ismernek valamilyen közös, titkos információt (a kulcsot), amelyet a támadó nem ismer. E titkos információ birtokában Alice elő tudja állítani a titkos üzenetet, Bob pedig értelmezni tudja azt. Eközben Eve és Mallory a titkos információ hiányában sem megérteni nem tudja a titkosított üzenetet, sem pedig rejtjelezni nem tud hamis üzenetet. Így csak Bob tudja elolvasni az üzenetet, aki így indirekt módon abban is biztos lehet, hogy az üzenetet Alice küldte.



16. ábra A szimmetrikus titkosítás modellje

Mindez persze csak addig igaz, amíg sem Eve, sem Mallory nem fejti meg az üzenetekhez használt kulcsot. Ez jó algoritmus esetben annál később következik be, minél hosszabb a kulcs. Más szavakkal: annak nehézsége, hogy egy titkosított üzenetet – adott algoritmus használata mellett – milyen nehezen lehet visszafejteni a kulcs ismerete nélkül, az a lehetséges kulcsok számával mérhető. Ha az algoritmus más módon nem támadható meg, a teljes kipróbálás (*brute-force*) módszerét hívják segítségül, ami az összes lehetséges kulcs kipróbálását jelenti. Azonban ez a feladat hosszú kulcs esetén ritkán végezhető el, mert az összes kulcs kipróbálásának időigénye a kulcs hosszának exponenciális függvénye (lásd: 17. ábra). Ez persze nem garantálja azt, hogy egy hosszabb kulcsot használó algoritmus egyben biztonságosabb is, de általánosságban igaz, hogy *a kulcsnélküli feltörés munkaigényessége a biztonság záloga.*

Kulcs hossza bitekben	Az összes kulcs kipróbálásához szükséges idő
40	10 másodperc
56	7 nap
72	1 381 év
88	90 544 142 év
96	23 179 300 571 év
104	5 933 900 946 398 év
112	1 519 078 642 278 015 év
128	99 554 337 900 332 014 087 év

17. ábra Különböző hosszúságú kulcsok feltöréséhez szükséges idő

A vastag vonal az Univerzum kora, amely Stephen W. Hawking szerint 10 - 20 milliárd év [19]. Ez a bűvös – bár teljesen lényegtelen – határ a 95. – 96. bit környékén van a Deep Crack sebességét feltételezve. A „mért" algoritmus egy olyan elvi DES, amely 56 bitnél hosszabb kulccsal is működik, és bármely kulcsméret mellett ugyanannyi ideig tart a DES-művelet.

A kulcsként használt információ tehát a rejtjelezéshez használt algoritmus egyik paramétere. Ha m a titkosítandó üzenet, és k a titkos kulcs, akkor az

$$M = C_k(m)$$

összefüggés adja meg a titkosított üzenetet. A C_k titkosító függvény vagy algoritmus a következő tulajdonságokkal bír:

1. A titkosított M üzenet a k kulcs ismeretében könnyen kiszámítható – ez a titkosítás folyamata.
2. A titkosított M üzenetből könnyen kiszámítható az eredeti üzenet, de csak akkor, ha ismerjük a k kulcsot – ez az üzenet megoldása.
3. A titkosított M üzenetből nem lehet meghatározni az eredeti üzenetet, ha nem ismerjük a k kulcsot. Ez akkor sem végezhető el, ha ismerjük a titkosító függvény felépítését, vagyis a C titkosító algoritmus csak a k kulcs ismeretében invertálható. Ez a tulajdonság garantálja a Kerckhoffs-elv betartását.

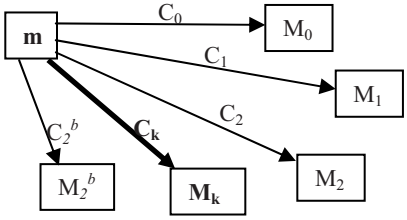
Az ilyen trükkös eljárásokat *csapdafüggvényeknek* (*trapdoor functions*) nevezzük. A visszafelé vezető út, vagyis a titkosított üzenet visszaállítása, elolvasása az

$$m = D_k(M) = C^{-1}_k(M)$$

egyenlettel írható le, ahol D_k a megoldóalgoritmus. Tulajdonképpen a C titkosító algoritmus inverze, ezért C^{-1} módon is jelölhetjük. A vele szemben támasztott elvárások megegyeznek az imént felsorolt tulajdonságokkal és a lényeg, hogy kulcs nélkül nem működik. Általában nagyon hasonlít a titkosítóeljárásra, de nem feltétlenül egyezik meg vele. Például az RSA és a DES esetében ugyanaz a megoldóalgoritmus, mint a titkosító, de az AES inverze jelentősen eltér a titkosítómódban működő AES-től. Egyébként a titkosító C algoritmus és a megoldó D algoritmus kriptográfiailag egyenértékű biztonságot ad, mert kulcs nélküli „megfordításuk" egyformán nehéz. Ez azoknál az algoritmusoknál látható be a legkönnyebben, amelyek C és D algoritmusa egyforma.

Felmerülhet a kérdés, hogy a paraméterként értelmezett kulcsot miért $_{\text{alsóindexben}}$ jelöljük, miért nem írjuk a zárójelek közé, az *m* üzenet mellé, így: *C(m,k)*? Nos, bizonyos értelmezés szerint:

- egy *b* bit hosszú kulcsú algoritmus használata esetén egy adott *m* üzenethez 2^b darab *M* üzenet tartozhat és a *k* kulcs ezekből választ egyet;
- valamint a C függvény vagy algoritmus sem egyetlen algoritmus, hanem egy algoritmuscsoport, melynek 2^b tagja van;
- ilyen értelemben a *k* kulcs nem az algoritmus működését befolyásolja, hanem kijelöli azt az adott algoritmust, amelyik a transzformációt elvégzi.



Gyakorlati, kissé szubjektív szempont, hogy az indexes jelölésmód könnyebben is olvasható, vagyis könnyebb ránézésre belátni az $m=D_k(C_k(m))$, mint az $m=D(C(m,k),k)$ egyenlőséget.

Tervezésnél érdemes elfogadni, hogy az algoritmus nem titkos, hanem nyilvános. Erre azért is szükség van, mert a legtöbb esetben az algoritmusok nem is tarthatók titokban. Ha egy olyan módszerről van szó, amit nemcsak egy szűk körben, hanem egymásnak ismeretlen emberek is használnának, a módszer csak akkor lehet elterjedt, ha az algoritmus mindenki által megismerhető. Így mindenki saját maga döntheti el, hogy megbízik-e az algoritmusban vagy sem. A nyilvánosság egyúttal megmérettetést is jelenet. Ha valaki titokban tartja algoritmusának részleteit, vagy egyáltalán semmit sem hajlandó róla elárulni, akkor vagy „kiskapu" van benne, vagy gyenge. A nyilvános algoritmusokat pedig kriptográfusok tucatjai boncolgatják, így a gyenge algoritmus elbukik, az erősbe vetett bizalom pedig tovább erősödik. A nyilvánosság igénye egyébként összhangban van Kerckhoffs második követelményével is. Ha egy gyenge, egyébként a jótól nehezen megkülönböztethető algoritmust használ valaki, könnyen a „hamis biztonság" csapdájába kerül és nagyobb veszélynek teszi ki magát, mintha nem használna titkosítást. Utóbbi esetben ugyanis más, általa megbízhatónak tartott módszerrel gondoskodna adatainak védelméről. A jó és a rossz titkosítás nem különböztethető meg egymástól pusztán a nyílt szöveg és a rejtjelezett szöveg vizsgálata alapján. Pásztor Miklós a „*Fenyegetettség és védekezés a hálózaton*" című cikkében a kilátó korhadt korlátjához hasonlította a hamis biztonság kérdését: ha nincs korlát, vigyázunk, mert tudjuk, hogy veszélyes helyen tartózkodunk, de ha van korlát csak korhadt, azonban ez nem látszik rajra, akkor belekapaszkodunk vagy nekidőlünk, aminek végzetes következménye lehet. (Az angol szakirodalomban ezt a helyzetet a „snake oil" vagy „snake oil trap" kifejezéssel jegyzik.) Az algoritmusok nyilvánossága természetesen nemcsak etikai vagy elvi szintű kérdés. Az egyes implementációk vizsgálatával, analízisével és visszafejtésével előállítható egy olyan algoritmus, amely

- *logikailag (lehet, hogy) alternatív, de*
- *funkcionálisan egyenértékű.*

Amennyiben az alapalgoritmus tervezési és megvalósítási logikája letisztult, elképzelhető, hogy az efféle szintézis logikája is igen közel áll az eredeti algoritmushoz, így gyakorlatilag az algoritmus nyilvánosságra kerülhet, bár ez – sem etikailag, sem technikai szempontból – sohasem egyenértékű azzal, mintha az eredeti algoritmus elve és tervezési lépései kerültek volna nyilvánosságra. Ilyen „baleset" egyik legjobb példája az RC4 algoritmus, amely széles körben ismert – és elismert – bár hivatalos publikációja a mai napig nincs, mert az RSA Inc. üzleti titokként kezeli.

3.1. A LEGISMERTEBB TITKOS KULCSÚ ALGORITMUS: A DES

Napjaink legismertebb és legelterjedtebb, legtöbb tanulmányt megért és egyik legrégibb szimmetrikus kulcsú titkosítási algoritmusa a DES.

> Maga a szabvány a DES (*Data Encryption Standard*) nevet viseli. A DES, mint szabvány a DEA (*Data Encryption Algorithm*) algoritmust használja. Az évek során a két kifejezésből a DES terjedt el és „DES"-t mondunk akkor is, ha az algoritmusról beszélünk. Természetesen a szabványok következetesen különbséget tesznek a két kifejezés között. (DEA – DES, TDEA – TDES, stb.)

A DES kifejlesztése a hetvenes évek elején kezdődött az IBM blokkos titkosító algoritmusával, a *Luciferrel* (1972). Az időközben gyors fejlődésnek indult elektronikus adatfeldolgozás lehetővé tette, hogy egyre több adatot tároljanak, továbbítsanak elektronikus úton, a létrejött adatbázisokhoz vagy más erőforrásokhoz pedig a helyi terminálokon vagy hálózaton keresztül csatlakoztak, akárcsak napjainkban. A kereskedelmi, kormányzati területeken hamar felismerték az új igényt: szükség van egy szabványosított algoritmusra. Ha a rejtjelezési igények kilépnek egy adott körből, vagyis addig ismeretlen emberek akarnak egymással védelem alatt kommunikálni, szükség van egy mindkét fél által biztonságosnak tartott és mindkettőjük számára hozzáférhető, megismerhető algoritmusra. Szabványos, általánosan elfogadott megoldás hiányában a biztonságos kommunikációt biztosító eszközök is csak egyedi fejlesztéseknek tekinthetők. Ezért a *National Bureau of Standards* (*NBS*, később *NIST*) pályázatot írt ki, erre többek között az IBM a *Lucifert* nevezte, és ez volt az egyetlen algoritmus, amelyet elfogadtak a kutatások alapjául. Az alapvető elvárások az algoritmussal szemben a következők voltak:

- Nyújtson magas szintű biztonságot.
- Egyszerű felépítésű, könnyen megérthető legyen.
- A biztonság csak a kulcstól függjön, ne az algoritmustól.
- Gazdaságosan alkalmazható legyen elektronikus eszközökben.

A későbbi fejlesztésekben részt vett az NSA (*National Security Agency*, az USA nemzetbiztonsági hivatala[18]) is, és ez a hivatal tette meg azt a két lépést, ami a DES-sel szemben bizalmatlanságot váltott ki: az eredeti 128 bites kulcs- és blokkméretet lecsökkentették 64 bitre, illetve az eredeti *S*-dobozok konstansait lecserélték. A 64 bites kulcsból 8 bitet (8. 16. … 56.

[18] Az NSA titkosságára jellemző, hogy a rossznyelvek szerint az NSA feloldása nem is a fenti „National Security Agency", hanem a „Never Say Anything" (*Soha ne mondj semmit*).

és 64.) végül kizártak („ellenőrzési célokra fenntartva" - paritásbitek), így az effektív kulcsméret 56 bit lett. Sokan feltételezik, hogy a kulcstér ilyen durva csökkentése (gyakorlatilag az eredeti kulcstér 99,6%-át kidobták) azért történt, hogy az NSA még feltörhesse a DES-t, de kisebb szervezet képtelen legyen rá. Az algoritmust 1975 márciusában hozták nyilvánosságra, bár az *S*-dobozok tervezési részleteit továbbra is titokban tartották. Az eredeti algoritmushoz képest ezek igen durva változtatások voltak, és emiatt sokan úgy gondolták, hogy az eljárásban kiskapu van, amin keresztül az NSA jóval könnyebben feltörheti a kódot. A kritikák ellenére 1977-ben a DES-t a *FIPS46-3-ban*[19] szabványként publikálták, és néhány évvel később több szabványügyi szervezet is elfogadta az algoritmust (pl. ANSI X3.92, 1981). A szabványosításnak köszönhetően hamarosan LSI és VLSI áramkörök tucatjai jelentek meg a piacon, melyeket az *NBS* is bevizsgált és elismert. Hamarosan a kereskedelmi és az üzleti területeken egyaránt de facto szabvánnyá vált. Bár a DES kifejlesztése óta több mint 20 év telt el, ma is élő, engedélyezett szabvány, széles körben használják a polgári élet minden területén[20]. Az algoritmust ötévenként felülvizsgálták, erre utoljára 1997-ben került sor és már nem is lesz több ilyen felülvizsgálat. A kezdetekkor is kritizált relatíve kicsi kulcstér ma már nem nyújt megfelelő védelmet, mert viszonylag rövid idő alatt feltörik az ezzel a kulcsmérettel kódolt üzeneteket. (Igaz, csak tekintélyes erőforrás-befektetéssel és *brute-force,* nem pedig elvi módon.) A kriptográfusok kutatásainak állandó tárgya a DES, ennek ellenére sokáig nem találtak gyakorlatban is használható támadási módszert. Az 1990-ben *Eli Biham* és *Adi Shamir* által bevezetett *differenciális kriptoanalízis* néhány évvel később eredményesebbnek bizonyult az addigi próbálkozásoknál. A *brute-force* 2^{55} átlagos DES műveletével szemben körülbelül 2^{37}-2^{38} speciálisan választott nyílt szövegből és annak kódolt párjából ki tudja találni a kulcsot. [17,18]) A *differenciális kriptoanalízis* módszer párja, a *lineáris analízis* (1992, *Mitsuru Matsui*), hasonló eredményeket ért el. (A differenciális és lineáris analízishez mintaadat, tehát ismert titkos-nyílt szövegpárok kellenek.)

Végül 1997-ben útjára indította a *NIST* az *Advanced Encryption Standard* (AES) projektet, amely célja, hogy olyan algoritmust találjon, ami felválthatja a DES algoritmusát. A DES algoritmust sokan az egykulcsos titkosítási rendszerek hosszú történetének összegzéseként nézik, mégis más, mint bármelyik elődje. A kriptográfia hagyományosan egy „titokzatos" tudomány, csak a XX. század végén nyilvánosságra került elvek (amelyek a II. Világháború német és japán eszközeinek működési és feltörési elvei) enyhítettek ezen. A DES viszont egy teljesen nyilvános algoritmus, a *FIPS46-3*-ban bárki találhat elegendő információt ahhoz, hogy saját szoftveres vagy hardveres megoldását elkészítse. A DES ma is kiválóan teljesít, viszont nyilvánosságának van egy furcsa, kissé paradox következménye: *a kriptográfia eddigi legsikeresebb titkosító algoritmusa a legkevésbé sem titkos.*

[19] *Federal Information Processing Standard*
[20] Érdekes közbevetés, hogy az eredeti kormányzati ajánlás kereskedelmi célokra ajánlotta ugyan a DES-t, de minősített adatok védelmére már nem. A működési módok közül az ECB nem javasolt, csak a CFB vagy CBC. Lásd még: *„7. A blokkos rejtjelezők működési módjai"*

3.1.1. A Feistel-struktúra

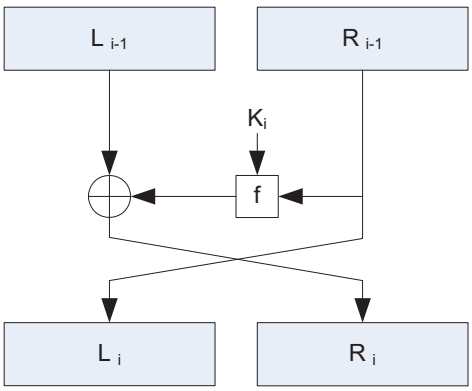
Az algoritmus lavinahatással rendelkezik (*lásd: később, illetve az üzenetpecsételnél*), így digitális aláírásrendszerekben is használható. Számos hardvermegvalósítás látott napvilágot, ezek elsősorban digitális telefonokban és más kommunikációs eszközökben működnek és többségük oly módon használja az alapalgoritmust, hogy az eredeti blokkos működést folyó titkosítás elvégzésre teszi alkalmassá. A hardvermegoldások teljesítménye egyébként jóval nagyobb a szoftveres megoldásoknál, mert a DES rengeteg bitszintű műveletet használ, ami hardverben rendkívül könnyen kivitelezhető. A szakértők véleménye szerint a DES egy jól átgondolt algoritmus. Egyébként egyszerű logikai műveleteket végez bitek kis csoportján. A DES egy 64 bites blokkos rejtjelező algoritmus, vagyis a nyílt szöveg egy 64 bit méretű blokkjához egy ugyanekkora rejtjeles blokkot rendel. A hozzárendelés csak a kulcstól függ, tehát ha a nyílt szövegben két nyílt blokk azonos, akkor a rejtjeles blokkok is azonosak lesznek.

Az algoritmus 19 különálló fokozatból épül fel, ebből 16 szolgálja a titkosítást úgy, hogy minden fokozat az előző eredményét használja fel, ami iteratív működésmódot eredményez (*iterated block cipher*). Egy ilyen iterációs lépést „*kör*”-nek (*round*) nevezünk és a körök száma az effajta algoritmusok egyik jellemzője. Minden fokozat azonos elvi felépítésű, csak a paraméterei mások. A DES működése alapján, az úgynevezett ***Feistel*-titkosítók** (*Horst Feistel*, IBM) csoportjába tartozik, melyek köreinek száma tipikusan $r \geq 3$ és páros. Bemenetükre $2t$ bites nyílt szöveget adunk. A t hosszúságú részekre a továbbiakban L (left, bal) és R (right, jobb) betűkkel hivatkozunk. Egy teljes adatblokk jelölése (L_i,R_i) módon történik.

$2t$ bites bemeneti adatblokk	
t bites bal oldali rész	t bites jobb oldali rész
L_i	R_i

A DES az (L_0,R_0) nyílt szöveget az (R_r,L_r) rejtjeles szövegbe képzi le, ahol r az iterációs körök száma. Az alsóindexben szereplő szám azt jelzi, hogy hányadik körnél járunk, illetve hányadik kör kimenetén megjelent adatról beszélünk. Az L és R betűk sorrendje is hordoz információt, mert a természetes sorrendet így jelöljük: (L,R). Ezzel szemben, ha a bal oldalt felcseréljük a jobb oldallal, akkor a jelölés (R,L). A Feistel-struktúra formális leírása és blokkvázlata a következő:

$$1 \leq i \leq r \qquad \begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1},K_i) \end{aligned}$$



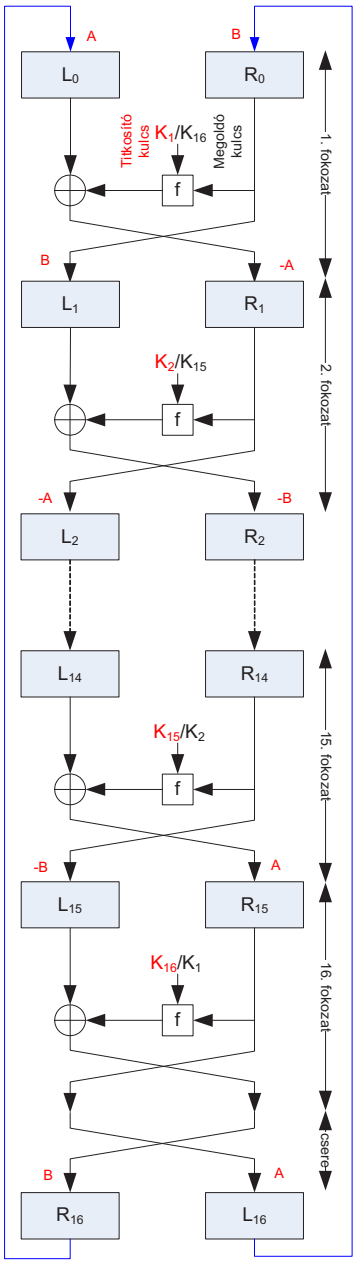
Az utolsó iteráció után a jobb és bal oldali kimenetet felcserélik – ezt jelzi a végeredmény (R_r,L_r) formája is –, így a megfejtés pontosan ugyanazzal az algoritmussal történhet, mint a titkosítás, csak a körkulcsokat kell fordított sorrendben alkalmazni. Az ábra melletti két összefüggésben a következő elemeket találhatjuk:

- K_i az eredeti kulcsból származtatott, az aktuális körhöz használt körkulcs.
- Az L_0R_0 a nyílt szöveget jelenti.
- A $\oplus$ jel XOR műveletet jelzi.
- Figyeljük meg az alsóindexeket: egy kör eredményének előállításához felhasználjuk az előző eredményét!
- Fontos megjegyezni, hogy mindig csak az adatblokkok egyik fele dolgozik! Az $L_i=R_{i-1}$ jelentése: „*az aktuális kör kimenetének bal oldala az előző kör kimenetének* ***változatlan*** *jobb oldalával egyenlő*”.
- Ezzel szemben az új jobb oldali eredmény előállításához felhasználjuk az előző kör kimenetének mindkét oldalát.
- Az algoritmus erejét az itt alkalmazott $f(R_{i-1},K_i)$ függvény adja, amit *körfüggvénynek* (*round function*) nevezünk és **nem kell invertálhatónak lennie**. Ennek ellenére maga a teljes kódoló invertálható. A körfüggvény belsejének részletezésére hamarosan sor kerül.

Álljunk meg egy pillanatra! Azt állítom, hogy az előző oldali ábrán lerajzolt műveleteket egymás után ismételgetve egy nem invertálható, meg nem fordítható művelet eredményét rázippzározzuk az adatfolyamra és az mégis visszanyerhető? Nos, megdöbbentő, de a válasz: igen! A megoldás a fél adatblokkok cserélgetésében rejlik. Nézzünk két speciális esetet:

1. Tegyük fel, hogy az f() körfüggvény mindig zéró biteket ad vissza. Ebben az esetben semmi sem történik, csak az adatblokkok keringőznek egy jót, végül változatlanul bújnak ki az ismételgetett iterációból...
2. Amennyiben az f() körfüggvény minden esetben egyes értékű biteket ad a kimenetén, az első iterációban az egyik fél blokk negálódik, a másik változatlanul halad át. A második iteráció után ez a másik fél blokk is negálódik. A harmadik iteráció „leszedi” az első negálást, a negyedik iteráció pedig a másodikat, vagyis minden negyedik iteráció változatlanul visszaadja az eredeti adatblokkot. (Lásd a következő ábra A, -A és B, -B betűjeleit!)
3. De mi történik, ha az f() függvény a kulcs és a fél adatblokk függvényében szinte bármit visszaadhat? Ezt a bármit ráhúzzuk az adatfolyamra, majd ugyanazt visszafelé eljátszva leszedjük róla. Ezt, és a teljes megfejtés menetét láthatjuk a következő ábrán, és a hozzáfűzött magyarázatban.

A Feistel-struktúra invertálása – a megfejtés menete



A bal oldali ábrán csak a legfontosabb lépések vannak, de ez az elnagyolt blokkvázlat is elegendő arra, hogy megnézzük, a fordított sorrendben alkalmazott körkulcsok miként jelentik a művelet invertálását. A titkosítás során a lánc bemenetére az (L_0,R_0) kerül, és az egyes fokozatok rendre $K_1,K_2,...K_{16}$ körkulcsokat használják. Az eredmény (R_{16},L_{16}). Megfejtéskor a lánc bemenetére az iménti (R_{16},L_{16}) kerül, és az egyes fokozatok rendre $K_{16},K_{15},...,K_1$ kulcsokat használják. A várt eredmény (L_0,R_0). Nosza, adjuk az első fokozat bementére a megfejtendő adatblokkot!

$$L_0 = R_{16}$$
$$R_0 = L_{16}$$

Mi jön ki ebből a fokozatból? Az L_1 meghatározásához elég, ha „ütközésig” követjük a nyilakat, mégpedig visszafelé:

$$L_1 = R_0 = L_{16} = R_{15}$$

Az adatblokk másik felének családfája már nem ilyen egyszerű. Mit várunk egyáltalán? Ha a bemenetre a 16-os sorszámú részeredmény került, és a bal oldalt sikerült a 15.-ig visszavezetni, jogos a feltételezés, hogy a 15-ös másik felét fogjuk kapni, vagyis L_{15}-öt. Nem feledve, hogy megoldó módban vagyunk, ezért az első fokozat a K_{16}-ot használja:

$$R_1 = L_0 \oplus f(R_0,K_{16})$$

Hát ez még messze van az áhított L_{15}-től, sőt nem is hasonlít rá. Csakhogy már tudjuk, az első fokozat bemenetén tulajdonképpen a 16-os sorszámú blokkok vannak! Helyettesítsük L_0-át R_{16}-tal és R_0-át L_{16}-tal!

$$R_1 = R_{16} \oplus f(L_{16},K_{16})$$

L_{16}-ról a nyilakat visszafelé követve láthatjuk, hogy leánykori nevén R_{15}-nek is szólíthatjuk:

$$R_1 = R_{16} \oplus f(R_{15},K_{16})$$

Most tüntessük el az R_{16}-ot is! Nem is bonyolult, mert tudjuk honnan jött, amikor titkosították:

$$R_{16} = L_{15} \oplus f(R_{15},K_{16})$$

Ha ezt visszahelyettesítjük R_1 kifejezésébe:

$$R_1 = L_{15} \oplus f(R_{15},K_{16}) \oplus f(R_{15},K_{16}) = L_{15}$$

És itt a trükk: a két körfüggvény a xor művelet szabályai szerint kioltja egymást! Vagyis (a helyes kulcs birtokában), lényegtelen, hogy mit kavar az f() körfüggvény, mert a Feistel-struktúra önműködően lefejti az adatfolyamról. Emiatt nem kell f()-nek invertálhatónak lennie.

A folytatás akkor lenne szép, ha

- a 2-es fokozat kimenetén visszakapnánk a 14-es adatblokkokat, majd
- a 3-as fokozat kimenetén visszakapnánk a 13-as adatblokkokat, majd
- a 4-es fokozat kimenetén visszakapnánk a 12-es adatblokkokat, majd
- a 5-ös fokozat kimenetén visszakapnánk a 11-es adatblokkokat, majd
- ...
- a 14-es fokozat kimenetén visszakapnánk a 2-es adatblokkokat, majd
- a 15-ös fokozat kimenetén visszakapnánk a 1-es adatblokkokat, majd végül
- a 16-ös fokozat kimenetén visszakapnánk a 0-ás adatblokkokat.

Próbaképpen lássuk a következő, kettes fokozat kimenetét: mennyi (L_2,R_2)? Remélhetően a 14-es adatblokkokat kapjuk eredményül... A bal oldal kinyomozásához kövessük visszafelé a hozzávezető a nyilat!

$$L_2 = R_1$$

Ez idáig rendben, sőt éppen az imént láttuk be az R_1=L_{15} egyenlőséget. Használjuk fel!

$$L_2 = R_1 = L_{15}$$

De, honnan származik L_{15}? Nézzünk bele a 15. fokozatba és láthatjuk, hogy L_{15} tulajdonképp'

$$L_2 = R_1 = L_{15} = R_{14}$$

Nagyszerű! És most lássuk, hogy vajon R_2 egyenlő-e a várt L_{14}-gyel? Hasonlóan járunk el, mint eddig, vagyis megnézzük, honnan származik. (Közben nem feledjük, hogy megoldómódban vagyunk, ezért a kettes fokozat a 15-ös körkulcsot használja...)

$$R_2 = L_1 \oplus f(R_1,K_{15})$$

L_1-ről már tudjuk, hogy igazából R_{15}, de vajon R_{15} miből született, amikor titkosították?

$$R_{15}= L_{14} \oplus f(R_{14},K_{15})$$

Írjuk ezt be R_2 kifejezésébe, pontosan L_1 helyére!

$$R_2 = L_{14} \oplus f(R_{14},K_{15}) \oplus f(R_1,K_{15})$$

Jó lenne, ha kiderülne, hogy R_{14} és R_1 egyenlő, mert akkor a két f() függvény ismét kioltaná egymást... Az előző oldal alján már láttuk, hogy R_1=L_{15}. Ha a 15. fokozatba kukkantunk, látjuk, hogy L_{15} a titkosítás során R_{14} másolataként jött létre, vagyis vele egyenlő! És ha L_{15}-tel egyenlő, akkor R_1-gyel is egyenlő, így $R_{14} = R_1$. Írjuk be R_1 helyére R_{14}-et:

$$R_2 = L_{14} \oplus f(R_{14},K_{15}) \oplus f(R_{14},K_{15}) = L_{14}$$

Pontosan ezt az eredményt vártuk! A megoldómódban működő struktúra második fokozatának kimenetén megkaptuk a 14-es adatblokkot! A többi fokozat kimenete is hasonlóan alakul, és a lánc végére az eredeti nyílt szöveg *(L_0, R_0)* összetevői is megjelennek. Kicsit átgondolva a fentieket, arra a következtetésre juthatunk, hogy az invertálást a befejező csere teszi lehetővé, mert ez biztosítja azt, hogy:

- Az a félszó, amely változatlanul jött ki a titkosítás utolsó köréből, változatlanul haladhasson át az első megfejtő körön is, illetve
- az a félszó, amire ráXORoltuk az utolsó kör f() függvényének eredményét, pontosan ugyanezen a transzformáción fog átesni a megfejtés első körében is. És ha valamire kétszer ráXORoljuk ugyanazt, az olyan, mintha semmi sem történt volna.

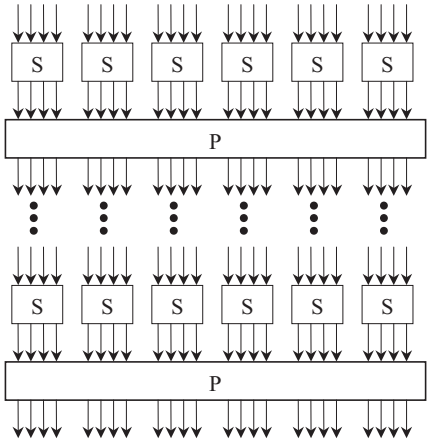
3.1.2. A DES lépései nagy vonalakban

A DES két alapvető működési elvet egyesít:

- *Feistel*-struktúra
- Produkciós-titkosító

  Produkciós titkosítónak (*production ciphers*) az olyan titkosító eszközöket nevezzük, amelyek kettő vagy több eltérő elvű művelet kombinálásával szolgáltatják eredményüket. A produkciós algoritmusok nagyobb biztonságot kínálnak, mint a benne szereplő műveletek külön-külön. Ha azonos elvű titkosítókat kötünk sorba, előfordulhat, hogy azok egymás hatását kioltják (például két, azonos kulccsal futó *OTP*) vagy a biztonságot nem növelik, csak a feldolgozási időt. Emiatt elfogadott az a tervezési elv, hogy a produkciós részegységek egymástól eltérő elven működjenek. Egyik speciális eset a helyettesítő-keverő hálózat (*substitution - permutation network*, *SP network*), mely helyettesítéseket (*S-doboz*) és keveréseket (*P-doboz*) végez egymás után. Emlékeztetőül idézzük fel az „ősi" módszerek közül:

  – a helyettesítő titkosítókat: Caesar, Vigenere kódolók, stb.
  – és a keverő titkosítókat: Skitali, keverősablon, stb.

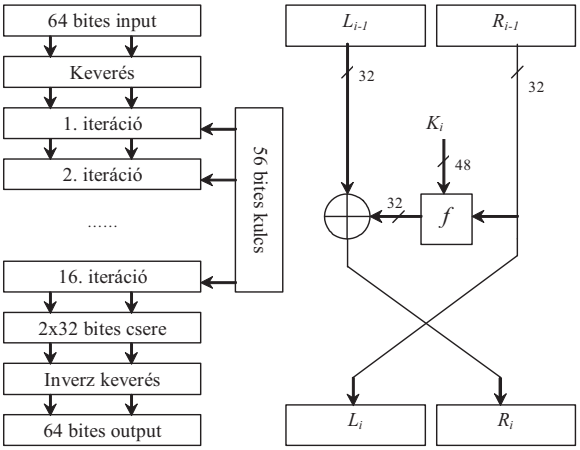


18. ábra Egy tipikus SP hálózat, a produkciós eszközök legáltalánosabb esete

A DES kulcsmérete 64 bit, azonban 8 darabot – minden nyolcadikat – kihagyunk a folyamatból, így a valódi kulcsméret csak 56 bit lesz. A kihagyott biteket különféle ellenőrzési célokra lehet használni, a hivatalos specifikáció szerint páratlan paritásbitek. Ez egyúttal azt is jelenti, hogy az algoritmus által nyújtott védelem is kisebb lesz: pontosan 256-szor hamarabb célt ér egy *brute-force* támadás, mintha mind a 64 bit valódi kulcsbit lenne. A feldolgozott blokkok mérete 64 bit.

Az első lépésben a kulcstól függetlenül a 64 bites bemenet bitjeit összekeveri, az utolsó lépés ennek pontosan az inverz művelete (19. ábra). Az iterációs lépések mindegyike 2 darab 32

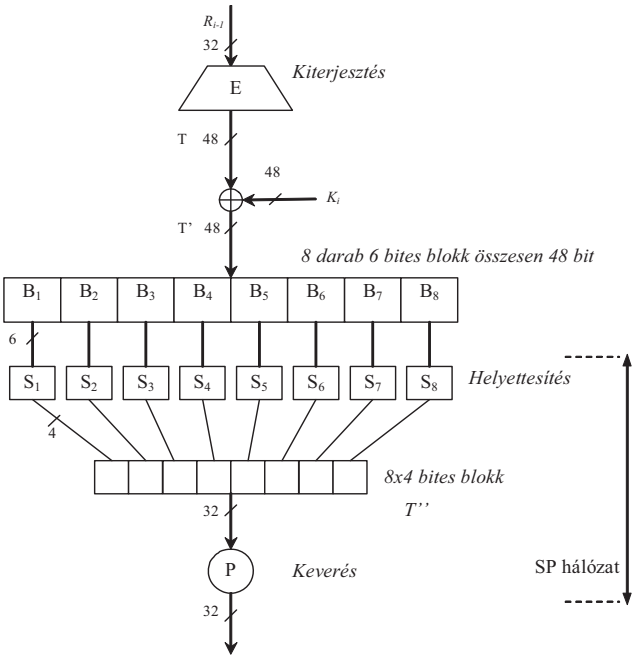
bites értékként értelmezi a bemenetére adott adatot és ennek megfelelően 2 darab 32 bites kimenetet ad. Az utolsó előtti lépésben a bal oldali 32 bites részt felcseréli a jobb oldali 32 bites résszel. A maradék 16 lépés működése egységes és mindegyik paramétere egy, a kulcsból származtatott érték (*körkulcs, K_i*) is: a kulcsot két 28 bites részre bontja az algoritmus, mindegyiket a körkulcs sorszámának megfelelő bittel jobbra forgatva. A K_i –t ezekből a darabokból egy újabb keveréssel képzi.



19. ábra A DES vázlatos felépítése és az iteráció egy lépése

Az *f()* körfüggvény négy lépésből áll (20. ábra). Első lépésben egy 48 bites számot képez R_{i-1}–ből, amit rögzített bitkeveréssel és bitek megduplázásával (*kiterjesztés, expansion*) kap meg. A másolt bitek kiválasztása nem a kulcstól függ, hanem előre rögzített. Ennek a lépésnek nagy szerepe van a lavinahatás elérésében, mert biztosítja, hogy egy-egy bit több helyen is szerepeljen, így a bit esetleges megváltozása több helyen is kifejti a hatását. A második lépésben a 48 bites szám és a körhöz tartozó K_i körkulcs között XOR műveletet végez. A kapott eredményt 8 darab 6 bites csoportra osztja ($B_1 ... B_8$), amiken különböző helyettesítéseket végez (*S*-doboz, $S_1 \ldots S_8$). A helyettesítések eredménye 4 biten keletkezik. Az így nyert 8×4 bitet végül megint összekeveri és az eredményül kapott 32 bites szám lesz a függvény kimeneti értéke. A megfejtő algoritmus ugyanez, de a körkulcsokat fordított sorrendben kell alkalmazni (az első körben a 16. körkulcsot, a másodikban a 15. körkulcsot és így tovább).

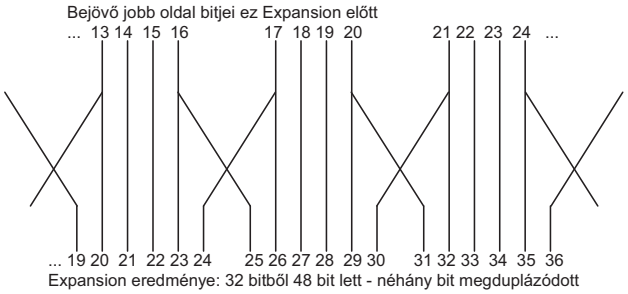
> Néhány emberben felmerült a kérdés, hogy akkor mégsem ugyanazt a kulcsot használjuk a megfejtéshez, mint a rejtjelezéshez. Tényleg nem azt, de vegyük figyelembe, hogy az implementáció képes figyelembe venni a felhasználó műveletét (titkosítás vagy megfejtés) és a kulcselőkészítés során ennek megfelelően generálja a körkulcsokat, vagyis a felhasználó ugyanazt a kulcsot használhatja mindkét folyamathoz. (Megfejtéskor a kulcselőkészítés a titkosító kulcs valamilyen értelemben vett inverzét állítja elő.)
>
> Itt jegyezném meg, hogy a DES algoritmusa bizonyos értelemben kulcsfüggetlen, mert az algoritmus lépéseit, működését nem befolyásolja a kulcs: a körök során az adatfolyamra „ráxoroljuk” a kulcsot, megfejtéskor pedig „lexoroljuk” azt róla, de magukat a lépéseket vagy azok végrehajtási sorrendjét nem érinti a kulcs.



20. ábra A DES körfüggvénye: $F(R_{i-1},K_i)=P(S(E(R_{i-1})\oplus K_i))$

Lavinahatás a DES-ben

Már említettük, hogy a DES rendelkezik lavinahatással *(avalanche effect)*. Mit is jelent ez? Azt, hogy ha a bemeneti blokk egy kicsit megváltozik, a kimeneti blokk jelentősen változzon meg. Pontosabban, ha a bemeneti blokk egy bitje megváltozik, a kimeneti blokk bitjeinek körülbelül a fele változzon meg. A már említett kiterjesztés (*E*) művelete az iterációs körök *f()* függvényében pontosan ezt biztosítja. Emlékezzünk vissza arra, hogy ez a művelet a bemeneti 32 bitből 48 bitet „készít", mégpedig úgy, hogy 16 kiválasztott bitet lemásol és megdupláz. (Ezek az 1., 4., 5., 8., 9., 12., 13., 16., 17., 20., 21., 24., 25., 28., 29. és 32. bitek.)



Ebből viszont az is következik, hogy ha a függvény bemenetén változás történik, az két helyen is kifejti a hatását. A kiválasztott bitek úgy helyezkednek el, hogy egy-egy ilyen „duplázott" bit kettő *S*-doboz bemenetére megy. Az *S*-dobozok egyébként is olyan felépítésűek, hogy a bemenetükön jelentkező egybitnyi változásra a kimenetükön több bitnyi változással válaszolnak. Ez azért is fontos, mert ha a bemenet nem a fenti bitek egyikén változik meg, az *S*-dobozok biztosítják a több bit megváltozását a következő fokozaton. Tehát így vagy úgy, de a fokozat kimenetén több bit is megváltozik, és ezt a hatást a következő fokozatok tovább erősítik. Az alábbi példában két titkosítás eredményét láthatjuk. A kulcs mindkét esetben a „*qwertzui*" karaktersorozat volt. A nyílt szöveg az első esetben (első sorok) „*abcdefgh*", a másodikban (második sorok) „*accdefgh*" volt, ami egy bit különbséget jelent. Az alábbi bitkülönbségek számszerű értékei csak erre a példára vonatkoznak, nem általánosak. Szerencsés esetben nagyobbak is lehetnek az eltérések az egyes lépések között, de lehetnek kisebbek is. Ami viszont általánosítható, hogy a legnagyobb „lavina" a 9. – 10. kör körül van.

```
P1: 6162636465666768=0110000101100010011000110110010001100101011001100110011101101000
P2: 6163636465666768=0110000101100011011000110110010001100101011001100110011101101000 Δ=1
  L 0:11111111000000000111100001010101,R 0:00000000111111111000000001100110
  L 0:11111111000000000111100001010111,R 0:00000000111111111000000001100110 Δ=1 bit
  L 1:00000000111111111000000001100110,R 1:10101101000110110000101011110110
  L 1:00000000111111111000000001100110,R 1:10101101000110110000101011110100 Δ=1 bit
  L 2:10101101000110110000101011110110,R 2:10011100101000000000101000111110
  L 2:10101101000110110000101011110100,R 2:10011100101000100001110001011110 Δ=4 bit
  L 3:10011100101000000000101000111110,R 3:10111111110001110011000110000100
  L 3:10011100101000100001110001011110,R 3:00001101101000110010010101001111 Δ=17 bit
  L 4:10111111110001110011000110000100,R 4:00011101000111010110011111011000
  L 4:00001101101000110010010101001111,R 4:00110011110110101000100110101001 Δ=33 bit
  L 5:00011101000111010110011111011000,R 5:11100010011010001011110100101011
  L 5:00110011110110101000100110101001,R 5:11101000000011111111100010100101 Δ=32 bit
  L 6:11100010011010001011110100101011,R 6:11010101011101011000101010111101
  L 6:11101000000011111111100010100101,R 6:11101101110001111000100011001110 Δ=26 bit
  L 7:11010101011101011000101010111101,R 7:10010101101000001001001100000010
  L 7:11101101110001111000100011001110,R 7:10000101010001110010011011100010 Δ=27 bit
  L 8:10010101101000001001001100000010,R 8:10001010011111111011000101111010
  L 8:10000101010001110010011011100010,R 8:11010000000111010100001111000110 Δ=32 bit
  L 9:10001010011111111011000101111010,R 9:00000000100000110011100011000000
  L 9:11010000000111010100001111000110,R 9:01001001101000011010011011001111 Δ=35 bit (max)
  L10:00000000100000110011100011000000,R10:11100000001010001011000010101001
  L10:01001001101000011010011011001111,R10:00011111111110000001011010110111 Δ=33 bit
  L11:11100000001010001011000010101001,R11:10001110101010110011110000001001
  L11:00011111111110000001011010110111,R11:11101011001010100001110000001011 Δ=27 bit
  L12:10001110101010110011110000001001,R12:00111010101000101111110010010001
  L12:11101011001010100001110000001011,R12:00101101101111011001100111111011 Δ=25 bit
  L13:00111010101000101111110010010001,R13:11110111111101100011011001111110
  L13:00101101101111011001100111111011,R13:11011110100100110100011111111010 Δ=30 bit
  L14:11110111111101100011011001111110,R14:00110111000100100101110111101011
  L14:11011110100100110100011111111010,R14:01111101111001011000110001111000 Δ=31 bit
  L15:00110111000100100101110111101011,R15:10101100101100011100101011000100
  L15:01111101111001011000110001111000,R15:11110010100001011100101011111000 Δ=29 bit
  L16:10101100101100011100101011000100,R16:00000010001100101011110111100110
  L16:11110010100001011100101011111000,R16:00001000110011101110100101100111 Δ=25 bit
C1:2459878C34B50BAF=0010010001011001100001111000110000110100101101010000101110101111
C2:2599315E82879FBE=0010010110011001001100010101111010000010100001111001111110111110 Δ=25
```

A most következő alfejezetet (körülbelül három oldalt) azok számára ajánlom, akik szeretnének ujjgyakorlatként készíteni maguknak egy DES-t. Megtalálható benne minden ehhez szükséges információ: algoritmusleírás, az E, P és IP műveletek táblázatai, stb. Akik számára rémisztőnek tűnik a következő oldal, nyugodtan lapozzanak tovább a következő alfejezetig, amely a DES feltörésének történetét meséli el.

3.1.3. A DES lépései applikációs mélységben

Data Encryption Standard, DES

Bemenet:
- 64 bit, mint nyílt szöveg blokkja: $m_1, m_2, \ldots m_{64}$
- 64 bit, mint kulcs: $k_1, k_2, \ldots k_{64}$

Kimenet:
- 64 bit, mint titkosított blokk: $c_1, c_2, \ldots c_{64}$

1. Kulcsgenerálás és kezdő keverés
   - Legyen $v_i = 1$, ha $i \in \{1,2,9,16\}$ egyébként $v_i = 2$, $1 \le i \le 16$. Ezen értékek (v_i) adják a későbbi forgatások lépésszámait.
   - $(C_0, D_0) = PC1(K)$, C_0 és D_0 egy-egy 28 bites részei a kulcsnak. $C_0=k_{57}k_{49}\ldots k_{36}$ és $D_0=k_{63}k_{55}\ldots k_4$
   - *Ciklus i=1-től 16-ig* az egyes K_i körkulcsok kiszámítására
     - $C_i = (C_{i-1} \leftarrow v_i)$,
     - $D_i = (D_{i-1} \leftarrow v_i)$,
     - $K_i = PC2(C_i||D_i)$. Ha $C_i||D_i$ bitjei rendre $b_1b_2b_3b_4\ldots b_{56}$, akkor $K_i=b_{14}b_{17}\ldots b_{32}$, összesen 48 bit. (a $\leftarrow$ szimbólum a balra forgatást, a || pedig a bitek összefűzését, konkatenációját jelöli)
   - *Ciklus vége*
   - $(L_0,R_0)=IP(m_1m_2\ldots m_{64})$, kezdő keverés $L_0=m_{58}m_{50}\ldots m_8$, $R_0=m_{57}m_{49}\ldots m_7$
2. Iterációk
   - *Ciklus i=1-től 16-ig*
     - $L_i=R_{i-1}$
     - $R_i=L_{i-1} \oplus f(R_{i-1},K_i)$, ahol $f(R_{i-1},K_i)=P(S(E(R_{i-1}) \oplus K_i))$ *a következők szerint:*
       - $T=E(R_{i-1})=r_{32}r_1r_2\ldots r_{32}r_1$, a 32 bites R-ből 48 bites T készül
       - $T' = T \oplus K_i$
       - Bontsuk fel T'-t 8 darab 6 bites részre: $(B_1,\ldots B_8)$
       - $y_1=S_1(B_1)$, $y_2=S_2(B_2)$, … ,$y_8=S_8(B_8)$. Az aktuális *S-box* táblázatának sorindexe B bitjei alapján $r=b_1b_6$ *és oszlopindexe* $c=b_2b_3b_4b_5$. Például $S_1(011011) = 5$, mert *r=1 c=13*.
       - $T'' = y_1y_2y_3\ldots y_8$
       - $T''' = P(T'')$. Ha $T''=t_1t_2\ldots t_{32}$, akkor $T'''=t_{16}t_7\ldots t_{25}$
       - $f(R_{i-1},K_i)=T'''$
   - *Ciklus vége*
3. Lezáró műveletek
   - $b_1b_2\ldots b_{64}=(R_{16},L_{16})$ (Végső csere)
   - $C=IP^{-1}(b_1b_2\ldots b_{64})=b_{40}b_8\ldots b_{25}$ (Kezdő keverés inverze)

Ez az algoritmus a titkosítást írja le, de ugyanezt kell használni megfejtéskor is, csak a körkulcsok sorrendjét kell felcserélni. Ha a titkosítás folyamán rendre a $K_1,K_2,K_3,\ldots,K_{16}$ körkulcsokat használtuk, akkor megfejtéskor a $K_{16},K_{15},K_{14},\ldots K_1$ sorrendet kell figyelembe venni.

A kezdő keverés (*IP*) és annak inverze (IP^{-1}):

	IP							
L_0	58	50	42	34	26	18	10	2
	60	52	44	36	28	20	12	4
	62	54	46	38	30	22	14	6
	64	56	48	40	32	24	16	8
R_0	57	49	41	33	25	17	9	1
	59	51	43	35	27	19	11	3
	61	53	45	37	29	21	13	5
	63	55	47	39	31	23	15	7

IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

A körfüggvényben alkalmazott kiterjesztés (*E*) és keverés (*P*):

E					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

A körkulcsok készítésénél használt PC1 és PC2 függvény:

	PC1						
C_i	57	49	41	33	25	17	9
	1	58	50	42	34	26	18
	10	2	59	51	43	35	27
	19	11	3	60	52	44	36
D_i	63	55	47	39	31	23	15
	7	62	54	46	38	30	22
	14	6	61	53	45	37	29
	21	13	5	28	20	12	4

PC2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Az *f()* függvényben alkalmazott helyettesítések (*S*-dobozok) a következők:

SOR (r)	oszlop sorszám (*c*)															
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
								S₁								
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	3	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
								S₂								
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
								S₃								
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
								S₄								
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
								S₅								
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
								S₆								
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
								S₇								
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
								S₈								
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Az S-dobozok tulajdonságai

Az *S*-dobozok eredeti tervezési módszereit mind a mai napig titok fedi, legalábbis nem nyilvánosak. Az évek során a tüzetes vizsgálatok során azért kiderült róluk egy-két tulajdonság. Bármely *S*-dobozra igaz, hogy

- nemlineáris,
- soraiban mindig a 0..15 számok szerepelnek valamilyen sorrendben,
- a bemenetét egy bittel megváltozatva a kimenete legalább kettő bitben megváltozik,
- $S(x)$ és $S(x \oplus 001100_h)$ legalább kettő bitben különbözik,
- $S(x) \neq S(x \oplus 11ef00)$, ahol „*e*” és „*f*” értéke „*0*” vagy „*1*”.

DES tesztvektorok

Minden algoritmus specifikációja tartalmaz olyan tesztadatokat, amelyekkel ellenőrizhető egy megvalósítás helyes működése. Ezeket a (kulcs, nyílt szöveg, titkos szöveg) hármasokat *tesztvektoroknak* hívjuk. Íme két példa a DES-re (a FIPS46 nem közöl hivatalos tesztvektort):

```
Nyílt szöveg : „Now is the time for all ”
K=0123456789ABCDEF
P=4E6F772069732074 68652074696D6520 666F7220616C6C20
C=3FA40E8A984D4815 6A271787AB8883F9 893D51EC4B563B53

Nyílt szöveg : „It's DES”
K=0123012301230123
P=4974277320444553
C=8FCF48864D378858
```

3.1.4. A DES feltörése

A DES – bonyolultságától eltekintve – alapvetően egy egybetűs helyettesítő (most egy „betű” = 64 bit). Az alkalmazott transzformációt még senkinek sem sikerült elméletileg feltörnie, jóllehet a *differenciális és a lineáris kriptoanalízis* ért már el eredményeket. A kulcstér viszonylag kis mérete, – ami már az elején is sok kritikát váltott ki, – a mai számítási kapacitásokat figyelembe véve már nem igazán véd a *brute-force* támadásokkal szemben. A kulcs rövidsége adta az ötletet, hogy egymás után alkalmazzák a DES algoritmusát két különböző kulccsal. Így a kulcs tulajdonképpen 2*56=112 bites lesz. Csakhogy *Merkle* és *Hellman* 1981-ben publikált egy módszert, amelynek megvalósítása ugyan gyakorlati nehézségekbe ütközik, de bebizonyította, hogy a kétszeres DES nem sokkal jobb elődjénél. A módszert **középen találkozó** (*meet in the middle*) feltörési kísérletnek hívják.

A középen találkozó feltörési kísérlet

Tegyük fel, hogy valaki kétszeresen titkosít egy üzenetet, valamint a kódtörő rendelkezik néhány (m_i, M_i) kódpárral, ahol m_i a bemeneti nyílt szöveg, M_i pedig a kimeneti, kétszeresen titkosított kriptoszöveg. Ezek kapcsolatát a következőképpen írhatjuk fel:

Az m_i üzenet titkosítva van K1 kulccsal és a titkosítás eredménye ismét titkosításra kerül K2 kulccsal:
m_i → C és K1 → x → C és K2 → M_i

$$M_i=C_{K2}(C_{K1}(m_i)) \qquad \text{I.}$$

Ha a D_{K2} megfejtő függvényt az egyenlet mindkét oldalán alkalmazzuk, akkor az eredmény:

Az m_i üzenet titkosítása és az M_i üzenet megfejtése ugyanazt az eredményt adja:
m_i → C és K1 → x ← D és K2 ← M_i

$$D_{K2}(M_i)=C_{K1}(m_i) \qquad \text{II.}$$

A középen találkozó feltörési kísérlet ezt az összefüggést használja fel a *K1* és a *K2* kulcs kitalálásához:

1. Kiszámítja az $R_i=C_i(m_1)$ értékeket mind a 2^{56} db kulcsra. Ez a II. egyenlet jobb oldalának összes lehetséges értékét jelenti, és ezek egyikére igaz, hogy $i=K1$ vagyis a keresett kulccsal van titkosítva m_1.
2. Kiszámítja az $S_j=D_j(M_1)$ értékeket mind a 2^{56} db kulcsra. Ez a II. egyenlet bal oldalának összes lehetséges értékét jelenti, és ezek egyikére igaz, hogy $j=K2$ vagyis a keresett kulccsal lesz megfejtve M_1.
3. Az első táblázatban olyan R_i értéket keres, amellyel egyező S_j érték van a második táblázatban. Ha ilyet talál, máris megvan egy potenciális kulcspár (i,j), amelyre igaz, hogy $C_i(m_1)=D_j(M_1)$ vagyis m_1 titkosítása ugyanazt az eredményt adja, mint M_1 megfejtése.
4. Ellenőrzi, hogy $C_j(C_i(m_2))$ egyenlő-e M_2-vel. Vagyis elvégzi a kétszeres kódolást, így győződik meg arról, hogy más meglévő nyílt-titkos szövegpárra is helyesen működnek a kulcsok. Ha igen, további ellenőrzéseket végez, egyébként folytatja a keresést.

Sok kulcspár jelentkezik, mielőtt a valódi kulcspárok előkerülnének, de ez előbb-utóbb megtörténik. A kétszeres DES elleni támadás műveletigénye a várt 2^{112} DES-művelet helyett „mindössze” 2^{57} DES-művelet. Az egyszeres, egykulcsos DES feltöréséhez legfeljebb 2^{56} DES-művelet szükséges, tehát a kétszeres DES csupán kétszeres védelmet nyújt, ami messze elmarad a várt 2^{56}–szoros védelemtől. Olyan, mintha mindössze 57 bites lenne...

A támadás fő problémája, hogy a táblázatok tárolásához 2^{57} DES-szó, azaz 2^{60} bájt szükséges, így ebben a formában nem kivitelezhető. Azonban *Merkle* és *Hellman* olyan optimalizálási lehetőségekre is rámutatott, melyekkel a tárigény csökkenthető a műveletszám rovására. Például, az *R* és *S* táblázatok indexei csak 55 bitesek, a táblázatok feltöltésénél a kulcsként használt indexeket kiegészítjük egy – egy „0”-val, így használjuk őket kulcsnak:

$$R_i=C_{i0}(m_1) \qquad S_j=D_{j0}(M_1)$$

Az $R_i = S_j$ összehasonlítás pedig igazából négy műveletet jelent majd:

$$R_{i0} = S_{j0} \rightarrow R_i = S_i$$
$$R_{i0} = S_{j1} \rightarrow R_i = D_{j1}(M_1)$$
$$R_{i1} = S_{j0} \rightarrow C_{i1}(m_1) = S_j$$
$$R_{i1} = S_{j1} \rightarrow C_{i1}(m_1) = D_{j1}(M_1)$$

ahol az index után írt „0” vagy „1” annak kiegészítését jelenti. Az összehasonlítás során az értékek egyik felét (R_i,S_i) ismerjük, másik felét menet közben (*on the fly*) kell kiszámolni. Ez a megoldás ugyan megduplázza a műveletek számát, de felére csökkenti a szükséges tárterületet.

A háromszoros kódolást már 1979-ben javasolta az IBM. A módszer, amit már azóta nemzetközi szabványként is elfogadtak (*ANSI X9.52*) a következő:

Az M titkosított üzenet előállítása:

1. *Az m nyílt szöveget titkosítjuk K3 kulccsal.*
2. *Az eredményt „megfejtjük" K2 kulccsal.*
3. *Az előző lépés eredményét ismét titkosítjuk, de most K1 kulccsal.*

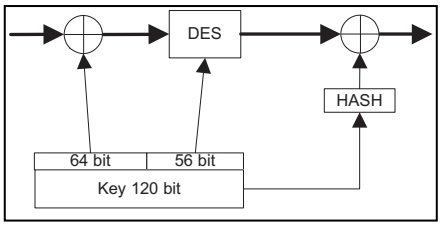
$$M=C_{K1}(D_{K2}(C_{K3}(m)))$$

A következő esetek fordulhatnak elő:

1. K_1, K_2 és K_3 között nincs egyforma.
2. K_1 és K_2 egyforma, K_3 különbözik tőlük.
3. Mindhárom egyforma. (Ez gyakorlatilag az egyszeres DES-sel egyenlő.)

Az első esetben három egymástól független kulcsot használunk és ez a *TripleDES*. Viszonylag ritkán használt megoldás, mert még a legóvatosabb titkosítási szakemberek is egyetértenek abban, hogy a 112 bites kulcs egyelőre bőven elég, a 168 bit csak feleslegesen továbbítandó adathalmaz lenne a tárolás és a kulcscsere során, bár ez a szempont manapság egyre kevésbé fontos. Gyakorlati megoldás a kettő kulcs használata. Ez a *3DES*. A *CDC* sorrenddel együtt a korábbi egykulcsos DES-rendszerekkel való kompatibilitást is biztosíthatja: ha $K_1=K_2$, akkor az első két fokozat eredménye az eredeti nyíltszöveg, így a három fokozat gyakorlatilag egy K_1 kulcsos egyszeres DES-ként működik. Ha a fokozatok *CCC* szerepet látnának el, ez a piaci jelentőségű kompatibilitás nem lenne igaz. (A sorrendeket szokás EDE és EEE betűkkel is jelölni.)

A *C* és *D* algoritmusok egyébként kriptográfiai szempontból egyenértékűek, kulcs nélküli invertálásuk egyformán nehéz. Ez a DES esetében egyszerűen belátható, hiszen ugyanazt az algoritmust használjuk a titkosításhoz és a megfejtéshez egyaránt.



Sajnos a *TripleDES* és a *3DES* egyaránt rendelkeznek azzal a nem túl kellemes tulajdonsággal, hogy az eredeti DES-nél háromszor lassabbak, hiszen az eredeti algoritmust használják három alkalommal. A DES számtalan variációja látott napvilágot az idők folyamán. Ezek többsége igyekszik olyan „körítésekkel" kiegészíteni a DES-t, hogy az effektív kulcshossz hosszabb legyen, mint a sokat kritizált 56 bit, és a sebesség se essen nagyon vissza. Ne feledjük: a DES matematikailag stabil, még senkinek sem sikerült elvi módon feltörnie. Így jelenleg egyetlen gyengeségnek a viszonylag rövid kulcsméret tűnik. Példaképpen lássuk egy elterjedt DES-variáns, a *DESX* egyszerűsített tömbvázlatát [42]!

A kétkulcsos megoldás azonban visszavezethető a kétszeres DES-re, így előbb-utóbb fel kell áldozni a kommunikációs csatornák sávszélességéből, és szorgalmasan három független kulcsot kell használni. Jelenleg nem ismert olyan algoritmus, amely a háromszoros (168 bites) DES-t valamilyen módon feltörné, illetve bizonyítható, hogy a három különböző kulcs nem helyettesíthető egyetlen kulccsal, vagyis nincs olyan K4 kulcs, amire $C_{K1}(D_{K2}(C_{K3}(m))) = C_{K4}(m)$ igaz lenne (a DES művelete nem zárt, így nem alkot csoportot sem).

Az egyszeres DES kiváltására rengeteg lehetőség van már (*TWOFISH, CAST, FEAL, RC6 stb.*), de az egyik legígéretesebbnek és legbiztonságosabbnak egy ideig az IDEA tűnt. Mára

azonban háttérbe szorult, például a PGP is áttért az IDEA használatáról a CAST-256-ra, sőt 2000. októberében a DES hivatalos utódját is kiválasztották, és az sem az IDEA lett.

A DES titkosításában 56 bit hosszúságú kulcsokat használnak, ami a szokásos számítási kapacitások mellett egyelőre megfelelő védelmet jelent, legalábbis addig, amíg csak a szomszéd kíváncsiskodik utánunk. Ha komolyabb, vagy jobban felszerelt ellenfelünk támad, ez a kulcsméret már nem biztos, hogy elég, hiába tűnik olyan rengetegnek a 2^{56} = 72 057 594 037 927 936 lehetséges kulcs, mégiscsak véges a számuk. Aki próbálgatással, *brute-force* módon szeretne megfejteni egy üzenetet, annak ennyi kulcsot kellene kipróbálnia. Némi számolgatás után belátható, hogy ennek a feladatnak hatalmas erőforrásigénye van (processzoridő, processzorsebesség), ezért lehetetlennek tűnik. Azonban ez korántsincs így.

„Elosztott hálózat" - Distributed.net

A számítási kapacitás növelésének egyik módja, hogy a csillagos égig növeljük a számítógép teljesítményét és így eljutunk a szuperszámítógépekhez, de ezek köztudottan kevesen vannak és borzalmas áruk van. A másik megoldás az, hogy igyekszünk lehetőség szerint igen gyors számítógépet alkalmazni, de nem egy darabot, hanem minél többet, sőt rengeteget, a feladatot pedig független részfeladatokra bontjuk és megosztjuk a résztvevő számítógépek között. A megosztott feldolgozás elve nem új. Hivatalos publikációban *Kínai lottó* néven vált ismertté (1991), és a DES feltörését oldotta meg „egyszerű" módon. Az ötlet szerint minden kínai háztartás rádió- vagy televíziókészülékét fel kellene szerelni egy olyan olcsó DES-chippel, amely 1 millió kódolást végezne másodpercenként. Feltételezve, hogy mind az 1,2 milliárd kínai lakos rendelkezik a készülékkel, a kínai kormány 60 másodperc(!) alatt feltörhet egy DES-üzenetet. A módszer működik, és nemcsak kódtörésre használják: a faktorizálásra vonatkozó adatokban (*30. ábra*) láthatjuk, hogy 1988 óta ezzel a módszerrel érnek el sikereket, de így keresik az új *Mersenne* prímeket vagy a SETI[21] program keretében a földönkívüli intelligenciák nyomait is. Ha egyszerre több száz (vagy százezer) számítógép számol, a számítási kapacitás a több százszorosára (százezerszeresére) nő, és akár több Tflops összesített sebesség is elérhető. Általában véve minden olyan esetben alkalmazható a módszer, ahol a megoldás egymástól független részfeladatokra bontható.

> A SETI projekt működésének alapja, hogy a rádióteleszkóp adatait feldarabolják, az egymástól független részek letölthetők az Internetről a feldolgozó ügyfélprogrammal együtt, ami a számítás elvégzése után az eredményt visszaküldi, és letölti az új csomagot.
>
> A SETI központba 1999.10.22. 12:00 és 1999.10.23. 12:00 között, 24 óra alatt, 268531 eredmény érkezett. Egy csomag feldolgozásának ideje átlagosan 20 óra 40 perc volt. Ez a 24 óra több mint 632 processzorévnek felelt meg. Két évvel később ezek az adatok a következőképpen alakultak: 2001.12.22. 16:00 és 2001.12.23. 16:00 között 646039 eredmény érkezett. A csomagok feldolgozási ideje 15 óra 04 percre csökkent, míg az összteljesítmény 29,1Tflops volt. Ez az időszak 1110 proceszszorévnek felelt meg. A gyorsulás részben a résztvevők számának növekedésével, részben pedig a gépek teljesítményének növekedésével magyarázható.

[21] *Search for Extraterrestrial Intelligence - Kutatás földünkívüli értelem után* - http://www.setiathome.ssl.berkely.edu

A DES-kódok feltörésében, ha az *brute-force* módon történik, a kulcstér egyes részei egymástól függetlenek, így azok a résztvevők között kioszthatók. Amikor az RSA kódtörő versenyt hirdetett a DES, az RC5[22] és az RSA feltörésére is, hasonló módon értek el eredményt a megfejtők. Ezek komoly versenynek tekinthetők főként, ha figyelembe vesszük a 10 000 dolláros jutalmat, amit az a kódtörő (vagy csapata) kapott, aki megnyerte a versenyt, azaz elsőként visszaküldte a visszafejtett üzenetet[23]. Az RSA-nak jelenleg is vannak megoldatlan (faktorizálás) pályázatai 10 000 – 200 000 dolláros díjjakkal. Az 56 bites kulcs feltöréséhez 1997-ben az Interneten keresztül együttműködő számítógépek ezreinek 96 napra volt szüksége. A második verseny eredménye 40 napon belül megérkezett. A *distributed.net* szervezett keresése a kulcstér 85%-t vizsgálta át, mire a megfejtés előbukkant.

DES Challenge I.
96 nap – *„The unknown message is:"*
1997. január – distributed
DES Challenge II-1.
41 nap – *„Many hands make light work"*
1998. január – Distributed.net – 17 *milliárd kulcs/sec*
DES Challenge II-2.
56 óra – *„It's time for those 128-, 192-, and 256-bit keys"*
1998. július – Deep Crack – *107 milliárd kulcs/sec*
DES Challenge III .
22 óra 15 perc – *„See you in Rome (Second AES Conference), March 22-23,1999"*
1999. január – (≈100 000 PC + Deep Crack) – *245 milliárd kulcs/sec*

Az fenti időpontok, időtartamok és a valóság között lehet eltérés. Nem tudom, melyek a pontos dátumok és időtartamok, mert más szerepel az RSA és más az EFF és megint más a Distributed.net honlapján és más a [42]-ben. Talán a dátumok már helytállóak, de például az időtartamok [42] szerint rendre: 5 hónap, 39 nap, 56 óra, 22 ¼ óra

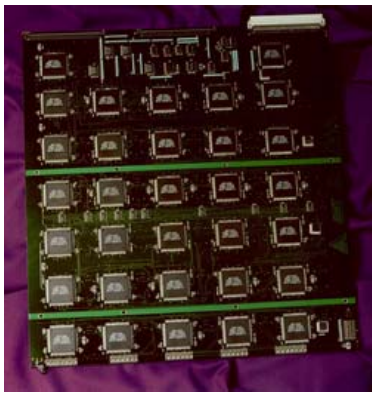
A *Deep Crack – DES törő célgép*

Az 1998. július 13.-án kiírt *DES Challenge II-2* megoldása már 3 napon (pontosabban 56 órán) belül megérkezett. 1993-ban *Michael Wiener* adta a legrészletesebb leírást egy kódtörő gép és annak különleges alkatrészeinek felépítésére, működésére vonatkozóan [29]. Körülbelül 1 000 000 dollárra becsülte annak a gépnek a költségét, amivel a számítások szerint 3,5 óra alatt megkereshető a kulcs. 1998-ban kiegészítette és aktualizálta tervét[24]: megszületett az elvi számítógép, ami egy ismert nyíltszövegből és egy ismert rejtjelezett szövegből rövid idő alatt kitalálta a használt 56-bites kulcsot. Az EEF már 1997-ben elkezdett befektetni a gép megépítésébe, ami az eredeti tervek alapján először FPGA áramkörök felhasználásával készült, később az *Advanced Wireless Technologies* elkészítette azt a végleges áramköri lapkát, melyből 1792 darabot használtak fel a gép építéséhez.

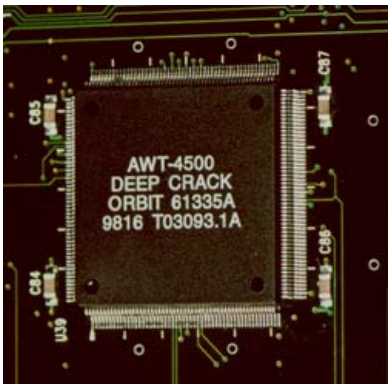
[22] Közel ötéves küzdelem után 2002. augusztusában jelentették be az RC5 64 bites verzióját törő verseny végét. (A kulcs már állítólag júliusban meglett, de némi technikai hiba csúszott a lebonyolításba...)
[23] A díjnak csak egy része kerül kiosztásra, a többit sorsolással választott jótékonysági célra fordítják.
[24] A folyamatos fejlődés következtében ekkor az 1.000.000$ már 1 óra körüli időt tett volna lehetővé.

		A feltöréshez szükséges becsült idő[25]	
Költség	*Technológia*	*40 bites kulcs*	*56 bites kulcs*
nincs	idle time	1 hét	1260 év
10 000$	FPGA	12 perc	556 nap
300 000$	FPGA vagy spec. IC	24 sec	19 nap
10 000 000$	FPGA vagy spec. IC	0,7 sec	13 óra
300 000 000$	FPGA	0,2 msec	12 sec
A brute-force kulcskeresés lehetőségei			

A projekt – amely 18 hónap alatt megvalósult – összköltsége körülbelül 210 000 dollár volt, ami jóval elmaradt az eredetileg Wiener által becsült összegtől. (Igaz, a sebesség is...) Ebből 80 000 dollárt az AWT számlázott ki, a maradék fedezett minden más költséget: az áramköri lapokat, kábeleket, hűtést, stb. Az eredmény pedig nem más, mint egy speciális számítógép, a *DEEP CRACK* lett [23,52].



21. ábra *Deep Crack* egy alaplapja



22. ábra A Deep Crack chip

A *Deep Crack* architektúrája dióhéjban

A rendszer a „keresőegységek” köré épült. Ezek a speciális DES-chipek nem tudtak mást, csak egymástól függetlenül próbálgatták a lehetséges kulcsokat, és ha nem találtak „érdekes szöveget”, vették a következő kulcsot, és újra próbálkoztak. Érdekes szövegnek számított az alfanumerikus karakterek egymást követő felbukkanása. A keresőegységek órajele mindössze 40 MHz volt, és 16 óraciklus alatt vizsgáltak meg egy kulcsot (egy iterációs kör ciklusonként).

Egység		*Kulcs / sec*	*DT*[26]
Keresőegység	-	2 500 000	333 600,0
Chip	= 24 kereső egység	60 000 000	13 900,0
Alaplap	= 64 chip	3 840 000 000	217,0
EFF *Deep Crack*	= 28 alaplap	107 520 000 000	7,7

[25] *Daniel J. Ryan*, a *Science Applications International Corporation* társelnöke által összeállított táblázat, *1999.*
[26] DT : Adott szinten ennyi idő szükséges a 2^{56} darab kulcs kipróbálásához (napokban)

Minden *Deep Crack* chip 24 ilyen keresőegységből állt. Az egész gép 28 alaplapot tartalmazott, minden alaplapon 64 darab *Deep Crack* chip ücsörgött néhány vezérlőlogika és óragenerátor társaságában[27]. Az egyes alaplapok között egy Linuxos Pentium PC játszotta a kommunikációs busz és a vezérlőegység szerepét. Bármelyik chipet le tudta állítani, újra tudta indítani és azok regisztereit írhatta, olvashatta. A keresés megkezdésekor ez a PC osztotta ki a vizsgálandó kulcstartományokat, és ez fogadta a feltételezett kulcsokat is. Ez a felépítés lehetővé tette a könnyű bővítést, valamint igen jó hibatűrést is biztosított, hiszen néhány chip kiesése nem okozott jelentős fennakadást vagy teljesítménycsökkenést. Az chipek belső felépítése is erős párhuzamosságot mutat: a visszafejtés, a nyílt szöveg tesztelése, az új kulcs előállítása mind párhuzamosan történt, ez tette lehetővé, hogy a DES-motor folyamatosan, teljes sebességgel működjön.

Az igazság pillanata?

Többször szóba került bizonyos beszélgetéseken, hogy megtörték-e a DES-t vagy sem. Erre kétféle válasz adható:

- nem törték meg, mert senkinek sem sikerült olyan hatékony algoritmust találnia, amely a kulcs nélküli invertálást megvalósítja.
- megtörték, mert például a *Deep Crack* csak maga alig több, mint két nap alatt megfejtett egy üzenetet.

Nos, mindkét válasz igaz. De helyesebb lenne talán így fogalmazni: az algoritmust nem törték meg, csak az általa nyújtott védelem már nem elegendő. A *brute-force* módszerrel történő kulcskeresés nem mindig tekinthető sikeres feltörési módszernek, mert a nyílt szövegjelöltek nem ellenőrizhetők le minden esetben. Egy vélt értelmes karaktersorozat ASCII kódokkal leírva könnyen ellenőrizhető annak elolvasásával vagy statisztikai elemzésével, de egy véletlen bitsorozatot vajon hogyan kell leellenőrizni a kulcsok próbálgatása során? Vagyis a *brute-force* csak akkor használható, ha a támadó tudja, hogy mit keres. Ha a nyílt szöveg nem ASCII, vagy a támadó nem ismeri a nyílt szöveg jellemzőit, szerkezetét (formátumát, struktúráját), akkor nem ér semmit a *brute-force* támadással, hiszen nem tudja felismerni a helyesen visszafejtett eredményt. Jegyezzük meg: a *brute-force* csupán tengernyi próbálkozás és csak akkor tudjuk felismerni a helyes eredményt, ha pontosan tudjuk, hogy mit keresünk, és annak milyen lesz a megjelenési formája!

Másrészt ne felejtsük el, hogy a *brute-force* nem hatékony algoritmus, csak univerzális. A mindennapi életben sem valószínű, hogy sokaknak lenne otthon egy-egy *Deep Crack* masinájuk a sarokban. Néhány izmosabb asztali géppel vagy vállalati szerverrel sem érdemes nekiállni a kulcsok próbálgatásának, mert reménytelen feladat.

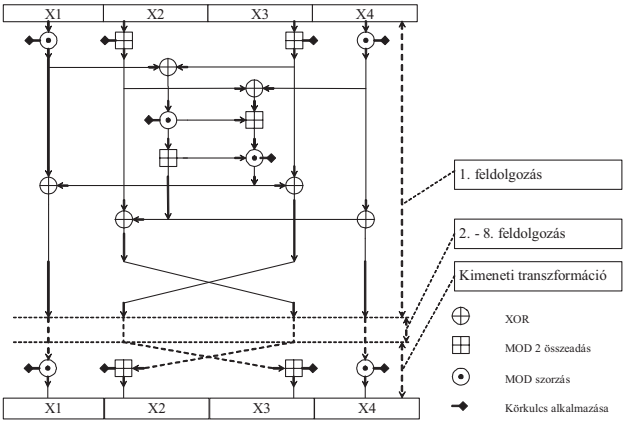
A fentiek miatt a *brute-force* eljárást egy olyan elvi megoldási lehetőségnek kell tekinteni, ami a gyakorlati feladatok során néha használható, néha pedig nem.

[27] Mivel minden általam látott fényképen egy alaplapon mindössze 32 chip van, viszont minden irodalom 64 chipről ír, így kénytelen vagyok feltételezni, hogy az alaplap kétoldalas szerelésű, de ezt eddig semmi sem erősítette vagy cáfolta meg.

23. ábra EFF Deep Crack és a vezérlő PC

3.2. IDEA

Az IDEA (1990-1992 Dr. Lai és Prof. Massey, *The Swiss Federal Institute of Technology*) szimmetrikus kulcsú rejtjelező algoritmus, tehát a DES-hez hasonlóan mind a küldőnek, mind a címzettnek ismernie kell ugyanazt a titkos kulcsot. Az algoritmus a DES-hez hasonlóan 64 bites blokkos eljárás: a nyílt szöveg 64 bitjéhez a rejtjeles szöveg 64 bitjét rendeli. A DES-sel ellentétben az IDEA 128 bites kulcsot használ, melyből 52 darab 16 bites körkulcsot származtat. A feldolgozást 8 iterációban végzi és egy végső, kimeneti transzformációval zárja le. Minden lépéshez hat darab, a kimeneti transzformációhoz pedig négy darab kulcs tartozik.



24. ábra IDEA blokkvázlat

Az algoritmus gyors, jól implementálható hardverkörnyezetben is. Kifejezetten adatátvitelhez tervezték, beleértve a digitalizált hang és kép valósidejű titkosítását is. Jelenleg nem ismert olyan algoritmus, ami az IDEA-t elvi módon feltörné, az egyetlen támadási mód a *brute-force*.

A számítógépek folyamatos sebességnövekedése igen gyors folyamat, de feltehetően soha nem fognak addig a pontig eljutni, hogy egy IDEA üzenetet *brute-force* módon fel tudjanak törni – ezt kulcsméretének köszönheti. Egy ideig a DES utódjaként tekintettek rá, azonban lassan a háttérbe szorult – ma már csak egy algoritmus a sok közül.

3.3. RC5

Az RC5 a *Ron Rivest* által tervezett algoritmus-sorozat tagja[28]. Sok alkalmazásban még RC2-t használnak, bár ez egyre kevésbé jellemző, és főleg nem ajánlott. Az RC6 algoritmust nevezték az AES-pályázatra is. Az RC5 olyan blokkorientált algoritmus, mely a *w=16, 32* vagy *64* bites környezetben érzi jól magát. Igen egyszerű felépítése van, hardver- és szoftver-implementációi egyaránt jó hatékonyságúak. Nemcsak a blokkméret változhat (ami $2\times w$ bit), hanem a körök száma (r) és a kulcs hossza (b) is paraméterezhető. Emiatt egy-egy implementáció pontos jelölésére az RC5-w/r/b az elfogadott forma (w bitekben, b bájtokban mérve). Ajánlott értékek: 32/12/16, 64/16/16. Az algoritmus összeadás (*mod w*), XOR, forgatás műveleteket használ. Az algoritmus érdekessége, hogy a szokásos rögzített vagy származtatott lépésszám helyett a pillanatnyi adattól függő lépésszámmal hajtja végre a forgatásokat (*data depending rotating*): *X ror Y*, ahol *Y* az előző *X* értéktől függ. Gyakorlatilag nem lehet előre megjósolni, hogy mikor hány bittel fog történni a forgatás.

3.4. AZ AES PÁLYÁZAT

Mintegy két évtizedes múlt után a DES elérte életciklusának végét, és lassan nyugalomba vonulva átadja helyét az új generációnak. Mivel a különböző számítógép-hálózatokon és egyéb kommunikációs csatornákon áramló adatok mennyisége egyre csak növekszik, az örökség nem kicsi. A DES jól helytállt az információ-feldolgozó rendszerekben (*ANSI X3*), banki alkalmazásokban (*ANSI X9*) és sok egyéb kereskedelmi vagy polgári alkalmazásban. 1985-ben úgy nézett ki, ISO szabvány lesz belőle. 1997-re az USA titkosítási rendszereinek majdnem fele (48%) tartalmazta a DES-t. Hardver-, és szoftvermegoldása minden kombinációban kifejlődött.

> *Diffie* és *Landau* a következő élettartam-fogalmakat definiálta a kriptorendszerekre:
> - *Mennyi ideig van használatban a rendszer vagy az algoritmus?* Ez általában egy bevált algoritmus vagy rendszer esetén 20 év körüli időtartam.
> - *Mennyi ideig maradnak titkokban a rendszer által titkosított üzenetek?* Ez az idő jó esetben hosszabb a használati időnél, és az elvárt minimális mérték 10-50 év. Ennek az időtartamnak az az érdekessége egyébként, hogy egy-egy új algoritmustól elvárjuk ugyan, hogy mondjuk 40 év múlva is védje a mai titkokat, holott fogalmunk sincs róla, milyen is lesz a számítási kapacitás akárcsak 10-20 év múlva.

[28] Biztos, ami biztos a márkabejegyzés RC1-től RC9-ig tart. Az RC rövidítés feloldása egyébként nem egyértelmű: Rivest's Cipher vagy Ron's Code – de az RSA mindkettőt „elismeri"... Az RC sorozat tagjainak egyébként nem sok közük van egymáshoz, csak a nevük hasonló. Például az RC4 nem elődje az RC5-nek, hanem attól egy gyökeresen eltérő algoritmus. Történetesen nem is blokkos, hanem folyamtitkosító!

A DES mára egyszerűen idejétmúlt algoritmus lett. Végül azok, akik ebben érintettek, belátták, hogy szükséges a DES felváltása, különös tekintettel az elektronikus kereskedelem térnyerésére. Bár a *TripleDES* és a *3DES* jó alternatívának látszik, 1996-ban a NIST (*National Institute for Standards and Technology*) megkezdte egy új algoritmus előkészítését, majd 1997. januárjában elindította az AES-projektet. 1997 szeptemberében az alapelvárásokat is deklarálták és a pályázat ezzel kiírásra került. Az eredeti elképzelések szerint nem volt cél egy adott algoritmus elfogadása, csak olyan algoritmust kerestek, ami – a DES egykori kifejlesztéséhez hasonlóan – alapot ad a jövő algoritmusához. Az algoritmus neve *AEA – Advanced Encryption Algorithm* és az ebből előálló szabvány az *AES – Advanced Encryption Standard.* Az AES fejlesztése (a hivatalos álláspont szerint) a teljes nyilvánosság előtt kell, hogy történjen. Az egész projektet a NIST koordinálta, jelentős szerepet vállalva a szabványok felhasználásában és a tesztelésben. A NIST algoritmusértékelést (nyilvánosan) nem végzett, döntéséhez a kriptográfus közösség eredményeit használta fel.

3.4.1. Pályázati követelmények, események

A kiírt pályázatban az alapvető elvárások is szerepeltek, és természetesen az egész AES-projekt menetét is ismertették.

- *Elvárások*
  - Szimmetrikus kulcsú, blokkos algoritmust kell megvalósítani
  - 128 bites blokkokat kell használnia (eredetileg a 192 és 256 bites blokkok kezelése is követelmény volt, de a végső kiírásban ez már nem szerepelt)
  - 128 – 192 – 256 bites kulcsmérettel kell dolgoznia
  - Gyorsabb legyen, mint a 3DES és nyújtson annál jobb védelmet
  - Hatékony megvalósíthatóság (tekintettel a számítási teljesítményre, a kód és adatmemória szükségletre, az előkészítő számításokra)
  - Flexibilitás az egyes platformok és későbbi fejlesztések tekintetében
- *A nevezéshez szükséges dokumentumok*
  - Az algoritmus specifikációjának komplett leírása
  - A teljesítményekre vonatkozó számítások, becslések, mérések
  - Teszt-vektorok (nyílt – titkosított szövegpárok)
  - Analízis az ismert támadási módszerekkel szembeni helytállásról
  - Az algoritmus lehetőségei és korlátai
  - Referenciamegvalósítás ANSI C-ben
  - Optimalizálási lehetőségek C illetve JAVA megvalósításban
- *Kiválasztás*
  - Három konferencia - Három tesztidőszak

Az alábbi táblázatban az AES-projekt legfontosabb dátumait és eseményeit gyűjtöttem össze.

1997. 01. 02.	A projekt indulása
1997. 09. 12.	A pályázat kiírása
1998. 05. 15.	A pályázatok beérkezésének határideje.
1998. 08. 20.	**I. konferencia, Ventura**, Kalifornia. 15 jelölt bemutatása (hatot eleve kizártak)
1. kiértékelési periódus (fő szempont: biztonság és szoftverhatékonyság)	
1999. 03. 22.	**II. Konferencia, Róma**. Az első kiértékelési periódus vizsgálatai és a konferencián bemutatott eredmények alapján a következő algoritmusok estek ki: FROG, MAGENTA, LOKI97, DEAL, HPC (sebességi és biztonsági problémák)
1999. 04. 15.	1. kiértékelési periódus és a II. Konferencia lezárása
1999. 08. 09.	A NIST kihirdeti az 5 döntőst (*finalist*): MARS, RC6, RIJNDAEL, SERPENT, TWOFISH Az első kiértékelési periódus további kiesői: CAST256, CRYPTON, DFC, E2, SAFER+ (hatékonysági, megvalósítási költség- és sebességproblémák miatt)
2. kiértékelési periódus (fő szempont: biztonság és hardverhatékonyság)	
2000. 04. 13.	**III. konferencia, New York** (különösebb áttörés vagy eredmény nélküli, csak gyakorlatban nem használható törési kísérletek)
2000. 05. 15.	2. kiértékelési periódus és a III. Konferencia lezárása
3. kiértékelési periódus	
2000. 10. 02.	Eredményhirdetés: a RIJNDAEL a győztes
2001. 11. 26.	FIPS197 kibocsátása, ezzel (már majdnem) szabvánnyá vált a Rijndael.
2002. 05. 26.	A FIPS197 hatályba lépésének dátuma, ez *előtt* még nem beszélhetünk szabványról!

3.4.2. Az AES-pályázat jelöltjei

A pályázatra 21 nevezés érkezett, de csak az alábbi 15-öt fogadták el [13]. (A kiemelt algoritmusokat választották ki a második konferencián.)

Név	***Fejlesztő***	
CAST-256	*Entrust Technologies Inc.*	Kanada
CRYPTON	*Future System Inc.*	Korea
DEAL	*Richard Outerbridge, Lars Knudsen*	Kanada
DFC	*Centre National pour la Recherche Scientifique*	Franciaország
E2	*Nippon Telegraph and Telephone Corp.*	Japán
FROG	*TecApro Internacional S.A.*	Costa Rica
HPC	*Rich Schroeppel*	USA
LOKI97	*Lawrie Brown, Josef Pieprzyk, Jennifer Seberry*	Ausztrália
MAGENTA	*Deutsche Telekom AG*	Németország
MARS	*IBM*	USA
RC6	*RSA Laboratories*	USA
RIJNDAEL	*Joan Daemen, Vincent Rijamen*	Belgium
SAFER+	*Cylink Corporation*	USA
SERPENT	*Ross Anderson, Eli Biham, Lars Knudsen*	Anglia, Izrael, Norvégia
TWOFISH	*Bruce Scheiner, John Kelsey, Doung Whiting, David Wagner, Chirs Hall, Niels Ferguson*	USA

CAST-256

A CAST-256 a CAST-128 kiterjesztése. Negyvennyolc körben képzi az eredményt, és általánosított Feistel-struktúrát használ. A titkosító és a megfejtő algoritmus ugyanaz.

CRYPTON

A CRYPTON egy SP-háló alapú algoritmus. Tizenkét körben, két – egymással váltakozva használt – körfüggvénnyel dolgozik. A körfüggvények erősen párhuzamosíthatók, ez jó teljesítményt jelent hardver-, és a párhuzamos végrehajtást támogató szoftverkörnyezetben. Gyakorlatilag kétszer olyan gyors, mint a (szimpla) DES. Ellenáll a differenciális és a lineáris kriptoanalízisnek. A titkosítás és a megfejtés egyforma.

DEAL

A DEAL (*Digital Encryption Algorithm with Larger blocks*) egy Feistel-hálózat. Körfüggvényként a DES-t használja. 128 és 192 bites kulcsokhoz hat körrel, 256 bites kulcsokhoz nyolc körrel dolgozik. Az utolsó kör után a két fél adatrészt nem cseréli fel, ez némi aszimmetriát jelent a titkosítás és a megfejtés között. A DEAL-t úgy mutatták be, mint a jól ismert DES evolúciójának következő, finomított lépését, kiküszöbölve annak gyengeségeit. A körkulcsok generálása úgy lett kialakítva, hogy nincs több ekvivalens kulcs[29], sem komplemens tulajdonság[30]. A körkulcsok elkészítése viszont jóval több időt vesz igénybe, mint a DES esetében, ami korlátot jelent olyan alkalmazásokban, ahol gyakran van szükség kulcscserére.

DFC

A DFC (*Decorrelated Fast Cipher*) egy nyolckörös Feistel-struktúra. Moduláris összeadás, szorzás, kivonás, keverés műveleteket használ. A keverés moduláris összeadás és XOR műveletek valamint konstansok felhasználásával történik. A táblázatban tárolt konstansok közül hat adatbit választja ki az aktuálisat (*data depending constant selection*). A titkosítás és a megfejtés algoritmusa egyforma.

E2

Az E2 (*Efficient Encryption*) szintén a Feistel-hálózaton alapul és 12 kört alkalmaz. A körfüggvény: két kulcs alapján történő helyettesítés közé ágyazott keverés valamint bájtforgatás. Az algoritmus bemutatója szerint tervezéskor az elsődleges cél a biztonság, a hatékonyság, a rugalmasság volt. Az alkalmazott *S*-dobozok minden platformon – akár a 8 bites processzorokon is – hatékonyan megvalósíthatók. Már a nyolc körös E2 is ellenáll a differenciális és a lineáris analízisnek. A titkosítás és a megfejtés algoritmusa egyforma.

FROG

A FROG egy nyolckörös, SP hálózaton alapuló, rendhagyó működésű algoritmus. Lényege, hogy a felhasználó által megadott kulcsból indexelésekkel, XOR műveletekkel, helyettesítéssel és keveréssel egy nagy belső kulcsot generál, amely függ a kódolandó adattól és

[29] A kulcs ekvivalens, ha $C=E_k(P)=P$.

[30] A DES komplemens tulajdonsága: ha $C = E_k(P)$, akkor $\overline{C} = E_{\bar{k}}(\overline{P})$.

a felhasználó kulcsától egyaránt. Az algoritmus bemutatója külön kiemelte, hogy az algoritmus nem a hagyományos sémákon alapul. Az alapötlet az, hogy a kulcs meghatározza az egész számítási eljárást, azonban maga az eljárás rejtetten működik abban az értelemben, hogy nem mondható meg, mikor milyen művelet hajtódik végre: ez az aktuális kulcs és az adat együttes függvénye. Az algoritmus ellenáll a differenciális és a lineáris analízisen alapuló támadásoknak, mert a helyettesítések inicializálása gyakorlatilag véletlen értékek felhasználásával történik. A titkosító és a megfejtő algoritmus nem egyforma.

HPC

A HPC (*Hasty Pudding Cipher*) öt alkódolóból áll, amiből a „középső" teljesíti az AES 128 bites blokkméret előírását. Az algoritmus összeadások, kivonások, XOR műveletek, fix lépésszámú forgatások, adatfüggő forgatások komplikált sorozatát használja. A titkosító és a megfejtő algoritmus nem ugyanaz. A tervező szerint az algoritmus hátránya a hosszú kód, a változó memóriaigény, a lassú kulcselőkészítés. Viszont az algoritmus nem „elegáns", így a vizsgálata – ergo a visszafejtés is – nehezebb a szokásosnál (Micsoda előny?!). A 64 bites architektúrákra optimalizált. („doesn't favor Pentium")

LOKI97

Két előző verzió – a LOKI89 és a LOKI91 – alapján készült. Módosított Feistel-struktúrát használ: mind a körfüggvény (kétszeres SP háló) előtt, mind a körfüggvény után a körkulcsok egy részét hozzáadja a jobb vagy a bal részhez. A megfejtés algoritmusa a titkosításhoz hasonló, de nem azonos vele. Ellenáll a differenciális és a lineáris támadásnak.

MAGENTA

A MAGENTA – (*Multifunctional Algorithm for General-purpose Encryption and Network Telecommunication Applications*) – Feistel alapú kódoló, viszont az utolsó kör után nem cseréli meg a jobb és a bal blokkot. A 128 bites és a 192 bites adatblokkhoz hat kört, a 256 bites adatblokkhoz nyolc kört használ. A körkulcsok egyszerűen az eredeti kulcsok 64 bites feldarabolásai. A megfejtő algoritmus gyakorlatilag megegyezik a titkosítóval.

MARS

Harminckét Feistel-struktúrájú kört használ, pontosabban 4×8-at, de mind a négy típusnak más a működése, ami heterogén felépítéshez vezet. A megfejtő algoritmus nem ugyanaz, mint a titkosító, bár hasonló a felépítése. (Az algoritmus gyengébb változatához, ahol a körök száma 11 vagy kevesebb, létezik a brute-force-nál hatékonyabb törési eljárás.)

RC6

Az RC6 a paraméterezhető titkosítók családjába tartozik, módosított Feistel-struktúrát használ, 20 körön keresztül. Az adatblokkot négy darab 32 bites blokkra vágja, és ezekkel operál. A szokásos összeadásokon, XOR műveleteken kívül adatfüggő forgatást (*data depending rotation*) is tartalmaz. Egyszerű, áttekinthető tervezés jellemzi. A megfejtő algoritmus az egyes lépések inverzéből származtatható. Intel platformon a leggyorsabb algoritmus volt minden jelölt közül. (Az algoritmus gyengébb változatához, ahol a körök száma 15 vagy kevesebb, létezik a brute-force-nál hatékonyabb törési eljárás.)

RIJNDAEL

A RIJNDAEL egy helyettesítő és lineáris transzformációkat ötvöző hálózat. A körök száma 10, 12 vagy 14 a kulcsméret függvényében. A bemeneti adatblokkokat 4×4, 4×6 vagy 4×8 bájtos darabokra vágja a feldolgozás előtt (szintén a kulcs méretének függvényében). A körfüggvények három elkülönülő részre bonthatók:

- nemlineáris réteg = minden egyes bájtra egy *S*-dobozt alkalmaz
- lineáris keverő réteg = oszlopok és sorok felcserélése, eltolása
- XOR réteg (*key addition layer*)

A körkulcsok készítése gyors, a megvalósítás párhuzamosítható, ami jelentős teljesítménytöbbletet jelenthet egy hagyományos, szekvenciális elvű működéssel szemben. Az algoritmus kulcsszervezése az összes jelölt közül a leggyorsabb volt. (Az algoritmus gyengébb változatához, ahol a körök száma 7 vagy kevesebb, létezik a brute-force-nál hatékonyabb törési eljárás.)

SAFER+

A SAFER+ a SAFER (*Secure and Fast Encryption Routines*) család korábbi tagjain alapul. A kulcsméret függvényében 8, 12 vagy 16 körben végzi a feladatát. Ezenkívül az utolsó kör után még egy végső kimeneti transzformációt alkalmaz. A körfüggvények egy kulcs által meghatározott helyettesítést végeznek az adatblokk 16 bájtján, miután az egész adatblokk átesett egy invertálható transzformáción. A helyettesítés minden egyes bájtra külön-külön hajtódik végre a következő műveletekkel: kulcshozzáadás, XOR az adat és a kulcs között, rögzített keverések és azok inverzei. A megfejtés a titkosítás lépéseinek megfordításával lehetséges. A tervezők szerint már az algoritmus nyolckörös változata is ellenáll a lineáris és a differenciális kriptoanalízisnek.

SERPENT

Harminckét körrel dolgozik, plusz egy kezdő és egy befejező keverés van az adatfeldolgozásban. A körök során XOR műveletet és S-dobozokat alkalmaz. Az algoritmus ciklikusan váltogat nyolc különböző *S*-doboz között, így mindegyikre négy alkalommal kerül sor a 32 kör folyamán. A megfejtés folyamata a titkosítás lépéseinek megfordításából származtatható. Az algoritmust bemutató *Eli Biham* kiemelte, hogy a tervezés során szándékosan ragaszkodtak a már jól bevált helyettesítés-keverés elvéhez. Már több évtizedes tapasztalat halmozódott fel az ilyen elvű kódolókkal kapcsolatban mind a tervező, mind az analizáló oldalon. A SERPENT emiatt a már jól ismert DES-sel rokon lett, de a körök számát, a kulcsméretet a mai igényekhez alakították és működése is gyorsabb. (Az algoritmus gyengébb változatához, ahol a körök száma 9 vagy kevesebb, létezik a brute-force-nál hatékonyabb törési eljárás.)

TWOFISH

Egy kissé módosított Feistel-hálózat 16 körrel. A módosítás abban nyilvánul meg, hogy a XOR művelet előtt és után az adatot egy bittel elforgatja. Ez a változtatás némi aszimmetriát okoz a titkosító és a megfejtő oldalon a körkulcsok sorrendjét tekintve. A körfüggvény egyébként négy darab kulcsfüggetlen *S*-dobozzal, moduláris összeadással és mátrixműveletekkel dolgozik. Az algoritmus kulcsszervezése az összes jelölt közül a leglassabb. (Az algoritmus gyengébb változatához, ahol a körök száma 6 vagy kevesebb, létezik a brute-force-nál hatékonyabb törési eljárás.)

3.5. AZ ÚJ KIRÁLY: RIJNDAEL

Az AES-projekt 2000. október 3.-án tartott eredményhirdetésén hivatalosan is bejelentették, hogy az AES-pályázat győztese – és így a DES hivatalos utóda – a *Rijndael* lett (ejtsd körülbelül így: réjndáel). Az algoritmus keresztszülői a tervezők voltak, *Vincent Rijmen*, egyetemi kutató és *Joan Daemen*, számítógépes szakértő. (***Rij***men '***n***' ***Dae***men) Az indokolás szerint mind az öt döntős versenyző kiállta a támadásokat és a vizsgálatokat, lényeges biztonságot érintő különbség nem volt köztük, de a Rijndael nyújtotta

- a biztonság
- a teljesítmény és a hatékonyság
- és a rugalmasság legjobb kombinációját.

A Rijndael struktúrája a titkosító és a megoldó oldalon hasonló (de nem egyforma), a műveletek jól párhuzamosíthatók. A mai nagyteljesítményű RISC és CISC processzorokon éppúgy alkalmazható, mint egy 8 bites chip-kártyán. Sőt lényegében az egyetlen olyan döntős pályázó volt, amely valóban hatékony és gyors megoldást nyújtott a 8 bites processzorokon. Az algoritmus nem áll szabadalom alatt és ilyen jellegű védelmét nem is tervezik. Jelenlegi ismereteink szerint ellenáll a jelenleg ismert összes támadási módnak, így pillanatnyilag nincs jobb gyakorlati módszer a feltörésére, mint a brute-force.

3.5.1. Alapok

A Rijndael a legtöbb pályázó algoritmussal szemben nem a már megismert csereberélős Feistel-struktúrát használja, hanem a kör eredményének kialakításában az összes bemeneti adatbit részt vesz – nem csak a fele. A körfüggvényeket négy, egymástól független transzformációból építi fel. Ezeket a Rijndael terminológiában rétegeknek nevezzük (*layers*). Az egyes rétegek a következők [30]:

- A *lineáris keverőrétegek* feladata, hogy speciális P-dobozként nagyfokú keveredést biztosítsanak a körök között mászkáló adatokban. Ezekből kettő van:
  - `MixColumns` (oszlopszinten párhuzamosítható)
  - `ShiftRows` (sorszinten párhuzamosítható)
- A *nemlineáris réteg* feladata, hogy egyetlen S-doboz alkalmazásával minden lehetséges linearitást eltüntessen. Egy ilyen réteg van:
  - `SubBytes` (bájtszinten párhuzamosítható).
- A *kulcsfüggő réteg* (*key addition layer*) feladata, hogy egy egyszerű XOR művelettel kulcsfüggővé tegye az egész körművelet eredményét. Az összes többi réteg kulcsfüggetlen! Egy ilyen réteg van:
  - `AddRoundKey` (bájtszinten párhuzamosítható)

Az utolsó kör némi eltérést mutat a többihez képest, mert a keverőrétege kis mértékben eltér (nincs `MixColumns` réteg). Ez a jelenség szintén megtalálható a DES-ben, hiszen az utolsó kör eredményének jobb és bal oldalát ott „visszacseréljük”, mintha az utolsó kör meg sem cserélte volna azt, vagyis ott szintén hiányzik egy lépés.

3.5.2. Az algoritmus specifikációja

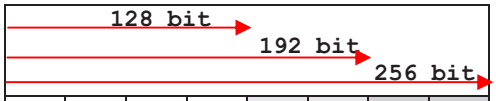
Paraméterek: körök száma, adatblokk- és kulcsméret

A Rijndael algoritmus blokkmérete és kulcsmérete egymástól függetlenül paraméterezhető: 128-tól 256 bitig terjedhet mindkettő, 32 bites lépésekben. A FIPS197 szabvány azonban csak a 128 bites blokkméretet, és a 128, 192, 256 bites kulcsméretet fogadta be, más kulcs- és blokkméret használata így nem szabványos.

- Az algoritmus gyakran használt mértékegysége a „szó”, mely 32 bitet jelent.
- A blokkméretet N_B képviseli, melynek pontos értéke N_B=*blokkméret/32*, vagyis a blokkméret szavakban mérve. Egyetlen szabványos értéke van: N_B=*4*.
- A kulcsméretet N_K képviseli, melynek pontos értéke N_K=*kulcsméret/32*, vagyis a kulcsméret szavakban mérve. Három szabványos értéke van: N_K = *4,6,8*. A szabványos algoritmus megnevezése a kulcsméret függvényében AES-128, AES-192, AES-256 lehet.
- A körök számát N_R-rel jelöljük, értéke a blokk- és kulcsmérettől egyaránt függ. (AES esetén csak a kulcsmérettől, hiszen csak egy blokkméret van.)

A State változó

Az egyes körök (ezen belül az egyes rétegek) ki- és bemeneti adatait egy *State* nevű struktúrában tároljuk. Ábrázolására a specifikáció egy négyzethálót ajánl, amelynek mindig négy sora van (1 szó magas), oszlopainak száma a blokkmérettől függ: N_B. (Vagyis szabványos AES esetén a state-struktúra egy 4x4-es négyzet.) Egy-egy cella egy-egy bájtot jelent. A kulcs ábrázolása hasonló, a sorok száma szintén négy, az oszlopok száma pedig N_K .



A_{00}	A_{01}	A_{02}	A_{03}	A_{04}	A_{05}	A_{06}	A_{07}
A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}
A_{20}	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}
A_{30}	A_{31}	A_{32}	A_{33}	A_{34}	A_{35}	A_{36}	A_{37}

A state-struktúra feltöltésekor mind a kulcs, mind a titkosítandó adat esetén fentről lefelé és balról jobbra kell haladni: A_{00}, A_{10}, A_{20}, A_{30}, A_{01}, A_{11}, A_{21}, ... Legyen *i* az aktuális adatbájt indexe az adatfolyamban, ekkor *row:=i and 3* és *col:=i >> 2* számításokkal megkapjuk az adott bájt sorát (row) és oszlopát (col): $A_{row,col}$ = *a[row,col] = bytestream[i]*.

A körfüggvény rétegei: SubBytes, ShiftRows, MixColumns, AddRoundKey

A Rijndael az iteratív kódolók családjába tartozik, ami a közbülső részeredmények többszöri újrafeldolgozását jelenti. A DES 16-szor ismételte körfüggvényét, a Rijndael 10-12-14-szer teszi ezt a kulcs- és blokkméret függvényében:

N_R	**N_B=4**	N_B=6	N_B=8
N_K=4	**10** (AES-128)	12	14
N_K=6	**12** (AES-192)	12	14
N_K=8	**14** (AES-256)	14	14

A Rijndael körfüggvénye (*round transformation*) a már korábban is említett négy rétegből áll. Minden körfüggvény egyforma, kivéve az utolsót, amely egy kicsit eltér a többitől, mert kimaradt a `MixColumns` lépés:

```
Round(State, Roundkey)
begin
  SubBytes(State);
  ShiftRows(State);
  MixColumns(State);
  AddRoundKey(State, RoundKey);
end;
```

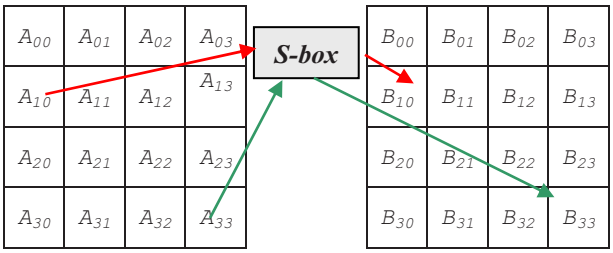
```
FinalRound(State, Roundkey)
begin
  SubBytes(State);
  ShiftRows(State);

  AddRoundKey(State, RoundKey);
end;
```

25. ábra Körfüggvények titkosítómódban

A „SubBytes” transzformáció – bájtszintű művelet

A `SubBytes` transzformáció nemlineáris, invertálható S-dobozt alkalmaz, pontosan egyet: minden egyes bájt helyettesítése ugyanazzal az S-dobozzal történik.



Ez az S-box – működés szempontjából – egyenértékű az ősi Caesar-titkosítással, de az egykori +3 helyett a következő összefüggés határozza meg az S-doboz kimeneti bitjeit (a bitek számozása a szokásos, tehát jobbról-balra történik, 0-7):

$$O_i = B_i \oplus B_{(i+4) \bmod 8} \oplus B_{(i+5) \bmod 8} B_{(i+6) \bmod 8} \oplus B_{(i+7) \bmod 8} \oplus C_i$$

$C = 01100011$, és $0 \leq i \leq 7$

A bonyultnak tűnő definíció ellenére az S-dobozoknál megszokott szótárszerű működést kapjuk: ha például az A_{00} értéke 74, az S-doboz eredménye 92, és ez csak a bemeneti értéktől függ. Szerencsére ezt mind előre ki lehet számolni, mivel ez erősen nem „bájtbarát” művelet, ezért ajánlott is. Maga a szabvány is teljes egészében közli a 256 bájt tárigényű táblázatot (lásd 1. Táblázat, jelen alfejezet végén).

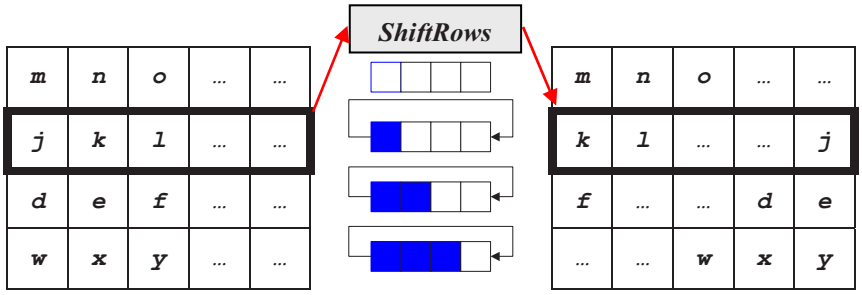
Az „InvSubBytes” transzformáció – bájtszintű művelet

Az `InvSubBytes` transzformáció szintén egy S-dobozt használ, a `SubBytes` S-dobozának inverzét. Előre kiszámolt értékeit a 2. Táblázat tartalmazza. Az előbbi számpéldát alapul véve ha az A_{00} értéke 92, az inverz S-doboz eredménye 74. (Szintén Caesar tiszteletének jeleként...)

A „ShiftRows” transzformáció – sorszintű művelet

Ez az egyik legegyszerűbb rétegfüggvény, a state-struktúra *sorait* forgatja el különböző mértékben. Az első sort nem forgatja, a többi forgatásának mértéke pedig az adatblokk méretétől függ, és előre definiált. A specifikáció idevonatkozó értékei következők:

N_B	C_1	C_2	C_3
(AES) 4	**1**	**2**	**3**
6	1	2	3
8	1	3	4

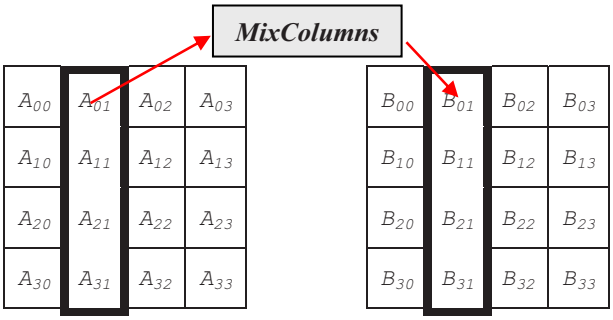


A „InvShiftRows” transzformáció – sorszintű művelet

A specifikáció szerint a `ShiftRows` inverz művelete ugyanez a balraforgatás, de a forgatások lépésszámai a blokkméret függvényében rendre: `0, `N_B-C_1`, `N_B-C_2`, `N_B-C_3 pozíció.

A „MixColumns” transzformáció – oszlopszintű művelet

A `MixColumns` a legbonyolultabb rétegfüggvény és – áttekintve a szabványon – szinte csak ennek a kedvéért érdemes megismerkedni egy új fogalommal, a polinommal és a rajta értelmezett speciális • gombóc-szorzás művelettel. A `MixColumns` a state-struktúra *oszlopait* alakítja át. Az oszlop minden egyes bájtját, mint polinomot megszorozza egy-egy másik, előre definiált polinommal, majd ezek összege adja az új oszlop egy-egy bájtját. Az új bájt kialakításában az oszlop minden bájtja részt vesz, így ha az oszlop 32 bitjének egyike megváltozik, megváltozik az egész oszlop: vagyis ez a művelet nagyfokú függőséget teremt az egyes bájtok között.



$B_{0c} = (\{02\} \bullet A_{0c}) \oplus (\{03\} \bullet A_{1c}) \oplus A_{2c} \oplus A_{3c}$
$B_{1c} = A_{0c} \oplus (\{02\} \bullet A_{1c}) \oplus (\{03\} \bullet A_{2c}) \oplus A_{3c}$
$B_{2c} = A_{0c} \oplus A_{1c} \oplus (\{02\} \bullet A_{2c}) \oplus (\{03\} \bullet A_{3c})$
$B_{3c} = (\{03\} \bullet A_{0c}) \oplus A_{1c} \oplus A_{2c} \oplus (\{02\} \bullet A_{3c})$

Kriptográfiai kifejezésekkel azt mondhatjuk, hogy olyan S-dobozokat alkalmaz, melyek bemenete és kimenete egyaránt 4-bájtos. A bájtok „képlete" elég bonyolultnak tűnik, de barátságossá tehető. Lássuk az B_{0c} kiszámolását szétbontva, az **xtime**() használatával!

$B_{0c} = (\{02\} \bullet A_{0c}) \oplus (\{03\} \bullet A_{1c}) \oplus A_{2c} \oplus A_{3c} =$
$= \text{xtime}(A_{0c}) \oplus (\text{xtime}(A_{1c}) \oplus A_{1c}) \oplus A_{2c} \oplus A_{3c}$

Aki kíváncsi arra, hogy pontosan mit is jelent ez a fekete gombóc, és mennyire egyszerű az **xtime**(), az ne lapozzon el, hanem folytassa itt az olvasást. Azok a gyengébb idegzetűek, akik minderre nem kíváncsiak, lapozzanak nyugodtan 3 oldalnyit, az `Addroundkey` transzformációig. (Ezzel ugyan kihagyják az `InvMixColumns` leírását is, de az pontosan ugyanúgy néz ki, mint az iménti `MixColumns`, csak mások az együtthatók.)

A polinomok ideát vannak

A Rijndael algoritmus lényegében bájtorientált, mert az elemi lépéseket bájtokon hajtja végre. Ezek egy része ugyan (alternatív algoritmus kialakításával) optimalizálható például 32-bites processzorokra, de ettől még az alapelv nem változik. Az algoritmus legfurcsább művelete a `MixColums` és párja az `InvMixColumns`, ezek megértését segítendő vezessük be a következő speciális módszert a bájtok értelmezésére.

Egy B bájthoz, melynek bitjei rendre a $B_7B_6B_5B_4B_3B_2B_1B_0$ bitek, rendeljük az alábbi polinomot! Tekintsünk úgy erre a bájtra, mintha az nem lenne más, mint az alábbi polinom rövid, tömör felírási módja![31]

$$b(x) = B_7x^7 + B_6x^6 + B_5x^5 + B_4x^4 + B_3x^3 + B_2x^2 + B_1x^1 + B_0x^0 \qquad (1.)$$

[31] A műveletek alapja igazából a véges testek – most éppen GF(2^8) – feletti polinomalgebra. Arra, hogy ez a mondat mit is jelent pontosan, nem térek ki, mert teljes értékű implementáció készíthető enélkül is. Ha valakit mégis érdekelnek a részletek, akkor [30,57]-ben vagy szinte bármelyik „Algebra" című főiskolai, egyetemi jegyzetben megtalálhatja azokat.

Mivel minden B_i együttható értéke vagy „0” vagy „1” általában nem is kell külön kiírni őket. A polinomszerű írásmód nehézkes, ezért egyszerűsítésképp bevezethető egy hexadecimális számokkal történő jelölés, amely a B_i együtthatókon alapul. Néhány példa:

A polinom	$B_7B_6B_5B_4\,B_3B_2B_1B_0$ együtthatók	A polinom rövidített írásmódja
X^0	{0000 0001}	{01}
X^1	{0000 0010}	{02}
$x^7 + x^4 + x^3 + x^1$	{1001 1010}	{9A}
$x^6 + x^4 + x^2 + x^1 + x^0$	{0101 0111}	{57}
$x^6 + x^5 + x^4 + x^2$	{0111 0100}	{74}

A jelölésmód elsőre is szembetűnő jó tulajdonsága, hogy egy-egy ilyen polinom egyetlen bájton ábrázolható. A polinom együtthatói, a bájtok bitjei és a hexa írásmód között egyértelmű megfeleltetés lehetséges, hiszen a polinom rövidített írásmódja nem más, mint a bájt hexadecimálisan felírt értéke. Így szépen visszakanyarodtunk bájtokhoz. Az algoritmusban definiált műveletek nagy része bájtszintű, így ez az ábrázolásmód kiválóan illeszkedik majd a műveletekhez, akár bájtról, akár polinomról beszélünk éppen.

Műveletek a polinomokkal

Összeadás

Két polinom összege a megfelelő helyértéken lévő együtthatók mod 2 összege, vagyis XOR kapcsolata. Ezt leírni bonyolultabb, mint kiszámolni… Mennyi *c(x)=a(x)+b(x)*, ha

$a(x) = x^7+x^6+\quad\quad x^2+x^1$
$b(x) = \quad x^6+x^5+x^4+x^3+x^2+x^1+x^0$?
$c(x) = x^7+\quad x^5+x^4+x^3+\quad\quad +x^0 \qquad C_i = A_i \oplus B_i$

És mindez a hexadecimális jelöléssel:

$a(x) = \{11000110\} = \{C6\}$
$b(x) = \{01111111\} = \{7F\}$
$c(x) = \{10111001\} = \{B9\} \qquad C_i = A_i \oplus B_i$

Mivel a bitszintű műveleteket bájtonként is elvégezhetjük, egyszerűen számolhatunk így is:

$C = A \oplus B = C6 \oplus 7F = B9$

Szorzás

Még egy művelettel meg kell ismerkedni, ez pedig a polinomok szorzása, amely jelen esetben egy kicsit a modulo aritmetikához hasonlítható. (Ha elfelejtjük a polinomokat és programozói szemmel csak bájtoknak látjuk a bájtokat, akkor tekintsük ezt egyszerűen egy speciális műveletnek, amit az alább leírt módszer szerint kell végrehajtani!) Első lépésként vegyünk egy speciális polinomot: $m(x) = \{01\}\{1B\}$, vagyis $m(x) = x^8+x^4+x^3+x^1+x^0$. E polinomnak saját magán és 1-en kívül nincs más osztója. Olyan, mintha prímszám lenne. (Megjegyzem a

$011B_{16}=283_{10}$ valóban prímszám, de a kettőnek semmi köze egymáshoz.) Sok ilyen polinom van, de a Rijndael pont ezt használja. A Rijndaelben használt szorzás művelete (melynek jelölése a továbbiakban •) a polinomok szorzatának fenti m(x)-szel történő osztási maradékát adja eredményül: *c(x)=a(x) • b(x) mod m(x)*

Sajnálatos módon két tetszőleges polinom szorzatának kiszámítására nincs egyszerű bájtszintű műveletsor, bár nem is lesz rá szükség, mert a Rijndael fejlesztői feltalálták magukat. Az algoritmus működése során csupán néhányszor végez ilyen műveletet, és akkor is csak konstans polinomokkal szoroz, tehát nem kell el egy általános szorzóalgoritmust leprogramozni, csak egy speciálisat…

Szorzás egy bizonyos polinommal - xtime()

Ha az alábbi általános polinomot megszorozzuk az $x^1 = \{02\}$ polinommal, az eredmény:

$$(B_7x^7+B_6x^6+B_5x^5+B_4x^4+B_3x^3+B_2x^2+B_1x^1+B_0x^0)\bullet x^1 =$$
$$B_7x^8+B_6x^7+B_5x^6+B_4x^5+B_3x^4+B_2x^3+B_1x^2+B_0x^1$$

Ne feledkezzünk meg arról, hogy a • művelet előírja az m(x) modulusképzést is! Mit jelent ez a gyakorlatban? Ha $B_7=0$, akkor semmi teendő nincs, hiszen az eredmény biztosan kisebb, mint m(x), így a maradékképzés úgysem változtatna az eredményen. Ha viszont $B_7=1$, az m(x) polinomot egyszerűen vonjuk ki a szorzatból! Mivel a ⊕ művelet önmaga inverze, a kivonást ugyanúgy kell elvégezni, mint az összeadást: a szorzatot XORoljuk össze m(x)-szel. A fenti eredményből láthatjuk még, hogy minden együttható egy hellyel balra vándorolt. Vagyis hogyan is kell egy tetszőleges b(x) polinomot egy ilyen {02} polinommal megszorozni? Egyszerűen a b(x) „ábrázolását" egy hellyel balra toljuk, és ha a bájtból kilépő bit „1", az eltolást még egy ⊕{011B} is követi. Ezt a műveletet a Rijndael specifikációja **xtime**()-nak hívja, megvalósítható függvényként is, de akár előre ki is számolható egy táblázatba az összes **xtime**(0-255) érték. A szabvány a számítás egyszerűsége miatt nem ajánlja a táblázatos előkészítést, de nem is tiltja meg. Sajnálatos módon ismert olyan támadási módszer, ami azt használja ki, hogy a függvényként megvalósított **xtime()** végrehajtási ideje attól függően változik, hogy végrehajtjuk-e a XOR műveletet vagy sem. Ezért a gyakorlatban érdemes egy táblázatba előre kiszámolni a lehetséges függvényértékeket, így a végrehajtási idő állandó marad [67].

```
function xtime(b:byte):byte;
begin
  if b > 127 then xtime:=(x shl 1) xor $011b else xtime:=(x shl 1);
end;
```

Egy polinomot {02} magasabb hatványaival az **xtime**() művelet ismételt alkalmazásával lehet megszorozni. Íme egy példa:

{57}•{01} = {57}

{57}•{02} = **xtime**({57}) = {ae}

{57}•{04} = **xtime**({ae}) = 15c ⊕ 11b = {47}

{57}•{08} = **xtime**({47}) = 8e

{57}•{10} = **xtime**({8e}) = 11c ⊕ 11b = {07}

És ez nekünk nagyon, nagyon, nagyon jó, mert e részeredmények összegéből azt is ki tudjuk számolni, mennyi például {57} • {13}?

```
{57}•{13} = {57}•{ {10} ⊕ {02} ⊕ {01} } =
          {57}•{10} ⊕ {57}•{02} ⊕ {57}•{01} = {07} ⊕ {ae} ⊕ {57} = {fe}
```

A „InvMixColumns" transzformáció – oszlopszintű művelet

A polinomok terén tett kis kitérő után térjünk vissza az AES rétegeire. Legutóbb a `MixColumns`-ot láttuk, és ennek inverze az `InvMixColumns`. A két réteg között csak az előre definiált szorzó polinomok jelentik a különbséget, az elvégzendő művelet mindkét esetben azonos:

$B_{0c} = (\{0e\}\bullet A_{0c}) \oplus (\{0b\}\bullet A_{1c}) \oplus (\{0d\}\bullet A_{2c}) \oplus (\{09\}\bullet A_{3c})$
$B_{1c} = (\{09\}\bullet A_{0c}) \oplus (\{0e\}\bullet A_{1c}) \oplus (\{0b\}\bullet A_{2c}) \oplus (\{0d\}\bullet A_{3c})$
$B_{2c} = (\{0d\}\bullet A_{0c}) \oplus (\{09\}\bullet A_{1c}) \oplus (\{0e\}\bullet A_{2c}) \oplus (\{0b\}\bullet A_{3c})$
$B_{3c} = (\{0b\}\bullet A_{0c}) \oplus (\{0d\}\bullet A_{1c}) \oplus (\{09\}\bullet A_{2c}) \oplus (\{0e\}\bullet A_{3c})$

Szedjük szét most is az B_{0c} kiszámolását, az **xtime**() használatával! Ehhez előbb bontsuk fel az előforduló polinomokat {02} hatványösszegére:

```
{09}={08}⊕{01}   {0b}={08}⊕{02}⊕{01}   {0d}={08}⊕{04}⊕{01}   {0e}={08}⊕{04}⊕{02}
```

$B_{0c}=(\{0e\}\bullet A_{0c})\oplus$ $(\{0b\}\bullet A_{1c})\oplus$ $(\{0d\}\bullet A_{2c})\oplus$ $(\{09\}\bullet A_{3c})$	$xtime(xtime(xtime(A_{0c})))\oplus xtime(xtime(A_{0c}))\oplus xtime(A_{0c})\oplus$ $xtime(xtime(xtime(A_{1c})))\oplus xtime(A_{1c}) \oplus A_{1c} \oplus$ $xtime(xtime(xtime(A_{2c})))\oplus xtime(xtime(A_{2c}))\oplus A_{2c} \oplus$ $xtime(xtime(xtime(A_{3c})))\oplus A_{3c}$

Hát ez elég ronda és hosszú kígyó lett... A `MixColumns` és az `InvMixColumns` transzformációkban összesen 7 különböző polinommal kell szorozni, így talán érdemes egy szorzó függvényt írni. Például az alábbi is ilyen:

```
function mpl(x, y: byte):byte; // Eredmény:{x}•{y}
begin
  a:=y;
  r:=0;
  while x>0 do begin
     if x and 1 > 0 then r:=r xor a;
     a:=xtime(a);
     x:=x shr 1;   // shift right
  end; // while
  mpl:=r;
end; // function
```

E függvény alkalmazása az iménti borzalmas **xtime**() kígyót az alábbira egyszerűsíti:

```
B0c = mpl($0e,a0c) xor mpl($0b,a1c) xor mpl($0d,a2c) xor mpl($09,a3c);
```

Az „AddRoundKey” transzformáció – bájtszintű transzformáció

Ez a legegyszerűbb művelet. Feladata, hogy mindazt, ami eddig történt a state-struktúrában, kulcsfüggővé tegye. A bemeneti state-struktúra bájtjait egyszerűen összeadja (xorolja) a körkulcsot tartalmazó state-struktúra bájtjaival:

$B_{00} = A_{00} \oplus R_{00}$ $\quad$ $B_{01} = A_{01} \oplus R_{01}$, stb.

A_{00}	A_{01}	A_{02}	A_{03}		R_{00}	R_{01}	R_{02}	R_{03}		B_{00}	B_{01}	B_{02}	B_{03}
A_{10}	A_{11}	A_{12}	…	$\oplus$	R_{10}	R_{11}	R_{12}	…	=	B_{10}	B_{11}	B_{12}	…
…	…	…	…		…	…	…	…		…	…	…	…
…	…	…	…		…	…	…	…		…	…	…	…

Kulcsszervezés – a körkulcsok előállítása, a kiterjesztett kulcs születése

A titkos kulcsból egy igen hosszú, ún. kiterjesztett kulcsot készít az algoritmus. Ennek feldarabolása szolgáltatja majd a körkulcsokat. Mekkora ez a kiterjesztett kulcs?

- Az `AddRoundKey` rétegben láttuk, hogy körkulcs N_B szót tartalmaz.
- Mivel N_R*-1* darab rendes `Round`, 1 darab `FinalRound`, és egy kezdeti `AddRoundKey` van, ez összesen (N_R*+1*) darab körkulcsot jelent.
- Ha mindegyik N_B szó hosszúságú, a kiterjesztett kulcs hossza pontosan $N_B(N_R+1)$ szó.

Az egyes körökben felhasznált körkulcsokat ebből a kiterjesztett kulcsból származtatja a Rijndael: az első N_B szót a 0. körhöz, a második N_B szót az 1. körhöz használja fel, és így tovább. A kiterjesztett kulcs jelölése a továbbiakban *EXP[i]*.

A kiterjesztett kulcs elejére a titkos kulcs másolatát helyezik, majd minden további szó rekurzív módon a korábbi szavakból származtatható. Bár a kiterjesztett kulcsot mindig a titkos kulcsból származtatjuk, a szabvány nem ajánlja (sőt, majdnem tiltja), hogy azt közvetlenül specifikáljuk – valószínűleg azért, mert nemcsak a kulcstól, hanem a körök számától és ablokkmérettől is függ. Mint később látni fogjuk, a kiterjesztett kulcsot nem szükséges teljes egészében előre kiszámolni, ezért az olyan alkalmazásokban, amelyek szűkös a memóriával bírnak, akár menet közben is számolható. A kiterjesztett kulcs előállítására két – hasonló – algoritmus van attól függően, hogy mekkora a titkos kulcs hossza. Szerencsére a két algoritmus egy feltételes ággal összegyúrható és a következő programkód már ezt az egyesített kulcskiterjesztést mutatja.

Rijndael kulcsszervezés – Szóalapú változat

Bemenet: `Key`: a titkos kulcs, $4*N_K$ darab bájtban

Kimenet: `Exp`: kiterjesztett kulcs, $N_B*(N_R+1)$ darab szóban

```
Rijndael KeyExpansion(Key, Exp)
begin
  // Titkos kulcs bemásolása a kiterjesztett kulcs elejére
  for i:= 0 to Nk-1 do
    EXP[i] := (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
  // A következő szavak előállítása az előzőek alapján
  for i:=Nk to Nb*(Nr + 1)-1 do
  begin
    temp := EXP[i-1];

    if (i mod Nk = 0) then
      temp := SubWord(RotWord(temp)) xor Rcon[i div Nk]
    else if (Nk > 6) and (i mod Nk = 4) then
            temp := SubWord(temp);
    EXP[i] = EXP[i-Nk] xor temp;
  end; // for
end; // Rijndael KeyExpansion
```

A fenti algoritmusban két újabb függvényt láthatunk: a `SubWord()` függvény bemenete és eredménye egyaránt egy 4 bájtos szó, a bemeneti négy bájt mindegyikére a `SubBytes` S-doboza kerül alkalmazásra. A `RotWord()` függvény eredménye szintén 4 bájtos szó, és a bemenetére adott `(a,b,c,d)` bájtsorrendű szót `(b,c,d,a)` sorrendűre alakítja.

A kiterjesztett kulcs első N_K szava a titkos kulcsot tartalmazza. Minden további *EXP[i]* szó, az egyel korábbi és az N_K-val korábbi szavak között végzett XOR művelet eredménye. Azoknál a szavaknál, melyek N_K egész számú többszörösének megfelelő pozícióban helyezkednek el ($i \bmod N_K = 0$), ott az algoritmus a XOR művelet előtt a EXP[i-1] szót a `SubWord` és `RotWord` függvények, és az adott körhöz tartozó konstans (`Rcon`) alkalmazásával átalakítja. 192 bitnél hosszabb kulcs esetén, ha *i-4* az N_K egész számú többszöröse *(*$i \bmod N_K = 4$*)*, akkor az algoritmus a XOR művelet előtt EXP[i-1] szót egy `SubWord`-del átalakítja.

```
  RC : array[1..30] of word32 =  // Indexelés 1-től !!!
    ( 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
      0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
      0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91 );
```

Érdekes módon az egész Rijndael algoritmus tervezése során szem előtt tartották a bájtorientáltságot, de a kulcsszervezésnél mintha elfelejtették volna. Az AES minden művelete egyszerűen elvégezhető bájtokkal bűvészkedve, de a kulcsszervezésről a szabvány is 32 bites szavakban beszél. Szerencsére ez a mozzanat is átszervezhető – újabb példa az AES rugalmasságára – ezért egy 8-bites processzoron érdemes az alábbi bájtalapú kulcsszervezést használni:

Rijndael kulcsszervezés – bájtalapú változat (a szerző átirata)

Bemenet: `Key`: a titkos kulcs, $4*N_K$ darab bájtban

Kimenet: `Exp`: kiterjesztett kulcs, $4*N_B*(N_R+1)$ darab bájtban

```
Rijndael KeyExpansion(Key, Exp)
begin
  for i:=0 to (NK-1)*4+3 do Exp[i]:=Key[i];
  for i:=NK to NB*(NR+1) do
  begin
    tmp0:=Exp[ (i-1)*4 + 0 ];
    tmp1:=Exp[ (i-1)*4 + 1 ];
    tmp2:=Exp[ (i-1)*4 + 2 ];
    tmp3:=Exp[ (i-1)*4 + 3 ];
    if (i mod NK)=0 then begin
      x:=tmp0;
      tmp0:=RCON[i div NK] xor SBOX[tmp1];
      tmp1:=SBOX[tmp2];
      tmp2:=SBOX[tmp3];
      tmp3:=SBOX[ x  ];
    end
    else if (nk > 6) and (i mod NK = 4) then begin
      tmp0:=SBOX[tmp0];
      tmp1:=SBOX[tmp1];
      tmp2:=SBOX[tmp2];
      tmp3:=SBOX[tmp3];
    end;
    Exp[ i*4 + 0 ] := Exp[ (i-NK)*4 + 0 ] xor tmp0;
    Exp[ i*4 + 1 ] := Exp[ (i-NK)*4 + 1 ] xor tmp1;
    Exp[ i*4 + 2 ] := Exp[ (i-NK)*4 + 2 ] xor tmp2;
    Exp[ i*4 + 3 ] := Exp[ (i-NK)*4 + 3 ] xor tmp3;
  end;
end;
```

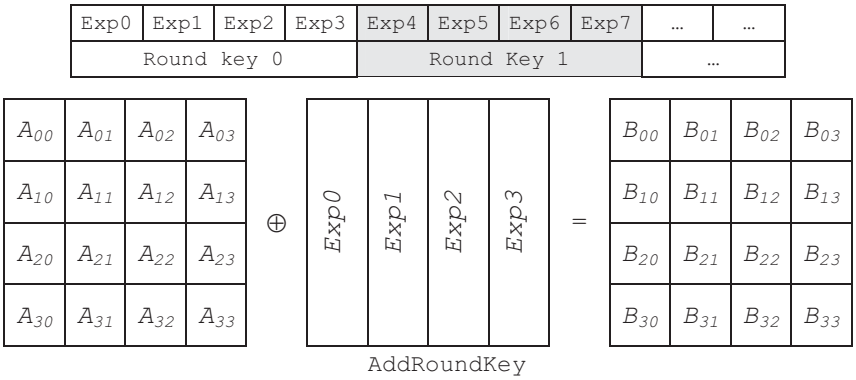
A fenti, elvi kérdéstől eltekintve elmondható, hogy a Rijndael / AES egy igazi alakváltó. Nincs még egy olyan algoritmus, amelynek ennyiféle megvalósításával találkoztam volna. Az egyik véglet, amikor a szabványt szó szerint leprogramozzuk. A másik, amikor mindent-mindennel összevonunk, táblázatokban keresgélünk, stb. Ez utóbbi megoldás egyébként – amely mintegy 4 kilobájtnyi tárterületet igényel –a leggyorsabb működést eredményezi, bár a megvalósított kódból eltűnik az algoritmus lényege: 32-bites platformon csak 16 XOR és 16 table-lookup marad körönként, 8 bites processzornál kicsit rosszabb a helyzet, mert 48 XOR és 48 lookup marad, de ott van más optimalizálási lehetőség is, melyekre egyébként [55]-ben kaphatjuk a legtöbb útmutatást. (Egy kis kuriózum: egy 472 bájtos implementáció x86 asm-re található az [URL68]-on.)

Körkulcs kiválasztása – szóalapú kiterjesztett kulcsból

A körkulcs (*round key, RK*) kiválasztása a kiterjesztett kulcsból a blokkmérettől függően történik. Minden egyes körkulcs N_B darab szóból áll, vagyis a state-struktúrával egyező méretű darabokra kell vágni a kiterjesztett kulcsot:

- Az első körkulcs az első N_B darab szó,
- a második körkulcs a második N_B darab szó és így tovább.

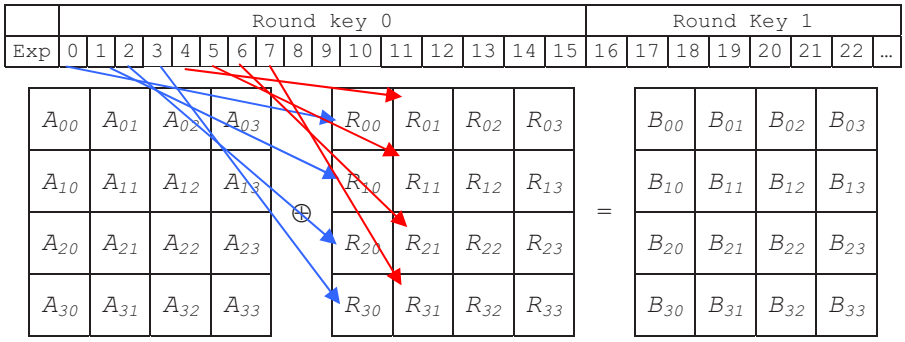
Egy körkulcsban az első szó az első state-oszlophoz, a második szó a második state-oszlophoz tartozik, és így tovább. Az alábbi ábra ezt a kiválasztást és a körkulcsként való felhasználást mutatja $N_B=4$ (128 bites adatblokk) esetén:



Körkulcs kiválasztása – bájtalapú kiterjesztett kulcsból

A bájtokból felépülő kiterjesztett kulcsból – 32-bites szavak felhasználása nélkül – megfelelő indexelést használva vehetjük ki a bájtokat:

```
AddRoundKey(r : integer);
begin
  for col:=0 to NB-1 do
    for row:=0 to 3 do
      State[row, col]:=State[row, col] xor Exp[ r*NB*4 + col*4 + row ];
end;
```



Ezzel végére értünk az AES alapműveleti ismertetésének, nekiláthatunk a titkosító és a megfejtő algoritmus felépítésének.

3.5.3. A titkosítás

A megismert elemekből most már felépíthetjük magát a Rijndael titkosító algoritmust. Alkalmazzuk az egyes lépéseket a következők szerint:

- Ha új kulcsot állít be a felhasználó, rögtön készítünk egy kiterjesztett kulcsot.
- A Rijndael kezdő lépése: `AddRoundKey`
- N_R-1 körben a `Round` függvény alkalmazása
- majd a `FinalRound` függvény alkalmazása

```
Rijndael(Data)
begin
   State:=Data;
   AddRoundKey(State, 0);     // 0. körkulcs és a kezdő lépés
   for i:=1 to NR-1 do
   begin                      // Körfüggvény
      SubBytes(State);
      ShiftRows(State);
      MixColumns(State);
      AddRoundKey(State, i);  // i. körkulcs alkalmazása
   end;
   SubBytes(State);           // Utolsó körfüggvény három rétege következik
   ShiftRows(State);
                              // hiányzó MixColumns réteg!!!
   AddRoundKey(State, NR);    // utolsó körkulcs
   Data:=State;
end;
```

3.5.4. Az inverz művelet

A megoldóalgoritmus az eredeti lépések inverzeit használja, az egyes lépéseket fordított sorrendben alkalmazza, a körkulcsok fordított sorrendben kerülnek felhasználásra. A két algoritmus ezért nem egyforma, sőt a körfüggvény egyes lépései is különbözőek, mert azok – az `AddRoundKey` kivételével – nem saját maguk inverzei.

```
InvRound(State, Roundkey)
begin
   InvShiftRows(State);
   InvSubBytes(State);
   AddRoundKey(State, RoundKey);
   InvMixColumns(State);
end;
```

```
InvFinalRound(State, Roundkey)
begin
  InvShiftRows(State);
  InvSubBytes(State);

  AddRoundKey(State, RoundKey);
end;
```

26. ábra Körfüggvények megfejtő módban

Akinek az az első gondolata, hogy ez a műveleti sorrend nem az eredeti körfüggvény műveleteinek visszafelé játszása (hiszen az `sub-shift-mix-add` és `sub-shift-add` volt, ez pedig `shift-sub-add-mix` és `shift-sub-add` lett), az ugyanabba a hibába esett, mint én először... A félreértések elkerülése végett vegyünk egy háromkörös Rijndael variánst, titkosítómódban!

Írjuk fel a lépéseit körönként kifejtve!

```
   AddRoundKey(State, 0);     // Ez a lépés fix, nem tartozik a körhöz.

   SubBytes(State);           // 1. kör
   ShiftRows(State);
   MixColumns(State);
   AddRoundKey(State, 1);

   SubBytes(State);           // 2. kör
   ShiftRows(State);
   MixColumns(State);
   AddRoundKey(State, 2);

   SubBytes(State);           // Utolsó kör
   ShiftRows(State);
   AddRoundKey(State, 3);
```

És ugyanezek a lépések visszafelé, megfejtő módban:

```
   AddRoundKey(State, 3);     // Ez a lépés fix, nem tartozik a körhöz.

   ShiftRows(State);          // 1. kör
   SubBytes(State);
   AddRoundKey(State, 2);
   MixColumns(State);

   ShiftRows(State);          // 2. kör
   SubBytes(State);
   AddRoundKey(State, 1);
   MixColumns(State);

   ShiftRows(State);          // Utolsó kör
   SubBytes(State);
   AddRoundKey(State, 0);
```

Hát ezért vannak az inverz körök lépései olyan „össze-vissza”! Nem egyszerűen a körfüggvények lépéseit kell megfordítani, hanem az egész adatfeldolgozást: a kezdeti `AddRoundKey` réteggel, kulcssorrenddel, műveleti sorrenddel, mindenestül! Az inverz rétegekből és körökből felépülő inverz kódoló tehát a következőképpen néz ki:

```
InvRijndael(Data)
begin
  State:=Data;
  AddRoundKey(State, NR);     // Utolsó körkulcs
  for i:=NR-1 downto 1 do     // A körkulcsokat is visszafele vesszük!
  begin                       // Inverz körfüggvény
    InvShiftRows(State);
    InvSubBytes(State);
    AddRoundKey(State, i);
    InvMixColumns(State);
  end;
  InvShiftRows(State);        // Utolsó körfüggvény három rétege következik
  InvSubBytes(State);
                              // hiányzó MixColumns réteg!!!
  AddRoundKey(State, 0);
  Data:=State;
end;
```

Ennyi a Rijndael. Nem több, nem kevesebb. Ha valaki további irodalmat böngész, vegye figyelembe azt az apró eltérést, hogy a korai cikkekben másként hívják a rétegfüggvényeket:

Rijndael eredeti elnevezései	FIPS197, végleges elnevezések
ByteSub	SubBytes
ShiftRow	ShiftRows
MixColumn	MixColumns
AddRoundKey	AddRoundKey

A NIST az eredményhirdetést követően egy évvel később, 2001. november 26.-án bocsátotta ki a FIPS197 szabványt [57], mely hat hónappal később, 2002. május 26.-án lépett életbe. Az algoritmust a DES-hez hasonló 5 évenkénti felülvizsgálat igyekszik védeni az idő ellen. Jelenleg az AES-ről szóló cikkek, tanulmányok száma nem túl nagy, legalábbis a DES-hez képest szinte elenyésző, amely főként az algoritmus fiatal korának tudható be. Ma legtöbben a különböző hardveres és szoftveres megvalósításokkal foglalkoznak, hiszen az algoritmus felépítése igen rugalmasan viszonyul a különböző optimalizálásokhoz, alternatív megvalósításokhoz.

3.5.5. Máris AES-törés?

A fejezet elején azt írtam, hogy „így pillanatnyilag nincs jobb gyakorlati módszer a feltörésére, mint a brute-force”. Hát, ez így ebben a formában igaz is, meg nem is. És elég kényes kérdésnek tűnik a szabványosítás után alig egy évvel, de rendkívül fontos. Az AES-projekt ideje alatt a nevezett algoritmusok fejlesztői a többiek algoritmusaival foglalkoztak. Mindenki azt próbálta bizonyítani, hogy a saját algoritmusa miért jobb, mint a másik. Ma már más a helyzet, hiszen csak egy algoritmust kell támadni, ezért az erre fordított erőforrás is sokkal nagyobb, mint mondjuk két-négy évvel ezelőtt, az AES-projekt közepén. 2002 végén *Nicolas Courtois* (Franciaország) és *Josef Pieprzyk* (Ausztrália), publikáltak egy cikket [68], amely igen nagy port kavart és tartalmával jelentősen megosztja a neves kriptográfusokat is. Az XSL-nek nevezett eljárás lényege, hogy az AES-t egy többezer tagot számláló egyenletrendszerre konvertálja, többezer ismeretlennel. Az AES-128 feltörésének komplexitása 2^{100} körüli, szemben a nyers erő 2^{128}-jával. Az eljárás (egyesek szerint csak elgondolás) fő ereje abban rejlik, hogy a feltörés komplexitása nem exponenciálisan arányos a kulcshosszal: vagyis az AES-256 hasonló módon történő megtörésének nem $2^{100+128}$ az erőforrásigénye, hanem jóval kevesebb: kb. háromszoros. Szerencsére mindkettő jelentősen túlmutat a gyakorlati lehetőségeken (így egyelőre nem kell félteni az AES-t) és a DES gyengeségén (a DES leváltása nem egy még rosszabb algoritmussal történt). Sajnos ezek a számok még megválaszolandó kérdésként állnak a kutatók előtt, mert igazából még senki nem tudta megbecsülni, főleg kiszámolni az algoritmus valódi lépésszámát, műveleteinek mennyiségét, egyszóval komplexitását. (Idézet az eredeti cikkből: „*The exact complexity of this attack remains an open problem.*” – E támadás komplexitása továbbra is nyitott kérdés marad). Megosztottság jellemzi most a kriptográfusokat, egyik felük az igyekszik bizonyítani, hogy az elgondolás alapjaiban hibás, a másik oldal viszont azt állítja, hogy a kételkedők meg sem értették az XSL lényegét. A kérdéskörhöz igen jó kiindulópont lehet az [URL59, URL60] honlap.

3.5.6. Néhány tesztvektor a FIPS197-ből

Szabványos tesztvektorok

```
AES-128 (Appendix B)
   Key = 2b7e151628aed2a6abf7158809cf4f3c
   Plaintext = 3243f6a8885a308d313198a2e0370734
   Ciphertext = 3925841d02dc09fbdc118597196a0b32
AES-192 (Appendix C.2)
   Key = 000102030405060708090a0b0c0d0e0f1011121314151617
   Plaintext = 00112233445566778899aabbccddeeff
   Ciphertext = dda97ca4864cdfe06eaf70a0ec0d7191
AES-256 (Appendix C.3)
   Key = 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
   Plaintext = 00112233445566778899aabbccddeeff
   Ciphertext = 8ea2b7ca516745bfeafc49904b496089
```

Nem szabványos tesztvektorok

```
AES-128
   Key = 00000000000000000000000000000000
   Plain = 00000000000000000000000000000000
   Ciphertext = 66e94bd4ef8a2c3b884cfa59ca342b2e
AES-192
   Key = 000000000000000000000000000000000000000000000000
   Plain = 00000000000000000000000000000000
   Ciphertext = aae06992acbf52a3e8f4a96ec9300bd7
AES-256
   Key = 0000000000000000000000000000000000000000000000000000000000000000
   Plain = 00000000000000000000000000000000
   Ciphertext = dc95c078a2408989ad48a21492842087
```

3.5.7. A SubBytes, InvSubBytes táblázata

```
SBOX : array[0..255] of byte = (
$63, $7c, $77, $7b, $f2, $6b, $6f, $c5, $30, $01, $67, $2b, $fe, $d7, $ab, $76,
$ca, $82, $c9, $7d, $fa, $59, $47, $f0, $ad, $d4, $a2, $af, $9c, $a4, $72, $c0,
$b7, $fd, $93, $26, $36, $3f, $f7, $cc, $34, $a5, $e5, $f1, $71, $d8, $31, $15,
$04, $c7, $23, $c3, $18, $96, $05, $9a, $07, $12, $80, $e2, $eb, $27, $b2, $75,
$09, $83, $2c, $1a, $1b, $6e, $5a, $a0, $52, $3b, $d6, $b3, $29, $e3, $2f, $84,
$53, $d1, $00, $ed, $20, $fc, $b1, $5b, $6a, $cb, $be, $39, $4a, $4c, $58, $cf,
$d0, $ef, $aa, $fb, $43, $4d, $33, $85, $45, $f9, $02, $7f, $50, $3c, $9f, $a8,
$51, $a3, $40, $8f, $92, $9d, $38, $f5, $bc, $b6, $da, $21, $10, $ff, $f3, $d2,
$cd, $0c, $13, $ec, $5f, $97, $44, $17, $c4, $a7, $7e, $3d, $64, $5d, $19, $73,
$60, $81, $4f, $dc, $22, $2a, $90, $88, $46, $ee, $b8, $14, $de, $5e, $0b, $db,
$e0, $32, $3a, $0a, $49, $06, $24, $5c, $c2, $d3, $ac, $62, $91, $95, $e4, $79,
$e7, $c8, $37, $6d, $8d, $d5, $4e, $a9, $6c, $56, $f4, $ea, $65, $7a, $ae, $08,
$ba, $78, $25, $2e, $1c, $a6, $b4, $c6, $e8, $dd, $74, $1f, $4b, $bd, $8b, $8a,
$70, $3e, $b5, $66, $48, $03, $f6, $0e, $61, $35, $57, $b9, $86, $c1, $1d, $9e,
$e1, $f8, $98, $11, $69, $d9, $8e, $94, $9b, $1e, $87, $e9, $ce, $55, $28, $df,
$8c, $a1, $89, $0d, $bf, $e6, $42, $68, $41, $99, $2d, $0f, $b0, $54, $bb, $16
);
```

1. Táblázat – A SubBytes lépés S-doboza

```
InvSBOX : array[0..255] of byte = (
$52, $09, $6a, $d5, $30, $36, $a5, $38, $bf, $40, $a3, $9e, $81, $f3, $d7, $fb,
$7c, $e3, $39, $82, $9b, $2f, $ff, $87, $34, $8e, $43, $44, $c4, $de, $e9, $cb,
$54, $7b, $94, $32, $a6, $c2, $23, $3d, $ee, $4c, $95, $0b, $42, $fa, $c3, $4e,
$08, $2e, $a1, $66, $28, $d9, $24, $b2, $76, $5b, $a2, $49, $6d, $8b, $d1, $25,
$72, $f8, $f6, $64, $86, $68, $98, $16, $d4, $a4, $5c, $cc, $5d, $65, $b6, $92,
$6c, $70, $48, $50, $fd, $ed, $b9, $da, $5e, $15, $46, $57, $a7, $8d, $9d, $84,
$90, $d8, $ab, $00, $8c, $bc, $d3, $0a, $f7, $e4, $58, $05, $b8, $b3, $45, $06,
$d0, $2c, $1e, $8f, $ca, $3f, $0f, $02, $c1, $af, $bd, $03, $01, $13, $8a, $6b,
$3a, $91, $11, $41, $4f, $67, $dc, $ea, $97, $f2, $cf, $ce, $f0, $b4, $e6, $73,
$96, $ac, $74, $22, $e7, $ad, $35, $85, $e2, $f9, $37, $e8, $1c, $75, $df, $6e,
$47, $f1, $1a, $71, $1d, $29, $c5, $89, $6f, $b7, $62, $0e, $aa, $18, $be, $1b,
$fc, $56, $3e, $4b, $c6, $d2, $79, $20, $9a, $db, $c0, $fe, $78, $cd, $5a, $f4,
$1f, $dd, $a8, $33, $88, $07, $c7, $31, $b1, $12, $10, $59, $27, $80, $ec, $5f,
$60, $51, $7f, $a9, $19, $b5, $4a, $0d, $2d, $e5, $7a, $9f, $93, $c9, $9c, $ef,
$a0, $e0, $3b, $4d, $ae, $2a, $f5, $b0, $c8, $eb, $bb, $3c, $83, $53, $99, $61,
$17, $2b, $04, $7e, $ba, $77, $d6, $26, $e1, $69, $14, $63, $55, $21, $0c, $7d
);
```

2. Táblázat – Az InvSubBytes lépés S-doboza

3.6. A KULCSOK CSERÉJE A SZIMMETRIKUS ALGORITMUSOKBAN

A titkos kulcsú rejtjelezésre nagyon jó módszerek vannak, de ezek legfőbb hátránya, hogy a titkos kulcsban Alicenek és Bobnak előre meg kell egyezni. Ez valamilyen kommunikációs folyamatot igényel, annak veszélyével, hogy közben a támadó esetleg megszerzi a titkos kulcsot (például Eve egyszerű lehallgatással vagy Mallory komolyabb eszközökkel). Ezért olyan kommunikációs csatornát kell használni, amely biztonságos, nem hallgatható le. Ez a csatorna viszont valamilyen értelemben biztosan drága vagy nem állandóan biztonságos, ellenkező esetben nem kellene titkosítani, hanem ezen a csatornán lehetne az üzenetet is elküldeni.

3.6.1. A kommunikációs csatornák jellemzői

A csatorna az a kommunikációs közeg, amely az információt egyik féltől a másikig szállítja. Ez lehet:

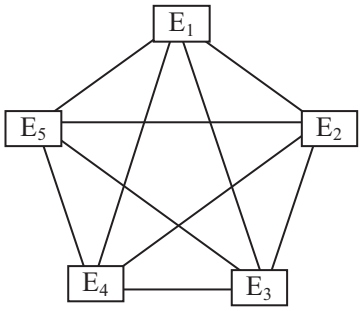
- Fizikailag ***biztonságos*** csatorna (*physically secure channel, secure channel*) A támadónak nincs lehetősége a csatlakozásra csak akkor, ha *a fizikai védelmet megbontja*, ez pedig valamilyen mértékű fizikai rongálást jelent. Szabotázs-védelemmel figyelhető. (Például a csatorna szerepét betöltő kábelt egy kis- vagy nagynyomású gázzal töltött csőben vezetik. Ha a támadó csatlakozni akar a kábelhez, a csövet meg kell bontania, a nyomásváltozást egy nyomásmérő azonnal jelezheti.) A fizikailag biztonságos csatorna csak kisebb földrajzi távolságokban kivitelezhető, dedikált vonalnak tekinthető és meglehetősen drága.

- ***Biztosított*** csatorna (*secured channel*) A csatornának nincs fizikai védelme, így a támadónak lehetősége van hozzáférni a csatornához, ahhoz tetszése szerint csatlakozhat. De a kapcsolatban lévő felek olyan protokollok segítségével kommunikálnak, melyek védelmet nyújtanak az adatok törlése, beszúrása vagy olvasása ellen. Fontos megjegyezni, hogy ez a csatorna fizikailag nem védett, így a küldött (és fogadott) adatok manipulálhatók. A végberendezések vagy ezek szoftvereinek valamint a kommunikációt felügyelő protokoll feladata a kapott adatok figyelése, ellenőrzése.
- ***Nem biztonságos*** csatorna (*insecure channel*) A támadó nemcsak csatlakozni tud a csatornához, hanem ott adatokat megváltoztatni, törölni vagy beszúrni is képes. Az ilyen csatornából fizikai védelemmel lehet *biztonságos* csatornát, vagy megfelelő protokollokkal és titkosító algoritmusokkal lehet *biztosított* csatornát készíteni. A legtöbb kommunikációs közeg ilyen.

Abszolút biztonságos csatorna (*Shannon-féle csatorna*), mely abszolút biztonságot garantál, nem létezik. Olyan, mint az abszolút 0 °K: elvileg van, gyakorlatilag nincs. Hasonló vágyálom a tökéletes biztonságot nyújtó titkosító algoritmus is (amivel már lehetne tökéletesen biztonságos csatornát is készíteni). Ha egy ilyen algoritmust használunk, a támadó – akármennyi üzenetet fog el és azok töréséhez akármennyi erőforrást igénybe tud venni –, csak a nyílt szöveg hosszát tudja meghatározni (*perfect forward secrecy*).

3.6.2. Hány kulcsra van szükség?

Ugyancsak probléma, hogy a titkosított kommunikációban résztvevő minden párnak, minden egyes feladónak minden egyes címzettel meg kell egyeznie egy titkos kulcsban: *n* kommunikációs partner esetén, ha mindenki mindenkivel kommunikálni akar úgy, hogy a másik *n-2* partner számára is titkos legyen az üzenet, akkor *n(n-1)/2* kulcsban kell megegyezni, ez már *n=100*-nál is 4950 kulcs biztonságos menedzselését kívánja meg. Sőt, ha figyelembe vesszük, hogy a kulcsokat „illik” gyakran cserélni, a helyzet tovább romlik. A problémát a nyilvános kulcsú titkosítási módszerekkel jelentősen enyhíteni lehet.



27. ábra Kulcsok és partnerek kapcsolta szimmetrikus algoritmusok esetén

A vonalak az egyeztetett kulcsokat jelentik, az ábra 10 vonala az 5 partner 10 kulcsát jelenti. Ha nem öt, hanem például 1000 partner lenne, 499 500 vonalat kellene behúzni, ennyi kulcsot kellene a rendszernek kezelnie.

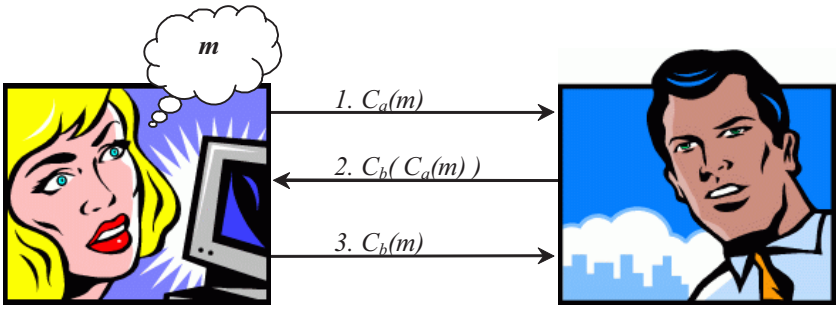
Azonban lehet egy másik ok is, ami a sok kulcs igényét felveti, ez pedig a feladó azonosítása. Korábban azt mondtuk, hogy a titkos kulcsú rejtjelezés használatakor élünk azzal a feltételezéssel, hogy a kulcs csak a felek birtokában van, más nem ismeri azt. Ebből közvetett módon az következik, hogy ha egy – a közös kulccsal – megfejthető üzenetet kapunk, egyúttal azt is

bizonyítva látjuk, hogy kitől jött. Ha azonban az iménti öt résztvevő egyetlen kulcsot használ, hogyan tudjuk megmondani illetve bizonyítani, hogy kitől jött az üzenet?

3.6.3. Háromutas kulcsforgalom – szirének éneke

Az előzetes *kulcscsere nélküli* kommunikációra érdekes megoldást nyújt a gyakorlatban ritkán alkalmazott *háromutas kulcsforgalom.* Nevével ellentétben a protokoll alkalmazásakor egyetlen kulcs sem kerül a kommunikációs csatornára, mert úgy küldünk el egy üzenetet Bobnak, hogy az ő kulcsáról semmit sem tudunk és eközben ő sem tud semmit a mi kulcsunkról. Az így küldött üzenetet egy olyan ládához lehetne hasonlítani, aminek két lakatja van. Előbb ráteszi Alice a sajátját és elküldi Bobnak. Ekkor más nem nyithatja ki, csak Alice. Bob nem tudja kinyitni a ládát, de ráteszi a saját lakatját, majd visszaküldi Alicenak. Most már Alice sem tudja kinyitni a ládikát, hiszen Bob is lezárta. Ebbe beletörődve leveszi a saját lakatját, amit elsőnek ő helyezett fel. A láda most már csak egy lakattal van lezárva, újra elküldi Bobnak, aki így már ki tudja nyitni a ládikót. Figyeljük meg, hogy a kommunikáció során az üzenet egyszer sem jelenik meg védelem nélkül és a lakatok kulcsait sem kell soha elküldeni, ami az egyszerű passzív lehallgatást kiküszöbölheti. A módszer egyetlen feltétele, hogy a két kódolás (a címzett- és a fogadó oldali) felcserélhető legyen:

$$C_a(C_b(m))=C_b(C_a(m))$$



28. ábra A háromutas kommunikáció

A senki sem ismeri a másik kulcsát, az üzenet mégis megérkezik.

Az üzenetváltás tehát megtörtént, jóllehet egyetlen kulcs sem jelent meg a nyilvánosnak tekinthető kommunikációban. A módszer késleltetése, ha a felek nincsenek on-line kapcsolatban, jelentős lehet. Egyszerű példaként nézzük meg a módszert a Caesar-titkosítás használatával és tanuljunk egy kicsit. Az elküldendő üzenet legyen egy „A” betű. Ekkor a forgalom lépései a következők:

1. Alice elküldi a kulcsával titkosított $„A”+k_a$ üzentet.
2. Bob is kódol egyet: $„A”+k_a+k_b$ és visszaküldi.
3. Alice a kapott üzenetet megfejti $„A”+k_a+k_b–k_a = „A”+k_b$ és újra elküldi.
4. Bob végre megfejt és olvas: $„A”+k_b–k_b = „A”$

És mit csinál eközben Eve vagy egy megbízhatónak tartott álpostás? Megjegyzi mindhárom csomagot, mert *(1)+(3) – (2)* kifejezés értéke egyenlő az eredeti nyílt szöveggel. Tehát vigyázzunk, hogy milyen módszert használunk egy-egy háromutas kommunikációban, nehogy a fentihez hasonlóan pórul járjunk! A módszer működik az egyébként biztonságos OTP-vel is, de a postás pontosan ugyanezekkel a lépésekkel hallgathatja le az üzenetet.

A Függelék jelen fejezethez kapcsolódó alfejezetei

14.8. Szabványok összefoglaló táblázata

14.12. Néhány szám és nagyságrend

14.13. Moore törvénye

További kiegészítések folyamatosan bővülő helye a http://www.netacademia.net/konyv webcím.

„A titkos leveleket illetően ezeknek mindenféle küldési módja van, de a küldőnek és a címzettnek egymás között előzőleg meg kell állapodnia." – Ezeket a sorokat Aineasz Taktikosz, az ókori görög hadművészeti irodalom kiemelkedő művelője több mint kétezer évvel ezelőtt írta le. Ebben az egyetlen mondatban benne foglaltatnak a tikosírás, a rejtjelzés, a rejtjelfejtés – a kriptográfia – napjainkban is érvényes törvényszerűségei. [...] Ez a kriptográfia egyik alaptörvénye, ma is érvényben lévő gyakorlata.

Révay Zoltán:
Titkosírások, 1978

4. NYILVÁNOS KULCSÚ MÓDSZEREK

Az eddigi szimmetrikus titkosító rendszerek ugyanazt a kulcsot használják a titkosításhoz, mint a megfejtéshez. Ez azt jelenti, hogy a kulcsot meg kell osztani egy titkos csatornán, mielőtt a titkosított üzenetek a nem titkos úton elindulnak. Ez némi ellentmondást hordoz magában, mert ha van egy titkos csatorna, amin kulcsot lehet cserélni, akkor ezen keresztül lehetne kommunikálni is. A korábbiakban beláttuk, hogy ez igaz ugyan, de a biztonságos csatorna nem használható sokáig vagy rendszeresen, így a kulcsok cseréjét is egy nem biztonságos úton kell megoldani, esetleg egy megbízható harmadik felet (*trusted third party*) kell bevonni. Mindenesetre az elv, miszerint a titkosan kommunikáló feleknek van egy közös titkuk, sokáig (a kezdetektől a XX. század végéig) „szent elv" volt. A feltételezés, hogy másként is lehetne, és a közös titkok nem feltétel, gyakran felháborodást és elutasítást váltott ki, egyszerűen képtelenségnek tűnt.

A nyilvános kulcsú rejtjelezés alapötlete az, hogy a titkosítás folyamatát elválasztja a megfejtéstől, és olyan algoritmust használ, ahol a titkosító paraméter nem azonos megfejtéshez használt paraméterrel.

4.1. DIFFIE – HELLMAN KULCSCSERE

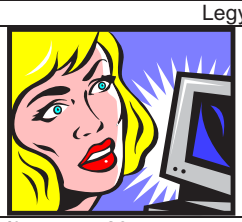
Diffie, *Hellman* és *Merkle* tett közzé először olyan kriptográfiai eljárást, ami a nyilvános kulcsú elméleten alapult (1976). Ez volt az első lépés az új rendszerek felé: megmutatták, hogyan tud két résztvevő a kommunikációhoz szükséges közös titokban megegyezni anélkül, hogy bármit előzetesen egyeztetniük kellene.

> Ma már bizonyított, hogy az angolok néhány évvel *Diffie, Hellman és Merkle* előtt már 1970-ben kitaláltak egy hasonló módszert, azonban katonai titokként kezelték és nem publikálták. Úgy látszik, nagy lehetett a hajtás akkoriban, mert *Bobby Inman*, az NSA egykori igazgatója még erre is „rálicitált" 5 évet, amikor azt állította, hogy az ügynökség már Diffie-ék előtt egy évtizeddel rátalált a módszerre. Ami a lényeg, hogy *Diffie-Hellman-Merkle* publikációja volt a nyíltkulcsú kriptográfia *első nyilvános megjelenése*.

Az eddigiek során a szimmetrikus algoritmusoknál feltételeztük, hogy Alicenak és Bobnak van egy megosztott kulcsa. Ha mégsincs, meg kell beszélniük egyet valahogyan, és ebben segít a *Diffie-Hellman - féle kulcscsere*. (Akinek még ismeretlen a moduláris aritmetika és a modulusképzés fogalma, a Függelék *14.4. Moduláris aritmetika nagyon dióhéjban* című alfejezetében megismerkedhet vele. Szükség van rá, mert a Diffie-Hellman-féle kulcscsere és lényegében az egész nyilvános kulcsú kriptográfia alapja.)

4.1.1. Két résztvevő – a klasszikus algoritmus

Az eljárás röviden a következő: Alice választ két *speciális* nagy prímszámot, *n*-t és *g*-t, melyek tipikusan 192-512 bit hosszúak. Ezután mindketten előállítanak egy-egy hasonló méretű véletlenszámot, amiket titokban tartanak, legyenek ezek *x* és *y* (mindkettő kisebb, mint *n-1*). Először Alice elküldi Bobnak (1) $k_1=(g^x \bmod n)$ *és* (n,g)-t. Bob válaszában (2) $k_2=(g^y \bmod n)$-t küld vissza. Alice a kapott válasz alapján kiszámolja (3) $k_3=k_2^x \bmod n = (g^y \bmod n)^x \bmod n$ értékét, Bob pedig az első üzenet alapján (4) $k_4= k_1^y \bmod n= (g^x \bmod n)^y \bmod n$ értékét. Mindkét kifejezés (k_3 és k_4) egyenlő $g^{xy} \bmod n$ –nel, ami közös, titkos kulcsként használható. A kissé tömör matekos leírást szemlélteti az alábbi számpélda:

Nyilvános paraméterek:	Legyen g = 13 és n = 37	
		
0. Titkos paraméterek:	Alice választ: x = 23	Bob választ: y = 14
1. Kulcselőkészítés:	Alice számol: $K1 = g^x \bmod n = 13^{23} \bmod 37 = 2$	Bob számol: $K2 = g^y \bmod n = 13^{14} \bmod 37 = 25$
2. Kommunikáció:	Alice elküldi az eredményt: K1 →	Bob elküldi az eredményt: ← K2
3. Egyeztetett kulcs:	$K2^x = (g^y)^x = g^{xy} = K \pmod n$ $25^{23} \bmod 37 = 30$	$K1^y = (g^x)^y = g^{xy} = K \pmod n$ $2^{14} \bmod 37 = 30$

Az algoritmus használatához figyelembe kell venni *g* és *n* viszonyát. Az ismertetés elején csak annyit mondtam: „speciális”. Ez mit jelent? A *g* számnak olyannak kell lennie, hogy az *n-nél* kisebb számok előállíthatók legyenek $g^x \bmod n$ alakban. Más szavakkal, ha sorban kiszámoljuk a $g^x \bmod n$ hatványt minden *n-nél* kisebb *x-re*, az eredmények egymástól különbözőek és szépen bejárják az összes *n-nél* kisebb számot. Egy számpéldával könnyebben megérthető, miről van szó:

Legyen n=11 és g=8, n=11 és g=5, n=11 és g=10.

Ekkor $g^x \bmod n$ értékei rendre a következők:

$8^1 \bmod 11 = 8$	$5^1 \bmod 11 = 5$	$10^1 \bmod 11 = 10$
$8^2 \bmod 11 = 9$	$5^2 \bmod 11 = 3$	$10^2 \bmod 11 = 1$
$8^3 \bmod 11 = 6$	$5^3 \bmod 11 = 4$	$10^3 \bmod 11 = 10$
$8^4 \bmod 11 = 4$	$5^4 \bmod 11 = 9$	$10^4 \bmod 11 = 1$
$8^5 \bmod 11 = 10$	$5^5 \bmod 11 = 1$	$10^5 \bmod 11 = 10$
$8^6 \bmod 11 = 3$	$5^6 \bmod 11 = 5$	$10^6 \bmod 11 = 1$
$8^7 \bmod 11 = 2$	$5^7 \bmod 11 = 3$	$10^7 \bmod 11 = 10$
$8^8 \bmod 11 = 5$	$5^8 \bmod 11 = 4$	$10^8 \bmod 11 = 1$
$8^9 \bmod 11 = 7$	$5^9 \bmod 11 = 9$	$10^9 \bmod 11 = 10$
$8^{10} \bmod 11 = 1$	$5^{10} \bmod 11 = 1$	$10^{10} \bmod 11 = 1$

Az eredmények között minden 11-nél kisebb szám előfordul, mégpedig egyszer.

Egy csomó szám hiányzik, ami viszont van, az ismétlődik.

Ha Alice és Bob egy *n=11* mellé *g=10*-et választana, mindössze kettő(!) kulcs generálására lennének képesek. Ha *g=5* a választásuk, csak ötféle kulcsuk lehetne. Azokat a *g* számokat, melyek teljesítik a fenti feltételt, „*generátor*”-nak vagy „*n-re nézve primitív*”-nek hívjuk. Sajnos annak a kiderítése, hogy egy szám generátor-e vagy sem, nem egyszerű és szükséges *n-1* prímtényezős bontása is... [24,11]

Ha Eve lehallgatta az üzenetváltást, a „Kommunikáció” lépésből ismeri g^x és g^y értékeit, valamint a nyilvános n és g paramétereket. A g^x–ből elméletileg ki tudja számolni x-et és Bob válaszüzenetéből y-t is. Azonban a módszer pontosan azon alapul, hogy a moduláris aritmetikában bizonyos esetekben igen nehéz logaritmust számolni: nem ismert olyan használható algoritmus, amely nagy prímszám modulus mellett a kívánt logaritmust (viszonylag) gyorsan előállítaná ($x=log_g\ k_1\ mod\ n$ és $y=log_g\ k_2\ mod\ n$ értékét szeretné Eve kiszámolni).

4.1.2. Három vagy több résztvevő – az algoritmus általánosítása

A klasszikus eljárás két partner számára készít közös kulcsot. De mi történjen akkor, ha a konferenciahívás mintájára „konferencia-bizalmas-kommunikációt” szeretnénk, amelynek több résztvevője lenne kettőnél? Némi általánosítással a Diffie-Hellman kulcssere képes ellátni ezt a feladatot is. A példa kedvéért tételezzük fel, hogy három résztvevőnk van: Alice, Bob és Carol. A kulcscsere a következőképpen zajlik (a modulusképzést nem jelölöm, de ne tessék elfeledkezni róla!):

Nyilvános paraméterek:	**n** és **g** – a már megismert tulajdonságokkal		
	Alice	**Bob**	**Carol**
0. Titkos paraméterek:	x	y	z
1. Kulcselőkészítés:	g^x	g^y	g^z
Alice elküldi Bobnak g^x-t, Bob Carolnak küldi g^y-t, Carol pedig Alicenak küldi el g^z-t. Mint egy körhinta...			
2. Kommunikáció után:	g^z	g^x	g^y
Újabb kulcselőkészítő lépés: a kapott csomagokra ismét mindenki a titkos paraméterét alkalmazza!			
3. Kulcselőkészítés:	g^{zx}	g^{xy}	g^{yz}
Megint befizetünk egy körre, vagyis az eredményt mindenki továbbküldi az előbbi séma szerint!			
4. Kommunikáció után:	g^{yz}	g^{zx}	g^{xy}
Utolsó kulcselőkészítő lépés: a kapott csomagokra ismét mindenki a titkos paraméterét alkalmazza!			
5. Kulcselőkészítés:	g^{yzx}	g^{zxy}	g^{xyz}

Ezt a táblázatot egy mondatban úgy lehetne összefoglalni, hogy a g szám addig vándorol a partnerek között, amíg mindenki titkos paraméterét a hátára nem veszi.

Az algoritmus továbbra is a jól ismert szimmetrikus algoritmusokat segíti, és semmit nem változtat annak tekintetében, hogy Alice és Bob közös titkot birtokol-e. Igen, továbbra is van közös titkuk, csak nem kell bizalmas csatornán előre egyeztetni azt.

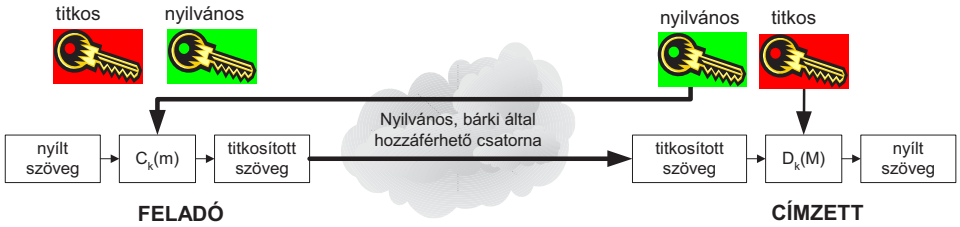
> Ez nem egészen igaz, hiszen továbbra is meg kell beszélni a titkot, ezt a célt szolgálja a *Kommunikáció* lépése. Ami alapvető változást jelent, hogy e megbeszélésnek nem kell titokban zajlania, nincs szükség semmiféle biztonságos csatornára, mert a felek nem a teljes titkot, csak egy-egy speciális részét küldözgetik egymásnak.

A további kutatások ezt a problémát is megoldották, ma már nincs szükség arra, hogy ugyanaz a kulcs legyen a felek birtokában. Ezt az aszimmetrikus rejtjelezést nevezzük *nyilvános kulcsú titkosításnak*.

4.2. A NYILVÁNOS KULCSÚ ALGORITMUSOK ALAPELVEI

A nyilvános kulcsú titkosítással elküldött üzenetek egy olyan ládához hasonlíthatóak, melyet bezárni a nyilvános kulccsal, de kinyitni már csak egy másik, titkos kulccsal lehet. Ez egyúttal azt is jelenti, hogy a módszer biztonsága a titkos kulcs biztonságán alapul, és további feltétel, hogy a bezáráshoz használt kulcsból nem határozható a kinyitáshoz szükséges kulcs.

A nyilvános kulcsot használó rendszerekben tehát minden résztvevő kettő kulcsot birtokol: ezeket *nyilvános* és *titkos kulcsnak* nevezzük. Ezek a kulcsok egymással összefüggnek: a titkos kulccsal lehet megfejteni azt az üzenetet, amit a nyilvános kulccsal kódoltak és fordítva. A nyilvános kulcsot közzé lehet tenni, míg a titkosat – nevéhez illően – titokban kell tartani. Az ismert nyilvános kulcs nem ad segítséget a titkos kulcs kitalálásához, sem pedig a rejtjelezett üzenet visszafejtéséhez, az üzenetet csak az tudja elolvasni, akinek birtokában van a titkos kulcs. Amíg ez a kulcs csak a címzettnek van meg, addig rajta kívül senki más nem férhet a neki címzett rejtjelezett üzenetek tartalmához, még a feladó sem. Ez akkor kínos, ha a levél visszajön, például rossz címzés miatt...



29. ábra A nyilvános kulcsú titkosítás modellje

A nyílt szövegeken és a titkos kulcsokon kívül minden más nyilvános

Az elválasztott titkosítás és megfejtés folyamatát az

$$M=C_e(m) \quad és \quad m=D_d(M)$$

egyenletekkel adhatjuk meg. Itt e és d a különböző titkosító és megfejtő kulcsok, C és D pedig a megfelelő titkosító és megfejtő eljárások. Egy ilyen algoritmust és a nyilvánossá tett e kulcsot használva bárki titkosított üzenetet tud küldeni a címzettnek (akihez az e nyilvános kulcs tartozik), de elolvasni csak a címzett tudja, amennyiben a privát d kulcsot csak a ő ismeri.

A már megismert Caesar-kódolót is tekinthetnénk ilyen rendszernek. Amennyiben a titkosító kulcs $k=3$, akkor a megfejtő kulcs $k= -3$ változatlan algoritmus mellett. Azonban ezt a módszert mégsem tekintjük nyíltkulcsos rendszernek, mert:

- A titkosító algoritmust megfordítva, lépéseit (pontosabban azok inverzét) visszafelé lejátszva megkaphatjuk a kívánt eredményt.
- A k=3 kulcsból egyetlen előjelváltással megkaphatjuk a megoldó kulcsot.

Ha most valaki arra gondol, hogy ugyanezek teljesülnek az RSA-ra is, az ottani műveleteknek is van inverze és a titkos kulcs kiszámolható, annak maximálisan igaza van. De a nagy különbség abban van, hogy egy jól paraméterezett RSA-ban az invertáláshoz szükséges műveletek és a titkos kulcs kiszámolása nem végezhető el belátható időn belül.

Tételezzük fel, hogy a kulcsok szerepe felcserélhető a *C* és *D* algoritmusokban, pontosabban a titkosító algoritmus elfogadja a megoldó kulcsot és fordítva, a megfejtő algoritmus is működik a titkosító kulccsal. Ekkor lehetőség van arra is, hogy a címzett *biztonsággal* ellenőrizni tudja az üzenet küldőjét.

1. Küldés előtt a feladó a titkosító algoritmussal és a saját titkos (tulajdonképpen megoldó) kulcsával rejtjelezi az üzenetet. Ezzel olyan átalakítást végez, amelyet csak az ő képes elvégezni, mert a titkos kulcsot csak ő ismeri.
2. Következő lépésben az iménti eredményt titkosítja a címzett nyilvános kulcsával, így biztosítva, hogy csak a címzett képes kibontani a küldeményt.
3. A címzett a küldemény kézhezvétele után a saját tikos kulcsával és a megfejtőalgoritmussal kibontja a csomagot.
4. A kapott eredményt még nem tudja elolvasni, mert az a feladó által titkosítva van, ezért a feladó nyilvános (tulajdonképpen titkosító) kulcsával megfejti azt.
5. A feladó biztos lehet abban, hogy csak a címzett tudja majd elolvasni az üzenet, mert a (3)-as lépéshez szükséges kulcsot csak ő ismeri.
6. A címzett pedig biztos lehet a feladó személyében, mert a (4)-es lépésben csak akkor tudja megfejteni az üzenetet, ha azt a feladó titkos kulcsával rejtjelezték. (Más szavakkal egy nyilvános kulccsal csak az az üzenet oldható meg, amit hozzátartozó a titkos kulccsal titkosítottak.)

Ilyen értelemben viszont nincs titkosító és megfejtő kulcs, csak titkos és nyilvános. A kulcsok szerepe az adott felhasználástól függ:

- egy titkosítás során a nyilvános kulcsunkkal csomagolják be a nekünk szánt üzenetet, amit majd a titkos kulccsal fogunk kibontani;
- ha szeretnénk aláírni az általunk elküldött üzenetet, akkor a titkos kulccsal csomagoljuk be, amit az aláírás ellenőrzője a nyilvános kulcsunkkal bont ki.

Úgy tűnik, hogy a nyilvános kulcs „beszerzése” érdekében továbbra is szükség van az előzetes megbeszélésre, amely során megkérjük a címzettet, hogy küldje el nyilvános kulcsát. Ez így igaz, de ezt a kommunikációt nem kell védeni. Sőt e kulcs publikus tulajdonsága akár azt is lehetővé teszi, hogy létrehozzunk egy telefonkönyvhöz hasonló kulcskönyvet, amely a résztvevők nyilvános kulcsait tartalmazza. A könyvben lévő személyek számára bárki tud levelet küldeni anélkül, hogy előtte bármiben is megállapodtak vagy találkoztak volna. A szimmetrikus módszerekkel ellentétben itt *n* partner esetén nem *n*(n-1)/2* kulcsot kell kezelni, mindösszesze *n* darabot. De ezek sem titkosak, így akár osztott használatú adatbázisba is helyezhetők (mint az előbb a kulcskönyv), de biztosítani kell a kulcsok és tulajdonosaik közötti kapcsolat hitelességét. Ha valakinek a nyilvános kulcsára kíváncsiak vagyunk, ebből az adatbázisból lekérjük. A küldeményt a kulcsszerver a titkos kulcsával aláírja, így annak nyilvános kulcsával lehet ellenőrizni a küldemény hitelességét. Egy felhasználó helyileg tárolt kulcsgyűjteményét találóan *kulcskarikának* (*keyring*) nevezik sok helyen. A helyi tárolás lehetővé teszi, hogy ne kelljen mindig a viszonylag lassú szerverhez fordulni, azonban figyelni kell arra is, hogy a tárolt kulcsok szinkronban legyenek a szerveren tároltakkal, különös tekintettel azok érvényességére. A hitelesség témakörére még visszatérünk, mivel ez a nyíltkulcsos rendszerek legsebezhetőbb pontja.

4.3. RSA

A figyelmes olvasónak talán feltűnt, hogy sajnos továbbra sem mellőzhető egészen az élmény, amit az előzetes megbeszélés öröme nyújt a feleknek. Továbbra sem úgy működik a titkosított üzenet küldése és fogadása, hogy a feladó gondol egyet, ír egy levelet a címzettnek, titkosítja, és elküldi azt. És mindenki boldog lenne, ha a címzett megkapva az üzenetet pikkpakk megfejtené és már el is olvashatná. Gondoljunk csak bele: a nyilvános kulcs célzott vagy megosztott átadása tulajdonképpen a korábbi *közös titok egy részének* megbeszélését jelenti! Viszont a korábbi módszerekkel ellentétben ennek a kulcsátadásnak nem kell biztonságos csatorna. Sőt, a nyilvános kulcsú módszert használó felek már azt sem mondhatják, hogy ismernek *egy közös titkot*, csupán azt, hogy mindketten ismerik *a másik fél titkának egy részét*, ami nem más, mint a nyilvános kulcs.

4.3.1. A probléma: faktorizáció

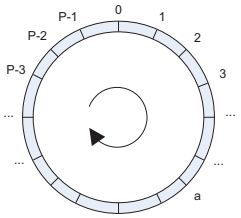
A nyilvános kulcsot használó módszerek gyakran a nagy prímszámok szorzásán és az így létrejövő még nagyobb szám prímtényezőkre bontásának nehézségén alapulnak. A most bemutatásra kerülő algoritmus, az RSA biztonságát is ez a problémája adja.

Első gondolatok

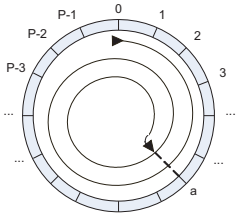
Ha p prímszám, és nem osztója egy a egésznek, akkor $a^{(p-1)}-1$ osztható p-vel. Matekul írva $a^{p-1}-1 \equiv 0 \pmod p$. Ez a kis *Fermat*-tétel, melyről bővebben a Függelékben olvashatunk. A tételt egyenletének mindkét oldalát szorozzuk meg a-val, így egy szemléletesebb összefüggéshez jutunk: $a^p \equiv a \pmod p$, vagyis **ha egy *a* számot egy *p* prímhatványra emelünk, akkor az eredmény *p*-vel való osztás utáni maradékaként visszakapjuk az eredeti *a* számot**.

$5^7 \bmod 7 = 5$
$4^{11} \bmod 11 = 4$
$2^3 \bmod 3 = 2$
$12^{17} \bmod 17 = 12$
$84^5 \bmod 5 = 4$

> Akinek még mindig idegen a ≡ jel és a *mod* szócska, az bizonyára még nem olvasta a *14.4. Moduláris aritmetika nagyon dióhéjban* című fejezetet. Tessék pótolni! Fontos!



1) A moduláris aritmetikát képzeljük el úgy, mintha a szokásos számegyenes helyett lenne egy p beosztású óránk. Ahogy a számegyenesen lépkedünk a különböző műveletek végrehajtásakor, tegyük most ugyanezt az óra kerülete mentén!



2) A kis Fermat-tétel azt állítja, hogy ha az órán kijelölünk egy tetszőleges a számot, majd a „0”-tól elindulva a hatványozás miatt kismilliószor körbefutunk, végül érdekes módon a választott számra érkezünk – abban az esetben, ha az óra kerületét meghatározó p szám prím.

$5^7 \bmod 7 = 5$	$\mathbf{5}^7=78125$ 78125:7=11160 és maradék az **5**!	Ha egy 7 osztású órán 0-tól indulva 78125 lépést teszünk, az 5-ös számra érkezünk.
$4^{11} \bmod 11 = 4$	$\mathbf{4}^{11}=4194304$ 4194304:11=381300 és maradék a **4**!	Ha egy 11 osztású órán 0-tól indulva 4194304 lépést teszünk, a 4-es számra érkezünk.
$2^3 \bmod 3 = 2$	$\mathbf{2}^3=8$ 8:3=2 és maradék a **2**!	Ha egy 3 osztású órán 0-tól indulva 8 lépést teszünk, a 2-es számra érkezünk.
$12^{17} \bmod 17 = 12$	$\mathbf{12}^{17}=2218611106740436992$ 2218611106740436992:17= 130506535690613940, maradék a **12**!	Ha egy 17 osztású órán „nagyon sokat" lépkedünk, akkor a 12-es számra érkezünk.
$84^5 \bmod 5 = 4$	$\mathbf{84}^5=4182119424$ 4182119424:5=836423884 és maradék a **4**!	Ha egy 5 osztású órán 4182119424 lépsét teszünk 0-ból indulva, a 4-es számra érkezünk.

A fenti számpéldák is azt mutatják, hogy

- ha egy tetszőleges számot (most éppen az 5-öt, 4-et, 2-t, 12-t és 84-et)
- egy prímszám hatványra emelünk (most 7.-re, 11.-re, 3.-ra, 17.-re és 5.-re), majd
- a kapott eredményt ugyanazzal a prímmel elosztjuk, visszakapjuk az eredeti számot.

Az utolsó példa kakukktojásnak tűnik, mert 84 helyett 4-et kaptunk eredményül... Az igazsághoz hozzátartozik, hogy ez a fenti egyszerűnek tűnő trükk csak akkor működik szó szerint, ha $a<p$. Egyébként bizony egy maradékot kapunk, mégpedig *a osztva p*-vel maradékát. A helyzet olyan, mintha valaki a karóráján meg szeretné mutatni, hol van a 13 óra... Nos, hol? Pontosan a 13-12=1 helyen. Hasonlóképpen, ha a 84-et szertnénk megmutatni egy 5 osztású órán, kénytelenek lennénk a 4-esre bökni. Az adatvesztések elkerülése érdekében a továbbiakban csak azokkal az esetekkel foglalkozunk, amikor $a<p$.

A *Fermat*-tételnek vagy egy általánosabb formája is, amelyet Euler fogalmazott meg. Nos, Euler azt állította és bizonyította, hogy

$$a^{\varphi(n)} \equiv 1 \ (mod\ n) \qquad \text{ha } a \text{ és } n \text{ relatív prímek}$$

A kitevőben lévő furcsa $\varphi(n)$ függvényt Euler φ függvényének hívjuk, és eredményül minden n természetes számhoz a nála kisebb, hozzá relatív prím természetes számok számát rendeli. (Részletesebben lásd a Függelékben.) Emlékszünk még, mit jelent a „relatív prímek" viszony? Azt, hogy nincs más közös osztójuk, csak és kizárólag az *1*.

Néhány példa φ értékeire:

- $\varphi(8) \rightarrow$ *1, 3, 5, 7* nem osztója 8-nak, míg 2 és 4 igen, tehát $\varphi(8) = 4$
- $\varphi(9) \rightarrow$ *1, 2, 4, 5, 7, 8* nem osztója 9-nek, ez összesen 6 darab, tehát $\varphi(9) = 6$
- $\varphi(14) \rightarrow$ *1, 3, 5, 9, 11, 13* összesen 6 darab, tehát $\varphi(14) = 6$
- $\varphi(17) \rightarrow$ *1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16* összesen 16 darab, tehát $\varphi(17) \rightarrow 16$. Az alábbi táblázatban egy-két példát látunk arra, hogy igazat mondott-e Euler? (Ahol az eredmény nem *1*, ott *a* és *n* sem relatív prím!)

n=9 és $\varphi(9)=6$	
a	$A^6 mod\ n$
1	1
2	1
3	0
4	1
5	1
6	0
7	1
8	1

n=17 és $\varphi(17)=16$	
a	$a^{16} mod\ n$
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	1
13	1
14	1
15	1
16	1

n=14 és $\varphi(14)=6$	
a	$a^6 mod\ n$
1	1
2	8
3	1
4	8
5	1
6	8
7	7
8	8
9	1
10	8
11	1
12	8
13	1

Figyeljük meg a középső számpéldát: a 17 prímszám és $\varphi(17)$ eredménye 16 lett, ami pont eggyel kevesebb. Hoppá! Ilyet már láttunk, a *Fermat*-tétel hatványkitevője is hasonló: *p-1*. A függelékben megtalálható a φ függvény kiszámításának módja, de nekünk egyelőre elég annyi, hogy prímekre $\varphi(prím)=prím-1$, hiszen egy prímhez minden más szám relatív prím.

> A fenti számpéldák még egy fontos tulajdonságra hívhatják fel a figyelmet: Euler tételének feltétele csak relatív prímekre „engedélyezi" a működést! Nézzük meg a példákat! Ha n=8, akkor mindössze 4 számra igaz a tétel, ha n=14, akkor arányaiban még kevesebbre, csak 6-ra. Legszimpatikusabb esetnek az tűnik, ha *n* prím, mert akkor minden nála kisebb számra működik a tétel.

4.3.2. Több felhasználó kellene

Az már látszik, hogy a titkosítás során Euler tételét

$$a^{\varphi(n)} \equiv 1 \ (mod\ n) \qquad \text{I.}$$

$$a^{\varphi(n)+1} \equiv a \ (mod\ n) \qquad \text{II.}$$

fogjuk használni, valószínűleg a (II.) alakban. De akárhogyan is írjuk, rendezgetjük az egyenleteket, a kérdés még nyitva áll, hogy hogyan lesz ebből kétfelhasználós rendszer? Mert ugye ketten használnák: valaki elküldi az üzenetet, valaki más pedig elolvassa. Ha a bal oldalon szereplő kitevőt fel tudnánk bontani két szám (mondjuk *e* és *d*) szorzatára, akkor az milyen jó is lenne, mert a küldő a^e-et küldene, a fogadó pedig $(a^e)^d$-t számolna és máris el tudná olvasni a küldött üzenetet. Ehhez ezt kellene megoldanunk:

$$ed=\varphi(n)+1$$

Sajnos ezt gyakran nem tudjuk megtenni. Például a fenti számpéldákban $\varphi(n)+1$ minden esetben prím, így felbontása sem lehetséges. De van kiút! Vegyük elő az (I.) alakot és bűvészkedjünk egy kicsit vele!

$a^{\varphi(n)} \equiv 1 \pmod n$

$a^{2\varphi(n)} \equiv 1^2 \pmod n$

$a^{3\varphi(n)} \equiv 1^3 \pmod n$

$a^{4\varphi(n)} \equiv 1^4 \pmod n$

$a^{k\varphi(n)} \equiv 1 \pmod n$

Mindkét oldalt emeljük négyzetre! Hogy miért? Azért tegyük meg, mert már volt valaki, aki megoldotta ezt a problémát, de szerencsére nekünk nem kell végigjárni azt a sok vakvágányt, amit neki kellett. Mi már tudjuk, hogy ez a helyes út... Szóval négyzetre fel!

Vagy köbre, vagy akármire. Lássuk be, hogy a jobb oldal nem változik meg, akárhányadik hatványra is emeljük. Ezért a kitevőben az 1, 2, 3, 4 stb. számok helyett akármi is állhat:

Most szorozzuk meg mindkét oldalt a-val: $a^{k\varphi(n)+1} \equiv a \pmod n$

Ez jó! Mert azt jelenti, hogy az $ed=k*\varphi(n)+1$ alakot is felbonthatjuk, ami már sokkal nagyobb szabadságot ígér. Minden tudásunkat összeszedve az alábbi egyenletet írhatjuk fel:

$$a^{k*\varphi(n)+1} \equiv a \pmod n \qquad \text{III.}$$

Mi kell még a működéshez?

1. Legyen n olyan modulus, amely relatív prím minden szóba jöhető ($a<n$) a-hoz.
2. Legyen n nagy! Ennek jelentőségét a későbbiekben látjuk majd, pillanatnyilag elégedjünk meg azzal a magyarázattal, hogy így nagy blokkokat tudunk titkosítani, ami a szótárszerű megfejtést vagy feltörést és a titkosítást lehetetlenné teszi. Vagyis sem a támadó, sem a legális felek nem tudnak a lentihez hasonló szótárat tárolni.)
3. Ki kell tudnunk számolni a $\varphi(n)$-t.
4. A $k*\varphi(n)+1= ed$ felbontható két egész szám szorzatára (vagyis nem prím). A felbontandó kifejezés másként is írható: $ed \equiv 1 \bmod \varphi(n)$

Íme egy példa: Legyen $n=17$. Ekkor $\varphi(n) = 16$. Az $ed=k*16+1$ felbontása $k=1$-re nem megy, de $k=2$-re igen: $ed=3*11$. Legyen Bob nyilvános kulcsa ($e=3,\ n=17$) titkos kulcsa pedig ($d=11$). Alice a nyilvános e-vel tud üzenetet küldeni, amelyet Bob a saját d kulcsával olvashat el. Korábban már vázoltuk, hogy Alice $a^e \bmod n$-t küld Bob pedig $(a^e)^d \bmod n$-t olvas.

Küldött	Alice	Bob	Olvasott
0	$0^3 \bmod 17 = 0$	$0^{11} \bmod 17 = 0$	0✓
1	$1^3 \bmod 17 = 1$	$1^{11} \bmod 17 = 1$	1✓
2	$2^3 \bmod 17 = 8$	$8^{11} \bmod 17 = 2$	2✓
3	$3^3 \bmod 17 = 10$	$10^{11} \bmod 17 = 3$	3✓
4	$4^3 \bmod 17 = 13$	$13^{11} \bmod 17 = 4$	4✓
5	$5^3 \bmod 17 = 6$	$6^{11} \bmod 17 = 5$	5✓
6	$6^3 \bmod 17 = 12$	$12^{11} \bmod 17 = 6$	6✓
7	$7^3 \bmod 17 = 3$	$3^{11} \bmod 17 = 7$	7✓
8	$8^3 \bmod 17 = 2$	$2^{11} \bmod 17 = 8$	8✓
9	$9^3 \bmod 17 = 15$	$15^{11} \bmod 17 = 9$	9✓
10	$10^3 \bmod 17 = 14$	$14^{11} \bmod 17 = 10$	10✓
11	$11^3 \bmod 17 = 5$	$5^{11} \bmod 17 = 11$	11✓
12	$12^3 \bmod 17 = 11$	$11^{11} \bmod 17 = 12$	12✓
13	$13^3 \bmod 17 = 4$	$4^{11} \bmod 17 = 13$	13✓
14	$14^3 \bmod 17 = 7$	$7^{11} \bmod 17 = 14$	14✓
15	$15^3 \bmod 17 = 9$	$9^{11} \bmod 17 = 15$	15✓
16	$16^3 \bmod 17 = 16$	$16^{11} \bmod 17 = 16$	16✓

Hurrá kész is vagyunk! Ez az RSA? Sajnos nem. És a fenti példa matematikailag ugyan jó, csak éppen azonnal elbukik egy törési kísérleten. Ugyanis (*e,n*) nyilvános és tudjuk, hogy *n* prím. A kíváncsiskodó Eve is szeretné *d*-t tudni, mert akkor ő is elolvashatná a Bobnak címzett üzeneteket. És az a baj, hogy gond nélkül meg is teheti, mert ki tudja számolni $\varphi(n)$-t! Emlékszünk? Ha *n* prím, akkor $\varphi(n)=n-1$. Az *n*, *e* és *d* összefüggése pedig: $ed \equiv 1 \bmod \varphi(n)$ És innentől csak egy ugrás a feladat: $3d \equiv 1 \bmod 16$, melynek megoldása $d=11$. (Bár most mellőztem *d* kiszámításának menetét, de higgyék el, nem egy nagy dolog!)

Modulust keresünk

Milyen modulust válasszunk, hogy mindenki jól járjon? Azt láttuk, hogy nem jó a prímszám, mert nem nyújt védelmet, tehát összetettnek kell lennie. De nem jó bármilyen összetett szám sem, mert $\varphi(n)$ kiszámításához szükség van az *n* szám prímtényezős bontására (lásd Függelék), ami nagy modulus esetén problémás feladatnak tűnik. Ugyanakkor ez jó védekezési módszer is, hiszen az előbb Eve csak azért tudta kiszámolni Bob titkos kulcsát, mert képes volt $\varphi(n)$ kiszámítására. De ha jó nagy számot választunk, hogy Eve ne tudja kiszámolni $\varphi(n)$-t, akkor mi miért lennénk rá képesek? Zsákutcának tűnik, hacsak nem fordítjuk meg a gondolatmenetet, és ahelyett, hogy egy nagy számot próbálnánk felbontani tényezőire, ismert számokból, mint tényezőkből magunk állítjuk elő azt. Így vélekedhettek az RSA kitalálói is, akik olyan modulust választottak, amely mindössze kettő prímszám szorzata: $n=pq$. Euler függvényét sem nehéz kiszámolni az ilyen számokra: $\varphi(n)=(p-1)(q-1)$. Legyen az új szereplőkkel a modulus $n=pq$ és a kitevő szorzata $ed=k*(p-1)(q-1)+1$ alakban felírva! Mindezt a (III.) egyenletbe beírva azt kapjuk, hogy:

$$a^{ed} \equiv a \bmod pq$$

ahol $ed \equiv 1 \bmod (p-1)(q-1)$

Legyen a nyilvános kulcs az (*e,n*) számpáros, a titkos kulcs pedig a (*d,n*) páros. Mit tud tenni most Eve? Nem sokat, mert $\varphi(n)$ kiszámolásához szüksége lenne *n* prímtényezős bontására: *p*-re és *q*-ra. De mi ravasz módon nem áruljuk el neki, sőt a kulcsgenerálás után eldobjuk őket és olyan nagyra választjuk ezeket a számokat, hogy *n* gigantikusan nagy legyen. Így – a tudomány mai állása szerint – Eve feladata reménytelen: nem képes kitalálni *p* és *q* számokat, így $\varphi(n)$ kiszámítására sem képes.

4.3.3. Az RSA titkosítás és megfejtés

Fenti eredményünket a következőképpen használhatjuk a gyakorlatban: a titkosításhoz az üzenetet először számokká alakítjuk úgy, hogy a számok (blokkok) mindegyike kisebb legyen, mint *n*. Az egyes üzenetdarabokat a fenti képletekben *a* helyére behelyettesítjük Ezután az egyes *m* számokból (mint üzenetekből) az

$$M=m^e \bmod n$$

képlettel előállítjuk a rejtjelezett *M* üzenetet, amit

$$m=M^d \bmod n$$

képlet alapján lehet megfejteni. És ez az RSA? És ilyen egyszerű? Igen, ez az RSA és valóban ilyen egyszerű! De ne feledjük, az idáig vezető út azért volt ilyen rövid és járható, mert mi már tudtuk, merre kell elindulni! Amikor 1976-77 környékén az algoritmus tervezői fejében hosszú munka, kutatás és rengeteg próbálkozás után végül mindez megszületett, a Diffie-Hellman algoritmushoz hasonlóan egyszerre volt zseniális újdonság és szentségtörő mutatvány. Az algoritmus elnevezése a tervezők nevének első betűjét őrzi: *Rivest*, *Shamir*, *Adleman*, munkájukat 1977-ben publikálták.

> Az RSA algoritmust szabadalom védte az USA-ban, de ez a védelem 2000-ben lejárt. Ma már bárki licenszdíj-mentesen készíthet RSA-algoritmuson alapuló hardver- vagy szoftvereszközt.

4.3.4. RSA kulcsgenerálás

A kulcsgenerálás lépései tehát a következők:

1. Válasszuk ki P és Q prímszámokat!
2. $N=P*Q$ és $\varphi(N)=(P-1)*(Q-1)$
3. Válasszunk egy véletlen E számot úgy, hogy relatív prím legyen $\varphi(N)$-re. (Különben nem lesz invertálható $\varphi(N)$-re és D sem lesz kiszámolható.)
4. Számoljuk ki E multiplikatív *modulo* inverzét $\varphi(N)$-re nézve, ez lesz D. (Ez a nyelvtörő nem káromkodás volt, hanem keressünk egy olyan D-t, amelyre $ED \equiv 1 \bmod \varphi(N)$ teljesül vagyis az ED szorzat $\varphi(N)$-nel osztva 1-et ad maradékul. Például 43 multiplikatív inverze 1590-re nézve 37, mert $43\times37=1591$, ami 1590-nel osztva 1-et ad maradékul. Ezt így írjuk: $43\times37\equiv1 \pmod{1590}$. Általános jelöléssel: $a\times a^{-1}\equiv1 \bmod m$, ahol a^{-1} az a-nak m-re vonatkozó inverze.)

Lássunk egy számpéldát a kulcskészítés és a titkosítás műveletére:

1. Legyen $P=17$ és $Q=23$!
2. $N=P*Q=391$ és $\varphi(N)=(P-1)*(Q-1)=352$
3. Legyen $E=21$, a (21,352)=1 teljesül. Az $E=21$ multiplikatív inverze $\varphi(N)$-re: $D=285$, mert 285*21 mod 352 = 1.
4. Első lépésként átalakítjuk az üzenetet számokká. Ehhez használhatjuk az ASCII táblát, a számként felírt üzenet számjegyeinek csoportosítását, de más módszert is kigondolhatunk. Egy a fontos: minden üzenetdarabnak kisebbnek kell lennie, mint 391. Ha $p=239$ és $q=277$, választásunk eredményeképpen $N=66203$ lenne, akkor a betűket kettesével is csoportosíthatnánk.
5. Az átkódolás és a hatványozások eredményét az alábbi táblázat mutatja:
   a. A „T” ASCII kódja: 84.
   b. Az ő titkosított párja: $84^{21} \bmod 391 = 135$, ezt kell elküldeni.
   c. A fogadó oldalon pedig a $135^{285} \bmod 391 = 84$ számítást kell elvégezni.

	m_i	M_i		M_i	m_i	
T	84	135		135	84	T
I	73	167	→	167	73	I
T	84	135		135	84	T
O	79	214		214	79	O
K	75	96		96	75	K
$M_i = m_i^{21} \bmod 391$				$m_i = M_i^{285} \bmod 391$		

Ezek az értékek a gyakorlatban nem használhatók biztonságos kódolásra, mert a választott prímek kicsik. A példát a Windows számológépével még lehet ellenőrizni, de a szokásos zsebszámológépek valószínűleg kudarcot vallanak, mert a megfejtő oldalon 10^{600} nagyságrendű számok is előfordulnak. (Addig, amíg nem optimalizáljuk a hatványozást.) A felhasznált számoknak olyan nagyoknak kell lenniük, hogy az N számot ne lehessen prímtényezőkre bontani. Ha ugyanis az N számot fel tudjuk bontani $N=PQ$ alakra, akkor $\varphi(N)$ kiszámolható lenne és így e ismeretében inverzszámítással meg lehetne határozni d-t is. ***Fontos tehát megjegyezni, hogy a titkos és a nyilvános kulcs között pontosan definiált matematikai összefüggés van. A nyilvános kulcsból mindig kiszámolható a titkos kulcs is, csak elég idő és számítási kapacitás kell hozzá!!!***

Valaki megkérdezte tőlem, hogy szerintem nem elég az, ha a generátorszámok nem prímek, csupán egymáshoz viszonyítva relatív prímek? A válasz az, hogy de igen, elég. Ha az általános esetet bemutató *Euler* kongruenciatételt vesszük szemügyre, láthatjuk, hogy az egyetlen feltétel $(a,n)=1$ teljesülése. A nyilvános e számot továbbra is úgy kell megválasztani, hogy relatív prím legyen $\varphi(n)$-re, ekkor invertálható és kiszámolható a megfelelő d. A bökkenőt csak $\varphi(n)$ kiszámítása jelenti. Értékét mindenképpen tudni kell, hiszen az e és d kulcsok számításához szükség van rá. Ehhez pedig ismerni kell n prímtényezős bontását, amit viszont nem tudunk, mert szándékosan olyan nagyra választottuk, hogy ne lehessen tényezőkre bontani. Így nem marad más hátra, hogy beletörődünk: az n modulust két prímszám szorzataként állítjuk elő.

Kiterjesztett Euklideszi legnagyobb közös osztó algoritmus

Hogyan is lehet kiszámolni a sokat emlegetett D titkos kulcsot, pontosabban egyik kulcsot a másikból? Keresünk egy olyan számot, amelyre igaz, hogy: $D \times E \equiv 1 \bmod \varphi(n)$, persze $\varphi(n)$ és E ismert, sőt relatív prímek, D a kérdés.

Álljon itt (bizonyítás nélkül) egy algoritmus, amely megfelel a célnak [11]. Adott két szám: x, y. Az algoritmusnak három visszatérési értéke van: a, b és v. A visszatérési értékek a következő kapcsolatban állnak egymással:

1. $v=(x,y)$ és 2. $v=ax+by$

Jó lesz nekünk ez az algoritmus? Bűvészkedjünk egy kicsit a paraméterekkel! Legyen $x=E$, $y=\varphi(n)$, és az a eredmény a D-ként értelmezve. Ekkor a fenti két egyenletet a következő alakban írhatjuk:

1. $v=(E, \varphi(n)) = 1$ és 2. $1=D \times E + b \times \varphi(n)$

Fordítsuk meg a (2)-es egyenletet, csak a vizuális típusú olvasók kedvéért, majd vegyük az egyenlet mindkét oldalának $\varphi(n)$-el történő osztás utáni maradékát:

$$D \times E + b \times \varphi(n) = 1$$
$$(D \times E + b \times \varphi(n)) \bmod \varphi(n) = 1 \bmod \varphi(n)$$
$$D \times E \equiv 1 \bmod \varphi(n)$$

Vagyis jó lesz, pontosan ez az a kifejezés, amit kerestünk, mindössze az algoritmus be- és kimeneti paramétereket kell megfelelően értelmezni!

Binary extended Eudidean GCD algorithm

Bemenet: • x,y – két pozitív egész szám

Kimenet: • (a, b, v) úgy, hogy $v=ax+by$ és $v=gcd(x,y)$

1. G=1
2. ciklus amíg X és Y egyaránt páros
   - X:=X/2
   - Y:=Y/2
   - G:=2×G
3. U:=X, V:=Y, A:=1, B:=0, C:=0, D:=1
4. Ciklus, amíg U páros
   - U:=U/2
   - ha A≡B≡0 mod 2, akkor A:=A/2, B:=B/2 különben A:=(A+Y)/2, B:=(B-X)/2
5. Ciklus, amíg V páros
   - V:=V/2
   - ha C≡D≡0 mod 2, akkor C:=C/2, D:=D/2 különben C:=(C+Y)/2, D:=(D-X)/2
6. Ha U ≥ V, akkor U:=U-V, A:=A-C, B:=B-D különben V:=V-U, C:=C-A, D:=D-B
7. Ha U = 0, akkor vissza(C,D,g×v) különben vissza a 4. lépésre

> Az iménti algoritmus az „eredeti” EGCD-nek egy optimalizált változata: a fő ciklusmagokban ugyanis nincs szorzás! A rengeteg kettővel való osztás pedig nem más, mint a tárolt szám bitjeinek eltolása jobbra. Ezért hívják „binárisnak”, ez a gyors módszer csak a binárisan tárolt számok esetében alkalmazható hatékonyan. Az algoritmus mindkét változata megtalálható [11]-ben. (14.61 és 2.107 pontok alatt.)

4.3.5. Kitevők és modulusok

Napjaink legismertebb aszimmetrikus kulcsú titkosítása az RSA lett, amelynek a fenti algoritmus az alapja: könnyű két nagy prímszámot összeszorozni, de nehéz a szorzatot felbontani. A DES-hez hasonlóan blokkos titkosító, a blokkok méretét n határozza meg. Az algoritmusban e és d szerepe felcserélhető, ezért az RSA alkalmas digitális aláírásra is. A gyakorlati alkalmazások során jelenleg az 1024-3072 bites modulusokat tekintjük biztonságosnak. Ha az RSA kulcsainak hosszáról beszélünk, az alatt mindig a modulus N hosszát értjük, mert N felbontása jelenti az algoritmus törését, így N hossza határozza meg a kulcsok és az algoritmus biztonságát. A faktorizáció területén elért újabb eredmények (pl. *elliptikus görbe faktorizáció*) miatt ajánlott közel azonos méretű prímszámokat felhasználni a kulcsok generálásához: például egy 1024 bites modulust célszerű két 512 bites prím alapján kiszámolni.

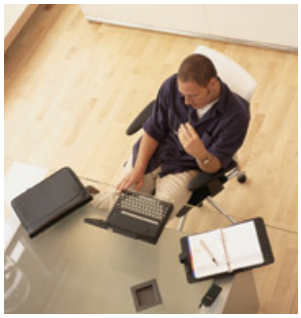
Azonban vigyázni kell arra is, hogy a két szám ne kerüljön „túlságosan közel” egymáshoz. (*lásd: később*) Másik gyakorlati megvalósításhoz kapcsolódó megjegyzés, hogy a nyilvános *e*-t sokszor nem véletlen számként állítják elő, hanem fixen 3 vagy 17 vagy 65537. Közös tulajdonságuk (amellett, hogy prímek), hogy bináris ábrázolásban csak kettő darab „1”-es tartalmaznak, ami a hatványozás során jelentős sebességnövekedést jelent más prímekhez képest (a miértről lásd a *„4.6.3. A moduláris hatványozás”* fejezetet) Sajnos a kedvező teljesítmény csak a nyíltkulcsot igénylő műveletekre igaz: a titkosításra és az aláírás ellenőrzésére, hiszen ezeknél használjuk a nyilvános kulcsot. Az X.509 a *65537* használatát javasolja, a PEM a *3* mellett döntött, a PKCS#1 pedig a *3* vagy a *65537* értéket ajánlja. Nem lehet igazán egyik vagy másik mellett dönteni, mert néha nem jók: ha például $\varphi(n)$ hattal osztható, nem jó az $e=3$ választás. Ilyen eset lehet a ($p=5,q=7$) vagy ($p=31,q=41$) stb. 1998-ban *Boneh* és *Venkatesan* azonban megmutatta, hogy kis *e* használata esetén az $m=\sqrt[e]{M}$ (vagyis a *k*-ad gyökvonás) könnyebb, mint az *n* faktorizálásának problémája [URL34], így a fenti ajánlások fenntartással kezelendők.

4.3.6. Szempontok a prímszámok és a kulcsok kiválasztásához

A prímszámok kiválasztásához eddig nem adtunk meg semmilyen szempontot, teljesen véletlenszerűen választottuk ki azokat. Egyetlen követelményünk volt, hogy legyenek minél nagyobbak. Azonban az elmúlt években a különböző támadási módszerek vizsgálta során körvonalazódott néhány tulajdonság, melyekkel a kiválasztott prímeknek és a belőlük generált kulcsoknak rendelkezniük kell. Mielőtt ezeket megismernénk, tegyünk egy kis elvi kitérőt, és ismerkedjünk meg a speciális kulcsok speciális tulajdonságaival, már ha vannak ilyenek. Vajon minden kulcs egyformán jó? Minden kulcs egyformán erős?

Weak keys

A címbeli kifejezés szó szerint „gyenge kulcsot” jelent, de magyarul szerintem találóbb lenne a „szerencsétlen kulcs” fordítás. Amikor egy titkosító algoritmust használunk és választunk hozzá egy kulcsot, tulajdonképpen kijelölünk egy megoldandó feladatot a támadó számára. Az összes lehetséges feladat számát a kulcstér mérete határozza meg, és mi ebből választunk ki egyet. (A DES-nél ez 2^{56}, az IDEA-nál 2^{128}, míg egy RSA-1024 modulus esetében körülbelül 2^{512} lehetséges választást jelent. Lásd még a 3. fejezetben lévő algoritmus – kulcs kapcsolatáról szóló gondolatébresztőt...)



A támadónak alapvetően kétféle megoldása lehet egy-egy ilyen feladatra. Az egyik *általános megoldás*, amit minden esetben alkalmazni tud, nem függ a használt kulcstól vagy más paramétertől. Ilyen általános megoldás mindig van és ez a *brute-force*, vagyis az összes lehetséges kulcs kipróbálása, valamint a helyes visszafejtés eredményének kiválasztása. Természetesen lehetnek más általános megoldások is, például az RSA esetében általános megoldásnak tekinthető a nyilvános modulus tényezőkre bontásának algoritmusa is, ha nem tartalmaz semmilyen

feltételt a modulusra vonatkozóan. A faktorizáló eljárás azonban algoritmusfüggő megoldás, hiszen egy DES esetében nem tudunk vele mit kezdeni. A *brute-force* ilyen értelemben minden algoritmustól független, mert elve szinte minden esetben használható.

A másik típusú megoldás a *speciális megoldás*, amely feltételezi, hogy a kulcs valamilyen formának megfelel. Ilyen feltétel lehet, hogy a 23. bit a kulcsban „0", vagy az utolsó öt bit „1" értékű legyen. Ha az alkalmazott kulcs megfelel az elvárásoknak, a speciális megoldás nagyságrendekkel hatékonyabb lehet az általános megoldásnál. Az ilyen kulcsokat nevezzük „*weak-key*"-nek, vagyis gyenge kulcsnak.

> Megjegyzem, hogy a DES-terminológiában kicsit mást jelent a „weak keys" kifejezés. Ott azokat a kulcsokat nevezzük így, amelyek azért nem alkalmasak titkosításra, mert az eljárás az eredeti nyílt szöveget adja eredményül, vagyis: $C=E_k(P)=P$. Ez azonban nem jelent elvi különbséget a fenti gondolatmenethez képest, mert ezt tekinthetjük egyfajta triviális megoldású feladatnak is.

Ha a támadó speciális megoldással próbálkozik, olyan szerencsejátékba kezd, amelyben feltételezi, hogy a titkosításhoz használt kulcs a megfelelő tulajdonságokkal rendelkezik. Vagy nyer vagy veszít, attól függően, hogy az összes kulcs közül mennyi a gyenge kulcs. Ha nyer, egy hatékony algoritmus könnyen megoldja a feladatot. Ha veszít, csak az idejét vesztegeti – bár előfordulhat, hogy a vesztes speciális megoldás végül általános megoldássá fajul.

A támadó a speciális és az általános megoldás között aszerint dönthet, hogy melyik ad előbb eredményt (*time-to-first solution*) és az is elképzelhető, hogy előbb megpróbálkozik a speciális megoldásokkal, és utána az általánossal. (Felvetődő kérdések: Mennyi gyenge kulcs van? Mennyi az összes lehetséges kulcs? Mennyi ideig tart kipróbálni az összes speciális megoldást? Mennyi ideig tart az általános megoldás? stb.)

Erős prímek az RSA-ban

Az RSA titkosítási rendszerben is lehetnek gyenge kulcsok, melyek bizonyos támadási módszereket előnyösebb helyzetbe hoznak az általános megoldásnál. Számos vizsgálat eredményeképpen a prímeknek (p és q) a következő tulajdonságokkal kell rendelkezni [26]:

1. A választott prím (továbbiakban R) *nagy,* legalább 400-500 bit hosszú.
2. $R-1$ legnagyobb prímosztója (továbbiakban R^-) nagy.
3. $R^- -1$ legnagyobb prímosztója (továbbiakban R^{--}) nagy.
4. $R+1$ legnagyobb prímosztója (továbbiakban R^+) nagy.

Azokat a prímeket, amelyek mindegyik feltételnek megfelelnek *erős prímeknek* (*strong primes*) nevezzük. Az eredeti RSA dokumentáció javaslata az volt, hogy a használt prímek R^{--} erősek legyenek, vagyis teljesüljön rájuk a (3)-as feltétel. Az ilyen tulajdonságú prímek előállítására útmutatást találhatunk a [26] irodalomban. A '70-es évek második felétől számos kutatás folyt, illetve folyik a faktorizálás terén, melyek eredményeképp kialakult az általános nézet, miszerint az RSA-hoz erős prímeket kell használni. Azokat az RSA kulcsokat, melyeket nem erős prímekből számoltak, gyenge kulcsoknak tekintették, mivel egyes faktorizáló algoritmusok hatékonyan ki tudják használni ezt a „hiányosságot". 1985-ben *Hendrik W. Lenstra* kidolgozott egy olyan – elliptikus görbén alapuló – eljárást, amely alapvetően felforgatta az

addig kialakult nézetet. A fő érv az erős prímek használatára *Pollard* faktorizáló algoritmusa volt, *Lenstra* eljárása viszont annak általánosításaként is felfogható. Előnye, hogy „nem zavarja", ha a vizsgált összetett szám erős prímekből lett számolva, viszont nem szereti, ha a vizsgált szám közel *egyforma hosszúságú* prímek szorzata. Hatékonyabb is a korábbi algoritmusoknál, mert erősen párhuzamosítható és a *nagy* számoknál jön igazán formába, ami viszont az RSA-nál egyébként is kívánatos. *Rivest* és *Silverman* mindezt egy 1998-as közös tanulmányukban foglalták össze [26]. Egyúttal megmutatták, hogy az erős prímek használata felesleges, mert alkalmazásuk véd ugyan bizonyos faktorizálási módszerek (és az iteratív támadás) ellen, de nem véd azok általánosítása, *Lenstra* módszere ellen. Használatuk tehát nem baj, de nem szükséges.

4.4. ELGAMAL

Az *ElGamal* kriptorendszer a diszkrét logaritmus problémáján alapul. Ennek kiszámítása a moduláris aritmetikában a modulus tényezőkre bontását igényli. Ha a modulust nem tudjuk tényezőkre bontani (mert prím vagy olyan nagy, hogy ez nem lehetséges), akkor a feladat nem egyszerű, sőt határozottan nehéz. Az algoritmus hasonlít a már látott *Diffie-Hellman* kulcstovábbítási megoldáshoz. Alkalmas digitális aláírások készítésére és ellenőrzésére. Az RSA-nál átlagosan kétszer lassabb, mert két moduláris hatványozást végez minden egyes kódoláskor. További hátrány, hogy a titkosított szöveg kétszer olyan hosszú, mint az eredeti nyílt szöveg.

A rendszer nyilvános paraméterei egy p prím és egy g generátor szám. (Mi az a generátor szám? Lásd: *4.1. Diffie – Hellman kulcscsere* alfejezetet vagy [24,11]-et. Sajnos a [24]-ben lévő ElGamal-ismertetés elfeledkezik a g szám generátor voltáról, de próbáljuk ki az alábbi algoritmust $p=11$ és $g=10$ választással: egyszerűen katasztrófa...) Egy résztvevőnek két kulcsa van, egy titkos a és egy nyilvános y, ahol $y=g^a \bmod p$. Ha ennek a résztvevőnek egy m üzenetet akarunk elküldeni, generálunk kell egy olyan véletlenszerű k számot, amely kisebb, mint p, majd kiszámoljuk az

$$y_1=g^k \bmod p \qquad \text{és} \qquad y_2=m \oplus (y^k \bmod p)$$

értékeket, melyeket elküldünk a címzettnek, aki az

$$m = y_2 \oplus (y_1^a \bmod p)$$

számítással megkapja az üzenetet. Miért is? Azt kell belátnunk, hogy $y_1^a \equiv y^k \bmod p$, mert csak ekkor működhet a fenti összefüggés. Nézzük meg, mit is rejt magában a megfejtés folyamatában szereplő y_1^a hatvány?

$$y_1^a \equiv (g^k)^a \equiv (g^a)^k \equiv y^k \bmod p.$$

Az algoritmus törését az jelentené, ha ki tudnánk számolni:

- y_1-ből a véletlen k-t: $k=\log_g y_1 \bmod p$ vagy
- y-ból titkos a-t : $a=\log_g y \bmod p$.

Az *ElGamal* titkosításon alapuló *ElGamal* aláírás a *DSS* aláírási szabvány alapját képezi.

4.5. HIBRID KRIPTORENDSZEREK

A gyakorlati megvalósítások során azonban nem az egész üzenetet szokták nyilvános kulcsú algoritmussal kódolni, mert ezek a módszerek (nemcsak az RSA) lassúak. A hagyományos szimmetrikus algoritmusok hardvermegoldásban átlagosan ezerszer gyorsabbak, de szoftveres megoldás esetén is legalább százszor [23,24]. Ezért az aszimmetrikus eljárások nem igazán alkalmasak arra, hogy magát az *m* üzenetet titkosítsák, ha az hosszú. A szokásos eljárás az, hogy az üzenetet egy gyorsabb titkos kulcsú algoritmussal, az ehhez használt – véletlenszerűen generált – kulcsot pedig a nyilvános kulcsú módszerrel titkosítják, és a kettőt együtt küldik el. Az ilyen alkalmankénti, egyszer használt kulcsot **viszonykulcsnak** (*session key*) nevezzük. Tehát a titok megfejtéséhez szükséges kulcs paradox módon éppen a rejtjelezett szöveggel együtt utazik. Így dolgozik a PGP is, az RSA segítségével titkosítja a szimmetrikus kulcsot, majd a tömörített üzenet tényleges kódolása az IDEA/CAST algoritmussal történik. Ne felejtsük el, hogy a tömörítés nemcsak a transzmissziós időt csökkenti, hanem a titkosítás biztonságát is jelentősen növeli: nehezíti a mintakeresését, növeli az entrópiát, elrejti a nyílt szöveg legtöbb jellemző tulajdonságát, stb! Ez a módszer – amit *hibrid kriptorendszernek*, borítékolásnak (*enveloping*) vagy egyutas kulcsforgalomnak is neveznek – ötvözi a titkos és nyilvános kulcsú algoritmusok előnyeit. Nem kell előre megállapodni a titkos kulcsban, hiszen azt a szöveggel együtt el lehet küldeni, ugyanakkor a titkosítás és a megfejtés gyors, mert a lassú nyilvános kulcsú titkosító algoritmussal csak a kisebb méretű titkos kulcsot kell titkosítani az egyik, és megfejteni a másik oldalon. Ez a módszer igen jó kulcscserét biztosít a titkos kulcsú algoritmusok számára. Az előnye mellett sajnos van egy kényelmetlen tulajdonsága is a megoldásnak, mert – bár sokaknak úgy tűnhet – mégsem ez a módszer a kriptográfusok Szent Grálja. A nyilvános kulcs hosszából származó előny – a *brute-force* nyilvánvaló lehetetlensége – ugyanis elveszik. A támadó számára nem is érdekes a titkosított viszonykulcs, hanem a titkosított üzenetet egyszerűen úgy kezeli, mintha nem ismerné a kulcsot, ami igaz is: az aszimmetrikus algoritmus megvédi a kulcsot. De mi védi meg az üzenetet a szokásos támadási módszereke egyikével szemben? Lényegében semmi. Meg kell érteni, hogy a nyilvános kulcsú algoritmus itt most a kulcsot védi és nem az üzenetet, tehát a kulcselosztásban segít! Ezért a hibrid rendszer megvalósításakor fontos figyelni a használt szimmetrikus algoritmusra, mert lényegében annak erőssége határozza meg az egész rendszer erősségét: ha a szimmetrikus algoritmus például *brute-force* módon megtörhető, akkor a rendszer is. Persze a kulcsot védő aszimmetrikus titkosítást sem szabad elhanyagolni, mert ha az gyenge, a támadó nem fog *brute-force*-szal szórakozni, hanem a mellékelt kulcs megszerzését részesíti előnyben.

4.5.1. Biztonságos levelezés nyilvános hálózaton – hibrid kriptorendszerrel

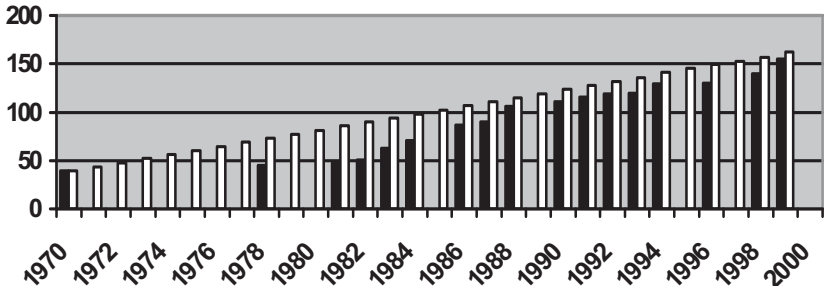
A bevezetőben említett egyik példaprobléma az volt, hogy hálózaton keresztül levelezünk, korántsem biztonságos módon. Mit lehet tenni, hogy a levél tartalma csak a címzett számára legyen hozzáférhető? Eddigi ismereteink már elégségesek a válaszhoz: titkosítsunk! Egy kissé gyakorlatiasabb megközelítésben a következőket tehetjük:

1. Válasszunk titkosítási módszert!
   - Ha személyesen is *találkoztunk* már a címzettel és *előrelátó* módon megegyeztünk vele egy titkos kulcsban, használhatunk valamilyen szimmetrikus eljárást, bár így nem fogjuk tudni megvédeni levelünket egy galád támadótól, aki kihallgatta a kulcsot egyeztető megbeszélésünket.
   - Ha aszimmetrikus eljárást használunk, nem kell a címzettel előtte kulcsot egyeztetni, csak el kell tőle kérni az ő *hitelesített* nyilvános kulcsát. (Ez történhet személyesen, kulcsszerverről vagy elküldheti emailben is, stb.)
   - Csakhogy mi szép hosszú leveleket szoktunk írni, meg jó nagy mellékleteket küldözgetünk, így a tisztán nyilvános kulcsos eljárás nagyon lassú lesz. Használjunk viszonykulcsot a titkosításhoz, vagyis dolgozzunk hibrid kriptorendszerrel! Ez azt jelenti, hogy minden levél titkosítása előtt generálunk egy véletlenszerű kulcsot, amelyet a szimmetrikus algoritmushoz használunk, a kulcsot pedig a kétkulcsos algoritmussal titkosítva küldjük el, a levéllel együtt!
2. Szerezzük be a címzett nyilvános kulcsát és tanúsítványát, majd írjuk meg a levelet!
3. Generáljunk egy véletlenszerű kulcsot és a választott szimmetrikus algoritmussal titkosítsuk a levelet! A viszonykulcsot pedig titkosítsuk a címzett nyilvános kulcsával!
4. Küldjük el a titkosított levelet és a titkosított kulcsot a címzettnek!

4.6. AZ RSA FELTÖRÉSE

Egyes számítások szerint egy 1024 bites nyilvános kulcs egy 80 bites szimmetrikus kulcsnak, egy 128 bites szimmetrikus kulcs egy 3000 bites nyilvános kulcsnak felel meg [23]. Az RSA feltöréséhez nem érdemes a DES esetében bevált kulcspróbálgatással nekifogni, mert egy 512 bites (elavultnak tekinthető) kulcsmérettel is bőven védekezhetünk a *brute-force* ellen. Ugyanezt a számot prímtényezőkre bontani, bár nem egyszerű és igencsak erőforrásigényes, de nem reménytelen feladat. Az alábbi táblázatban – ami [10,11]-ből való – azt láthatjuk, hogy az egyes új módszerek és a számítógépek teljesítménynövekedése milyen haladást tettek lehetővé a faktorizálás terén. Az adatsor érdekessége, hogy a digitben mért számméret és az évek között nagyjából lineáris kapcsolat van, ami feltehetően annak köszönhető, hogy a számítási kapacitások növekedése és az egyre nagyobb számok faktorizálásának erőforrásigénye egyaránt exponenciális jellegű. Az adatok alapján egy 1024 bites szám faktorizálása 2037 körül várható, bár egyes elméleti fejtegetések – *Moore* törvényével alátámasztva és egyéb okfejtések alapján – ezt 2018-ra jósolják.

Év	Digit	Bit	Ki	Eljárás	Hardver
1970	39	129	Brillhart/Morrison	CFRAC	IBM MainFrame
1978	45	150	Wunderlich	CFRAC	IBM Mainframe
1981	47	156	Gerver	QS	HP-3000
1982	51	170	Wagstaff	CFRAC	IBM Mainframe
1983	63	210	Davis/Holdridge	QS	Cray
1984	71	240	Davis/Holdridge	QS	Cray
1986	87	290	Silverman	MPQS	LAN Sun – 3' s
1987	90	299	Silverman	MPQS	LAN Sun – 3' s
1988	100	332	Internet	MPQS	Distributed
1990	111	369	Lenstra/Manasse	MPQS	Distributed
1991	116	386	Lenstra/Manasse	MPQS	Distributed
1992	119	429	Atkins	MPQS	Distributed
1996	130	432	Montgomery	GNFS	Distributed
1998	140	466	Montgomery	GNFS	Distributed
1999	155	512	Montgomery	GNFS	Distributed



30. ábra Eddigi faktorizálások

A függőleges tengelyen az adott évben faktorizált szám mérete decimális digitben. A fekete oszlopok a valós adatokat jelölik, a fehér oszlopok egy lehetséges lineáris közelítést jelölnek: Méret=4.23*(Év-1970)+39

Az idő pénz

A következő táblázat alapja az, hogy van 10.000.000 dollár pénzünk…(már akinek van). Azután olyan gépet építünk, ami tudja a *GNFS*-t – a faktorizálás ma ismert egyik legjobb módszerét – mégpedig annyit, amennyire csak futja a pénzből. A gép teljesítménye legyen akkora, mintha a Wiener-féle gép 100 másodperc alatt kitalálná a DES-kulcsot, tehát körülbelül 2/3 elméleti Kína számítási kapacitása legyen benne[32]. A gépek fő költségét a memória jelenti, de a szükséges mennyiséget – a bontandó szám mérete alapján – előre meg tudjuk határozni.

[32] Ez a mondat csak azoknak mond valamit, akik olvasták a „3.1.4. A DES feltörése" fejezetet. Aki még nem tette, itt az ideje pótolni…

A táblázatban az egy gépbe építendő memória mérete van megadva. A gépek alapköltsége (processzor, alaplap, csatlakozók, stb.) legyen 100 dollár és 1 Mb memória 50 centbe kerüljön. Az egyéb költségeket, mint például a hálózati összeköttetések költségét nem vesszük figyelembe.

```
Összes_memória := F(szám_méret)
Gépek_száma := (10.000.000 - Összes_memóra * 0,5)/100
Egy_gép_memóriája := Összes_memória / Gépek_száma
```

Ha a szám mérete	*akkor a teljes memóriaigény alapján legfeljebb ennyi gépre elég a pénzünk:*	*Így az egy gépbe jutó memória mérete:*	*és a feltörés ideje:*
430 bit	86200	32Mb	kevesebb mint 5 perc
760 bit	4300	4Gb	600 hónap
1020 bit	114	170Gb	3 000 000 év
1620 bit	0,16	120Tb	10^{16} év

Az utolsó gépet nem tudjuk megépíteni, hiszen egyhatod géppel igen nehéz boldogulni. Az 512 bites RSA modulus mai – és a fentieknél kisebb költségű – eszközökkel is faktorizálható, belátható időn belül. Tehát használjunk jó nagy (>1024 bit) számokat az RSA titkosításban és egy jó ideig biztonságban vagyunk.

4.6.1. Néhány RSA elleni egyszerűbb támadás

Ha az üzenettér kicsi, próbálgatással könnyebben meghatározható mind az üzenet, mind a használt kulcs. Másrészt gyakrabban fordulhat elő, hogy a hatványozás elvégzése után az eredmény kisebb marad, mint N ($m^e<N$) így a hatványozás hagyományos hatványozás marad, és a visszafejtés is gyökvonássá egyszerűsödik. A lehetséges védekezés: az üzenetrészeket ki kell egészíteni véletlenszerű információval. Ez a kitöltés (*padding, salting*) egyébként mindentől függetlenül bármikor, bármilyen algoritmus használata esetén is ajánlható.

Ha a p-q különbség kicsi, akkor p és q jó közelítéssel az N modulus négyzetgyökével egyenlő. Ez szerencsétlen esetben jó becslést jelenthet p-re és q-ra vonatkozóan, jelentősen könnyítve így a faktorizáció problémáját. Ha a p-q különbség nagy, a veszély csökkenthető.

Trükkösebb támadást mutatok be a következő sorokban, és ezt *választott titkosított szöveg alapú támadásnak* nevezzük.

Trükkös Eve esete

1. Eve elfog egy $c=135$ üzenetet és tudja, hogy Alice (az üzenet feladójának) nyilvános kulcsa: $(e,n)=(21,391)$. Eve szeretné elolvasni az üzenetet, ennek legkényelmesebb módja lenne, ha rendelkezne Alice tikos kulcsával. Ezzel csak Alice rendelkezik ($d=285$), Eve a meglévő adatokból kiszámolni nem tudja, így az is elég lenne, ha Alicet rábírná az üzenet megfejtésére. (Pontosabban fogalmazva arra, hogy alkalmazza titkos kulcsát.)

2. Eve ennyire nyílt kérést nem intézhet Alicehoz, mert Alice esetleg ráismer az általa korábban küldött üzenetre és megtagadja a megfejtett üzenet visszaküldését. Ezért Eve cselhez folyamodik, és megváltoztatja a *c* üzenetet: megszorozza egy tetszőlegesen választott számból számított értékkel! (Úgy is mondhatjuk, álcakabátot ad rá...)
3. Véletlenszerűen választ egy $x=3$ számot, majd ebből előállít egy hamis *c'* kriptoszöveget: $x^e \bmod n = 3^{21} \bmod 391 = 192$, ezzel megszorozza az eredeti titkosított üzenetet, így $c'=135\times192 \bmod 391 = 114$-et kap eredményül. Ezt elküldi a Alicenak olyan kérés kíséretében, ami a titkos kulcs alkalmazására készteti őt.
4. Alice megfejti $c'=114$-et: $m'=114^{285} \bmod 391=252$. Mivel az eredmény nem emlékezteti őt korábbi üzeneteire, ezért gyanútlanul visszaküldi Eve-nek.
5. Eve, amíg Alice eredményére vár, kiszámolja x inverzét is Alice modulusára nézve: $x^{-1} \bmod 391 = 261$. (Ellenőrzésképpen: $3\times261 \bmod 391 = 1$)
6. Amikor Eve végre megkapja Alice válaszát, egyszerűen megszorozza azt az imént kiszámolt x^{-1}–nel (leveszi róla az álcakabátot): $m=m' \times x^{-1} \bmod n = 252\times261 \bmod 391 = 84$ és ha visszalapozunk a korábbi számpéldára, láthatjuk, hogy ez a „T” betű ASCII kódja, aminek rejtjeles párja valóban 135 volt.

A módszert az alábbi összefüggésekkel igazolhatjuk:

$$
\begin{aligned}
m' &= c'^{d} && \bmod n \\
&= (cx^{e})^{d} && \bmod n \\
&= c^{d} x^{ed} && \bmod n \\
&= mx && \bmod n \\
m &= m'x^{-1} && \bmod n, \quad \text{ha } (x,n)=1
\end{aligned}
$$

Elsőre naiv gondolatnak tűnik, hogy a feladó egy számára ismeretlen üzenetet a titkos kulcsával rejtjelezni fog. Azonban ez a művelet nem más, mint a digitális aláírás egy változata: a feladó az elküldött üzenetet nem a címzett nyilvános kulcsával, hanem a saját titkos kulcsával kódolja, így az üzenet csak a feladó nyilvános kulcsával fejthető vissza, bárki ellenőrizheti a feladó személyét. Az x^e–nel való szorzást *c*-re nézve *vakításnak* hívjuk, mert az „áldozat” nem ismeri fel az egyébként tőle származó üzenetet. Fontos felismerni, hogy Eve nem az algoritmus valamilyen gyengeségét használta ki, hanem cselesen becsapta Alicet, akit így sikeresen rábírt az üzenet megfejtésére!

Közös modulus [40] esete forog fenn, ha ugyanazt az üzenetet kétszer ugyanazzal az *n* modulussal, de különböző nyilvános exponenssel titkosítjuk. Amennyiben a nyilvános kulcsok relatív prímek, létezik két olyan szám (u,v), amire $ue_1+ve_2\equiv1 \bmod n$. Ezek a számok megkereshetők. (lásd: *4.3.4. RSA kulcsgenerálás*). Ebből és a két elfogott titkosított üzenetből kiindulva: $m=m^{ue1+ve2}=c_1^{u}\times c_2^{v} \bmod n$. Ne tekintsük ezt az esetet nagyon ritkának! A prímszámkeresés igen lassú művelet, ezért ha valamilyen alkalmazásban gyakran kell kulcsokat generálni, előfordulhat, hogy időtakarékossági okokból ugyanazokat a prímszámokat többször is felhasználja az alkalmazás. Ugyanazok a prímek viszont mindig ugyanazt a modulust adják, hiszen azok szorzataként áll elő.

4.6.2. Nagy prímek keresése

Elegendő prímszám van?

Egy 1024 bites RSA modulushoz körülbelül $3{,}778\times10^{151}$ prím áll rendelkezésre. Ez sokkal több az elégnél. Másrészről a RSA biztonságának megtartásához egyre nagyobb és nagyobb kulcsok kellenek, úgyhogy jó lenne, ha tényleg nagyon, nagyon, nagyon, nagyon sok prímszámunk lenne. A legjobb az lenne, ha végtelen sokan lennének, így biztosan nem fájna a fejünk! (Persze, amikor a kulcs úgy 1 megabájt lesz, az már nagyon kínos lesz…) Szerencsére a prímszámok száma valóban végtelen és ezt Euklidész már réges-régen bebizonyította!

1. Tételezzük fel, a legnagyobb prímszám P!
2. Írjuk fel a P-nél kisebb prímszámokból alkotott következő gigantikus szorzatot, majd adjunk hozzá 1-et: $Q = 2\times3\times5\times7\times11\times13\times\ldots\times P + 1$
3. Próbáljuk meg elosztani Q-t 2-vel! A hányados $3\times5\times7\times11\times13\times\ldots\times P$, de a +1 miatt maradék az 1.
4. Próbáljuk meg elosztani Q-t 3-mal! A hányados $2\times5\times7\times11\times13\times\ldots\times P$, de a +1 miatt most is 1 a maradék.
5. Öttel? Héttel? P-vel? Nem, nem, mert a fránya +1 miatt mindig egy a maradék!
6. Ha Q összetett szám, akkor valamelyik prímtényezőjének nagyobbnak kell lennie P-nél, hiszen az előbb láttuk, hogy P-vel bezáróan semmi sem osztja Q-t. Ebben az esetben ez a bizonyos tényező lesz a legnagyobb prím.
7. Ha Q nem összetett, hanem prím, akkor ő maga üti le a trónról P-t.

És ugyanezt kezdhetjük elölről, az újonnan megtalált „legnagyobb” prímre…

Egy szám prím vagy összetett?

A nyilvános kulcsú rejtjelezéshez nagy prímeket kell előállítani. Mivel nincs olyan módszer, amellyel közvetlen módon lehetne meghatározni egy prímszámot, próbálgatással kell egyet találni. Generálunk egy nagy véletlen páratlan számot és megnézzük, hogy prím-e. Ha nem, veszünk egy másikat, és azt ellenőrizzük. Mindezt addig ismételjük, amíg a megfelelő prímszámot meg nem találjuk az algoritmushoz. Ha viszont egy tetszőleges számot kell ellenőriznünk, vajon prím-e, felmerül a kérdés, hogyan tudjuk ezt viszonylag egyszerűen eldönteni róla? A válasz az, hogy teljes biztonsággal sehogy[4]. Erre ugyanis csak egy biztos módszer van: az, hogy végignézzük 1 és a szám négyzetgyöke közötti egészek mindegyikét, hogy osztója-e valamelyik a számnak. Ha egyik sem osztója, akkor a szám prím, ellenkező esetben az első osztó megtalálásakor abbahagyhatjuk a keresést, mert a szám összetett. Ezt a sorozatos osztást azonban nem tudjuk elvégezni, hiszen pont olyan nagy prímet szeretnénk találni, amelyikkel ez a próbálgatás már nem tehető meg belátható időn belül. És ugyanez a probléma más, komolyabb faktorizáló módszerekkel is. Természetesen ez a sorozatos osztás is egyszerűsíthető, mert ha egy szám nem volt osztható 3-mal, akkor természetesen 9-cel sem lesz osztható (*szitamódszerek*). A legjobb az lenne, ha valahogy generálni tudnánk a vizsgált számnál

kisebb prímeket, és csak azokkal végeznénk el az osztást. Ezzel a gondolattal legalább két baj van. Az egyik, hogy jelenlegi ismereteink szerint nem tudunk olyan algoritmust vagy függvényt készíteni, amely visszaadná a paraméterként átadott számnál kisebb valamennyi prímet. Ilyen eljárás egyszerűen nincs. A másik gond az, hogy ha lenne sem tudnánk használni. Miért nem? *C.F.Gaufy* a 18. században sejtette (és a következő évszázadban *Hudamand* és *Poussin* bizonyította is), hogy ha x egy tetszőleges szám, akkor az x-nél kisebb prímszámok száma:

$$\pi(x) \cong \frac{x}{\ln x}$$

Vagyis ha egy 1024 bites prímszámot vizsgálnánk, akkor 2^{512}-ig $\pi(2^{512}) = 3{,}778\times10^{151}$ prímet generálna a „varázsalgoritmus", ezekkel lehetne az osztásos próbát elvégezni. Ez annyiban segítene, hogy az eredeti számmennyiségnek (kb. $1{,}341\times10^{154}$) kevesebb, mint a 0,3%-a maradna meg, de még ez is olyan borzasztóan sok, hogy kivitelezhetetlen ennyi osztás elvégzése.

A sorozatos osztás, mint klasszikus eljárás biztos eredményt szolgáltatna (sőt még a vizsgált szám egy prímosztóját is megadná), de a gyakorlatban nem tudjuk alkalmazni. Teljes bizonyossággal tehát nem tudjuk eldönteni egy számról, hogy prím-e vagy sem, ugyanakkor *tetszőleges biztonságú* becslést adhatunk erre annak árán, hogy a prímteszt algoritmusok nem adják meg a vizsgált szám egyetlen prímosztóját sem, de erre nincs is szükségünk.

Valószínűségi prímtesztek

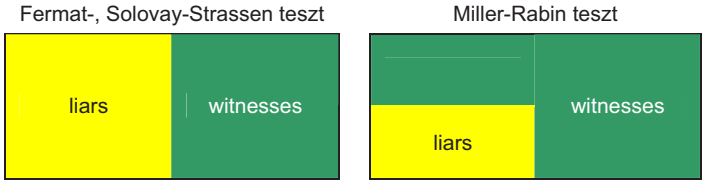
Ha egy p szám prímszám, igaz rá a *Fermat*-tétel. Ha viszont nem, úgy jó esélyünk van rá, hogy találunk olyan a számot p mellé, amelyre a *Fermat*-tétel nem igaz. Ha találunk egy ilyen a számot, akkor a vizsgált p szám biztosan összetett, mert egy prímszám mellé nem tudnánk olyan a-t találni, amely megsérti a *Fermat*-tételt. A gyakorlati próba során egy p prímjelölt mellé véletlenszerű a számokat generálunk és megnézzük, hogy igaz-e a tétel. Ha úgy találjuk, hogy a tétel nem igaz, a p számot összetettnek nyilvánítjuk és befejezzük a tesztet. Bizonyítható, hogy ha p összetett, akkor a nála kisebb a számok legfeljebb felére igaz a tétel [11]. Így ha a vizsgált p-hez már 100 olyan a-t találtunk, amire a feltétel teljesült, annak a valószínűsége, hogy p mégsem prím: 2^{-100}. A próba nem dönti el 100%-os biztonsággal egy szám prím voltát, ezért arra a számra, ami már egy meghatározott számú esetben kielégítette a feltételeket, azt mondjuk, hogy *prímgyanús (pseudo prime, probable prime)*. Ha a teszt összetettnek nyilvánítja a vizsgált számot, akkor az biztosan összetett.

A vonatkozó szakirodalom a következő kifejezésekkel illeti az a számokat, attól függően, hogyan viselkednek:

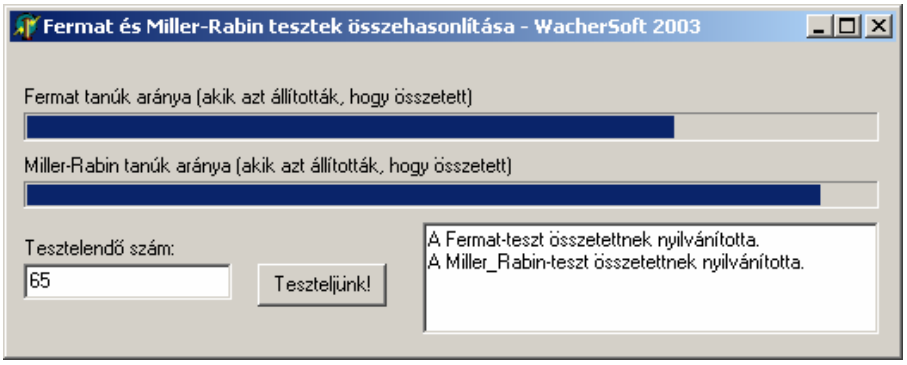
- Ha a p szám összetett, de egy a számon elbukik a teszt, akkor az a szám *tanú* (*witness*) – az összetettség mellet.
- Ha a p szám összetett, de egy a számon nem bukik el a teszt, akkor az a szám *hazug* (*liar*) – a prímtulajdonságra vonatkozóan.

Természetesen léteznek más prímtesztek is, például: *Solovay-Strassen* teszt, *Miller-Rabin* teszt, de ezek sem tudnak egyelőre garantált eredményt adni, csak biztosabbat: ha egy szám a

Miller-Rabin teszten átmegy százszor, a bizonytalanság mindössze $4^{-100} = 2^{-200}$, vagyis az *a* számok legfeljebb ¼-e hazug (általában sokkal kevesebben vannak).



A tanúk (*witnesses*) és a hazugok (*liars*) aránya általános esetben.



A gyakorlatban a *Fermat*-tesztet és a *Solovay-Strassen*-tesztet viszonylag nagy (egyébként egyforma) bizonytalanságuk miatt nem használják. Jelenleg legelterjedtebb algoritmusnak a *Miller-Rabin* teszt számít, amely a következő (matematikai háttér nélkül, ami [11]-ben megtalálható, a 4. fejezetben):

Miller-Rabin valószínűségi prímteszt

Bemenet: • Egy tesztelendő páratlan $n \geq 3$ szám, és egy $t \geq 1$ szám, amely a végrehajtott tesztek számát határozza meg.

Kimenet: • *„prím”* vagy *„összetett”*, mely *n* prím voltát jelzi.

```
1.  Írjuk fel az n-1 számot r × 2^s alakban, ahol r páratlan!
2.  for i := 1 to t do
        a := random_min_max(2,n-2)
        y := a^r mod n
        if y ≠ 1 and y ≠ n-1
            j:=1
            while j ≤ s-1 and y ≠ n-1
                    y := y^2 mod n
                    if y = 1 return := ”összetett”
                    j := j+1
            if y ≠ n-1 return := ”összetett”
3.  return := ”PRÍM”
```

Tipp az első lépéshez: mivel *n-1* páros, osszuk el 2-vel! Ha ismét páros, újra osszuk el, és így tovább. Számoljuk meg, hányszor tudtuk elosztani: ez lesz *s*, az osztások végeredménye pedig *r*. Valahogy így:

```
r:=n-1; s:=0;
while (r and 1)=0 do
begin
  r:=r shr 1;    // shift right
  s:=s + 1;
end;
```

Egy ilyen tesztelés nagy számítási kapacitást igényel, hiszen a *Fermat*-teszt esetében egy valamilyen *a* számhoz ki kell számítani a^{p-1}*-1*-et, illetve ami ezzel ekvivalens, a^{p-1}-nek *p*-vel adott osztási maradékát. A vizsgált *p* akár 64..256 bájtos (512..2048 bites) egész szám is lehet, ami decimális alakban 155-660 jegyű, és ez van a kitevőben! Ezért a teszteket csak olyan *p* számokra célszerű alkalmazni, amelyekről más, egyszerűbb módszerek még nem derítették ki, hogy összetett. A teljes négyzetek például kihagyhatók a tesztelésből. Egy négyzetszám utolsó két jegye mindig *00, s1, s4, 25, t6,* vagy *s9*, ahol *s* páros, *t* páratlan számjegy. Nem biztos, hogy minden így végződő szám négyzetszám, de ha az, akkor így végződik [4]. Egy gyökvonást elvégezni viszont jóval rövidebb ideig tart, mint a *Fermat* próba. Régi, jól ismert és hatékony gyorsítási lehetőség, ha kihagyjuk a 2-vel osztható (utolsó számjegyből eldönthető), 3-mal osztható (a decimális számjegyek összegéből eldönthető), 5-tel osztható (a decimális utolsó számjegyből eldönthető) számokat. E számjegyösszeges vizsgálatok természetesen csak akkor alkalmazhatók, ha a vizsgált szám decimális jegyei rendelkezésre állnak. Általánosabban fogalmazva, mielőtt a számításigényes prímtesztet elkezdjük, érdemes előszűrésként megnézni, hogy a szám osztható-e valamilyen 2 és előre rögzített *H* közötti prímmel.

Az ilyen típusú teszteket *valószínűségi teszteknek* nevezzük, mert csak valamekkora valószínűséggel tudják megállapítani egy szám prím voltát. Csak az a biztos, ha a teszt összetett számnak nyilvánítja a vizsgált számot, ezért néha nem is prímteszteknek, hanem *összetettségi teszteknek* hívják ezeket a módszereket. Felmerülhet a kérdés, hogy érdemes-e véletlenszerűen választott páratlan számokat vizsgálnunk? Korábban láttuk, hogy egy *x*-nél kisebb prímek száma *x/ln(x)*. Annak az esélye hogy az összes *x* darab szám közül beletaláljunk egy prímbe *1/ln(x)*. Ezt meg is duplázhatjuk, mert páros számokat nem vizsgáljuk. Egy 1024 bites szám esetében ez a valószínűség *2/(512 *ln(2)) ~ 1/177* vagyis alig több, mint *0,56%*, ami elég kicsi „találati arány", de nem annyira, hogy reménytelen vállalkozássá tegye az ilyen módszereket.

Valódi prímtesztek különleges prímekre

Speciális prímszámokat másképp is elő lehet állítani, például a *Mersenne* prímek $M=2^n$-1 alakúak, ahol *n* prím és *n≥2*. Ezek kevesen vannak, és ha ilyen prímet használunk, próbálgatással gyorsan fel lehet törni a titkosítást. A jelenleg ismert legnagyobb prímek mind *Mersenne* prímek, és a jelenlegi (2003. december) rekorder: $2^{20\ 996\ 011}$-1, ami a 40. ismert *Mersenne* prím és 2003.11.17.-én találták meg [URL73]. El tudja valaki képzelni, mekkora ez a szám? Több mint 2,5 megabájtot foglal el bináris ábrázolásban és pontosan 6 320 430 decimális jegye van! A *Mersenne* számokhoz kapcsolódik a *Lucas-Lehmer* teszt, mely kifejezetten

a *Mersenne* prímek tesztelésére való, eredménye azonban biztos, nincs olyan bizonytalansága, ami a valószínűségi teszteket jellemzi. Maga a teszt igen egyszerű és a következő:

Lucas-Lehmer teszt Mersenne prímekhez

Bemenet: • n – kitevő a 2^n-1 formulában

Kimenet: • *„prím” vagy „összetett”,* mely $M=2^n-1$ prím voltát jelzi.

```
1. Ellenőrizzük n prím volt át! Mivel n kicsi, ez akár 2 és √n közötti
   sorozatos osztással is ellenőrizhető. Ha n nem prím, akkor M=2^n-1
   sem prím.
2. u := 4
3. for k:=1 to n-2 do u:=(u^2 - 2) mod M
4. if u=0 then return (”PRÍM”) else return(”összetett”).
```

4.6.3. A moduláris hatványozás

Az a moduláris hatványozás, amely az RSA alapja, olyan művelet, melyet széleskörűen használnak a numerikus titkosítás és egyéb kódolások területén. Több kriptorendszerben is előfordul az RSA mellett, például a *Diffie-Hellman* kulcscserében, *ElGamal* rendszerben, vagy a *NIST DSS*-ében (*Digital Signature Standard*). A moduláris hatvány elméleti támadásának matematikai háttere általában a nagy számok tényezőkre bontásán alapszik. Csakhogy az összetett számok prímszámokra való felbontására nincsen hatékony algoritmus [4], bár az sem bizonyított, hogy ilyen algoritmus nem létezik. (Ehhez legközelebb Lenstra algoritmusa jár...)

> Egy algoritmust akkor nevezhetünk hatékonynak, ha a probléma „méretének” (vagy más szavakkal valamelyik bemenő paraméter jellemzőjének vagy értékének) kismértékű növelésével az algoritmus erőforrásigénye nem nő „drasztikusan”. Az erőforrásigény általában memória vagy futási idő. Nem húzható éles határvonal a hatékonyan megoldható és a nem megoldható feladatok közé. Gyakran felvetődik az a kérdés is, hogy érdemes-e egy feladatot megoldani? Milyen erőforrásigénye van egy-egy megoldásnak? Ez nekünk azt jelenti, hogy érdemes-e egy kulcs nélküli üzenetet feltörni? Elavult-e az üzenet, mire megismerjük? Érdemes-e egy-egy üzenetet „végtelen” védelemmel ellátni?

A problémát most a felbontandó szám nagyságrendje jelenti. Az alapvető aritmetikai műveletek kiszámításhoz szükséges lépések száma a számjegyek számával arányosan, vagy kis kitevőjű hatvány szerint nő. Ha a számjegyek számát például kétszeresére növeljük, akkor az összeadáshoz kétszer, a szorzáshoz, osztáshoz négyszer annyi művelet, illetve idő kell. Ha *N* prímtényezős felbontáshoz a sorozatos osztás egyszerű módszerét választjuk, az összes egész páratlan számot kipróbáljuk *1* és gyök *N* között, aminek időigénye a szám méretének exponenciális függvénye. Emiatt ha növeljük az iménti titkosításban használt *N* értékét, olyan értékekhez jutunk, ahol a rejtjelezés folyamata pár pillanat, míg a feltörés évezredekig, esetleg évmilliókig tart. A hatványozás elvégzéséhez szükséges idő egyszerűen elenyésző a ma ismert legjobb faktorizáló algoritmusok futási idejéhez képest is. *A számítási kapacitások növekedése miatt időközönként felül kell vizsgálni az ajánlott N értéket, és ennek növelésével ismét biztonságos tartományba helyezhető a titkosítás.*

Azonban a „hatalmas” méretű, több száz vagy ezer bites számokkal való számolás nem igazán egyszerű feladat. A bonyolultabb – négy alapműveleten túlmutató – műveletekre gyakorlatilag csak moduláris aritmetika szabályai szerint fejleszthető ki.

Művelet[33]	t=8 bit ≅ 2,4 digit	t=16 bit ≅ 4,8 digit	t=24 bit ≅ 7,2 digit	t=1024 bit ≅ 308,2 digit	
Összeadás, kivonás	8	16	24	1024	O(t)
Szorzás, osztás	64	256	576	1048576	$O(t^2)$
Karatsuba-Ofman szorzás[34]	26	80	151	57052	$O(t^{1.58})$
Hatványozás, egész kitevő	16384	16777216	9663676416	$1{,}885\times10^{314}$	$O(2^t\times t^2)$
Hatványozás, intelligens	1024	8192	27648	2147483648	$O(2\times t\times t^2)$
Osztás 1 és √N között	1024	65536	2359296	$1{,}372\times10^{157}$	$O(2^{t/2}\times t^2)$

31. ábra Az egyes aritmetikai műveletek lépésigénye

A számok a művelethez szükséges bitenkénti összeadások, mint elemi műveletek számát jelölik.

Ahogy [5]-ben is olvasható, az első és egyben legegyszerűbb alapszabály az, hogy hogyan *ne számoljunk* moduláris hatványt:

$$M=m^e \bmod n$$

1. lépés $X=m^e$

2. lépés $M=X \% n$

Ez a megoldás kudarcra van, ítélve amint a használt számok elérik azt a tartományt, melyet az RSA titkosításhoz használni „illik”. Legyen $e=2^{512}$ és $n=2^{1024}$ nagyságrendű! Ekkor az m üzenettér körülbelül 1024 bites, így az X részeredmény tárolásához végül

$$\log_2 m^e = e*\log_2 m = 2^{512}*1024 = 1{,}3728*10^{157} \text{ bit}$$

kell, ami elképzelhetetlenül nagy tárolókapacitás. (Állítólag ez a szám több, mint a Föld atomjainak becsült számának kőbe.) Egyúttal azt is jelenti, hogy nem ez a megoldás. Ne feledjük, hogy a lebegőpontos számoktól eltérően itt nem engedhető meg a legkisebb helyértékű bitek elhagyása, minden bitet tárolni kell!

[33] Az szükséges lépésszámoknál nincsenek figyelembe véve az optimalizálási lehetőségek és a legrosszabb esetre (*worst case*) vonatkoznak.
[34] $1.58 \approx \log_2 3$

Bináris hatványozás

Igen sok módszer van a hatványozás egyszerűsítésére, én most a legegyszerűbbet – és műveletszám szempontjából a legrosszabbat – mutatom meg, melyet bináris hatványozásnak is (*addition chain*) hívnak. Kérdés: mennyi a^b értéke?

1. Számítsuk ki *a* következő hatványait, és *b*-t írjuk fel bináris alakban, jelölje *n* a kitevő *b* bitjeinek a számát!

a^1	a^2	a^4	a^8	...	$a^{2^{n-1}}$
b_0	b_1	b_2	b_3	...	b_{n-1}

2. Szorozzuk össze *a* azon hatványait, melyekhez nem nulla b_k bit tartozik:

$$a^b = f(b_0 * a^1) * f(b_1 * a^2) * f(b_2 * a^4) * ... * f(b_{n-1} * a^{2^{n-1}}),$$

ahol *f(x)* egy olyan függvény, amely minden egész számra magát a számot adja vissza, kivéve a nullát, ahol *f(0)=1,* vagyis programozástechnikailag egy feltételvizsgálat. Ez az összefüggés az azonos alapú hatványok szorzására vonatkozó azonosságon alapul:

$$a^b = a^{b_0*1+b_1*2+b_2*4+b_3*8+...} = a^{b_0*1} * a^{b_1*2} * a^{b_2*4} * a^{b_3*8} * ...$$

Tanulmányozva az alábbi programrészletet belátható, hogy legrosszabb esetben, ha $b=2^k-1$ alakú, $k*2$ darab szorzással, ha $b=2^k$ alakú (ami a legjobb eseteket írja le), mindössze $k+1$ szorzás megadja az eredményt.

```
function iPower( a,b : long_integer ):long_integer;
{ Vissza: a^b }
var r:long integer;
begin
  r:=1;
  while b<>0 do begin
    if (b and 1) = 1 then r:=r*a;
    b:=b shr 1;
    a:=a*a;
  end;
  iPower := r;
end;
```

Általános esetben a szorzások számát a *b* felírásához szükséges bitek száma és a *b* felírásában lévő „1” bitek számának összege adja. Ez legalább *n+1*, legfeljebb *2*n,* átlagosan *3/2*n.* A számítás időigénye tehát a *b* bitjeinek a számával arányos és nem *b* nagyságrendjével. (A korábbi $e=2^{512}$ példánál maradva ez körülbelül 1500 szorzást jelent, ami – másodpercenként 1 millió szorzást feltételezve – 0,015 másodperc)

Természetesen ez az algoritmus még tökéletesen használhatatlan, mert csak az egyik problémát oldotta meg: az eredmény kiszámolása most már nem időkritikus. Azonban a számok tárolásának problémája még mindig nem megoldott. Szerencsére a feladat általában nem a^b értékének kiszámítása, hanem csak a^b *mod n* értékét kell meghatározni. Ha az előbbi programkódot a megfelelő helyeken kiegészítjük a moduláris aritmetika szabályai szerint, már majdnem kész is vagyunk:

```
function iMPower( a,b,n : long_integer ):long_integer;
{ Vissza: a^b mod n }
var r:long_integer;
begin
  r:=1;
  while b<>0 do begin
    if (b and 1) = 1 then r:=(r*a) mod n;
    b:=b shr 1;
    a:=(a*a) mod n;
  end;
  iMPower := r;
end;
```

Sajnos még nem nyújtózkodhatunk elégedetten, mert a rutinban szereplő szorzó, eltoló, maradékképző operátorokat nekünk kell megvalósítani úgy, hogy a `long_integer` típusú, 512…4096 bit hosszú egész számokat képviselő adatstruktúrákat kezelni tudja, magyarul meg kell írni az aritmetikát is. Ennek az alaparitmetikának a gyakorlati megoldásokban a következő műveleteket kell tudnia: összeadás, eltolás illetve az ezekre épülő kivonás, szorzás, hatványozás és maradékképzés. Az összeadás, kivonás nem valószínű, hogy bárkinek is problémát okozna. A hatványozás egyik lehetséges megoldását pedig az előbb láthattuk. A szorzást eltolások és összeadások sorozatával – a „papír és toll" megoldáshoz hasonlóan – oldhatjuk meg. De érdekességképpen lássunk egy olyan szorzóalgoritmust, amely a szokásos t^2 helyett mindössze $t^{1,58}$ összeadással oldja meg a szorzás feladatát. (Ahol t a számábrázoláshoz használt bitek száma, *lásd 31. ábra*)

Karatsuba – Ofman szorzás

A most leírt rekurzív algoritmust 1962-ben publikálták a névadó orosz matematikusok. További részletek Knuth könyvéből tudhatók meg. [4] A feladat $a*b$ szorzat kiszámítása, ahol a és b egyaránt k bites egész szám. A k legyen kettő valameny hatványával egyenlő. Ez nem igazán jelent megkötést, úgyhogy elfogadhatjuk.

1. Első lépésben bontsuk fel az a és a b számokat két egyenlő méretű részre:

   $$a=2^h a_1+a_0 \quad és \quad b=2^h b_1+b_0$$

   ahol a_1 a magasabb helyértékű biteket tartalmazza, és a_0 az alacsonyabbakat. A b_1 és b_0 hasonló tartalmú. Mivel k páros és a részek egyenlők, ezért $h=k/2$.

2. Szorozzuk össze a számokat:

   $$\begin{aligned} t &= ab \\ t &= (2^h a_1+a_0)(2^h b_1+b_0) \\ t &= 2^{2h} a_1 b_1 + 2^h (a_1 b_0 + a_0 b_1) + a_0 b_0 \\ t &= 2^k t_2 \quad + 2^h t_1 \quad + t_0 \end{aligned}$$

3. Az eredeti egy darab k bites szorzást felbontottuk négy darab h bites szorzásra. Ezt az eljárást rekurzívan tovább folytatva egyre kisebb számokra vezetjük vissza műveleteket. A fenti algoritmust nevezzük el *Standard Rekurzív Szorzó Algoritmusnak (SRSA)* és kódban a következőképpen néz ki:

```
function SRSA(a,b)
   t0:=SRSA(a0,b0)
   t2:=SRSA(a1,b1)
   u0:=SRSA(a1,b0)
   u1:=SRSA(a0,b1)
   t1:=u0+u1
return ( [t2 << k] + [t1 << h] + t0 )
```

Ha két k bites szám összeszorzásához szükséges műveletek száma $O(k)$, akkor az *SRSA* lépéseinek száma körülbelül

$$O(k)=4O(k/2)$$

Azért csak körülbelül, mert a rekurzió adminisztrálása, az összeadások, a balra-léptetések mind valamennyit hozzátesznek a feldolgozás idejéhez. Ez a rekurzió $O(1)=1$ feltétellel a várt $O(k)=k^2$ eredményt adja. A *Karatsuba-Ofman* szorzó algoritmus (*KOSA*) olyan, mint az *SRSA*, csak másként számol: három részből építi fel az eredményt és nem négyből.

1. A számokat az előbbiekhez hasonlóan két részre bontja.
2. Majd kiszámolja a t_i részeredményeket:

$$t_0 = a_0 b_0$$
$$t_2 = a_1 b_1$$
$$t_1 = a_1 b_0 + a_0 b_1 = (a_1+a_0)(b_1+b_0) - t_0 - t_2$$

   A t_1 kiszámolása látszólag bonyolultabb, mint az *SRSA* esetében, ahol kettő szorzás és egy összeadás kellett, itt viszont öt művelet van: egy szorzó és négy összeadó. Azonban ne felejtsük el, hogy egy k bites összeadást körülbelül k-szor gyorsabban el lehet végezni, mint egy k bites szorzást.

3. Ezekkel a részeredményekkel és számításokkal az algoritmus kódja a következőképpen néz ki:

```
function KOSA(a,b)
   t0:=KOSA(a0,b0)
   t2:=KOSA(a1,b1)
   u0:=KOSA(a1+a0,b1+b0)
   t1:=u0-t0-t2
return ( [t2 << k] + [t1 << h] + t0 )
```

Ha két k bites szám összeszorzásához szükséges műveletek száma $O(k)$, akkor az *KOSA* lépéseinek száma körülbelül

$$O(k)=3O(k/2).$$

Ez a rekurzió $O(1)=1$ feltétellel nagyjából a $O(k)=k^{\log_2 3}=k^{1,58}$ eredményt adja. Lényeges, és néha kellemetlen különbség azonban a *SRSA*-val szemben, hogy amíg ott csak k bites számok $2k$ bites szorzatát kell kiszámítani, addig a *KOSA* algoritmusban $k+1$ bites számok $2k+2$ bites szorzatát is ki kell tudni számítani. A rekurziót akkor kell leállítani, amikor a számok mérete eléri azt a bitméretet, amivel a futtató architektúrán már könnyen lehet szorzást számolni. A rekurzió adminisztálása, a verem kezelése, a kiegészítő – előkészítő – műveletek igen sok időt visznek el, így a szép eredmény egyúttal elméleti is. Azonban minnél hosszabb számokkal dolgozunk, annál inkább megéri áttérni a *KOSA* algoritmusra a hagyományos *eltol-összead* módszerről. A tapasztalat azt mutatja, hogy az áttérésnek körülbelül $k=250$ *bit* felett van értelme.

4.7. GYAKORLATI PROBLÉMÁK

A korábban bemutatott Diffie-Hellman algoritmus akkor segít, ha egy szimmetrikus titkosításhoz közös kulcsot kell egyeztetni. A nyilvános kulcsú módszereknél természetesen nincs ilyen probléma, de van másik. A kulcsokat egy központi adatbázisban, az úgynevezett *kulcsszerveren* tárolják, ahonnan letölthető a címzett nyilvános kulcsa, ha az nekünk nincs meg, és küldeni szeretnénk neki valamit. Hasonlóan ahhoz, amikor a telefonkönyvben kikeresünk egy telefonszámot. A kliens-szerver architektúra azonban nem kötelező, saját nyilvános kulcsát a címzett maga is elküldheti (nem csökkentve ezzel a hitelesség kérdését).

4.7.1 Hitelesség

A szerver, amikor elküldi nekünk a kért kulcsot, titkos kulcsával aláírja vagy titkosítja azt. Így az megérkezéskor a szerver nyilvános kulcsával ellenőrizhető, és nem fordulhat elő, hogy a támadó a saját kulcsát küldi el nekünk a szerver nevében. (Hiszen a támadó nem ismeri a szerver titkos kulcsát, így sem titkosítani, sem aláírni nem tud a nevében.) Biztosítani kell az adatbázis hitelességét és megbízhatóságát, hiszen ha a címzett neve mellett nem a saját kulcsa van, hanem az üzenetet lehallgatni kívánó személy kulcsa, az egész nem ér semmit, hiába írja alá a szerver. Sőt, a szerver aláírása ilyen esetekben csak a hamis biztonság érzetét kelti.

A valódi adatbázis

Felhasználónév	***Nyilv. Kulcs***
Szabolcs	(n_s, e_s)
Kornél	(n_k, e_k)

A hamis adatbázis

Felhasználónév	***Nyilv. kulcs***
Szabolcs	(n_s, e_s)
Kornél	(n_t, e_t)

A *kulcsok hitelessége* – a kulcs és a valódi tulajdonos azonosságának, összetartozásának biztosítása – még mindig fő probléma, és a *nyilvános kulcsú rendszerek legsebezhetőbb pontja*. Nem beszélve arról, hogy a szerver kulcsát is hitelesítenie kell valakinek. Meg annak a kulcsát is és így tovább. Az ilyen kulcshitelesítő szerveket, melyek ellenőrzik és igazolják egy nyilvános kulcs és a tulajdonos összetartozását, *Certification Authory* (*CA,* a PGP-terminológiában *trusted introducers*)-nak nevezzük, míg a tulajdonos-kulcs összetartozást igazoló és a *CA* által kibocsátott igazolást hitelességi bizonyítványnak (*certificate*). Az ellenőrzés első lépése az lehet, hogy a hitelesítést kérő a saját nyilvános kulcsát aláírja a saját titkos kulcsával. Ez egyrészt biztosítja, hogy a tulajdonos elismeri, hogy a kulcs az övé, másrészt bizonyítja, hogy a nyilvános kulcs titkos párja valóban a tulajdonos birtokában van (*self-signed key*). Ebben a hierarchikus felépítésben (*hierarchial trust*) legalább egy olyan személy – vagy szervezet – lesz, akinek nem lesz hitelesített kulcsa: annak, aki az egész alá- és fölérendeltséget jelképező fa csúcsán van (*root of certification tree,* a PGP-terminológiában *meta-introducer*). Nincs senki, aki az ő kulcsát aláírhatná, így nyilvános kulcsa csak közvetett módon ellenőrizhető.

A kulcsok hitelesítésére más megoldás is használatos – főként a privát szférában, de már az üzleti megoldások is e felé haladnak – ahol a szereplők nem alá- és fölérendeltségi viszonyban vannak egymással: ez a „*bizalmi háló*” elv. Ha egy olyan személy által aláírt kulcsot kapunk, akiben megbízunk, megbízhatunk az általa aláírt kulcsban is. Az ilyen bemutatott

kulcs aztán újabb kulcsokat hitelesíthet. Sőt, saját kulcsunkkal mi is hitelesíthetünk kulcsokat azon partnerek felé, akik megbíznak a mi kulcsunkban és döntéseinkben. Szabályozhatjuk, milyen mélységben fogadunk el bemutatásokat, és hogy hány hitelesnek ismert bemutató aláírása kell ahhoz, hogy egy nyilvános kulcsot hitelesnek ismerjünk el. Bárki bárkinek a kulcsát hitelesítheti. Azt, hogy a hitelesítést elfogadjuk-e, csak rajtunk múlik. A bizalmi elvekről, hitelesítési hierarchiákról lásd még a *8.4.2. A hitelesség* című fejezetet.

Ha van egy olyan személy vagy szervezet, akinek az aláírását egy adott körben mindenki hitelesnek ismeri el, megoldható az is, hogy ez a személy aláírásával kulcsokat és dokumentumokat rendszeresen hitelesítsen. Ez a személy lesz az *elektronikus közjegyző (public e-notary).* Az általa hitelesített dokumentumokat nemcsak a polgári életben, hanem az államigazgatásban és a jogrendszerben is használhatjuk, ha megbízhatóságát ezen a területeken is elismerik.

4.7.2. Érvényesség

A kulcsok kezelésének másik problémája az érvényesség. Minden kulcskezelő rendszer lehetővé teszi, hogy saját kulcsait valaki érvénytelennek nyilvánítsa, ha azokat például eltulajdonították tőle. Minden érvénytelenítés az adott kulcsra esetlegesen kiadott hitelességi bizonyítványok visszavonását (*revoking of certification*) is eredményezi, és érvénytelenné teszi az e kulccsal aláírt bizonyítványokat is. Ha felmerül a gyanúja annak, hogy kulcsainkat valaki ellopta, az érvénytelenítést mindenképpen el kell végeznünk, mert amíg ezt nem tesszük meg, az illetéktelen felhasználó a nekünk szánt üzeneteket elolvashatja, és – ami talán a legfontosabb – bármit aláírhat a mi kulcsunkkal, ez pedig okirathamisítást jelent. Az érvénytelen kulcsot nem szabad a nyilvántartásból törölni, mert egy olyan dokumentum aláírásellenőrzésekor, ami még az érvénytelenítés előtt keletkezett, szükség lehet rá. Az ilyen kulcsokat egy „érvénytelenségi listán" kell tárolni, itt mindenki ellenőrizheti, hogy az általa használni kívánt kulcs érvényes-e. Hasonló lista tartalmazhatja a visszavont hitelességi bizonyítványokat is (*Certificate Revocation List, CRL*). Egy érvénytelen hitelességi bizonyítvány azonban nem jelenti egyértelműen azt, hogy a hozzátartozó kulcs kompromittálódott. Lehet, hogy csak határozott időre szólt az igazolás és emiatt járt le (*expired certification*). Összegezve: ha egy nyilvános kulcsot titkosításra vagy aláírásellenőrzésre használunk, győződjünk meg arról is, hogy a kulcs (vagy a hozzátartozó bizonyítvány) nem hamis, nem járt le, és nincs érvénytelenítve sem.

A titkos kulcsok biztonságos tárolása ugyanolyan fontos, mint a szimmetrikus rendszerekben, sőt a digitális aláírások jogkövetkezményei miatt talán még fontosabb. Ezért közös használatú gépet körültekintően, lehetőleg NE használjunk! Ha ez elkerülhetetlen, a titkos kulcsokat tároljuk lemezen, és mint egy igazolványt hurcoljuk magunkkal. (Azért tartsunk másolatot róla, a floppy nem a legbiztonságosabb eszköz, egy USB kulcs már jobb megoldásnak tűnik.) Ebben az esetben nekünk kell gondoskodni a kulcsok adathordozóra való exportálásáról és rendszerbe történő importálásáról, sőt a privát kulcs rendszerből való törléséről is. Az igazi megoldást az intelligens kulcstároló eszközök jelentik, hiszen azok az életük árán is megvédik az általuk generált vagy beléjük plántált titkos kulcsot. Erről lásd: *13.3. Biztonságos kulcstároló eszközök* fejezetet.

4.8. GYAKORLATI ALKALMAZÁSOK

Ebben az alfejezetben két gyakorlati alkalmazást mutatok be vázlatosan: SSL és mobilBank. Az elsőt már régóta használhatjuk biztonságos kommunikációra nyilvános hálózaton, a második nem túl régi terület, és jelentősen túlmutat az informatikai felhasználáson, a mindennapjaink része lehet, bárkié, aki rendelkezik mobiltelefonnal.

4.8.1. Hálózati adatforgalom nyilvános hálózaton

A nyilvános kulcsú algoritmusok megfelelő implementációban alkalmasak olyan hálózati kapcsolatok titkosítására is, amelyek eredetileg nem képesek titkos forgalom lebonyolítására (*unsecure channel* → *secured channel* ≠ *secure channel*).

Egy hálózaton az általunk generált forgalmat sokan olvashatják. Ez egy Ethernet hálózatban könnyen belátható, hiszen – koax kábel esetén – szó szerint ugyanazon a vezetéken lóg mindenki. Ezen alapul a „broadcast" üzenet is, mindenki hallja és veszi, de csak az válaszol, akinek kell. Számtalan olyan program létezik, mely képes figyelni a hálózati forgalmat statisztikai mérésekre (*ipmon*), illetve képes a lehallgatott forgalmat átalakítani ember által is könnyen értelmezhető formába (*Netmon, ShadowScan, SpyNet*). Ez utóbbi programok nemcsak elkapni tudják az Ethernet kereteket, hanem tartalmuk szerint fel is tudják dolgozni, így megkülönböztethetik a IPX, az IP stb. hálózati (*layer-3 szintű*) protokollokat, sőt például IP felett értelmezni tudják a különböző alkalmazásprotokollokat is (POP3, TELNET, stb.)[35]. Hogy egy-egy hálózati csomag hány másik munkaállomáshoz jut el, az a hálózat fizikai és logikai architektúrájától, valamint az alkalmazott hálózati eszközöktől függ.

Az Interneten a fentiek nem egészen igazak. Ennek egyik oka, hogy az Internet különféle fizikai felépítésű hálózatokat kapcsol össze, de a helyi hálózat fizikai forgalma általában a helyi hálózaton belül marad. Az összekapcsolt hálózatokban a fizikai réteg (*physical layer, layer-1*) és az adatkapcsolati réteg (*datalink layer, layer-2*) ritkán közös. Ezzel szemben az OSI harmadik rétegének, a hálózati rétegnek (*network layer, layer-3*), a hálózati protokollok rétegének forgalma már látható lehet más hálózatok és munkaállomások számára is a hálózat aktív eszközeinek beállításától függően. Sőt gyakran kell is, hogy látható legyen, hiszen nem tudnánk egy távoli szervert megszólítani, annak szolgáltatásait használni. Az ilyen módon vándorló adatokat a hálózati eszközök és a protokoll-stackek hol feldarabolják, hol összeillesztik, hol továbbítják, hol meg eldobják, de a lényeg az, hogy számtalan helyen hozzáférhető az eredeti adatcsomag és annak tartalma.

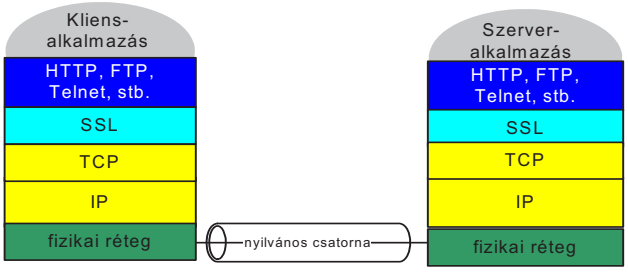
[35] Ez az alfejezet feltételez bizonyos hálózati ismereteket: OSI referencia-modell, hálózati és alkalmazás-protokollok közötti különbség, Ethernet hálózat, stb ismeretét. Aki ebben egyáltalán nem járatos, annak ajánlom a [2]-es irodalmat, ahol alapos magyarázatot találhat vagy két lapozással ugorjon a következő alfejezetre, ami a 4.8.2. Mobilbank szolgáltatások címet viseli.

Ha egy ilyen hálózati környezetben titkosítani szeretnénk, a felek valamilyen közös titkosító algoritmus használatával beszélgethetnek. A kérdés az, hogy a titkosítás hova épüljön be?

- Ha egy alkalmazáshoz vagy szolgáltatáshoz kötjük a titkosítást, akkor minden adminisztrációs műveletet a szolgáltatásnak kell végezni. Ha új szolgáltatás kerül a szerverre, azt külön fel kell készíteni a titkosítás használatára. Régi szolgáltatások nem lesznek képesek a biztonságos kommunikációra, ha arra eredetileg nem készítették fel őket. Ebben az esetben a titkosítás művelete az alkalmazásokhoz kötődik és független a hálózati forgalomtól.
- Ha a titkosítást nem az alkalmazásokhoz, hanem a hálózati protokollokhoz kötjük, a titkosítás alkalmazásfüggetlen lesz, lehetőséget teremtve arra, hogy olyan alkalmazások is használhassák a titkosítás szolgáltatásait, melyek eredetileg nem voltak felkészülve erre. Az alkalmazásoknak emiatt nem is kell mindig tudniuk, hogy ők egy titkosított csatornán beszélgetnek egymással.

Az SSL (TLS)

Ez utóbbi megoldást választották az *SSL* tervezői is. Az *SSL* (*secure socket layer,* vagy újabb nevén *TLS - transport layer security*) napjaink igen elterjedt, titkosított kommunikációt biztosító protokollja, amely nyílt hálózatokban, kapcsolatorientált (tehát nem broadcast) kommunikációban nyújt védelmet. Eltérően az olyan protokolloktól, mint például az *IPSec*, amely az egész hálózatot teszi biztonságossá, az *SSL* csak egy-egy kommunikációs csatornát biztosít. Az *SSL* a *protokoll-stackben* az alkalmazásréteg „alá" és a szállítási réteg fölé került, önmaga is egy protokollréteget alkot. Kliens- és szerveroldalon egyaránt szükség van támogatásra, hiszen hiába kiabál a szerver, hogy ő titkosított kapcsolatot szeretne kezdeményezni, ha erről a kliens nem vesz tudomást. Ez természetesen visszafelé is igaz. Ha ez biztosítva van, az e felett futó alkalmazások szempontjából az *SSL*-réteg átlátszó, bármilyen alkalmazáshoz használható.

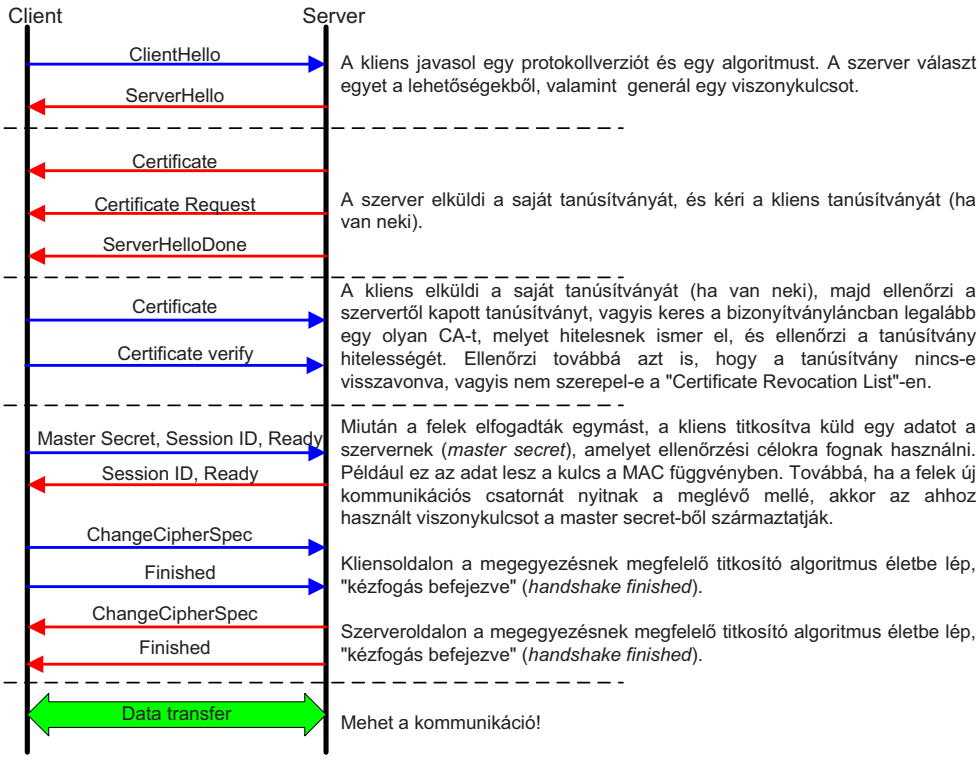


32. ábra Biztosított nyilvános csatorna – kicsit egyszerűsítve

Az *SSL* egyik legelterjedtebb alkalmazását a HTTP-protokoll egy kiterjesztése jelenti, a https://, amely biztonságos kommunikációt tesz lehetővé a webszerver és a kliense között. A https:// használatához olyan webszerver és olyan böngésző kell, amely támogatja az *SSL* használatát, azonban sem a kliensoldali, sem a szerveroldali alkalmazásoknak (például szerver vagy kliensoldali szkripteknek) nem kell tudniuk az őket támogató biztosított csatornáról. Az *SSL* minden egyes kapcsolatot egyedi kulccsal titkosít, függetlenül attól, hogy több kommuni-

kációs csatorna is végződhet ugyanannál a kliensnél. Sőt, a szerver→kliens, kliens→szerver irányok is más-más kulccsal kerülnek titkosításra. Az *SSL* azért ennél jóval több, mert a nyílt-kulcsos technológia – szinte – minden lehetőségét kihasználja:

- Minden egyes kommunikációs kapcsolat (*session*) titkosításához rövidéletű véletlen kulcsot használ. A véletlen kulcsot a szerver titkos kulcsával titkosítva küldi el a klienshez, a tényleges kommunikáció megkezdése előtt.
- Tanúsítvány igazolja a szervert (X.509). A tanúsítványban lévő nyilvános kulccsal a kliens kibonthatja a viszonykulcsot, ha a tanúsítványt rendben lévőnek találja. A kliens tanúsítványának alkalmazása általában nem kötelező, bár a szerver előírhatja.
- Azzal a szimmetrikus algoritmussal tikosítja a kliens és a szerver közötti adatforgalmat, melyben a felek megegyeznek. Ez a *DES, 3DES, TripleDES, RC2, RC4, IDEA, AES* algoritmusok valamelyike lehet.
- Biztosítja az adatintegritást (*MD5, SHA-1*), melyet kulcsolt *MD* függvényekkel ér el (*MAC*).



33. ábra Az SSL alapvető lépései

4.8.2. Mobilbank szolgáltatások

Hazánkban 2002. októberében vezette be a „*Mobilbank*” szolgáltatást a mobilpiac és a bankszféra két nagy szereplője. A kampány két jelmondata, „*Pénzügyek – saját kezűleg!*” és „*Tartsa kézben pénzügyeit!*” jól kifejezik a szolgáltatásban rejlő lehetőséget: az ügyfél – szerződéskötés után – maga intézhet bizonyos banki megbízásokat. A Telebank szolgáltatásoktól eltérően itt nemcsak arról van szó, hogy az ügyfél telefonon a bank egyik alkalmazottjával beszélve megbízást ad, hanem saját maga indítja a banki műveleteket (egyenleglekérdezések, eseti átutalások, betétlekötések és -feltörések, limitlekérdezések és -beállítások, árfolyaminformációk, stb.). Mégpedig saját mobiltelefonján keresztül, számítógép és internetelérés nélkül.

A mobiltelefon és a mobil kapcsolat egyébként is igyekszik védeni magát és a felhasználókat a nemkívánatos résztvevők ellen: egy-egy hívás felépítése előtt többlépcsős identifikáció történik a SIM kártya és a központ között, a hívás ideje alatt pedig titkosítva közlekednek az adatok (A5/1 és A5/2). A jelenlegi mobiltelefonok digitális, programozható voltuk miatt egyébként is alkalmasak – kisebb teljesítményű – információfeldolgozásra.

A technológia szempontjából ehhez „mindössze” a két végpontot kell felkészíteni, vagyis:

- a felhasználónak el kell tudni küldenie a kérését úgy, hogy az számára és a bank számára egyformán biztonságos legyen, illetve
- a banknak tudnia kell fogadni, ellenőrizni és feldolgozni a kéréseket.

A felhasználók számára alapvetően két felület szolgálhat az ügyintézésre: a WAP és a készülék menüjében megjelenő alkalmazás. A WAP előnye, hogy nem függ a mobilszolgáltatóktól, bármelyiknél is vagyunk, igénybe tudjuk venni. Ebben az esetben a telefonban lévő WAP-böngésző és a WAP-szerver közötti, WTLS alapú együttműködés biztosítja a megbízhatóságot, hasonlóan a HTTP-böngészőkben megszokott TLS-protokollhoz. A banki tranzakciókat egyébként is többszintű védelem védi: jelszó, titkosítás, napi limit, stb.

Amennyiben a készülék menüjébe beépülő lehetőségeket kívánjuk használni, az elküldött üzenet kezelése és biztonsága már nem ennyire egyértelmű. Nem is lehet akármilyen kártyával igénybe venni ezt a szolgáltatást, speciális funkciókkal ellátott SIM kártya szükséges, amely képes „alkalmazások” futtatására (SIMToolkit v2 támogatás). Egy ilyen alkalmazás kezeli a Mobilbank számára elküldött üzeneteket is:

- Az üzeneteket, azok elküldése előtt digitálisan aláírja (RSA-1024).
- Majd a SIM kártyán található TDES-kulccsal az aláírt üzenetet titkosítja.
- Így kerül elküldésre az aláírt, titkosított üzenet a bank számára, ahol megtalálható a SIM kártyához tartozó TDES-kulcs, illetve az RSA nyilvános kulcs.
- Először a bank ellenőrzi, hogy az adott számnak van-e egyáltalán szerződése.
- Ha igen, előkeresi a TDES-kulcsot, amivel megfejti az üzenetet.
- Ezek után ellenőrzi az aláírást.
- Ha mindez sikeres, ellenőrzi az ügyfél által megadott „telebank azonosítót”.
- Ha az ellenőrzés itt is sikeres volt, feladja a tranzakciót a banki rendszernek, amely a megfelelő adatok esetén végrehajtódik.

A kulcskezelés a szokásostól eltérően valósul meg, ugyanis nincs az eddigi értelemben vett kulcscsere, kulcselosztás. A felhasználó nem maga generálja a kulcsait, sőt hozzá sem fér azokhoz, ugyanis az RSA kulcspárt a gyártás során a SIM kártya gyártója generálja. Ez a folyamat a bankkártya gyártással megegyező biztonsági minősítéssel ellátott folyamat. A kulcsokat a gyártó nem tárolja el, sőt az ügyfél sem ismeri saját privát kulcsát, mert a kulcsot nem lehet kiolvasni. A publikus kulcsot (még néhány adattal együtt) a regisztrációs SMS-ben küldi el a SIM kártyáról a felhasználó, egy „egyszerű" SMS-ben. Mivel itt publikus kulcsról van szó, nem kell biztonságossá tenni ezt az utat, főként, hogy maga az SMS – a feladó telefonszámával együtt – „tanúsítványként" működik, és a mobilforgalom is védett.

A TDES-kulcs elhelyezése a SIM kártyán szintén a gyártó feladata, amit a kártya különböző egyedi paraméterei alapján generál. A bank egy erre alkalmas speciális eszköz (*HSM – Hardware Security Module*) segítségével a regisztrációs üzenet beérkezésekor annak adatai alapján rekonstruálja a kulcsot (az SMS természetesen tartalmazza a SIM kártya megfelelő adatait is). A TDES-kulcs küldésére így nincs szükség, az ügyfél implicit módon maga küldi el a banknak.

> A HSM modul a kulcsok biztonságos generálását és tárolását biztosítja. Fizikailag védett eszköz (akár PCI-kártya). A digitális aláírás vagy annak ellenőrzése a védett modulban történik. Ha a fizikai védelem megsérül, a modul tartalma megsemmisül. További általános jellemzője, hogy valódi véletlenszám-generátort tartalmaz és FIPS142-1-level 3 minősítésű. Mindennek persze megvan az ára is, kb. 2 000 000 Ft-nál kezdődik. (2003. januári adat)

Az állandó TDES-kulcs használata nem szerencsés, különös tekintettel arra, hogy a megfelelő adatok birtokában rekonstruálható. Arra az esetleges kérdésre, hogy az algoritmus miért nem a biztonságosabbnak tekinthető viszonykulcsos megoldást használja, nem találtam választ.

A Függelék jelen fejezethez kapcsolódó alfejezetei

14.4. Moduláris aritmetika nagyon dióhéjban

14.5. A kis Fermat-tétel bizonyítása

14.6. Euler-féle φ függvény

14.7. Hibrid kriptorendszer digitális aláírással, viszonykulccsal – logikai vázlat

14.8. Szabványok összefoglaló táblázata

14.9. Pollard-ρ algoritmus – UBASIC implementáció

14.12. Néhány szám és nagyságrend

14.13. Moore törvénye

További kiegészítések folyamatosan bővülő helye a http://www.netacademia.net/konyv webcím.

Kriptográfiai termékek illegálisnak nyilváníthatók,
de az információ soha.

Bruce Schneier
Counterpane Inc.

5. Elliptikus görbék

Egy jó ideje egyre több helyen találkozhatunk az ECC betűhármassal, mint egy kriptorendszer megjelölésével, nevével. Egyre több előnyéről hallunk: az RSA-nál rövidebb kulcsokkal érhető el ugyanakkora biztonság, sokkal gyorsabb a működése, kisebb a memóriaigénye. Mindezek a Smart Card technológia biztonsági eszközeit is az ECC felé fordítják. Mi is ez? *Elliptic Curve Cryptosystem*, bár gondolom ettől most sokan nem lettek okosabbak. Nekik (is) szól a következő fejezet, amelyben előbb megtanulnunk összeadni!

Az elliptikus görbékről egyre gazdagabb irodalommal és egyre több ismerettel bírunk. A Certicom szerint az elliptikus görbéket, mint algebrai és geometriai elemeket az elmúlt 150 évben behatóan tanulmányozták. Ezeken a tanulmányokon alapul a gazdag és mély ismeretek tárháza. Tekintve, hogy a Certicom Corporation az ECC egyik vezéregyénisége, sajnos ez a megállapítás csak féligazság, mert az igaz, hogy az elliptikus görbék régóta ismertek, azonban kriptográfiai szempontból csak ~15 éve vizsgálják őket. Az elliptikus görbék kriptográfiai alkalmazását – egymástól függetlenül – két kutató is javasolta: először *Neal Koblitz* (University of Washington) 1985-ben, majd tőle függetlenül *Victor Miller* (IBM). A jelenleg használt PKI rendszerek szinte mindegyike a következő három probléma valamelyikének megoldási nehézségén alapul:

- faktorizálás – IFP (~1970-től, *integer factorization problem*)
- diszkrét logaritmus – DLP (~1970-től, *discrete logarithm problem*)
- diszkrét logaritmus az elliptikus görbék felett – ECDLP (~1985-től, *elliptic curve discrete logarithm problem*)

Sajnos jelenleg senki sem tudja biztosan, hogy e három probléma elég erős-e, de ennek ellenkezőjét sem bizonyította még senki. Jelenleg csak olyan megoldóalgoritmusokról beszélhetünk, amelyek „napjaink legjobb algoritmusai”, de nem biztos, hogy elvileg is a legjobbak. Mindenesetre – figyelembe véve a ma ismert támadási módokat – az IFP megoldására ma ismert legjobb algoritmus hatékonysága messze leköröz a ma ismert legjobb ECDLP megoldóalgoritmus hatékonyságát.

NIST javaslata az egyes kriptorendszerek kulcshosszának összehasonlítására:

ECC modulus	AES	RSA modulus	RSA:ECC
112	56	512	5:1
161	80	1024	6:1
256	128	3072	12:1
384	192	7680	20:1
512	256	15630	30:1

A jobb oldali táblázatban három kriptorendszer általánosan vagy szabvány szerint használt kulcsméreteit láthatjuk. Az egy sorban lévő kulcsméretek közel azonos biztonságot nyújtanak. Látható, hogy a két legismertebb nyíltkulcsos algoritmus (RSA és ECC) kulcsméretei – azonos biztonság mellett – „köszönőviszonyban” sincsenek egymással... (Az RSA és az ECC mérési eredményeken alapuló összehasonlítását lásd például [URL55]-ben.)

Az a tény, hogy az ECC sokkal kisebb kulcsmérettel is megfelelő biztonságot nyújt, a következőket vonja maga után:

- gyorsabb algoritmusok (Azonban az RSA-aláírás ellenőrzése és az RSA-titkosítás szinte verhetetlen, ha kis nyilvános exponenst használunk, például *e=65537*!)
- kevesebb továbbítandó és kezelendő adat
- kisebb tárigény a kulcsok tárolásához
- kisebb tanúsítványok.

Mindezen vélt vagy valós előnyök miatt nemcsak a kutatók foglalkoznak az elliptikus görbékkel, hanem az üzleti élet számára szolgáltatásokat nyújtó cégek is. Ez eggyel több ok, hogy mi is megismerjük legalább az alapvető fogalmakat és módszereket.

A következő alfejezetekben az elliptikus görbéket először a valós számok halmazán definiáljuk és megnézzük az értelmezett műveleteket, a műveletek származtatását, esetenként geometriai jelentését is. Végül néhány kriptográfiai algoritmussal is megismerkedünk, melyek az elliptikus görbéken alapulnak. Ne ijedjünk meg a sok ábrától és képlettől, jóval egyszerűbb, mint amilyennek elsőre tűnik! (Ettől függetlenül sajnos igaz, hogy az ECC matematikailag jóval bonyolultabb, mint az RSA vagy a Diffie-Hellman, ezért sokkal nehezebb megérteni és elmagyarázni vagy bizonyítani, igazolni a felhasználók felé... Ha valaki a fejezet végére ér, és úgy érzi, hogy egy kukkot sem értett meg belőle, nyugodtan lapozzon vissza és kezdje elölről!)

5.1. VALÓS SZÁMOK HALMAZÁN JÁRVA

5.1.1. A görbe

Jelöljünk ki a koordinátasíkon egy **P**(x,y) pontot! Ha a koordináták kielégítik az alábbi egyenletet, akkor a **P**(x,y) pont az elliptikus görbe egy pontja.

$$y^2 = x^3 + ax + b$$

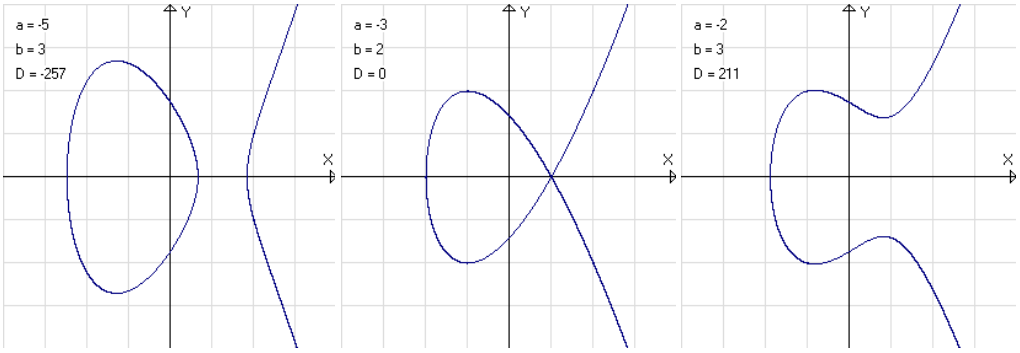
Az elliptikus görbéknek még egy – definíció szerinti – pontjuk van: a végtelenben lévő pont, melyet „**O**” betűvel jelölünk. Nem keverendő össze a valós számok végtelenjével, ami megszámlálhatatlanul sokat jelent, míg az „**O**” pont inkább mérhetetlenül messze van. És ezekkel a pontokkal fogunk dolgozni! A velük végzett műveletek adják az elliptikus görbéken alapuló kriptorendszerek elemi műveleteinek nagy részét.

> Talán sokaknak feltűnt, hogy az ellipszis másodfokú görbe, ez az egyenlet viszont harmadfokú. Nos, az elnevezés közvetett: egy általános ellipszis kerületét egy ún. elliptikus integrál adja meg. Az elliptikus integrálok általában nem fejezhetők ki elemi függvényekkel, de elvezetnek a fenti egyenletet kielégítő (x,y) pontok, így az elliptikus görbék tanulmányozásához.

Az *a* és *b* paraméterek változtatásával újabb és újabb görbéket definiálhatunk, de nem mindegyik felel meg nekünk. Vannak olyan (*a,b*) paraméterpárosok, amelyekre igaz, hogy

$$D = 4a^3 + 27b^2 = 0 .$$

Az ilyen görbék vagy elmetszik magukat, vagy csúcsban végződnek. Most nem használjuk ezeket, nem tekintjük őket elliptikus görbének (szinguláris görbék). Az alábbi három ábrán három görbét láthatunk azokra az esetekre, amikor $D<0$, $D=0$ és $D>0$.



34. ábra Elliptikus görbék a paraméterek függvényében

Az első és utolsó görbe jó lesz a munkánkhoz, ezek közös jellemzője, hogy $4a^3 + b^2 <> 0$

5.1.2. Műveletek a görbe pontjaival – geometriai megközelítésben

A görbék pontjaival mindössze három elvégezhető műveletet fogunk definiálni, lássuk most őket egyesével, részletesen!

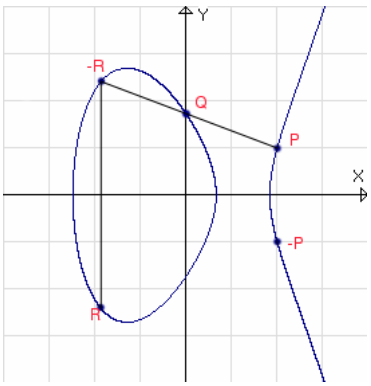
Előjelváltás – ellentett képzése

Egy **P**(x,y) pont ellentett párja **R** = **–P** nem más, mint a pont x tengelyre tükrözött képe, amely szintén rajta lesz a görbén: **R**(x,-y)

Összeadás

Legyen a görbe két különböző pontja **P** és **Q**! E két pont összegét jelölje **R**, vagyis **R=P+Q**. A műveletet a következőképpen kell elvégezni:

1. Kössük össze a **P** és **Q** pontot egy egyenessel!
2. Az egyenes egy harmadik pontban metszi a görbét, ez a pont lesz **–R**.
3. E pont x tengelyre tükrözött képe az előjelváltás szabálya szerint szintén rajta lesz a görbén és ez lesz az eredmény **R** pont.

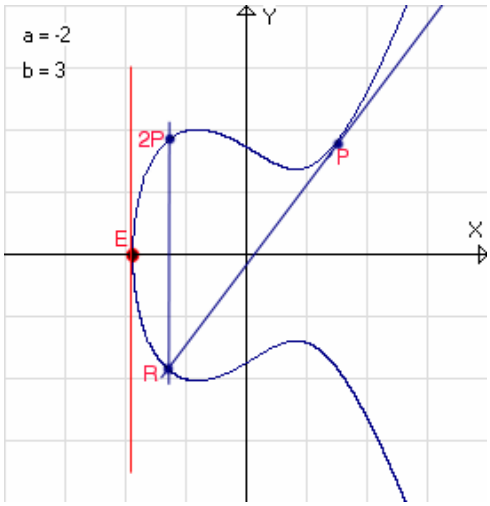


P + (–P) = ?

Ha eddig egyszerűnek tűnik, akkor most próbáljuk meg **P** és **–P** pontot összeadni! Mivel **P** és **–P** egymás tükörképei az x tengelyre nézve, a rajtuk keresztül húzott egyenes párhuzamos az y tengellyel és sajnos nincs olyan harmadik pont, amiben metszené a görbét. Az iménti ábrán látható még egy ilyen pontpáros (**–R** és **R**): az őket összekötő egyenes szintén párhuzamos az y tengellyel és hiába hosszabbítanánk meg bármelyik irányba, nem fogja harmadik pontban metszeni a görbét. Ezért **P** + (**–P**) nem számítható ki az összeadás módszerével, de szerencsére van nekünk egy **O** pontunk, amely a végtelenben van. A párhuzamos egyenesek a végtelenben metszik egymást, így definíció szerint **P + (–P) = O**. Ebből következik még, hogy elliptikus görbén értelmezve **P + O = P**. Hasonló azonosság a valós számok körében is van: $x * 1 = x$.

2P = ?

Egy pontot az előzőek mintájára adhatunk össze saját magával. Ha **P** és **Q** pontokat végtelenül közel visszük egymáshoz, akkor **P**=**Q** lesz. A **P** és **Q** ponton keresztül húzott egyenes pedig a **P** pontba húzott érintővé válik. A folytatást pedig ismerjük: az érintő metszi valahol a görbét és a metszéspont tükörképe lesz a **P** pont kétszerese.



Sajnos itt is van kivétel (lásd az ábrán **E** pontot): ha a pont az x tengelyen szíveskedik tartózkodni, az érintő függőleges, nem metszi másik pontban a görbét. Ebben az esetben definíció szerint a pont kétszerese a végtelenben van, vagyis 2**E**=**O**. Ezek a pontok – melyeknek y koordinátája zérus – elég furcsán viselkednek, ha 2**E** után kiszámoljuk 3**E**, 4**E**, 5**E** stb. pontokat:

2**E** = **O**

3**E** = **E**+2**E** = **E**+**O** = **E**

4**E** = **E**+3**E** = **E**+**E** = **O**

5**E** = **E**+4**E** = **E**+**O** = **E** és így tovább.

5.1.3. Műveletek a görbe pontjaival – algebrai megközelítésben

Az előzőekben az elliptikus görbén értelmezett műveleteket geometriai eszközökkel mutattuk be, de ez nem túl gyakorlatias a digitális számítógépek számára. Kicsit nehézkes érintőket és húrokat húzkodni, x tengelyre tükrözni, de szerencsére a geometriai műveletek eredményét ki is tudjuk számolni...

Előjelváltás – ellentett képzése

Ebben semmi újdonság sincs, ha a pont **P**(x,y), akkor **R** = –**P**

$\mathbf{x_R = x_P}$ és $\mathbf{y_R = -y_P}$.

Összeadás

A levezetést mellőzve, csak a végeredményre hivatkozva: ha **P**(x_P,y_P) és **Q**(x_Q,y_Q) nem egymás ellentettje, akkor **R** = **P** + **Q**

$s = (y_P - y_Q) / (x_P - x_Q)$

$\mathbf{x_R = s^2 - x_P - x_Q}$ és $\mathbf{y_R = s(x_P - x_R) - y_P}$

P + (–P) = ?

Korábban láttuk, hogy ha a két összeadandó pont egymás ellentettje, a rajtuk keresztül húzott egyenes függőleges, ami **O** pontban „metszi" a görbét. Az egyenes függőleges voltát jól jelzi az is, hogy az *s* kiszámításához használt tört nevezőjében zérus van. (Figyeljük meg, hogy *s* a PQ egyenes meredeksége!)

2P = ?

Ha y_P nem zérus, vagyis a duplázandó pont nem az x tengelyen van (lásd előző oldal **E** pontját!), akkor **R** = 2**P**

$s = (3x_P^2 + a) / (2y_P)$

$\mathbf{x_R = s^2 - 2x_P}$ és $\mathbf{y_R = s(x_P - x_R) - y_P}$

(Itt *s* a **P** pontba húzott érintő meredeksége.)

5.2. A MODULÁRIS ARITMETIKA KÖZBELÉP

5.2.1. A görbe

A már megismert egyenletünket egészítsük ki egy kicsit! Ne a valós számokon értelmezzük, hanem a moduláris aritmetika szabályai szerint:

$$y^2 \equiv x^3 + ax + b \pmod p,$$ ahol *p* prím

Vagyis ha az egyenlet mindkét oldala ugyanazt a maradékot adja *p*-vel történő osztás után, a **P**(x,y) pont a görbe pontjai közé tartozik. Két további feltétel:

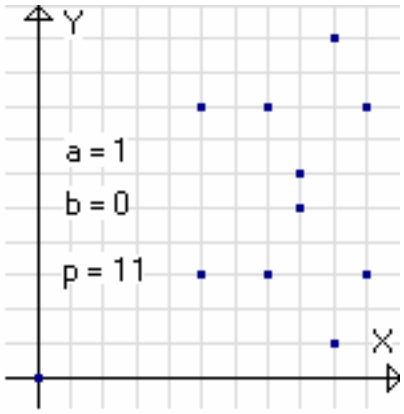
- $0 \le x \le p-1$ és $0 \le y \le p-1$
- valamint $4a^3 + 27b^2 \bmod p \neq 0$.

Igazság szerint *p* nem feltétlenül prím, bizonyos feltételek mellett akár összetett szám is lehet, de ez olyan speciális esetnek számít, amelynek megoldása sokkal egyszerűbb, mintha *p* prím lenne. Az ANSI X9.63 szabvány egyenesen kizárja az összetett *p*-vel definiált görbéket a kriptográfiai alkalmazásokból.

Néhány gyakorlati indok az áttérés mellet:

- A valós számokkal való számolás lassú és pontatlan. A moduláris aritmetika gyors és pontos, csak egész számokkal dolgozik.
- A „valós" görbének végtelen sok pontja van, a modulárisnak jóval kevesebb.
- A moduláris aritmetikában behatárolható a számok értelmezési tartománya, mert a műveletek operandusa(i) és eredménye mindig *0* és *p-1* közé esik.
- A moduláris aritmetika alkalmazása megnöveli a megoldások számát.

Ha a ponthalmazt az xy-síkon ábrázoljuk, már nem lesz olyan „szép", mint eddig, de megfigyelhető például, hogy továbbra is szimmetrikus. Igaz, most már nem az x tengelyre. A következő ábra *p=11, a=1 és b=0* paraméterek által kijelölt „görbét" mutatja. Figyeljük meg, hogy:



- 11 darab pontja van a görbének,
- ebből egy darab az origóban (mert *b=0*),
- 10 darab viszonylag véletlenszerűen, de az y=5,5-re szimmetrikusan helyezkedik el, ezért
- minden x értékhez továbbra is kettő y tartozik.

Megoldások: (0,0) (5,3) (7,3) (8,5) (9,1) (10,3) (5,8) (7,8) (8,6) (9,10) (10,8)

35. ábra Az *E: $y^2 \equiv x^3+1x+0$ (mod 11)* „görbe"

A *E* görbe pontjainak száma (amit egyébként a görbe kardinalitásának vagy rendjének is hívnak és általában #E(p)-vel jelölik) most csak véletlenül egyenlő *p*-vel. Azokat a görbéket, amelyek pontjainak száma megegyezik *p*-vel, rendhagyó görbéknek (*anomalous curve*) nevezzük és gyakorlatilag az összes szabvány tiltja használatukat, mert létezik hatékony támadási módszer az ilyen görbét használó ECC-rendszer ellen [61]. Hasse tétele szerint egyébként a pontok száma $(p+1) \pm 2\sqrt{p}$ között van.

Egy pontra jellemző még egy szám, amelyet a pont rendjének hívunk: hányszor tudjuk összeadni saját magával, mielőtt ***O***-ba érkezünk? Vagyis ha *nP=**O***, akkor *n* a pont rendje. Ha a pont rendje megegyezik a #E(p)-vel, a pontot generátorpontnak hívjuk. (A generátor tulajdonságról bővebben lásd: *4.1. Diffie – Hellman kulcscsere* alfejezetet) A pontok számának meghatározásához útmutatást találunk [60]-ban és az ANSI TG-17 *technical guide*-ban.

5.2.2. Műveletek a görbe pontjaival

Előjelváltás – ellentett képzése

Ha visszaemlékszünk a valós számokon értelmezett görbére, ott egy **P**(x,y) ellentettje az (x,-y) pont volt. Most sincs ez másként, csak modulárisan kell számolnunk: (x, -y mod p), ami nem más, mint: (x, p-y mod p). Ha megnézzük a 35. ábra megoldásait, láthatjuk, hogy az egymással szemben lévő pontok y koordinátáinak összege mindig *p=11*. Például (5,3) és (5,8) → 3+8=11. Tehát egy **R** = –**P** pont kiszámolása:

$\mathbf{x_R = x_P}$ és $\mathbf{y_R = -y_P \bmod p.}$

Összeadás

Kicsit nehézkes most olyat értelmezni, hogy „kössünk össze" két pontot és keressük meg a harmadik metszéspontot, főleg hogyan „húzzunk érintőt"? Ezért a korábbi algebrai eredményeket egyszerűen átvesszük a moduláris aritmetika szabályai szerint:

$s = (y_P - y_Q) / (x_P - x_Q) \bmod p = (y_P - y_Q)(x_P - x_Q)^{-1} \bmod p$

$\mathbf{x_R = s^2 - x_P - x_Q \bmod p}$ és $\mathbf{y_R = s(x_P - x_R) - y_P \bmod p}$

P + (–P) = ?

Ez továbbra sem változik: **P + (–P) = O**

2P = ?

Ha y_P nem zérus, vagyis a duplázandó pont nem az x tengelyen van, akkor **R** = **2P**

$s = (3x_P^2 + a) / (2y_P) \bmod p = (3x_P^2 + a)(2y_P)^{-1} \bmod p$

$\mathbf{x_R = s^2 - 2x_P \bmod p}$ és $\mathbf{y_R = s(x_P - x_R) - y_P \bmod p}$

5.3. A PROBLÉMA: DISZKRÉT LOGARITMUS

Minden kriptorendszer alapja egy olyan probléma, amit *gyakorlatilag* lehetetlen megoldani. Az ECC-nek is egy ilyen adja a biztonságát, mégpedig a „diszkrét logaritmus elliptikus görbék felett" kiszámolásának problémája (*Elliptic Curve Discrete Logarithm Problem – ECDLP*).

> 1991-ben néhány kutató elkészítette az RSA algoritmus elliptikus görbén alapuló változatát, de néhány évvel később többen is megmutatták, hogy az elliptikus RSA-nak (*ECC-like RSA*) nincs számottevő előnye a hagyományos RSA-val szemben. Az ECRSA problémája egyébként továbbra is a faktorizálás maradt. [62]

Eddig lényegében két műveletet definiáltunk a görbén: pontok összeadása és egy pont duplázása. Egy pont sorozatos összeadásával (**R**, **R**+**R**, 2**R**+**R**, 3**R**+**R**) tulajdonképpen már szorozni is tudunk. Az így képzett **Q**=*n***R** pontot a pont *skalár szorzatának* nevezzük, de *n* meghatározása a szorzat alapján nem egyszerű feladat főként, ha a görbét *mod p* felett értelmezzük:

- *Ha* **Q**=*n***R**
- *n = ?* (*Ismert:* **Q**, **R** és a *görbe egyenlete*)

Ebben az esetben *n* diszkrét logaritmusa **Q**-nak, *p* bázis felett.

> Legyen egy elliptikus görbe egyenlete $y^2 \equiv x^3 + 9x + 17 \bmod 23$
> Mennyi az **R** = (16,5) alapú diszkrét logaritmusa **Q** = (4,5) pontnak? (**Q**=n**R**)
>
> Első megközelítésben *n* kiszámolásához addig adogassuk össze a **R** pontot önmagával, amíg meg nem kapjuk **Q**-t:
>
> 1**R**=(16,5) 2**R**=(20,20) 3**R**=(14,14) 4**R**=(19,20) 5**R**=(13,10) 6**R**=(7,3) 7**R**=(8,7) 8**R** = (12,17) 9**R** = (4,5)
>
> Mivel **Q**(4,5) = 9**R**, ezért a **Q** pont **R** alapú diszkrét logaritmusa mod 23 felett: 9.

Ha most valaki azt mondja, hogy a logaritmus a hatványozás inverze és nem a szorzásé (főleg nem az összeadásé), akkor annak maximálisan igaza van. De...

1. A racionális számokon értelmezett „hagyományos” logaritmus pontosan ugyanarról szól, mint például a *Diffie-Hellman* kulcscsere megoldása. Ha egy *g* számot *x* alkalommal összeszorzunk önmagával ($a=g^x$), akkor az eredményből és *g* ismeretében: $x=\log_g a \bmod p$, ha létezik. Ez a DLP.
2. Az ECDLP esetén adott **P** és **Q** pont a *mod p* felett értelmezett elliptikus görbén. Feladat: megkeresni *x*-et úgy, hogy *x***P**=**Q** legyen, ha létezik ilyen szám.

Úgy tűnik a két probléma jelentősen eltér egymástól, hiszen eleve más műveletekről szólnak, az elsőben sorozatos *szorzás* van, a másodikban sorozatos *összeadás*. Azonban nálam hozzáértőbb matematikusok azt mondják, hogy a DLP szorzása és a ECDLP összeadása absztrakt módon ugyanaz. Általánosságban nem is szorzásnak és összeadásnak hívják a használt műveleteket, hanem csoportműveleteknek (*group operations*) [URL36]. Az elliptikus görbék pontjaira értelmezett összeadás és a „hagyományos” szorzás logikai hasonlóságát az is jelzi, hogy mindkét művelet ugyanazokkal a tulajdonságokkal rendelkezik a *pozitív számok* halmazán. Íme a három szóban forgó művelet:

Tulajdonság	**Elliptikus görbék**	**„Hagyományos”**	
	Összeadás	Szorzás	Összeadás
A művelet eredménye az operandusok halmazán belül marad.	✓	✓	✓
Az operandusok felcserélhetők: X @ Y = Y @ X	✓	✓	✓
Az operandusok csoportosíthatók: (X @ Y) @ Z = X @ (Y @ Z)	✓	✓	✓
Létezik egy olyan *e* elem az operandusok halmazában, hogy minden X-re igaz (*null elem*): X @ e = e @ X = X	✓ e=**O**	✓ e=1	✓ e=0
Minden X-hez létezik egy Y úgy, hogy (*inverz elem*): X @ Y = Y @ X = e	✓ Y= -X	✓ Y=1/X	✘

Egy mérnöki szemléletű embernek ez hajmeresztőnek tűnhet, ezért a „békesség kedvéért” ne keressünk most logikát az elnevezésekben, hanem vegyük tudomásul: az elliptikus görbéken értelmezett sorozatos összeadások inverzét logaritmusnak hívjuk. Pont.

Ugyancsak az [URL36]-on olvashatunk az ECDLP elleni főbb támadásokról és metódusokról is. Ugyancsak itt kaphatunk útmutatást arra, hogyan generáljunk véletlenszerűen elliptikus görbéket vagy görbén lévő pontokat, bár erre mi is visszatérünk néhány oldal múlva. Az álvéletlen bitsorozatok előállítására az SHA-1 algoritmust használja és az előállt bitsorozat alapján

- választja görbék esetén az *a* és *b* paramétereket, illetve
- pontgenerálás esetén *x=véletlenszerű érték* mellé keres *y* megoldást.

5.4. TITKOSÍTÁS ÉS ALÁÍRÁS AZ ELLIPTIKUS GÖRBÉKKEL

Az ECDLP-n alapuló rendszerek többsége aláíró (például ECDSA) vagy kulcscserélő (például ECDH) rendszer, mert gyors titkosításra ez a módszer is alkalmatlan. A most következő oldalakon ezekkel a rendszerekkel ismerkedhetünk meg [60,62,64]. (A folytatás megértéséhez szükséges ismeretek: *4.1. Diffie – Hellman kulcscsere, 4.4. ElGamal* valamint az üzenetpecsét algoritmusok ismerete – legalább nagyvonalakban. Ezek nélkül is érthető a folytatás, csak nehezebb lesz párhuzamot találni a már megismert algoritmusok és az EC-alapú párjuk között...)

5.4.1. ECDH – Elliptic Curve Diffie-Hellman kulcscsere

Az eredeti Diffie-Hellman algoritmus a szimmetrikus titkosító rendszerek kulcsmegosztási problémáját oldatta meg. Hogyan is? A két résztvevő ugyanazokat a műveleteket végezte el egyező nyilvános és különböző titkos paraméterekkel, de azonos eredményt kaptak, melyet kulcsként használhattak. Az ECDH is ugyanígy működik, csak nem moduláris hatványozást használ, hanem a fejezet eddigi részében megismert EC-műveleteket.

Alice és Bob megegyeznek egy *E* görbében és egy **G** pontban, utóbbit bázispontnak hívjuk. A továbbiakban eme paramétereket nyilvános rendszerparamétereknek tekintjük. Alice választ egy véletlen számot, (amely kisebb, mint a **G** pont rendje) és ugyanígy tesz Bob is: Alice száma legyen *a*, Bobé legyen *b*. Mindketten titokban tartják választásukat. A kulcscsere következő lépésében Alice kiszámolja *a***G** pontot, melyet elküld Bobnak, aki Alice műveletéhez hasonlóan kiszámolja *b***G** pontot és elküldi Alicenak. Végül Alice a Bobtól kapott *b***G**-t megszorozza *a*-val, így megkapja *ab***G** pontot, valamint Bob az Alicetól kapott *a***G** pontot szorozza meg titkos *b* számával és eredményül ő is az *ab***G** pontot kapja.

A közös pont valamely tulajdonsága (például x vagy y koordinátája vagy éppen x + y, x xor y, stb.) használható kulcsként. A kíváncsi Eve-nek az *ab***G** pontot kellene kiszámolnia, de csak **G**, *a***G** és *b***G** pontokat ismeri, magukat a titkos *a* és *b* számokat nem. Az elliptikus Diffie-Hellman működését és lépéseit az alábbi egyszerű számpélda alapján követhetjük:

Nyilvános paraméterek:	Legyen $E: y^2 \equiv x^3 + 5x + 8 \bmod 23$ és G(8,10)	
0. Titkos paraméterek:	Alice választ: a = 7	Bob választ: b = 3
1. Kulcselőkészítés:	Alice számol: 1G(8,10) 2G(13,4) 3G(20,9) 4G(22,18) 5G(6,1) 6G(2,18) 7G(7,15) **aG = (7,15)**	Bob számol: 1G(8,10) 2G(3,4) 3G(20,9) **bG = (20,9)**
2. Kommunikáció:	Alice elküldi az eredményt: aG →	Bob elküldi az eredményt: ← bG
3. Egyeztetett kulcs:	1bG (20,9) 2bG (12,18) 3bG (7,8) 4bG (22,5) 5bG (8,13) 6bG (13,4) 7bG (6,1) **abG (6,1)**	1aG (7,15) 2aG (13,19) 3aG (6,1) **baG (6,1)**

5.4.2. ECElGamal - Elliptic Curve ElGamal titkosítás

Ahogy az eredeti ElGamal titkosítás (*4.4. ElGamal*) a Diffie-Hellman algoritmus problémáján alapul, úgy építhető fel az elliptikus ElGamal is az ECDH-ra:

1. Alice és Bob választ egy *E* görbét és egy **G** bázispontot.
2. Mindketten választanak egy-egy véletlen *a* és *b* számot, mint titkos kulcsot.
3. Alice elküldi az *a***G** pontot, mint nyilvános kulcsot Bobnak.
4. Bob elküldi a *b***G** pontot, mint nyilvános kulcsot Alicenak.
5. Ha Alice üzenni akar Bobnak, az üzenetet leképzi a görbe egy (vagy több) **M** pontjára, és generál egy véletlen *k* számot, mint viszonykulcsot. Elküldi Bobnak a (*k***G**, **M**+*k*(*b***G**)) üzenetpárost.
6. Bob a következőképpen olvassa el az üzenetet: a kapott küldemény első felét megszorozza saját titkos *b* számával, így *bk***G**-t kap, amit egyszerűen kivon a küldemény második feléből.

Eve támadásának feltétele az lenne, ha Bob átküldött nyilvános kulcsából (*b***G**) ki tudná számolni *b* értékét vagy a titkosított üzenet első részéből (*k***G**) a *k* számot. Jelenlegi ismereteink szerint egyikre sem képes elfogadható időn belül.

5.4.3. ECDSA – Elliptic Curve Digital Signature Algorithm

A következő algoritmus a FIPS186-2-ben leírt DSA-hoz hasonlóan működik, hasonlóak a végzett műveletek, lépések is. Ahhoz, hogy Alice egy *M* üzenetet aláírva el tudjon küldeni, a következő paraméterek és eszközök szükségesek:

- egy elliptikus görbe *mod q* felett (nyilvános paraméter)
- egy **G** bázispont, melynek rendje *n* (nyilvános paraméter, $n \geq 160$ bit)
- egy véletlen *d* szám ($1 \leq d \leq n-1$) és egy $\mathbf{Q}=d\mathbf{G}$ pont. Alice kulcspárja $(d,\mathbf{Q})$, ahol *d* a titkos és **Q** a nyilvános kulcs.

Az aláírás algoritmusa

1. Alice választ egy *k* számot *1* és *n-1* között.
2. Kiszámolja $k\mathbf{G}=(x_1,y_1)$ pontot és $r=x_1 \bmod n$. Ha a pont x koordinátája zérus ($x_1=0$), akkor új *k* számot választ. A pont x koordinátája lesz az aláírás egyik komponense, ezért jelöltük meg külön egy *r* betűvel.
3. Kiszámolja *k* multiplikatív inverzét *n*-re ($k^{-1} \bmod n$).
4. Kiszámolja a küldendő üzenet pecsétjét, melyre a szabvány az SHA-1 algoritmust ajánlja. Legyen hát $e = SHA\text{-}1(M)$ (számként értelmezve)!
5. Az aláírás másik alkotóeleme: $s=k^{-1}(e + dr) \bmod n$. Abban a szerencsétlen esetben, ha $s=0$, akkor az egész algoritmust elölről kell kezdeni. Itt láthatjuk, hogy a 2. lépésben miért nem lehet $r=0$: az aláírás nem tartalmazná a titkos kulcsot!
6. Az *M* üzenethez és Alicehoz tartozó aláírás: (r,s).

Az ellenőrzés algoritmusa

Feltételezzük, hogy Bob, mint az aláírás ellenőrzője rendelkezik a hitelesített rendszerparaméterekkel és Alice hiteles nyilvános kulcsával. Bob a következő lépésekkel ellenőrizhet egy aláírást:

1. Ellenőrzi, hogy az (r,s) egész számok az [*1, n-1*] intervallumban vannak-e.
2. Kiszámolja a kapott üzenet pecsétjét. Ehhez ugyanazt az algoritmust kell használnia, amit Alice használt: $e = SHA\text{-}1(M)$.
3. Kiszámolja *s* multiplikatív inverzét *n*-re: $w = s^{-1} \bmod n$. Ezért nem hagyhattuk az aláírás 5. lépésében, hogy $s=0$ legyen: nem létezne inverze!
4. Kiszámolja a következő részeredményeket: $u_1=ew \bmod n$ és $u_2=rw \bmod n$.
5. Végül kiszámolja a $\mathbf{P}(x_1,y_1) = u_1\mathbf{G} + u_2\mathbf{Q}$ elliptikus pontot. Ha $\mathbf{P}=\boldsymbol{O}$, akkor biztosan nem jó az aláírás, egyébként legyen $v = x_1$!
6. Bob az aláírást csak akkor fogadja el, ha $v = r$.

Miért helyes az ellenőrzés?

Az aláírás-ellenőrzés helyességéhez az kell belátnunk, hogy az 5. pontban kiszámolt **P** pont nem más, mint az Alice által kiszámolt $k\mathbf{G}$ pont (aláírási folyamat második lépése). Alice aláírásának egyik része a következő kifejezés:

$$s = k^{-1}(e + dr) \bmod n$$

Ha az egyenlet mindkét oldalát megszorozzuk $s^{-1}k$ szorzattal, akkor

$$k \equiv s^{-1}(e+dr) \equiv s^{-1}e + s^{-1}dr \equiv we + wdr \equiv u_1 + u_2 d \pmod n$$

Már csak egy lépés választ el attól, hogy a Bob által kiszámolt **P** pontra belássuk állításunkat:

$$\mathbf{P}(x_1,y_1) = u_1\mathbf{G} + u_2\mathbf{Q} = u_1\mathbf{G} + u_2d\mathbf{G} = (u_1 + u_2d)\mathbf{G} = k\mathbf{G}$$

Ha Bob eredményül Alice véletlen pontját kapja vissza, azok x koordinátáinak is meg kell egyezniük. Ha Bob egy másik pontot kap eredményül, akkor az x koordináták is különbözőek, így az aláírás nem fogadható el.

5.5. PONTOK, GÖRBÉK ELŐÁLLÍTÁSA

Egy-egy ECC-rendszerhez szükség van egy *E* görbére, egy **G** bázispontra, véletlen pontokra stb. Az alábbiakban ezek előállítására láthatunk módszereket – némelyikre kettőt is. Sajnos a görbék és pontok választásánál sok olyan tulajdonságra kell figyelni, amelyekről már volt szó, vagy eddig nem tértem ki rájuk és nem is fogok, azok komplexitása miatt.

> Ha ezen ismeretek hiányában kívánunk üzleti célú biztonságos implementációt készíteni, a szabványokban javasolt görbékből és bázispontokból válasszunk! Ne feledjük: ezek egyébként is nyilvános paraméterek! Például [URL46]/SEC2–ben 113 bittől 571 bitig találunk paramétereket, URL[45]-ben pedig a szabványosnak tekintett bitméretekre: 112, 128, 160, 192, 224, 256, 384, 521 bit. Úgy vélem, hogy az ECC alapteremtő magyarázatához és megértéséhez nem szükségesek a most „elfelejtett" tulajdonságok, akit pedig érdekel, az Interneten tengernyi irodalmat találhat. Én eddig a legtömörebb, legegyszerűbb és „legimplementáció-barátabb" leírásokat az IEEE P1363-ban találtam [64]. Kérem a Kedves Olvasót, hogy nézze el ezt a hiányosságot nekem, de csak és kizárólag az ECC matematikai hátteréről több könyvet lehetne írni. És még néhányat a kriptográfiai alkalmazásukról, gyakorlati implementációikról... Csak két példa „elrettentésül":
>
> 1. Minden eddig leírt definíciót, szabályt és algoritmust még legalább kétszer le lehetne írni, mert az elliptikus görbéket nemcsak a természetes számok (*mod p*) felett lehet értelmezni, hanem polinomalgebra alapján is, amelynek legalább kétféle ábrázolásmódja ismeretes. (Ilyenkor a modulus nem prímszám, hanem kettőhatvány. Szoftverekben a prímmodulus használata, hardverben a kettőhatvány modulus biztosít gyorsabb működést.)
> 2. A kedvelt és szemléletes Descartes-féle koordinátarendszer mellett az elliptikus görbe pontjait a projektív síkon is szokták ábrázolni. Ekkor a végtelenben lévő O pont az origóba kerül.

5.5.1. Görbe generálása 1.

A görbék készítésére egyébként legalább háromféle módszer van. Az egyik speciális görbékkel dolgozik, amelyek együttható bizonyos szempontoknak megfelelnek: optimális hardvermegvalósítást tesznek lehetővé, vagy különlegesen ellenállóvá teszik a görbét valamelyik támadási forma ellen, stb.

5.5.2. Görbe generálása 2.

A második szerint a görbét meghatározó együtthatókat véletlenszerűen választjuk meg:

1. Válasszunk egy prímszámot: $p = 4294967861$ ($2^{32}+565$)
2. Majd egy véletlen a paramétert: $a = 1234567890$
3. És egy véletlen b paramétert: $b = 0987654321$
4. Ellenőrzés következik: $4a^3+27b^2 \bmod p = 752670551349406800391749 4107 \bmod 4294967861 = 1240368550 \neq 0 \rightarrow$ OK!
5. A kész görbénk egyenlete: $y^2 \equiv x^3+1234567890x+987654321 \pmod{429467861}$

5.5.3. Görbe generálása 3.

A harmadik módszer nem véletlen számokat használ, hanem egy kezdőértékből (*seed*) számított SHA-1 érték alapján állítja elő az együtthatókat. Az IEEE P1363 és a NIST a következő eljárást javasolja az ilyen típusú görbék (*pseudo-random curves*) konstruálásához [64, URL45]:

Álvéletlen görbék generálása

Bemenet: • Egy p prím, melynek hossza l bit, és $l \geq 161$ bit

Kimenet: • Az a, b, p görbeparaméterek és s az ellenőrzéshez.

1. Végezzük el a következő előkészítő számításokat:
   - v = egészrész((l – 1) /160)
   - $w = l – 160v – 1$
2. Válasszunk egy 160 bites s inicializáló értéket, és h = SHA-1(s).
3. Legyen h_0 = a 160 bites h legnagyobb helyértékű w darab bitje.
4. z = az s bitstring, egész számként értelmezve.
5. for i = 1 to v do
   - $s_i = (z + i) \bmod 2^{160}$
   - h_i = SHA-1(s_i)
6. Az előző h_0 , h_1, …, h_v értékeket, mint bitstringeket fűzzük össze: $h = h_0 \| h_1 \| \ldots \| h_v$
7. c = a h bitstring l-1 darab legnagyobb helyértékű bitje egész számként értelmezve.
8. Ha $c = 0$ vagy $4c + 27 \equiv 0 \pmod p$, akkor elölről kezdjük az egészet az első lépéstől.
9. Válasszunk két egész számot (melyek kisebbek, mint p) úgy, hogy $cb^2 \equiv a^3 \pmod p$ igaz legyen! A legegyszerűbb választás $a = c$ és $b = c$, de ez nem kötelező.
10. E kész görbénk E: $y^2 = x^3 + ax + b$. Ellenőrizzük le, hogy megfelel-e a követelményeknek (például elég magas-e a rendje)?
11. Ha jó, akkor készen vagyunk, ha nem, akkor elölről kezdjük az egészet az első lépéstől.
12. Kimenet: (a, b, p, s)

Az álvéletlen megoldás előnye, hogy reprodukálható és így ellenőrizhető, hogy egy görbe ezzel a módszerrel készült-e. Ha a kilencedik lépésben nem a legegyszerűbb megoldást választjuk, hanem állandó a értékekből választva keressünk egy b együtthatót, akkor az 1.-8. lépések ismételt végrehajtása után ellenőrizhető a $cb^2 \equiv a^3 \pmod{p}$ egyenlőség. (A generálás 7. lépésében van egy apró különbség a NIST ajánlás és az IEEE szabvány között, én az utóbbira szavazok… Ha valaki összehasonlítja a két forrást, vegye figyelembe, hogy ugyanolyan nevű változókat használnak, csak összekeverve, más-más feladatra!)

5.5.4. Pont generálása

Ha már van egy görbénk, akkor azon egy véletlen pontot is kereshetünk. Példaképpen a véletlen számokból készített görbénken keresünk egy pontot:

$$y^2 \equiv x^3+1234567890x+987654321 \pmod{429467861}$$

$x = 147896325$ (véletlenszerűen választott)

$$y^2 \equiv 370713451 \pmod{4294967861}$$

$y = ?$

Hát, izé… ennek nincs megoldása (mintha a 35. ábrán az $x=6$-ra keresnénk megoldást), próbáljuk meg újra egy másik x értékkel:

$x = 225589$

$$y^2 \equiv 376919525 \pmod{4294967861}$$

$y = 57372704$

Na ezzel megvolnánk: **P**(*225589, 57372704*)! E pontnak van még néhány tulajdonsága, amit jó lenne tudni (például generátor-e? Ha nem, mennyi a rendje?), de ezekkel most nem foglalkozunk (lásd korábban, hogy miért nem).

5.6. ÜZENET LEKÉPZÉSE EGY PONTRA ÉS VISSZA

Néhány kriptográfiai algoritmus tartalmaz egy olyan lépést, amikor az üzenetet le kell képezni egy olyan alakra, amit az adott algoritmus kezelni tud. Esetünkben ez azt jelenti, hogy egy adott E görbe alkalmazása mellett Alice **P** ponttá tudja alakítani az m üzenetet, és Bob egy megfejtett pontból ki tudja venni az üzenetet.

> Itt jegyzem meg: előfordulhat, hogy csak a pont x koordinátája közlekedik teljes egészében a kommunikációban. Az y-nak csak legnagyobb helyértékű bitje kíséri az x-et, mondván, az y kiszámolható. Ebben az esetben a pontot tömörített pontnak (*compressed point*) hívja az ANSI9.62, az IEEE P1363 és a SEC is ([64,65] és [URL46]. A három forrás legnagyobb különbsége, hogy a SEC csak használja a fogalmat, a többiek el is magyarázzák.

A számpéldához a korábban készített görbénket fogjuk használni:

$$\text{E: } y^2 \equiv x^3+1234567890x+987654321 \pmod{429467861}$$

Első próbálkozásunkkal alakítsuk ponttá a „Jó!” karaktersorozatot! Ehhez először számmá kell alakítani: a karakterek ASCII kódját hexa formában egymás mellé írjuk, és egyetlen hexadecimális számként értelmezzük. Más módszer is használhatunk, a lényeg, hogy:

- kölcsönösen egyértelmű megfeleltetés legyen, és
- a blokkméretnek kisebbnek kell lennie, mint a modulus hossza!

És ezután mi sem egyszerűbb, ez legyen az üzenetet képviselő **P** pont x koordinátája, csak ki kell számolni az y-t! Lássunk neki!

m = „Jó!” = 0x4A 0xF3 0x21 = 0x4AF321 = 4911905

Első próba

P(m,y) = (*4911905, ?*)

$y^2 \equiv x^3+1234567890x+987654321 \pmod{429467861}$

$y^2 \equiv 43578828$

$y \equiv 165701469$

P(m,y) = (*4911905, 165701469*)

Természetesen felvetődhet a kérdés, hogy mi van akkor, ha az üzenet alapján nem létezik pont? A pontgenerálás első próbálkozása is ezért volt sikertelen. Mi a teendő akkor, ha y^2-nek nincs megoldása?

Második próba

m = „Nem” = 0x4E 0x65 0x6D = 0x4E656D = 5137773

P(m,y) = (*5137773, ?*)

$y^2 \equiv x^3+1234567890x+987654321 \pmod{429467861}$

$y^2 \equiv 109210672$

$y \equiv$ nincs megoldása

Az ilyen esetekre felkészült alkalmazások nem tisztán az *m* üzenetet tekintik az x koordinátának, hanem az x koordináta felső bitjeibe helyezik el azt, majd az alsó bitekkel addig „szórakoznak”, amíg eredményre nem jutnak: **P**(m * eltolás + valami, y). Ilyenkor az üzenetet kibányászni a (P_x / *eltolás*) egészrészeként lehet. Példánkban a modulus 29 bites, az eltolás legyen 6 bit, így 23 bites üzeneteket tudunk továbbítani. Próbáljuk meg még egyszer a „Nem” szót ponttá alakítani, az eltolás használatával (*valami*=10, ha nem jutunk eredményre, majd választunk másikat…):

m = „Nem” = 0x4E 0x65 0x6D = 0x4E656D = 5137773

P$(m*2^6+10,y)$ = (*328817482, ?*)

$y^2 \equiv x^3+1234567890x+987654321 \pmod{429467861}$

$y^2 \equiv 275000646$

$y \equiv 125641602$

P(*328817482, 125641602*)

Ezt a pontot már Alice tetszőleges módon titkosíthatja és elküldheti Bobnak. Bob a megfejtés után szerencsés esetben ezt a pontot fogja visszakapni. Megragadja a pont x koordinátáját, elosztja 2^6-nal és az eredmény egészrésze: 5137773 ! Voilá!

5.7. TÉNYLEG BIZTONSÁGBAN VAGYUNK?

A kérdés megválaszolásához a bevezetőben szereplő gondolatokat kell továbbfűznünk az eddigi törési kísérletek és tapasztalatok fényében [63,62,URL33,URL47].

5.7.1. Certicom challenges

Az RSA Inc.-hez hasonlóan a Certicom Corporation is írt ki törési versenyeket [URL47], melyek egy része mára megoldott, másik része még megoldásra vár. A feladatok 1999 óta – a Certicom szándéka szerint – azt kívánják bizonyítani, hogy az ECC erősebb, mint az RSA- vagy a DLP-probléma. Másrészt marketingfogás, amely megpróbálja a potenciális ipari felhasználók, fejlesztők és a kutatók figyelmét az ECDLP felé terelni.

A versenykiírásban mintegy 20 nyilvános kulcs szerepel, a hozzájuk tartozó rendszerparaméterekkel együtt [URL51]. A feladat: meg kell keresni a titkos kulcsot. A Certicom három csoportra osztotta a feladványokat[36]:

Excercises:	79 bites	(néhány óra)
	89 bites	(néhány nap)
	97 bites	(néhány hét)
Level I:	109 bites	(néhány hónap)
	131 bites	(néhány hónapnál sokkal több)
Level II:	163 bites	(jelenleg megoldhatatlan feladatok)
	191 bites	
	239 bites	
	359 bites	

Eddigi megoldások néhány technikai adatát tartalmazza a következő felsorolás. (Érdekes, hogy a különböző források kis mértékben ellentmondóak egymásnak, akárcsak az RSA törési versenyek esetében...)

2002. November 6.: ECCp-109 Challenge megoldása (prím modulus)

10300 résztvevő, 10000 számítógép, 1,5 év időtartam

2000. Április 17.: ECC2K-108 Challenge megoldása (kettőhatvány modulus)

1300 résztvevő, 9500 számítógép, 4 hónap időtartam

1999. Szeptember 28.: A 97-bites ECC Challenge megoldása

200 résztvevő, 740 számítógép, 16 000 MIPS-év számításigény. Mindez körülbelül fele az ötször hosszabb RSA-512 feltöréséhez használt teljesítménynek.

[36] Zárójelben a Certicom által becsült megoldási idők, néhány ezer együttműködő gép esetére.

5.7.2. Pollard-ρ algoritmusa

Napjaink legjobbnak tartott algoritmusa a *Pollard-ρ* algoritmus (Pollard-ró). Az eljárás kis módosítással teljes mértékben párhuzamosítható, így ha 10 processzor áll rendelkezésre, 10-szer gyorsabban jut eredményre. Ha csak egy processzorunk van, $0.5*\sqrt{(\pi*p)}$ EC-összeadás (ahol p a modulus) kell a végrehajtásához, ha több, akkor ez a sok számítás megoszlik köztük. Akármilyen jó is az algoritmus, tetemes számításigénye[37] van [62]:

p mérete	$0.5*\sqrt{(\pi*p)}$	MIPS-év
97 bit	2^{49}	$1{,}6\times10^{4}$
160 bit	2^{80}	$8{,}5\times10^{11}$
186 bit	2^{93}	$7{,}0\times10^{15}$
234 bit	2^{117}	$1{,}2\times10^{23}$
354 bit	2^{177}	$1{,}3\times10^{41}$
426 bit	2^{213}	$9{,}2\times10^{51}$

Összehasonlításul a faktorizálás becsült számításigénye a szokásos modulusok méretére [62]:

modulus mérete	MIPS év
512 bit	3×10^{4}
768 bit	2×10^{8}
1024 bit	3×10^{11}
1280 bit	1×10^{14}
1536 bit	3×10^{16}
2048 bit	3×10^{20}

Az algoritmus további részleteitől és hátterétől nagyvonalúan tekintsünk el, jelentősen túlmutat a könyv, főleg a fejezet célkitűzésein. Mindenesetre a kíváncsi Olvasók számára az irodalom mellett ajánlom a Függelékben lévő UBASIC nyelven készült rövid implementációt is...

5.7.3. Válasz a kérdésre: nem tudjuk!

Napjaink minden széles körben elterjedt kulcscserére, titkosításra vagy digitális aláírásra használt PKI algoritmusa a faktorizálás vagy a diszkrét logaritmus problémáján nyugszik. A két probléma hasonlít egymáshoz, amit az is jól jelez, hogy a legjobb faktorizáló algoritmusok (bizonyos feltételek teljesülése esetén) felhasználhatók a DLP-problémák megoldásában is. Némi egyszerűsítéssel azt is mondhatjuk, hogy egyező kulcsméret mellett egyező biztonságot nyújtanak [63].

Sajnos a DLP és az ECDLP már nem hasonlít ennyire egymásra, a viszonylag jó és újabb DLP megoldóalgoritmusok (például „*index kalkulus*”) egyszerűen nem használhatók ECDLP esetére: ott meg kell elégedni a régebbi módszerekkel (például *Pollard-ρ*). Ebből az is következik, hogy elégséges, ha a kulcsok e régi és lassú „trükköknek” ellenállnak, tehát rövidebbek is lehetnek. Jelentősen rövidebbek. A minimálisan ajánlott kulcsméret ma 1024 bit az RSA esetében, 163 bit az ECC-rendszerekhez. Semmi sem garantálja azonban, hogy ez holnap is

[37] 1 MIPS-év az a számításigény, ami 1 darab 1 MIPS teljesítményű számítógéppel 1 év alatt teljesíthető.

így lesz. Semmi sem garantálja, hogy a közeljövőben valaki nem publikál egy olyan eljárást, amely valóban jó megoldást nyújt az ECDLP-re. Nem tudjuk, hogy elvileg sem működnek a DLP jó algoritmusai vagy csak a mi ismereteink a hiányosak. Igazság szerint az ECDLP-re ma is léteznek jó algoritmusok, de ezek mindegyike kizárólag valamilyen speciális tulajdonságú görbére alkalmazható és nem általánosan. Jelenlegi tudásunk szerint mindenestre az ECDLP alapú kriptorendszerek jobbak – gyorsabbak és biztonságosabbak –, mint az IFP-n alapuló RSA-szerű, vagy a DLP-n alapuló kriptorendszerek. Mindezekről *Menezes* [URL34]-en, *Schneier* pedig [63]-ban elmélkedik…

A Függelék jelen fejezethez kapcsolódó alfejezetei

14.4. Moduláris aritmetika nagyon dióhéjban

14.8. Szabványok összefoglaló táblázata

14.9. Pollard-ρ algoritmus – UBASIC implementáció

14.12. Néhány szám és nagyságrend

14.13. Moore törvénye

További kiegészítések folyamatosan bővülő helye a http://www.netacademia.net/konyv webcím.

A titkok megfejtésére szító késztetést az ember a génjeiben hordozza. Még a legkevésbé kíváncsi elme érdeklődését is felkelti annak ígérete, hogy mások elől eltitkolt értesülésekhez juthat. Akadnak szerencsés emberek, akik olyan munkakört találnak maguknak, amelyben rejtélyek megfejtése a dolguk, a túlnyomó többségnek azonban be kell érnie annyival, hogy a szórakoztatására kitalált rejtvények megfejtésével csillapítsa ezt a késztetést. A többségnek krimi és keresztrejtvény jut - a titkos rejtjelek feloldásának feladata csak kevesek osztályrésze.

John Chadwick (kriptográfus, nyelvész):
A lineáris B megfejtése

6. TITKOS VS. NYILVÁNOS KULCSÚ MÓDSZEREK – ÉS NÉHÁNY ZÁRÓ GONDOLAT

Mind a szimmetrikus, mind az aszimmetrikus algoritmusoknak megvan a maga előnye és hátránya. Egyértelműen nem lehet eldönteni, hogy melyik a jobb, mert a rossz és a jó tulajdonságok egymást kiegészítik. Ez a fejezet átfogó összehasonlítást kíván adni ezekről a tulajdonságokról, egy helyen összegyűjtve a szem előtt tartandó jellemzőket, követelményeket és ok – okozati összefüggéseket.

6.1. A SZIMMETRIKUS ALGORITMUSOK

6.1.1. Előnyök

- A szimmetrikus algoritmusok gyorsak, így jól használhatók olyan alkalmazásokban, melyek nagy adatátviteli sebességet igényelnek. Néhány hardvermegvalósítás sebessége a 10-200 Mbit/s sebességet is eléri. A szoftveres megvalósítások lassabbak, általában csak 1-10 Mbit/s sebességűek.
- Az alkalmazott kulcsok viszonylag rövidek (56-256 bit), így csak kevés tárhelyet foglalnak (Smart Card alkalmazások fő szempontja).
- A szimmetrikus algoritmusok nemcsak titkosításra alkalmasak, hanem többféle kriptográfiai feladatban is alkalmazhatók. Álvéletlen számok generálásához, hashfüggvények tömörítő-függvényeiként, stb. is használatosak.
- A különböző elvű szimmetrikus módszerek kombinálásával igen erős titkosító egységek hozhatók létre (*produkciós titkosítások*). Olyan egyszerű transzformációk, mint a helyettesítés vagy a keverés önmagukban könnyen elemezhetők, de ezek összekapcsolásával keletkező SP-hálók igen összetett (és erős kriptográfiai) működést eredményezhetnek.
- A titkosítás és a megfejtés feladata és folyamata logikailag és algoritmikusan is elkülöníthető egymástól. (Nem előny, de megjegyzendő, hogy a kommunikációhoz ettől függetlenül egy kulcs kell, ami ugyanaz a küldő és a fogadó oldalán.)
- A szimmetrikus titkosítóeszközöknek igen bőséges történelmi előzménye van. A gyakorlati és elméleti ismeretek bővülése elvezetett az ókori Caesar-kódolóktól a ’70-es évek DES algoritmusáig és napjaink AES-szabványáig.

6.1.2. Hátrányok

- Egy kommunikációban mind a feladó, mint a címzett oldalán titokban kell maradnia a kulcsnak, egészen a kommunikációs folyamat(ok) végéig.
- Nagy hálózatokban, vagy olyan szervezetekben, ahol sok ember kíván egymással érintkezésbe lépni, a kezelendő kulcsok száma a résztvevők (n) számával négyzetesen arányos.
- A kommunikációs folyamat megkezdése előtt egy biztonságos csatorna használatával kulcsot kell egyeztetni. Ha a kulcsot valamilyen okból meg kell változtatni, a kényes kulcscserét meg kell ismételni.
- A rövid kulcsok kedveznek a *brute-force* támadásnak, ezért azokat minél sűrűbben cserélni kell (legalább a feltételezett feltörési időn belül) vagy hosszabb – 128-256 bites – kulcsot kell használni. Sok partner esetén a gyakori kulcscsere nehézkes lehet, pontosan a kapcsolatok nagy száma miatt.

6.2. AZ ASZIMMETRIKUS ALGORITMUSOK

6.2.1. Előnyök

- A résztvevő feleknek két kulcsuk van, ezek feladata más és más. Az egyiket nyilvánosságra lehet (kell) hozni, a másikat titokban kell tartani.
- A titkosító és megoldó folyamat általában csak logikailag különbözik, így a kulcsok szerepe sem a fizikai feladathoz kötődik. Nem mondható ki egyértelműen, hogy a nyilvános kulcsot csak titkosításra használjuk, sem az, hogy a titkos kulcsot csak megoldáskor kell használnunk. Erre nagyon jó példa a digitális aláírás, ahol mindez éppen fordítva van.
- A titkosító és a megfejtő folyamatok a legtöbb aszimmetrikus rendszerben logikailag felcserélhetőek, ezért ezek az algoritmusok hatékonyan használhatók digitális aláírási rendszerekben. Az egyediséget biztosító titkos kulcs lehetővé teszi a jó digitális aláírás elvárásainak teljesítését.
- A nagy létszámú résztvevővel rendelkező kommunikációs hálózatokban sem jelent különösebb nehézséget a kulcsok megosztása. Ha n partner van, n darab nyilvános kulcsot kell kezelni.
- Az aszimmetrikus algoritmusok jól definiált, de nehéz matematikai problémákon alapulnak és kulcsaik sokkal hosszabbak, mint a szimmetrikus kulcsok. Általában egy-egy kulcspár évekig használható. (Hacsak nem lopják el azt, vagy meg nem oldják a matematikai feladatot.)
- Az igen hosszú kulcsok lehetetlenné teszik a *brute-force* támadást.

6.2.2. Hátrányok

- Az algoritmusok lassúak, nem képesek a gyakorlati igényeket kielégíteni. Emiatt gyakran a szimmetrikus algoritmusokkal együtt használják őket, hibrid kriptorendszert alkotva.
- A kulcsok mérete sokkal hosszabb, mint a szimmetrikus algoritmusok 56-128 bites kulcsa, például az RSA ajánlása szerint a napjainkban generált kulcsoknak legalább 1024-2048 bitesnek kell lenniük. A kulcsmenedzsment és egyéb műveletek nagyobb adatmennyiség mozgatását és tárolását igénylik, mint a szimmetrikus algoritmusok esetében.
- Nagyszám-aritmetika megvalósítás szükséges.
- Szükség van egy megbízható harmadik félre, aki garantálja és tanúsítja, hogy a nyilvántartásában szereplő felhasználónév és a hozzá tartozó nyilvános kulcs valóban összetartozik.
- Egyetlen algoritmus sem nyújt az OTP-hez hasonló elméleti titkosítást (lásd: *2.6. Az egyszer használt bitminta* fejezetet), mert a legtöbb megoldás valamilyen nehezen megoldható matematikai problémán alapszik. Az RSA titkosításban maga a titkos kulcs is kiszámolható a nyilvános (kitevő, modulus) párosból, de ehhez szükség lenne a modulus prímtényezős bontására. Azonban a modulus szándékosan olyan nagy, hogy faktorizálása időben lehetetlen feladat legyen. A nyíltkulcsos algoritmusok biztonságát általában az adja, hogy az inverz művelethez olyan részeredményekre van szükség, amelyek előállítása időben vagy társzükségletben lehetetlen.
- Nincs történelmi háttere az algoritmusoknak. A gyakorlati és az elméleti tapasztalatok mindössze néhány évtizedre, a ’70-es évekig nyúlnak vissza. Ekkor kerültek előtérbe azok a matematikai problémák, amelyek az algoritmusok alapját képezik. A számítástechnika fejlődése is ekkortól tette lehetővé az addig megoldatlan problémák vizsgálatát és újabb problémák felvetését.

6.3. ÖSSZESÍTÉS

Érdekes módon a szimmetrikus és az aszimmetrikus kódolások egyes tulajdonságai kiegészítik egymást. Napjaink titkosító rendszerei úgy kombinálják a két módszert, hogy mindegyiknek a saját jó tulajdonsága jelenjen meg. Gyakori példa erre a „borítékolás”, amikor magát a szöveget a gyors szimmetrikus módszerrel titkosítjuk és a titkosításhoz használt szimmetrikus kulcsot – aszimmetrikusan titkosítva – a rejtjelezett szöveggel együtt küldjük el. A titkosítás egy gyors módszerrel történik, a kulcs biztosítása viszont egy lassú, de nagyon biztonságos(nak tekintett) módszerrel. A nyilvános kulcsú algoritmusok inkább kiegészítik a szimmetrikus algoritmusokat, segítik azokat – különös tekintettel a közöttük lévő elvi különbségre, amely funkcionálisan is elhatárolja a két csoportot egymástól.

6.4. TITKOSÍTÁS, MINT FEGYVER?!

Az Amerikai Egyesült Államok korábban nagyon erősen szabályozta a titkosítási termékek használatát, és főleg az USA-ból történő exportját: a titkosítóeszközök a fegyverexport hatáskörébe tartoztak. A hatóságok attól tartottak, hogy a korszerű kriptográfiai módszerek ellenséges hatalom, terroristák vagy bűnözők kezébe kerülve veszélyt jelentenek, ezért törvényekkel korlátozták a legálisan használható algoritmusok kulcshosszúságát, illetve kikötötték, hogy csak az a cég exportálhat titkosító technológiát, aki a kulcsokat letétbe helyezte az államnál (*key escrow*). A hírhedt *Clipper chip* (1993) esetében minden chip egyedi kulcsáról volt egy „biztonsági" másolat a megfelelő állami szervnél. Sajnos ilyen esetben csak a hírszerzéssel foglalkozó szervezetek „becsületszava" az egyetlen biztosíték arra, hogy a kulcsokat csak a törvény által megengedett esetekben használják fel. Ha a kulcsokat nem helyezték letétbe, csak olyan kulcsméretet lehetett exportálni, ami az NSA számára még *brute-force* módon megfejthető volt. Ezért sokáig nem volt szabad olyan szimmetrikus titkosítási terméket exportálni az USA-ból, amely 40 bitnél hosszabb kulcsot tudott használni, vagy könnyen átalakítható volt erősebb titkosításra. Ez a korlátozás később csökkent, köszönhetően azoknak a mozgalmaknak, akik egyre erősebben követelték a kriptográfia teljes liberalizációját. Ezekben a mozgalmakban a kriptográfia terjedésében üzletileg érdekelt szoftvergyártókon kívül olyan emberek is részt vettek, akik az „elektronikus privátszféra" védelmét tűzték ki célul: az államnak nincs joga a megfejtéshez szükséges kulcsokhoz, hiszen a privát kommunikációt akkor is védi az alkotmány, ha az telefonvonalon vagy számítógéphálózaton zajlik[38]. A „kripto-aktivisták" attól tartottak, hogy a digitális korban az állam a kulcsok birtokában Nagy Testvér jellegű hatalomhoz juthat, ezért célul tűzték ki a kriptográfia kijuttatását a civil szférába. '91 előtt csak a kormányzati szervek és a nagy cégek használhattak titkosítást, de ebben az évben *Philip Zimmermann* olyan programot írt, amely az RSA algoritmust e-mail titkosítására tette alkalmassá, a PGP - *Pretty Good Privacy* - névre keresztelt programot pedig közkézre adta az Interneten. A hatóságok a fegyverexportra vonatkozó törvény megsértésével vádolták, és sokáig, mintegy három éven keresztül zaklatták a programozót. Átmeneti megoldásként a fejlesztők könyvben adták ki a program teljes forráslistáját. Az USA-ból viszont bármilyen tartalmú könyv exportálható volt, így Európában a könyvből szkennelt és lefordított változat kezdett terjedni. A program az egész világon kivívta az elismerést: részben azért, mert a PGP-vel már megjelenésekor is 128 bites kulcsokat lehetett használni, részben pedig azért, mert a felhasználói felülete mögött átgondolt, biztonságos, fontos ügyviteli szabályokat betartó programmag húzódott meg.

A kormányzat reakciója egyúttal értékelés is a PGP által használt algoritmusokról és magáról a PGP rendszerről. Az *NSA* egyik igazgatója a következőket nyilatkozta [23]: „*Ha a világ összes személyi számítógépét – 260 millió – arra használnánk, hogy feltörjünk egy PGP-vel titkosított üzenetet, becslések szerint még így is átlagosan 12 milliószor annyi időben tellene, mint ahány éves az univerzum.*" (*William Crowell*, 1997. március)

[38] Az amerikai Alkotmány negyedik kiegészítése (Fourth Amendment) a magánszférát védi, és a jogosulatlan megfigyelés és vizsgálat elleni védelmet garantálja. Jogi vitákban igen komolyan veszik.

> Érdemes egy kicsit elgondolkodni az amerikai kormányzat magatartásán. A kiviteli korlátozás valóban jogos volt abban a tekintetben, hogy eleve nem engedett olyan technológiát idegen kézbe, amit a védelmi szervek nem tudnak megfejteni. Azonban ez az indoklás feltételezi, hogy az amerikai technológia a legjobb a világon és mindenki más csak olyan módszerek kifejlesztésére képes, ami az amerikai illetékes hivataloknak nem okoz gondot. Azonban ez a gondolkodásmód az amerikai önérzet növelésén kívül semmire sem jó, hiszen rengeteg példa mutatta és mutatja ma is, hogy az USA területén kívül is kiváló titkosító algoritmusokat, rendszereket készítenek és terveznek. Arról nem is szólva, hogy a titkosító algoritmusok leírásai, specifikációi mindeközben (korábban is) nyilvánosak voltak…
>
> Másrészről a korlátozás piaci korlátozást is jelentett, mert megakadályozta, hogy jó nevű, vezető amerikai informatikai és szoftverkészítő cégek a nemzetközi piacon is megjelenhessenek kriptográfiai eszközeikkel és termékeikkel. [20,21]

A titkosító eszközök exporttilalma 1996-98 óta folyamatosan gyengült [21], illetve más besorolás alá került. Most már lehet hosszabb – például 128 bites – titkos kulcsú eszközöket is exportálni az USA-ból. Magyarországon a polgári titkosítás kérdéseit pillanatnyilag semmi nem szabályozza, nincsenek olyan megszorító határozatok, amelyek megtiltanák vagy korlátoznák (általában véve szabályoznák) a titkosítás használatát.

6.5. BIZALMASSÁG VAGY BIZTONSÁG?

Mi a különbség a címbeli két kifejezés között? Jelen könyv keretein belül nem sok: ha egy levelet bizalmasan el tudunk juttatni a címzetthez, biztonságos kommunikációt valósítunk meg. Nekünk ez elég, mással nem foglalkozunk. Azonban az általános informatikai biztonság nem mossa össze ennyire ezt a két szót. A biztonság alatt inkább a vírusok, a behatolás és trójai falovak észlelését, illetve ezek elleni védelmet érti.

Képzeljünk el egy céget, ahol mindenki számára engedélyezett a titkosított levelezés használata! Emellett vegyük figyelembe azt is, hogy a publikus kulcsok publikus természetét sokan úgy értelmezik, hogy azokat lépten-nyomon hirdetik, weblapra helyezik, stb. Így azután ismeretlen feladóktól is kaphatnak titkosított levelet, a titkosítás pedig megvédi a levél tartalmát a kíváncsiskodók elől. Azonban a kíváncsiskodás a védelem eszköze is lehet egyben: a repülőtéren sem azért ellenőrzik a csomagjainkat, hogy mit viszünk fel, hanem azt ellenőrzik, hogy tiltólistán szereplő tárgyakat nem viszünk-e fel! De mit szól egy központi vírusirtó szoftver egy titkosított bejövő levélhez? Semmit. Vagy beengedi, vagy nem. Ha beengedi a levelet és az vírust, férget vagy egyéb „kedves ajándékot” tartalmaz, akkor olyan, mintha ott sem lenne a védelem: a nemkívánatos tartalom biztonságban eljut a címzetthez és ott működésbe lép – ha nincs a munkaállomáson is víruskereső. A központi vírusellenőrzők jelenlétét nemcsak céges levelezőszerverek esetén tételezhetjük fel, hanem minden magára valamit is adó mailszolgáltató üzemeltet vírusvédelmi szoftvert a bejövő levelek ellenőrzésére és a felhasználók védelmére.

Ez azt jelenti, hogy a titkosítás alkalmazásával saját magunkat tesszük védtelenné és fel kell adni a bizalmasságot a biztonságért cserébe? Addig, amíg a kéretlen küldemények is használhatják a védett útvonalat, félig-meddig sajnos igen. Szerencsére ez a kényszerhelyzet csak az elektronikus levelezésre jellemző, illetve ott a legveszélyesebb. Általános gyógyszerként lássuk el munkaállomásunkat is védelemmel, és ami a legfontosabb – egyébként általános

levelezési – szabály: soha ne nyissunk meg olyan mellékletet, amelyet ismeretlen forrásból, kéretlenül kaptunk! Másrészt talán mégsem kell annyira publikusan kezelni a nyilvános kulcsot, csak annak adjuk oda, aki kéri és „megérdemli" ...

Másik szempont a sokat bírált, sok vita tárgyát képező Nagy Testvér érzés, amikor egy cég vezetése megköveteli a kimenő levelek (általában a kimenő hálózati forgalom) tartalomfigyelését. Sokan negatívan reagálnak az ilyen hírekre – akár érintettek, akár nem – és általában a cég döntése ellen vannak: támogatják a titkosítás használatát, mert az lehetetlenné teszi a szűrést. Persze, ha nekik lenne egy cégük, ahol nekik kellene felelni a biztonságért, már más véleményen lennének, és nem értelmeznék olyan leleményes módon a „személyes használatra átadott számítógép" és a „cég tulajdonában lévő eszközök használatának" fogalmát...

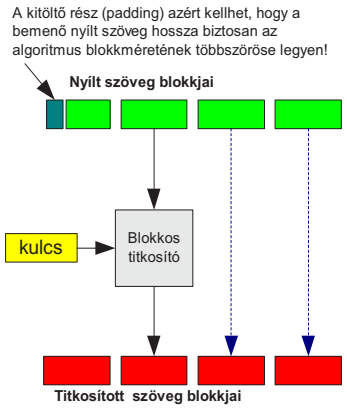
Van egyensúly? Ha valaha is döntési helyzetbe kerül az Olvasó – hogy használjon-e titkosítást vagy sem – legyen tekintettel a fenti néhány gondolatra is!

Nem igazak azok az állítások, amelyek szerint a kvantumkomputerek már csak pár évnyi távolságra vannak tőlünk. [...] Kínosan hasonlít a dolog ahhoz, mint amikor valaki összeállítja egy kártyavár legalsó szintjét, aztán magabiztosan kijelenti, hogy a többi tizenötezer emelet már csak merő formalitás.

Serge Haroche
francia kutató, 1998

7. A BLOKKOS REJTJELEZŐK MŰKÖDÉSI MÓDJAI, ÉS A FOLYAMTITKOSÍTÓK VILÁGA

Az eddigi algoritmusok közös jellemzője, hogy a titkosítás műveletét az üzenet egy valahány bites blokkján hajtják végre, emiatt – nem meglepő módon – blokktitkosítóknak (*block ciphers*) hívjuk őket. A blokkok mérete általában kötött, de például az RC5 blokkmérete változó lehet. Az RSA esetében a blokk mérete a modulus függvénye. A másik nagy csoportot a *folyamtitkosítók* (*stream ciphers*) alkotják, amelyek az adatokat kisebb – általában egybájtos vagy egybites – egységekben dolgozzák fel. Ilyen például a már megismert *OTP*, de ebben a fejezetben példát láthatunk arra is, hogy hogyan alakíthatunk át egy blokktitkosítót folyamtitkosítóvá.



36. ábra A blokktitkosítás

Az egyes üzemmódok sok más tekintetben is megváltoztatják az eredeti kódoló tulajdonságait, például:

1. Változhat a titkosítás sebessége.
2. Változhat a titkosító módszer nyílt szöveg sajátosságaira gyakorolt hatása. A blokktitkosítók alapesetben szótárként viselkednek, így ugyanaz a nyílt szöveg ugyanazzal a kulccsal mindig ugyanarra a titkosított szövegre képződik le.
3. Változhat még a rendszer hibatűrése is, tekintettel a kommunikáció során előforduló hibákra: egy blokk megváltozására vagy éppen egy blokk kiesésére vagy ismétlődésére. A továbbiakban azt is megvizsgáljuk, hogy az ismertetett módok hogyan reagálnak a következő hibatípusokra:

   - Ha egy küldött blokk tartalma megváltozik, de a továbbított bitek száma változatlan marad, *bithibáról* beszélünk.
   - Ha egy továbbított teljes blokk elvész vagy megismétlődik, *blokkszinkronhibáról* van szó. E hiba fontos tulajdonsága, hogy a vételi oldalon az adatbitek blokkhatárhoz viszonyított helyzete nem változik meg.
   - Amikor a szinkronhiba nem blokkokban, hanem csak bitekben mérhető: ilyenkor néhány bit kiesik vagy megismétlődik a továbbított bitfolyamból. Ez egyúttal azt jelenti, hogy a kieső vagy megduplázódott biteket követő adatbitek blokkhatárhoz viszonyított helyzete is megváltozik. Ezt a típusú hibát a jelenleg ismertetett üzemmódok nem tudják javítani. (*bitszinkron hiba*)

	101011|110101|011101	bithiba
101010|110101|011001 →	101010| |011001	blokkszinkon~
	101010|1.01010|11001	bitszinkon~

7.1. ELEKTRONIKUS KÓDKÖNYV

A blokkos rejtjelezők legegyszerűbb működési módja az *elektronikus kódkönyv* mód (*Electronic Code Book, ECB*). Ekkor egy nyílt blokkhoz az eljárás mindig ugyanazt a (kulcstól függő) titkosított blokkot rendeli. Ha a kódtörő egyszer már rájött, hogy például a „8B357CBB” nyílt blokkot egy adott titkosítási viszonyban mindig a „2D8FB366” képviseli, azt közvetlenül meg tudja fejteni, amikor lehallgat egy új üzenetet. Az egész titkosítási módszer egy szótárhoz lesz hasonló. Ha a kulcsot nem cseréljük le bizonyos időközönként, a lehallgató személy (ha tevékenységét rendszeresen végzi) egyre több kódpárt ismerhet meg.

1. nyílt szöveg blokkjai:	m_1 / M_1	m_2 / M_2	… / …	m_k / M_k
2. titkosítás/megfejtés:	↓	↓		↓
3. rejtjeles szöveg blokkjai:	M_1 / m_1	M_2 / m_2	…/ …	M_k / m_k

Az ECB mód tulajdonságai

- Sebessége azonos az algoritmuséval, egy titkosító vagy megfejtő lépésben (legalább) egy teljes blokkot kapunk eredményül.
- Az algoritmust C és D módban egyaránt használja, de a titkosítás és a megfejtés során egyforma a blokkok kezelése: a két folyamat felépítése megegyezik, csak az algoritmust kell a C és D módok között váltogatni.
- A nyílt szöveg nem minden sajátosságát rejti el, néhány statisztikai jellemző változatlan marad.
- A blokkok egymástól teljesen függetlenek, ezért a megfejtés és a titkosítás folyamata párhuzamosan végezhető. Ha több kriptoegység áll rendelkezésre, azok a beérkező blokkokat szintén párhuzamosan dolgozhatják fel. (Vagyis ha rendelkezésre áll m_0, m_1, m_2, m_3 ..., akkor M_1, M_2, M_3 ... egy lépésben megkapható és ez fordítva is igaz.)
- Az algoritmus blokkmérete megköti a kommunikáció blokkméretét.
- Hibák
  - Bithibák: nem terjednek szét (*error propagation*), csupán a sérült blokk lesz olvashatatlan.
  - Blokkszinkron-hibák: nem okoznak katasztrófát, a többi megfejtett adatblokk sértetlen marad, de értelemszerűen a vett blokkok sorában is megjelenik a szinkronhiba.

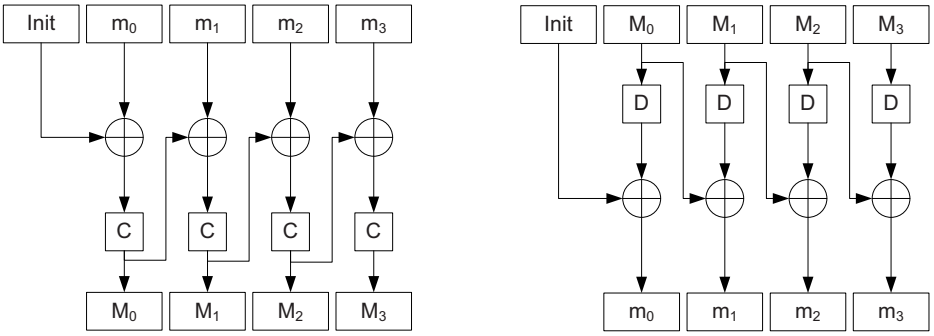
7.2. A REJTJELES BLOKKOK LÁNCOLÁSA

A rejtjeles blokk láncolása (*Cipher block chaining, CBC*) az aktuális blokk titkosításának eredményét felhasználja a következő blokk titkosításához is. A nyílt szöveg és az előző rejtjeles blokk között egy XOR műveletet hajtunk végre, mielőtt azt titkosítanánk. Ezért ugyanaz a

nyílt blokk nem mindig ugyanarra a rejtjeles blokkra fog átalakulni, mert minden egyes blokk titkosításának eredménye függ minden őt megelőző blokktól.

$$M_i=C_k(m_i \oplus M_{i-1}) \qquad m_i=D_k(M_i) \oplus M_{i-1}$$

Ha az adatokat ilyen módon dolgozzuk fel, szükség van egy véletlen kezdőblokkra is, amivel az első blokkot a titkosítás előtt XOR-oljuk. Az inicializáló vektort (IV) nem kell feltétlenül titokban tartani: lehet véletlenszám, egy sorszám, esetleg egy időpecsét is. Természetesen az IV használata nem kötelező – ekkor IV=0 –, de alkalmazásával elérhető, hogy ugyanaz a nyílt szöveg változatlan kulcs mellett is mindig más és más rejtjeles blokkra képződjön le.



37. ábra A rejtjeles blokkok láncolása.
Bal oldalon a titkosítás, jobb oldalon megfejtés logikai folyamata

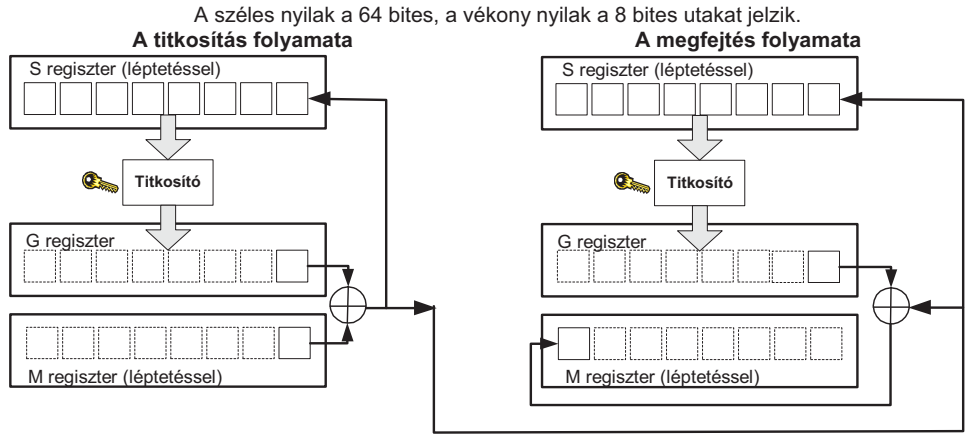
A CBC mód tulajdonságai

- Sebessége azonos az algoritmuséval, egy titkosító- vagy megfejtőlépésben (legalább) egy teljes blokkot kapunk eredményül.
- Az algoritmust *C* és *D* módban egyaránt használja, de a blokkok kezelése miatt csak a megfejtés párhuzamosítható. A titkosítás folyamán M_0, M_1, M_2...nem állítható elő egyszerre, mert M_i kibocsátáshoz szükség van M_{i-1}-re is. Megfejtéskor a rendelkezésre álló M_0, M_1, M_2... blokkokból egyszerre, egy ütemben előállítható m_0, m_1, m_2...
- A nyílt szöveg sajátosságait elrejti. Ha ugyanazt a nyílt szöveget ugyanazzal a kulccsal többször is elküldjük, a rejtjeles szöveg mindig más és más lesz, a kezdő vektor függvényében.
- A blokkok összefüggése miatt a rejtjeles üzenet nehezebben manipulálható.
- Hibák:
  - Bithibák: kis mértékű szétterjedés tapasztalható, mert nemcsak a rossz blokk megfejtése ad rossz eredményt, hanem a következőé is, mivel ahhoz felhasználjuk magát a hibás blokkot is.
  - Blokkszinkron-hiba esetén a kieső blokk helyére lépő vagy beékelődő blokk megfejtése rossz eredményt ad, de a következő blokkok megfejtése helyes lesz.

7.3. VISSZACSATOLÁSOS MÓDOK

7.3.1. A titkos szöveg visszacsatolása

A titkos szöveg visszacsatolása (*Ciphertext FeedBack, CFB*) nevű módban a kimenet visszacsatolása miatt az eddigiektől eltérő elvi működést kapunk eredményül. Az előző módszereknél a kulcsot közvetlenül a nyílt szöveg titkosításához használtuk fel. Az *CFB* esetében a kulcsot csak közvetett módon használjuk az üzenetek feldolgozásához. A visszacsatolás általában a blokkméretnél (például 64bit) kisebb egységekben (például 8bit) valósul meg. (Az alábbi magyarázatok is ezeket az értékeket tételezik fel.)



38. ábra A titkos szöveg visszacsatolása

A *38. ábra* segítségével nézzük meg a működést, elsőként a titkosító oldalt:

1. Elsőként a felső 64 bites *S* léptetőregisztert feltöltjük egy véletlen kezdővektorral.
2. A titkosítandó szöveget betöltjük az *M* léptetőregiszterbe, amely szintén 64 bites.
3. Az *S* regiszter tartalmát, mint nyílt szöveget titkosítjuk a *K* kulccsal.
4. Az eredmény a szintén 64 bites *G* regiszterbe kerül. Ennek első nyolc és az *M* regiszter első nyolc bitje közötti XOR művelet adja a titkosított blokk első nyolc bitjét.
5. A XOR művelet eredményét az *S* regiszterbe is betoljuk és az *M* regisztert is léptetjük. Ha rendelkezésre áll a nyílt szöveg következő nyolc bitje, az *M* regiszter végén lévő üres helyre máris berakható.
6. Az előző három lépést addig ismételjük, amíg a nyílt szöveg el nem fogy.

A megfejtés folyamata a következő:

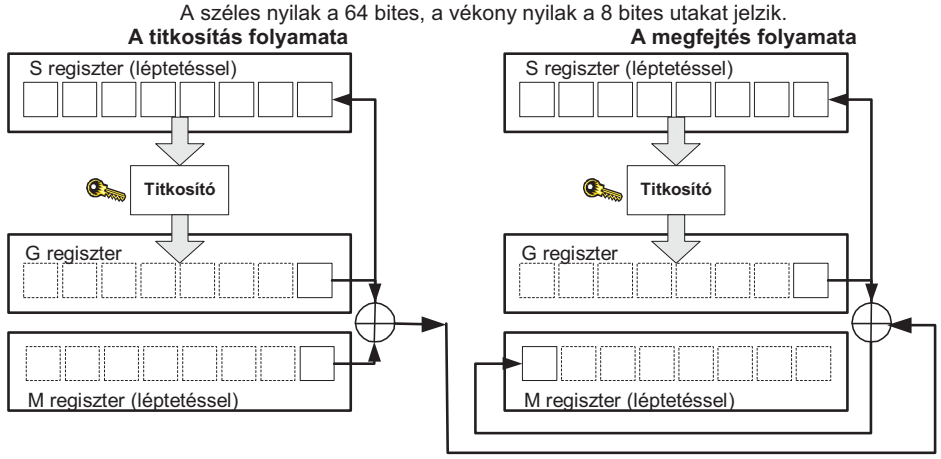
1. Első lépésként az *S* léptetőregisztert feltöltjük a titkosításhoz használt 64 bites kezdővektorral.
2. Az *S* regiszter tartalmát, mint nyílt szöveget titkosítjuk a *K* kulccsal és az eredményt *G* regiszterbe töltjük.
3. A *G* első nyolc bitje és a bemeneten beérkező rejtjeles nyolcbites blokk között XOR műveletet végzünk. Ez lesz a 64 bites nyílt szöveg első nyolc bitje, amit betolunk *M* regiszterbe is.
4. A következő rejtjeles blokkot beléptetjük az *S* regiszterbe.
5. Az előző három lépést addig ismételjük, amíg a rejtjeles szöveg el nem fogy.

A CFB mód tulajdonságai

- Lassabb adatmozgást biztosít, mint maga a titkosítóalgoritmus, mert egy kriptográfiai lépésben csak nyolc bit áll elő. Bár az eredeti algoritmus (például 64 bites) blokkokat használ, az adatforgalom kisebb egységekben (például 8 bit) is történhet. Alkalmazása ezért előnyös lehet terminál - hoszt kapcsolatokban (például `telnet`). Természetesen annak sincs akadálya, hogy a nyolc ütem eredményét összegyűjtsük egy újabb regiszterben és egyszerre továbbítsuk azt, megőrizve így az eredeti blokkméretet.
- Az algoritmust csak titkosító módban használja. (Így a gyakorlatban csak szimmetrikus kódoló használható, *nyilvános kulcsú nem*.)
- A blokkok összefüggése miatt a rejtjeles üzenet nehezebben manipulálható. A kimeneten megjelenő titkosított adat közvetlen módon függ a megelőző 8 bájttól. Mivel ez igaz az elődökre is, így végül minden őt megelőző bájt hatással van rá.
- A nyílt szöveg sajátosságait jól elrejti. Ha esetleg ugyanazt a nyílt szöveget ugyanazzal a kulccsal többször is elküldjük, a kimeneten mindig más és más eredményt kapunk a kezdővektor függvényében. Nem is szólva a korábbi titkosítás eredményéről, ami szintén hatással van az eredményre!
- Visszacsatolás csak a titkosító oldalon van.
- Hibák:
  - Bithibák: nem okoznak katasztrófát, legfeljebb 8 ütem eredménye lesz hibás, ami egy teljes, nagy blokk elromlását jelenti. Ezután a hibát okozó kis blokk kilép a *S* regiszterből, és nem befolyásolja tovább a megfejtő működését.
  - Blokkszinkron hiba esetén a folyamatosan belépő újabb és újabb rejtjeles blokkok miatt legkésőbb kilenc lépés után automatikusan korrigálódik a hiba, ezért a CFB módot önszinkronizáló (*self synchronizing*) módnak is hívják.

7.3.2. A kimenet visszacsatolása

A kimenet visszacsatolása (*Output FeedBack, OFB*) mód működése hasonló a *CFB* működéséhez, azonban nem a titkosított szöveget, hanem a belső titkosítás eredményét csatoljuk vissza. Ezért ezt a visszacsatolást gyakran *belső visszacsatolásnak* is nevezzük.



39. ábra A kimenet visszacsatolása

A fenti ábra segítségével nézzük meg a működést, elsőként a kódoló oldalét:

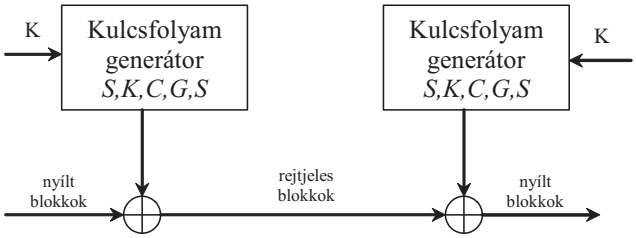
1. Első lépésként az *S* léptetőregisztert feltöltjük egy 64 bites kezdővektorral.
2. A titkosítandó szöveget betöltjük a 64 bites *M* léptetőregiszterbe.
3. Az *S* regiszter tartalmát, mint nyílt szöveget titkosítjuk a *K* kulccsal.
4. Az eredmény a *G* regiszterbe kerül. Ennek első nyolc bitje és a nyílt szöveg első 8 bitje között XOR műveletet végzünk, ami a titkosított blokk első nyolc bitje lesz.
5. A *G* regiszter XOR-oláshoz is felhasznált 8 bitjét betoljuk az *S* regiszterbe. Az *M* regisztert léptetjük. A nyílt szöveg újabb 8 bitje az üres helyre berakható.
6. Az előző három lépést addig ismételjük, amíg a nyílt szöveg el nem fogy.

A megfejtés folyamata a következő:

1. Első lépésként az *S* léptetőregisztert feltöltjük a kódoláshoz használt 64 bites kezdővektorral.
2. Az *S* regiszter tartalmát, mint nyílt szöveget titkosítjuk a *K* kulccsal, az eredményt *G*-be rakjuk.
3. A *G* első nyolc bitje és a bemeneten beérkező rejtjeles szövegblokk között (amely szintén 8 bites) XOR műveletet végzünk. Ez lesz a 64 bites nyílt szöveg első nyolc bitje, amit betolunk *M* regiszterbe is.
4. A *G* regiszter XOR-oláshoz felhasznált 8 bitjét betoljuk az *S* regiszterbe is.
5. Az előző három lépést addig ismételjük, amíg a rejtjeles szöveg el nem fogy.

A OFB mód tulajdonságai

- A belső visszacsatolás miatt a *G* regiszterben megjelenő értékek a kezdővektor függvényében determinisztikusak és periodikusak, illetve ami a legfontosabb: függetlenek a titkosítandó vagy titkosított szövegtől. A belső visszacsatolás tulajdonképpen egy bitsorozatot generál, amit a XOR művelethez használunk fel, így gyakorlatilag a már megismert *One Time Pad*-hez hasonló kódoló alakul ki (*40. ábra*). Amennyiben a periódus hosszú vagy a titkosítás hamarabb befejeződik, mint a periódus a végére érne és egy új titkosításhoz új kezdővektort használunk, a periodikus bitsorozatból következő biztonsági problémák enyhülnek. (A kezdővektor viszonykulcsként viselkedik.)
- Lassabb adatmozgást biztosít, mint maga a titkosító algoritmus. Bár az eredeti algoritmus (például 64 bites) blokkokat használ, az adatforgalom kisebb egységekben (például 8 bit) is történhet. Alkalmazása ezért előnyös lehet terminál - hoszt kapcsolatokban (például `telnet`). Természetesen annak sincs akadálya, hogy a nyolc ütem eredményét összegyűjtsük egy újabb regiszterben és egyszerre továbbítsuk azt, megőrizve így az eredeti blokkméretet. Másik megoldás a blokkméret megőrzésére, ha a belső visszacsatolás blokkmérete megegyezik a kódoló algoritmus blokkméretével.
- A nyílt szöveg sajátosságait elrejti. Ha esetleg ugyanazt a nyílt szöveget ugyanazzal a kulccsal többször is elküldjük, a kimeneten mindig más és más eredményt kapunk a kezdővektor függvényében.
- Az algoritmust csak titkosító módban használja. (Így a gyakorlatban csak szimmetrikus kódoló használható, *nyilvános kulcsú nem*.)
- A rejtjeles blokkok függetlenek egymástól (az egyik megváltozása nem befolyásolja a másikat), ezért az adatok viszonylag könnyen manipulálhatók.
- Mindkét oldalon van visszacsatolás.
- Hibák:
  - A bithibák nem terjednek szét, mivel az utazó csomagok nem kerülnek be sem regiszterbe sem a visszacsatolásba, kizárólag a hibás blokk lesz olvashatatlan.
  - Blokkszinkron hibák nem javíthatók. A *CFB* móddal ellentétben itt a blokkok nem befolyásolják a kulcsfolyam-generátor működését, így az nem tud egy esetleges szinkronhibához igazodni sem. Minden további blokk hibás lesz.
  - A kulcsfolyam generátorok egymáshoz viszonyított szinkronhibája (mivel teljesen függetlenek) szintén nem javítható.



40. ábra Az OFB függetlensége

7.4. A BLOKKOS MŰKÖDÉSI MÓDOK ÖSSZEHASONLÍTÁSA

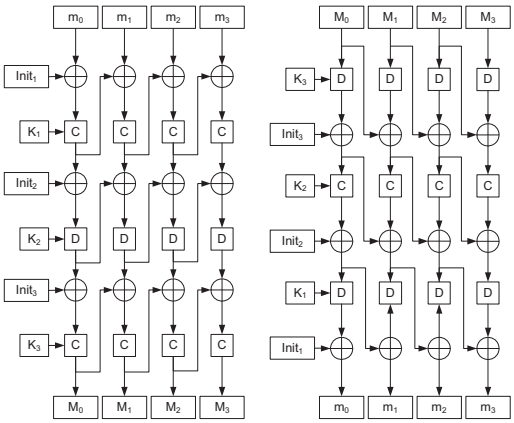
Az ismertetett tulajdonságok alapján láthatjuk, hogy a különböző működési módok sokszor nagyon különböző viselkedést eredményezhetnek. Hatással vannak a megvalósítás sebességére, hibatűrésére, de még a biztonságára is. Egyik módban csak szimmetrikus algoritmus használható, a másiknál szabadon választhatunk a szimmetrikus és az aszimmetrikus megoldás között. Nem kívánom különösebben értékelni vagy rangsorolni a bemutatott működési módokat (főként, hogy van még másmilyen is), de felhívnám a figyelmet arra, hogy az *ECB* mód használatát mindenki kerülje! Az a tény ugyanis, hogy egy adott nyílt szöveg egy adott kulcs mellett mindig ugyanazt a titkos szöveget adja, a forgalomfigyelésen alapuló (*traffic analysis*) támadásokat is segítheti. Az alábbi táblázatban összefoglalva láthatjuk ezeket az eltéréseket.

A blokkos működési módok fontosabb tulajdonságainak összehasonlítása

	ECB	**CBC**	**CFB**	**OFB**
Sebesség a titkosító algoritmushoz viszonyítva	=	=	<	<=
Segít elrejteni a nyílt szöveg sajátosságait?	N	I	I	I
Az alapalgoritmust milyen módban használja?	C/D	C/D	C	C
Több blokk párhuzamosan titkosítható?	I	N	N	N
Több blokk párhuzamosan megfejthető?	I	I	N	N
A rejtjeles blokkok függnek egymástól?	N	I	I	N
Egy rejtjeles blokk bithibája hány *további* blokkot tesz tönkre?	0	1	0	0
Egy rejtjeles blokk blokkszinkron hibája hány *továbbit* tesz tönkre?	0	1	1	∞
A kulcsfolyam-generátor szinkronhibája javítható?	-	-	I	N

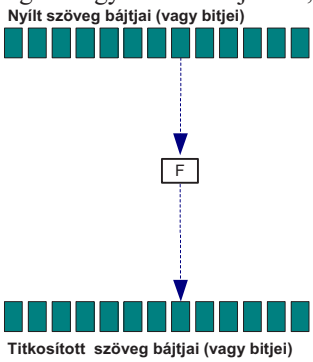
7.5. TÖBBFOKOZATÚ KÓDOLÓK

Az eddigi megoldások az olyan kódolókra vonatkoztak, amelyeknek egyszeres fokozatuk volt, vagy a kódolót egyszeres fokozatúnak tekintettük, függetlenül belső felépítésétől. Az olyan többfokozatú kódolóknak, mint például a *TripleDES*-nek más működési módja is lehet. Ilyen például a „belső - titkosított blokkok láncolása” mód *(inner-CBC mode)*. A jobb oldali ábrán a TripleDES-t láthatjuk *iCBC* működési móddal. Természetesen, ha van „belső”, van „külső” láncolás is *(outer-CBC mode)*, ez viszont nem más, mint a „sima” CBC, egyszerűen a három fokozatot fekete dobozként egyetlen fokozatként értelmezi.



7.6. STREAM CIPHERS - FOLYAMTITKOSÍTÓK

Az eddig bemutatott algoritmusok az adatokat sokbájtos csoportokban dolgozták fel. A másik nagy család a *folyamtitkosítók* (*stream ciphers*) családja, amely az adatokat kisebb – általában egybájtos vagy egybites – egységekben kezeli. A legnagyobb elvi eltérés, hogy a blokktitkosítók esetében egy adott blokk mindig ugyanarra képződik le (legalábbis ECB módban). Ezzel szemben a folyamtitkosítók transzformációja az időtől függ (pontosabban az adat adatfolyambeli elhelyezkedésétől), így egy blokknyi adat titkosított képe más lehet attól függően, hogy az adatfolyamban korábban vagy később jelenik meg. Ez egyúttal azt is jelenti, hogy a folyamtitkosító „emlékszik" arra, hogy hol is tart éppen, míg a blokkos titkosító nem (*memoryless*).



41. ábra Folyamtitkosítás

A folyamtitkosítók gyorsabbak és általában kisebb bonyolultságúak, mint blokkos társaik, különösen hardver implementációkban. Alkalmazásuk ott lehet különösen előnyös, ahol nincs lehetőség adatpufferelésre: egyszerűen nem várható meg, amíg 8-16 bájtnyi adat összegyűlik (GSM, szórt multimédia, telnet kapcsolatok, stb).

A szofveres megvalósítások univerzálisan használhatják a blokkos- és folyamtitkosítást: a fejezet eddigi részében több példát is láttunk arra, hogy egy blokkos algoritmus hogyan alakítható át folyamtitkosítássá. Tulajdonképpen elég, ha egy szoftveres alkalmazás a folyamtitkosításra rendezkedik be, ha a felhasználó valamiért valamelyik blokkos algoritmust választja, egyszerűen egy visszacsatolásos móddal folyamtitkosításként használja majd.

A kétféle elv másként viselkedik az egyes hibákkal szemben. A folyamtitkosítók bites-bájtos kicsi adatmennyiségéből az is következik, hogy egy-egy bit- vagy bájtszintű adathiba nem okoz nagy felfordulást. Sajnos ugyanez nem mondható el a szinkronhibáról, az általa okozott káosz általában végzetes.

7.6.1. Már megint egy régi elv: OTP

A folyamtitkosítók alapelve sem újdonság: a gyökerek a Vernam-titkosításhoz és annak „elvi" továbbfejlesztéséhez, az OTP-hez nyúlnak vissza. Mik is voltak a főbb jellemzők? Először is véletlen(szerű) kulcs használata, amely akármi lehet, nem függ senkitől és semmitől, ráadásul nem ismétlődik. Másodszor pedig a kulcs hossza legalább akkora, mint maga az üzenet, így egy b bites üzenet 2^b különböző kulccsal titkosítható.

Jó nagy kulcs

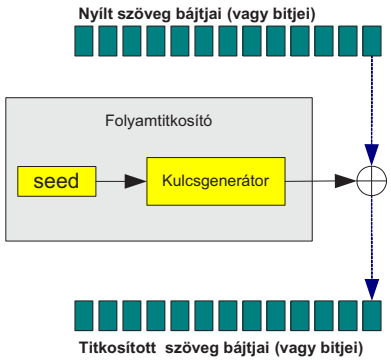
A legegyszerűbb megoldás szerint készítünk egy jó hosszú kulcsot, ebből kivágunk egy üzenetnyi darabot és összegyúrjuk az üzenettel. Az összegyúrás legtöbbször (de nem kizárólag) a XOR művelettel valósul meg, részben rendkívül egyszerű megvalósítása, részben pedig

amiatt, hogy a művelet önmaga inverze. (Ha a ⊕ műveletet összeadásnak tekintjük, akkor a kivonást szintén a ⊕ művelettel kell elvégezni!) De mi történjen, ha elfogy a kulcskészlet?

> Elegánsan elhanyagoljuk, hogy a kulcsfolyam és a nyílt szöveg összegyúrására nem kizárólag a XOR művelet alkalmas, hanem más függvények is. Mivel erre a feladatra a XOR a legelterjedtebb, mi is ezt a modellt követjük (*binary additive ciphers*). E megoldásnak azonban van egy kissé kényelmetlen tulajdonsága: ha valamelyik ciphertext bitet megpiszkáljuk, mondjuk „0”-ról „1”-re állítjuk, akkor a neki megfelelő nyíltszövegbeli bit is (és csak az) az ellenkezőjére változik az üzenet megfejtése során. Ezt a tulajdonságot, ami a titkos szöveg megváltoztatásának nyílt szövegre gyakorolt hatását ennyire kiszámíthatóvá teszi, *malleability*-nek nevezzük.

Generált kulcs

Persze, ha nem tudjuk előre, hogy mekkora az üzenet, felvetődhet néhány kérdés: milyen hosszú kulcsot készítsünk, ha nem ismerjük a titkosítandó üzenet hosszát, vagy hol tároljunk és hogyan továbbítsunk ilyen gigantikus kulcsokat? A gyakorlat megválaszolta a fenti kérdéseket, igaz, többé-kevésbé a biztonság rovására menő módon. Ha hosszú kulcs kell, de nem kívánjuk vagy nem tudjuk azt előre letárolni, hát generáljuk menet közben akkor, amikor szükség van rá! (Jelen fejezet egyébként nagyon szoros és kölcsönös barátságban van a 2.6.2. fejezettel.)



42. ábra Folyamtitkosítás kulcsgenerátorral

Az így készült kulccsal szembeni elvárások hasonlítanak az OTP eredeti, ideális kulcsának tulajdonságaira:

- legyen véletlen(szerű) és
- legyen (nagyon) hosszú!

A véletlenszámokkal foglalkozó alfejezetben láttuk, hogy aritmetikai (általában determináns) eszközökkel a véletlenség nem teljesíthető, ezért legalább véletlenszerűnek kell lennie. A hosszúság kérdése is ehhez kapcsolódik: amikor a kulcsgenerátor felvesz egy olyan állapotot, amiben már előzőleg járt, ugyanúgy fogja folytatni működését, ahogy az már egyszer tette, innentől pedig periodikussá válik a működése. Úgy tűnik, hogy a folyamtitkosítók lelke a kulcsgenerátor… és ez így is van!

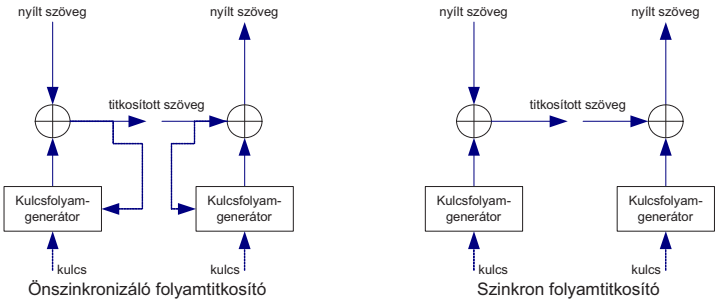
Szinkron és önszinkronizáló titkosítók

Attól függően, hogy a titkosítás folyamata miként reagál a szinkronhibákra, az alábbi csoportokat különbözetjük meg:

- Ha a műveletben a titkos és a nyílt szöveg is részt vesz, akkor önszinkronizáló generátornak (*self synchronizing keystream generator*) nevezzük. Az elnevezés arra utal, hogy a kulcsfolyam képes korrigálni az üzenetekből kieső vagy beékelődő bitek, bájtok hatását. (Ezt láthattuk a *CFB* bemutatásakor is!) Az érkező titkosított adatfolyam megfejtése bár-

mikor elkezdhető, a kulcsfolyam néhány lépésen belül automatikusan „rááll” a helyes értékre. E tulajdonság az adatszórásos kommunikációban használható ki leginkább (ahol ugyanazzal az adatfolyammal kell kiszolgálni több vevőt egyszerre). Egy bithiba kisebb-nagyobb mértékben szétterjed.

- Ha a művelet a titkos és a nyílt szövegtől függetlenül zajlik, akkor szinkron generátornak (*synchronous keystream or running key generator*) nevezzük. Az elnevezés arra utal, hogy az adó- és vevőoldalnak szinkronban kell lennie a helyes működéshez, így a szinkronhibák tönkreteszik a kommunikációt. (*OFB* mód!) A szinkronitás azt jelenti, hogy a kulcsgenerátort egyszerre indítják és a vevő mindig pontosan tudja, hol tart az adó. Ha elveszik a szinkron (és ezt észre is veszik!), akkor a generátorok újraindítása, a kommunikáció újrakezdése szükséges[39]. Egyszerre több vevő kiszolgálása ugyanazzal az adatfolyammal rendkívül nehézkes. Egy bithiba nem terjed szét.



43. ábra Folyamtitkosítók

Az önszinkronizációra képes titkosításokban a kulcsfolyam generálását a titkosított adatblokkok is befolyásolják

7.6.2. Léptetőregiszteren alapuló kulcsgenerátorok

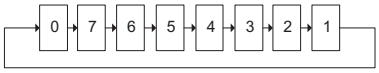
Most már csak az a kérdés, hogyan készítsünk egy ilyen bitkígyót? A fejezet első részében két olyan működési módot is láthattunk (*OFB, CFB*), amelyek annyira megváltoztatták blokktitkosítók jellemzőit, hogy végül folyamtitkosítókká alakultak. A céljaink eléréséhez valóban szükség van egy blokkos titkosítóra? Általában nincs.

Léptetőregiszter

Tároljuk el a bitjeinket olyan dobozkákba, melyeket egymás mellé helyeztünk. Valahogy így:

[39] Tisztára, mint az Enigma...

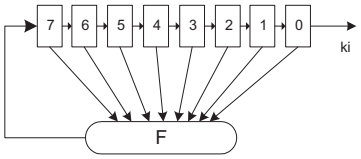
Miután minden bit a helyén van, valamilyen esemény bekövetkezésekor (például egy órajelre) a bitek vándoroljanak a jobb oldali szomszéd helyére! A kilépő 0. sorszámú bit kerüljön a 7. dobozba!



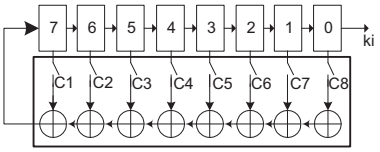
Ez nem nagy újdonság, ilyen művelettel eddig is találkoztunk, csak egyszerűen „forgatás”-nak hívtuk. Azonban vége a jó világnak, ebben a fejezetben ez a visszacsatolt léptetőregiszter! (Egészen pontosan: átvitel-visszacsatolt léptetőregiszter - *Feedback with Carry Shift Registers*)

Visszacsatolás

Most bonyolítsuk egy kicsit az életünket, és a belépő bit ne egyszerűen a kilépő másolata legyen, hanem a regiszterben lévő bitek aktuális értékétől függjön! Vezessük át a biteket egy olyan F függvényen, aminek egyetlen bit az eredménye, és ezt adjuk vissza a léptetőregiszternek! (Feltételezzük, hogy mire a léptetés elkezdődik, a függvény eredményének előállítása befejeződik.) Ezt valahogy így lehet szemléltetni:



Az F *állapot-függvény* (*next-state function*) helyén általában (de nem kizárólagosan) egy feltételes összeadót alkalmaznak, ami a beállításoktól függően az egyes biteket vagy összeadja, vagy nem. Az összeadást XOR művelet végzi. Ezt *lineárisan visszacsatolt léptetőregiszternek* (LFSR, *linear feedback shift register*) hívjuk. Ha az F helyén mondjuk egy DES van (és a kimeneti 64 bitből csak eggyet használunk fel), *nemlineárisan visszacsatolt léptetőregiszterről* (FSR, *feedback shift register*) beszélünk. A dobozkák és függvény közötti kapcsolatot fixen is be lehet állítani (huzalozni, programozni), de hosszútávon célszerű ezt szabadon programozhatóvá tenni. Ezt az alábbi ábrán kis kapcsolók (C1-C8) jelölik és a valóságban AND kapukkal valósítják meg:



Ha a kapcsoló nyitva van, akkor a XOR kapcsolódó bementére 0 kerül, egyébként pedig a hozzátartozó bit értéke. Ha az összes kapcsoló nyitva van, akkor a léptetőregiszter nyolc lépésen belül kinullázódik, akármi volt a tartalma előzőleg. Ha a kapcsolók közül legalább néhány

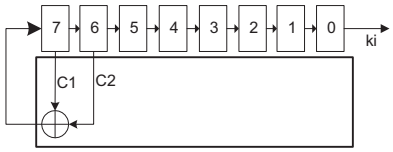
zárva van, a regiszterben tárolt érték elég „vadul” alakulhat, így a 0. pozíción megcsapolt kimenet is[40]. Ha ez elég véletlenszerű, kulcsfolyamként használhatjuk!

> A kapcsolók állását un. *kapcsolati polinommal* is szokták jelölni:
>
> $C(x)=1 + c_1x^1 + c_2x^2 + c_3x^3 + c_4x^4 + c_5x^5 + c_6x^6 + c_7x^7 + c_8x^8 + \ldots$
>
> Ha a legnagyobb helyértékű c_i=1 (vagyis a kapcsoló zárva van), a polinomot „*nem szinguláris*”-nak, egyébként „*szinguláris*”-nak nevezzük. A „nem szinguláris” beállítással mindig periodikus bitsorozatot kapunk, míg a másik esetben előfordulhat, hogy 00000…000-ra vagy 11111….1111-re fut a sorozat.

A leghosszabb bitsorozat

Íme egy példa a kis eszközünk működésére! Legyenek zárva a C1 és C2 kapcsolók, a regiszter kezdőértéke legyen 10101010! A kapcsolóállások miatt az iménti ábra az alábbi, egyszerűbb formába önthető:

$X7_{+1} = X7 \oplus X6$ vagyis



Hogy mi is történik a fenti kis egységben, azt a mellékelt táblázat segítségével követhetjük nyomon.

Lépés	Bitek 76543210	Ki
0.	10101010	0
1.	11010101	1
2.	01101010	0
3.	10110101	1
4.	11011010	0
5.	01101101	1
6.	10110110	0
7.	11011011	1
8.	**01101101**	1

- Indulás előtt feltöltjük a kezdőértékkel, ez a 0. lépés.
- A következő ütemben alakul ki az 1. lépés eredménye, majd abból a 2. lépésé és így tovább.
- Sajnos a 8. lépés eredménye megegyezik az 5. lépés eredményével, ezért az 5., 6., és 7. lépések eredményei ismétlődnek a továbbiakban, ez mindössze 3 ismétlődő állapotot jelent.
- A kulcsfolyam periódusa 3 bit hosszú, a kimeneten megjelenő bitsorozat: 101,101,101,101,…

Hát ez a periódus elég rövidre sikerült, de más kapcsolóállás más bitsorozatot eredményez, más ismétlődési hosszúsággal. A következő táblázatban néhány, különböző kapcsolóállások mellett jelentkező periódushosszt gyűjtöttem össze.

Kapcsolók								Periódus
1	2	3	4	5	6	7	8	
		✓		✓				31 bit
				✓	✓			63 bit
					✓	✓		127 bit
		✓	✓	✓			✓	17 bit
			✓	✓	✓		✓	255 bit

[40] Egy másik módszer szerint az egyes bitek következő ütembeli állapotát egy sejt-automata állítja elő, ilyenkor a bitek értéke önmaguktól, valamint a jobb és baloldali szomszédjuktól függ. [70]

Számunkra a fenti beállításokból különösen az utolsó érdekes, ugyanis a generátor csak 255 állapotváltás után tér vissza egy előző állapotba. Tekintve, hogy a léptetőregiszter 8 bites, e sorozat hossza egyben a kicsikarható leghosszabb is. Általában is igaz, ha a regiszter L bites, az elérhető leghosszabb periódus 2^L-1, és ez csak a kapcsolati polinomtól függ. (A regiszterben megjelenő értékek közül a `000…00` hiányzik, ezért nincs 2^L állapot. Ha a regiszter felvenné a csupa `0000` állapotot, az elég kínos lenne, mert úgy is maradna! Hoppá! Ezek szerint eggyel kevesebb nulla van a sorozatban, mint egyes? Hát izé, igen…)

> Íme a fenti, leghosszabb periódushoz tartozó 255 bit:
>
> ```
> 0101010111110010100001001111111100001011110001101000000010001110001
> 0010111000000110010010011011100100000101011011010110010110000111110
> 1101111010111010001000011011000111100111001100010110100100010100101
> 0100111011101100111101111101001100110101000110000011 1
> ```

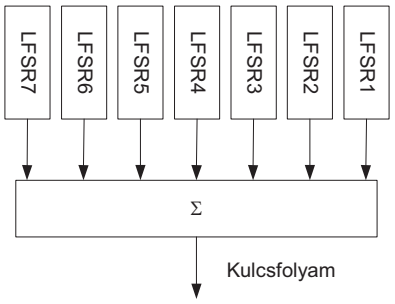
Léptetőregiszteren alapuló folyamtitkosítások

A fenti módon felépített kulcsfolyamgenerátor sok szempontból jónak mondható és használják is, ahol csak lehet, főként hardvereszközökben, szinkron módban. Ezt egyszerű felépítésének, sebességének, jó statisztikai tulajdonságainak, nagy periódusának köszönheti. Sajnos létezik azonban olyan algoritmus (Berlekamp-Massey algoritmus), amely a kimeneti bitfolyam hosszabb-rövidebb szakaszából képes kitalálni, hogy mi a generátor belső felépítése (mi a kapcsolati polinom) és képes egy olyan regisztertartalom meghatározására is, amellyel folytatható a bitfolyam. [11, 24]

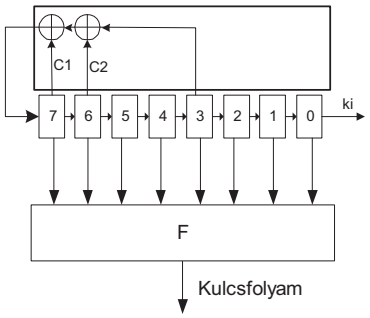
A linearitás problémája

Ennek legfőbb oka a lineáris visszacsatolásban keresendő, így a linearitás eltüntetése egyúttal gyógymód is a problémára:

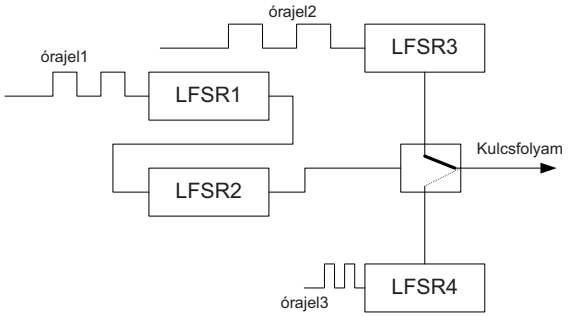
1. Több, egyszerre működő, különböző periódusú lineáris kulcsfolyamgenerátor kimenetét valamilyen nemlineáris módon „összegezzük” (például egy DES-re vezetjük), és a DES kimenetét használjuk kulcsfolyamként. Ez a nemlineáris függvény nem feltétlenül DES-szerűen bonyolult. Ha az egyes biteket b_i-vel jelöljük, lehet akár ilyen (egyszerűnek tűnő) is: $F=(x_1 \text{ and } x_2 \text{ and } x_3) \text{ or } (x_2 \text{ xor } x_4) \text{ or } (x_6 \text{ and } (x_4 \text{ xor } x_1))$



2. A kulcsfolyamgenerátor folyamatosan változó regisztertartalmát vezetjük egy nemlineáris függvényre (pl. DES-re), és annak kimenetét használjuk. Pontosan ezt a megoldást láttuk a fejezet elején, az OFB, CFB módokban.



3. Több, egyszerre működő, különböző periódusú lineáris kulcsfolyamgenerátort úgy kötünk össze, hogy azok egymást léptetik vagy külön órajel alkalmazásával váltogatunk közöttük. Az órajelet, így a léptetést befolyásolhatja még egy-egy regiszter tartalma is (erre lesz példa az A5 algoritmus). Az ilyen feltételes vezérlést *stop'n'go* vezérlésnek is hívják.



Az ilyen rendszerekben a felhasználó által szolgáltatott kulcs

- a kapcsolati polinomokat,
- a regiszterek kezdőértékeit vagy
- ha az összegző függvény kimenete sokbites, akkor a kimeneti bitet választhatja ki.

7.6.3. Léptetőregiszter-mentes kulcsgenerátorok

A szoftveres megoldásokban alkalmazott módszerek általában más algoritmusokkal készítik a véletlenszerű bitsorozatokat. Az, hogy a bitsorozat mennyire alkalmas biztonságos applikációk készítésére, előállításának módjától függ, ezért sokszor olyan algoritmuson alapulnak, amely már a gyakorlatban bizonyította megbízhatóságát, ezek a kriptográfiailag biztonságos álvéletlen bitgenerátorok (*cryptographically secure pseudorandom bit generators, CSPRBGs*). Sajnos jellemzően lassabbak, mint LFSR társak, így optimalizálásukra több figyelmet kell fordítani. A továbbiakban (mindössze) kettő ilyen algoritmust mutatok be, de hasonló módon a többi kriptográf algoritmusból is faragható bitstream…

RSA-alapú generátor 1.

Az RSA-alapú generátor biztonsága az RSA-probléma megoldásának nehézségén alapul, vagyis amíg az RSA biztonságos, addig ez a generátor is az. Ugyanolyan paraméterek kellenek a működéséhez, mint az RSA titkosításhoz vagy az aláíráshoz: két titkos (sőt, megsemmisített) prím és egy modulus. A paraméterek célzott megválasztásával (kis exponenssel, például $e=3$) elég jó sebesség érhető el. Emellett nem túl hatékony, sőt kifejezetten drága algoritmus, mert hatványozásonként mindössze egyetlen bitet kapunk.

RSA-alapú álvéletlen bitgenerátor (RSA PRBG)

Inicializálás:
- p és q prímek, melyek szorzata $n=pq$ valamint
- $f = (p-1)(q-1)$
- e = tetszőleges egész, e és f relatív prím, valamint $e<f$.
- L = a kért bitsorozat hossza

Kimenet:
- L darab álvéletlen bit: $z_1, z_2, \ldots z_l$

1. Válasszunk egy tetszőleges, de n-nél kisebb x_0 kezdőértéket!
2. for i:=1 to L do
   a. $x_i := x_{i-1}^e \bmod n$
   b. $z_i :=$ x_i legkisebb helyértékű bitje.
3. Kimenet: $z_1, z_2, \ldots z_l$

Az algoritmus sebességének sarokköve a moduláris hatványozás, amelynek sebessége tudatosan választott e kitevővel előre számítható illetve optimalizálható. Például e=3, 17, 65537 esetén rendre 2, 5, 17 moduláris szorzást kell elvégezni egy-egy bitért cserébe.

RSA-alapú generátor 2. – Micali-Schnorr generátor

Hatékonyabb megoldása is van a fenti elgondolásnak, ezt *Micali-Schnorr* generátornak hívják.

Micali-Schnorr álvéletlen bitgenerátor

Inicializálás:
- P és q prímek, melyek szorzata $n=pq$.
- N = n hossza bitekben mérve
- $f = (p-1)(q-1)$
- e = tetszőleges egész, e és f relatív prím, valamint $e<f$ és $80e \leq N$
- k = *egészrész(N(1 - 2/e)) és* $r=N - k$
- L = a kért bitsorozat hossza

Kimenet:
- $L*k$ darab álvéletlen bit: $z_1, z_2, \ldots z_l$

1. Válasszunk egy tetszőleges, de n-nél kisebb x_0 kezdőértéket, legfeljebb r bites hosszúságban!
2. for i:=1 to L do
   a. $y_i := x_{i-1}^e \bmod n$
   b. $x_i :=$ y_i legmagasabb helyértékű r darab bitje
   c. $z_i :=$ y_i legkisebb helyértékű k darab bitje
3. Kimenet: $z_1, z_2, \ldots z_l$

Az algoritmus sebessége ezzel jelentősen megnőtt: pontosan k-szorosra, vagyis minden egyes moduláris hatványozásért cserébe k darab bitet kapunk (az eddigi egy helyett). Például $e=3$, $N=1024$ szokásos értékek mellett, $k=341$ bitet készít minden ciklusban az algoritmus.

7.6.4. Titkosítsunk már!

Igazából még egy alfejezetet megérdemeltek volna az ún. alternatív generátorok. Az ilyen generátorok (némi túlzással) nem alapulnak semmin, nem használnak ki semmit, egyszerűen működnek. És jók. Ilyen például az RC4 is – amiről hamarosan szó lesz –, de tágabb értelemben pontosan ilyen folyamtitkosító volt az Enigma a II. Világháborúban. Most azonban itt az ideje elkezdeni a titkosítást!

Az általános modell

Csak az ismétlés kedvéért: miként is működik egy folyamtitkosítás?

- Az algoritmus a felhasználótól megkapja a titkosító kulcsot, amivel a (valamilyen modellen alapuló) kulcsgenerátort inicializálja. (*seed value*)
- A titkosítandó adat érkezésével párhuzamosan elkezdődik a kulcsfolyam generálása.
- Az éppen aktuális kulcsbiteket az éppen aktuális adatbitekkel összegyúrjuk (*combining*). Az éppen aktuális kulcsbiteket az éppen aktuális adatbitekkel összegyúrjuk (*combining*). Az éppen aktuális kulcsbiteket... és így tovább.

RC4 – alleged RC4

Az RC4 folyamkódolót Ron Rivest tervezte 1987-ben. Az algoritmust az RSA Inc. üzleti titokként kezeli, hivatalos publikációja a mai napig nincs, neve pedig kereskedelmi védjegynek minősül. 1994 szeptemberében valaki névtelenül elpostázott egy algoritmust az egyik levelezőlistára, ami villámgyorsan elterjedt. Bár az RSA eleinte tagadta, hogy a listán megjelenő algoritmus és az RC4 ugyanaz lenne, de később ez az állítás megdőlt: több kereskedelmi RC4 implementációval összehasonlítva az „alternatív" algoritmus mindig ugyanazt az eredményt adta, így lassan mindenki elfogadta a két algoritmus egyenlőségét. Ettől függetlenül, ha az alternatív algoritmus, mint RC4 vizsgálatára kerül sor, általában elé szokás tenni az „alleged" (állítólagos) jelzőt. Az „állítólagos RC4" változtatható kulcsméretű (8 bites lépésekben, 8-2048 bit), bájtorientált műveltekkel dolgozik. Szoftverkörnyezetben is rendkívül gyors, körülbelül 10-szer gyorsabb a DES-nél. Az algoritmus igazi csereberélős fajta, íme a rövid receptje:

- Vegyünk egy 0..255 sorozatot! (*initialization*)
- Tagjait a **kulcs függvényében** egyszerű szabályszerűséget követve összeturmixoljuk! Ez a kezdeti keverés (*initial permutation*) minden egyes elemet elmozdít a helyéről, legalább egy alkalommal.
- Amikor jön egy adatbájt, kavarjunk még egyet a sorozaton, válasszunk ki egyet és azzal, mint kulccsal XOR-oljuk össze az adatbájtot (*stream generation*)!

A kulcsfolyam periódusa borzasztó hosszú (~2^{1700}). Eddig nem sikerült érdemi törést találni [24, 69], biztonságossága széles körben elfogadott, maga az algoritmus pedig az alternatív folyamtitkosítók egy szép példája. *Mivel az SSL egyik leggyakrabban használt algoritmusa, így lényegében a világ legelterjedtebb titkosítóalgoritmusa.*

„alleged" RC4 folyamtitkosító – Inicializálás és kezdő keverés

Bemenet: • *keylen*8 bites K kulcs*
Kimenet: • *feltöltött S[] tömb*

```
1.  // Inicializálás
2.  for i:=0 to 255 do S[i]:=i
3.  // Kezdő keverés
4.  j:=0
5.  for i:=0 to 255 do
        a.  j = (j + S[i] + K[i mod keylen]) mod 256
        b.  swap_in_s_array(i,j)
```

„alleged" RC4 folyamtitkosító – Kulcsfolyam generálás

Bemenet: • *az S[] tömb*
Kimenet: • *a kulcsfolyam egy következő bájtja*

```
1.  i:=0
2.  j:=0
3.  while true do
        a.  i = (i + 1) mod 256
        b.  j = (j + S[i]) mod 256
        c.  swap_in_s_array(i,j)
        d.  t = (S[i] + S[j]) mod 256
        e.  output S[t] as keybyte
```

Ez az algoritmus egy kicsit elvi lett, mert a gyakorlatban természetesen nem egy végtelen ciklusban ontjuk a kulcsbájtokat, hanem akkor kérünk egyet, amikor van mit titkosítani vagy megfejteni: ez az „a"-„e" lépések végrehajtását jelenti. A `swap_in_s_array(i,j)` lépés az i. és j. indexű tömbelem tartalmát cseréli fel.

A5/1 – a GSM őre

Sokáig úgy gondolták, hogy az európai GSM[41] kommunikáció rejtjelezésére használt A5 algoritmus elég erős lesz a visszaélések elkerüléséhez. Azóta inkább gyengesége miatt féltik az általa biztosított csatornákat. A legendák szerint a NATO berkeiben sok vita övezte az algoritmus erejét, és amíg a Szovjetunióhoz közeli Németország az erős rejtjelezésre voksolt, a többi ország leszavazta. Végül a francia eredetű, 1987-es A5-re esett a választás. Később az egyik angol telefontársaság elküldte a dokumentációt a Bradford Egyetemnek –az algoritmus pedig az Interneten kötött ki, sok gondot okozva ezzel a szolgáltatóknak.

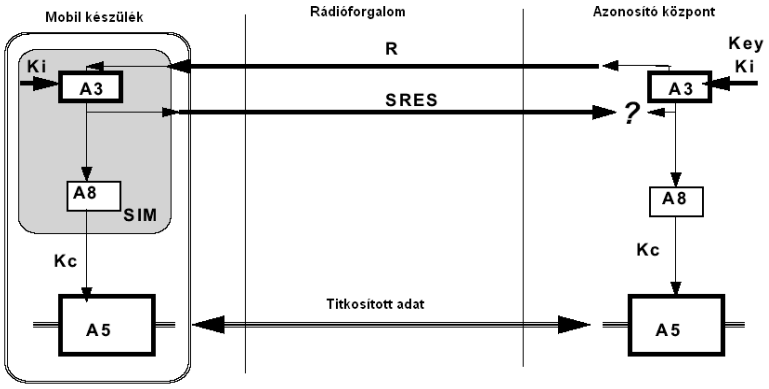
[41] *Global System for Mobile Communication,* korábban *Group Special Mobile*

Titkosítás a GSM kommunikációban

A kommunikáció biztosítására három algoritmust használ a GSM, ebből az A5 a telefon és a *bázisállomás* közötti kapcsolatot védi.

- Azonosító algoritmus: A3
- Viszonykulcs-generátor: A8
- Titkosító algoritmus: A5/1 és A5/2 és A5/3 (Az A5/1 erősebb, az A5/2 gyengébb algoritmus, az A5/3 pedig az új…)

Minden mobiltelefon tartalmazza ezt a három algoritmust. A telefonnal közvetlen kapcsolatot tartó bázisállomás szintén ismeri az A5 algoritmust, és össze van kötve az azonosító központtal (*authentication unit center, AuC*) is, ahol az A3 és A8 algoritmusok működnek. Az üzenetek rejtjelezése a hívás kezdeményezésekor indul: az A3 algoritmus a SIM-kártyában tárolt 128 bites egyedi *Ki* kulcsból és a központtól kapott véletlenszerű *R* kulcsból előállít egy válaszüzenetet (*SRES, SignResponse*), amit a telefon visszaküld. A központ ugyanezeket a lépéseket hajtja végre, így ha az SRES-ek egyeznek, az azonosítás sikeres. Az SRES értéken az A8-cal megint gyúrnak egyet és ebből lesz végül az a 64 bites *Kc* viszonykulcs, amivel a továbbiakban az A5 algoritmus védi a telefon és a bázisállomás közötti kommunikációt, így a beszélgetéseket és adathívásokat is [72]. A kommunikáció során továbbított 228 bites adatokat kereteknek (*frames*) hívjuk. Az algoritmus a *Kc* viszonykulcs mellé a pillanatnyi 22 bites keretsorszámot (*frame number*) is felhasználja. A 22 bit közel négyórás kapcsolat kereteit tudja megszámozni.[42]

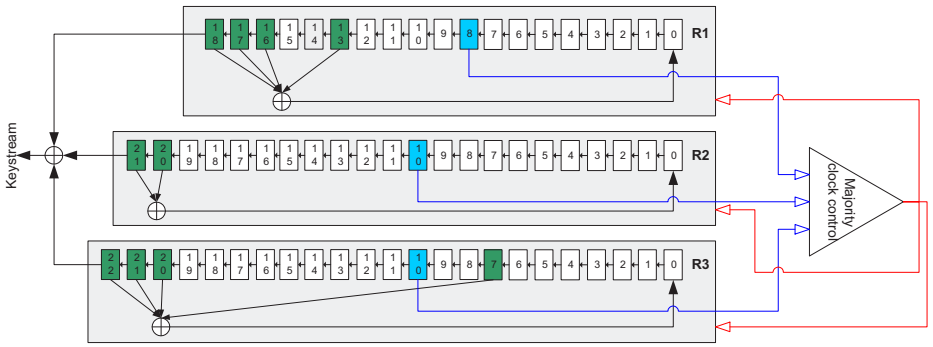


44. ábra A GSM challenge-response autentikációja

[42] A fenti leírás jelentősen leegyszerűsített változata a GSM azonosításnak, de a lényeget megmutatja: hol használjuk az A5 algoritmust?

Az A5/1 szíve – a kulcsfolyamgenerátor

Az A5 belsejében három LFSR található, nevük legyen R1, R2 és R3, hosszuk pedig rendre 19, 22 és 23 bit. A kapcsolati polinomok meglehetősen egyszerűek (*sparse polinoms*), mert a leghosszabb regiszter esetében is csak négy bit kell az új bit kiszámításához. A három regiszter kimenetének XOR-ja adja a végeredményt. A regiszterek léptetése kissé szokatlan az eddigi technikákhoz képest, mert a 8., 10., és 10. (továbbiakban középső) helyen lévő bitek döntik el, hogy adott regiszter forgatásra kerül-e, vagy sem. A biteket egy többségi függvénnyel (*majority function*) összegezzük, és azokat a regisztereket forgatjuk, amelyekben a középső bitek értéke megegyezik e függvény értékével. Vagyis ha a bitek között a „0”-ák vannak többségben, azokat a regisztereket forgatjuk, amelyekben középen 0 áll. Ha több az 1-es, akkor azokat, ahol 1 áll (*majority clock control*). Így minden ciklusban legalább kettő regisztert forgatunk. A kapcsolati polinomok maximális periódust biztosítanak, melyek hossza rendre 2^{19}-1, 2^{22}-1, 2^{23}-1.



45. ábra Az A5/1 kulcsgenerátora

Az algoritmusnak (változatlan néven) több változata is kering, például [24]-ben a fentiektől eltérő implementáció van. Jelen könyvben szereplő leírás főként [URL62]-re és [71]-re támaszkodott. Viszont a [71]-ben lévő ábra... feltehetőleg rossz, ugyanis a regiszterek egy cellával hosszabbak, mint amiről írnak, de a cellák részleges számozása összhangban van a szöveggel. Feltehetően egyszerű figyelmetlenségről van szó, mert a [71] egyik szerzője egyúttal az [URL62]-en lévő - GSM konzorcium által elismert - implementáció egyik szerzője is. Sok ellentmondás található az A5-tel foglalkozó cikkekben, előadásokban is. A szerzők gyakran hivatkoznak [URL62]-re, néha mellékelik azt, de a szövegben, ábrákon más bitek XOR-ja adja az új eredményt, vagy más pozícióban vannak a „középső” bitek, mint a mellékelt C kódban. Szóval [URL62]-re alapozva ismét a többségi elvet követtem az A5 algoritmus leírásakor – mint már oly sokszor a könyvben... Nesze neked majority function!

Az A5/1 kulcselőkészítése

A többi algoritmushoz hasonlóan az A5 is kulcselőkészítést végez indulása előtt. Ez jelen esetben nem újabb kulcsok (körkulcsok) előállítását jelenti, hanem egyszerűen az aktuális 64 bites kulcsot, utána a 22 bites keretsorszámot bitenként betölti mindhárom regiszterbe. E betöltés során a kulcs és a sorszám kisebb helyértékű bitjeivel kezdünk, és haladunk a magasabb

helyértékűek felé. A betöltendő biteket az aktuális regisztertartalmak LSB-jével összeXORoljuk. A regiszterek `000...00` tartalommal indulnak, minden egyes beXORolt bit után megforgatjuk őket. A forgatáskor a többségi függvényen alapuló vezérlést figyelmen kívül hagyjuk. Az így kialakuló állapotot hívjuk inicializált állapotnak. Ezután mindhárom regisztert még 100 alkalommal megforgatjuk, de itt már figyelembe vesszük a forgatási szabályokat. Ez a megoldás alaposan összemossa a kulcsot és a keretszámot, valamint biztosítja a lavinaeffektus meglétét is. A kulcsgenerátor kimenetét 228 bitenként összegyűjtjük, mert ennyi kell a GSM adatblokkhoz (ti. az is ilyen hosszú).

Bár minden lényeges elem és lépés megtalálható e leírásban, aki további (implementációs szintű) részletekre kíváncsi, azoknak ajánlom az [URL62]-n lévő (de a Függelékben is megtalálható) C kódot, amely széles körben a legelfogadottabb. A kód eredetileg 1999-ben visszafejtéssel készült (*reverse engineering*), mert az A5 jelenleg sem publikus.

Az A5/1 biztonsága

Az algoritmus feltörésére számos kísérlet született. Ezek futási ideje a valós idejű működéstől a több hétig terjed. A leggyorsabb algoritmus tárigénye 150-300 GB között van, ami alapvetően elfogadható, főként hogy ez nem memóriaigényt, hanem merevlemezkapacitást jelent. A beszélgetés első 1-2 percének analizálása után a további forgalmat gyakorlatilag real-time módon töri [71]. A támadás részben a regiszterek rövidségén, részben pedig azon alapul, hogy bizonyos feltételek mellett a folyamkódolók kulcsgenerátorának első néhány bitnyi vagy bájtnyi adata nagy valószínűséggel, helyesen megbecsülhető (*sampling resistance*). (Ez nemcsak az A5 esetében, de például az RC4 esetében is igaz.) Mindezek ellenére az A5 elve tiszta és jól érthető, maga az algoritmus igen gyors és hatékony. Az általa generált bitfolyam a korábbi LFSR-ek jó tulajdonságait is birtokolja, egyetlen gyengesége a regiszterek rövidsége, ami lehetővé teszi a viszonylag gyors támadást. Az algoritmus mind a mai napig használatos, csak 2002 júliusában jelentették be az A5/3-at (más néven KASUMI, amely a Mitsubishi cég MISTY algoritmusának utódja), amely minden eddiginél nagyobb biztonságot ígér. (Az A5/3 nem váltja le az A5/1 és A5/2 algoritmusra alapuló, már meglévő implementációkat, hanem az újakban, illetve az új generációs alkalmazásokban jelenik majd meg.)

> Korábban említettem, hogy egy algoritmus nyilvánossága nem kerülhető el. Ennek elvi okai is vannak, de az RC4 és az A5 példája jól mutatja: ha valaki kíváncsi az algoritmusra, visszafejt egy működő implementációt, és ezzel az algoritmus lényegében nyilvánossá válik.

A Függelék jelen fejezethez kapcsolódó alfejezetei

14.8. Szabványok összefoglaló táblázata

14.10. A5/1 – GSM titkosítás – C implementáció

További kiegészítések folyamatosan bővülő helye a http://www.netacademia.net/konyv webcím.

A Vernam-kódolás legnagyobb hátránya, hogy a feladónak (például a kémnek) pontosan annyi kódtáblát kell magával hordania, ahány üzenetet küldeni kíván - mindeközben ügyelnie kell arra, hogy ezek semmilyen körülmények között ne kerüljenek idegen kezekbe (vagyis a fejlövés és a halál közötti mozizásra fordított időt a kódtáblák elégetésével kell töltenie). Az eljárás így meglehetősen körülményes, a lapok az őserdőben szétáznak, a sivatagban megeszik a tevék.

Index.hu

8. DIGITÁLIS ALÁÍRÁSOK ÉS BIZONYÍTVÁNYOK

A nyilvános kulcsú algoritmusok kétkulcsos technikája nemcsak a kulcscsere problémáján enyhít, hanem van egy másik következménye is: megteremti a digitális aláírás feltételeit. Az előzőekben többször szó esett már a digitális aláírásokról, itt az ideje, hogy pontosan kijelöljük elvárásainkat, és definiáljuk a fogalmat. Mint látni fogjuk, a jó digitális aláírás a hagyományos aláírás minden jó tulajdonságát hordozza, sőt ki is egészíti azokat. Az elektronikus aláírás feladata, hogy tanúsítsa az *aláíró személyazonosságát* és azt, hogy *az üzenetet nem változtatták meg* az aláírás időpontja óta.

8.1. AZ ALÁÍRÁS TULAJDONSÁGAI: DIGITÁLIS VS. HAGYOMÁNYOS

Digitális aláírás	*Hagyományos aláírás*
Az aláírás nem helyezhető el más dokumentumokon, észrevétlenül nem vihető át.	
A digitális aláírás az egész dokumentumot vagy annak jellemző kivonatát kódolja, így az aláírás végső tartalma függ az aláírt dokumentumtól is. A dokumentum valamilyen formában az aláírás része. Azonban a hagyományos aláírástól eltérően az elektronikus aláírás csak logikailag kapcsolódik az aláírt dokumentumhoz és nem fizikailag. A digitális aláírás a tartalmat hitelesíti és nem a dokumentum hordozóját.	Az aláíró függetlenül az aláírandó dokumentumtól, mindig ugyanúgy ír alá. Ezért egy begyakorlott hamis, vagy megfelelő eszközökkel lemásolt aláírás felhasználható más dokumentumon is. A hagyományos aláírás az elektronikus aláírástól eltérően fizikailag kapcsolódik az aláírt dokumentumhoz és nem logikailag. Igazából a hordozó papírt hitelesíti és csak áttételesen annak tartalmát[43].
Az aláírt dokumentum tartalma nem változtatható meg észrevétlenül[44].	
Az előző pont szerint a dokumentum valamilyen formában az aláírás része, ezért ha a dokumentum az aláírást követően megváltozott, azt az aláírás ellenőrzése kimutatja.	A dokumentum tartalma megváltoztatható az aláírás után is, feltéve, ha az aláíró nem kap másolatot róla. Ez azonban csak jogviták forrása, mert ettől maga a dokumentum fizikailag megváltoztatható.

[43] Ha a tartalom nem változtatható meg a hordozó roncsolása nélkül.
[44] Ez a két tulajdonság csak úgy biztosítható, ha az aláíráshoz felhasználjuk az egész dokumentumot vagy annak egy leképzett részét. A digitális aláírás ekkor a feladó azonosítását és az adatintegritást egyszerre szolgálja.

Az aláírás bizonyítja, hogy az aláírás annak tulajdonosától származik és nem valaki mástól.

Csak akkor hamisítható, ha a titkos, aláírásra használt kulcsunkat valaki megszerzi. Egyébként senki sem tud a mi nevünkben aláírni, mert egy sikeresen ellenőrizhető aláírást csak a mi kulcsunk tud létrehozni. Következésképp **mi sem tagadhatjuk le** aláírásunkat és **nem is hamisítható az aláírásunk**.	Némi gyakorlással hamisítható és a tulajdonosa nélkül is „felhasználható" az aláírás. A jó hamis aláírást szinte semmi sem különbözteti meg a valóditól.

Következésképpen az aláíró nem tagadhatja le az aláírását.

Amennyiben egy aláírás ellenőrzése során a mi nyilvános kulcsunkkal sikeresen elolvasható egy aláírás, akkor azt a mi titkos kulcsunkkal írták alá, amiből az következik, hogy az aláírás tőlünk származik. Feltéve persze, hogy az aláíráshoz szükséges titkos kulcsunkat nem lopták el[45].	Ha elég bátrak vagyunk, nyugodtan letagadhatjuk aláírásunkat. Bár ezzel vigyázzunk, mert a jogi gyakorlatban az aláírás hitelességének vélelmével szemben a hamisítást bizonyítani kell …

További eltérések

Elektronikus aláírással bármilyen elektronikus (digitális) formában megjelenő dokumentumot (levél, kép, zene, program- vagy adatfájl stb.) alá lehet írni.	Hagyományos aláírással csak papíralapú dokumentumot lehet aláírni.
A digitálisan aláírt dokumentum másolata az eredetivel egyenértékű, hiszen egy digitálisan tárolt adat digitális másolata bitről bitre megegyezik az eredetivel.	A hagyományos aláírt dokumentum másolata az eredetitől jelentősen eltérhet, de egy kiváló minőségű fénymásolatot sem ismerünk el eredetinek, csak ha a másolat hitelesítve van, vagy az aláíró a másolatot is aláírja.

> Bár eddigi kínosan ügyeltem a kódolás-dekódolás és a titkosítás-megfejtés szópárok helyes használatára, jelen fejezetben egy kis általánosítással találkozhat az Olvasó. Ennek oka, hogy a digitális aláírásokban a titkos és a nyilvános kulcsok szerepe némileg felcserélődik: egy dokumentum aláírásakor a titkos kulcsot alkalmazza az aláíró, az ellenőrzés pedig a nyilvános kulccsal történik. Eddig ezeket a műveleteket rendre megfejtésnek és titkosításnak hívtuk, de ennek most semmi értelme nem lenne, mert a kriptoszöveg megfejtéssel, az értelmezhető nyíltszöveg pedig titkosítással jönne létre. Ezért a kulcspár egyik felének alkalmazását egyszerűen kódolásnak, az inverz műveletet (a kulcspár másik felének alkalmazását) pedig dekódolásnak hívom.

[45] Bár ezt nekünk kell bizonyítani, hasonlóan a hagyományos aláírás valódiságának kétségbe vonásához.

8.2. AZ ALÁÍRÁS LOGIKÁJA

Azok a feladatok, melyeket egy digitális aláírásrendszerben minimálisan meg kell valósítani a következők:

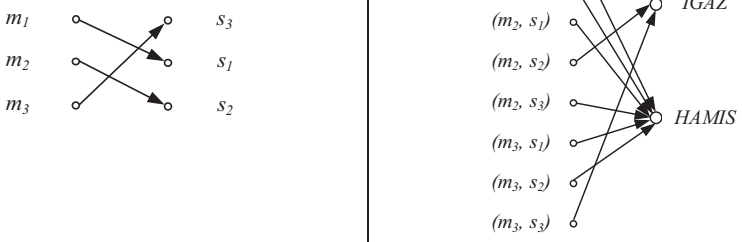
Feladat	Funkció
Aláírás	Előállítja az adott *m* üzenethez tartozó, *Alice* által készített digitális aláírást. Általában az aláírás tartalmaz az aláírás körülményeire vonatkozó adatokat is, például időpont, dátum, hely stb.
Ellenőrzés	Egy adott *m* üzenet és egy *s* aláírás kapcsolatát vizsgálja, *Alicera* nézve. Két eredménye lehet: vagy igaz – akkor, és csak akkor ha az üzenet és az aláírás összetartozik – vagy hamis, ha nem tartoznak össze.

Ezután Alice aláírási eljárása egy *m* üzeneten a következő:

1. Kiszámolja azt az *s* értéket, amit a továbbiakban *aláírásnak* hívunk. Tartalma, készítésének módja egyelőre nem érdekes. A lényeg, hogy Alice és az üzenet kapcsolatára jellemző értéket ad.
2. Továbbítja Bobnak az *(m,s)* párost, melyek csak logikailag tartoznak össze, nem kötelező a fizikai összeragasztásuk. Az *s* aláírás azonban önmagában nem sokat ér. Tartalmazhat ugyan az aláíró személyére, az aláírás körülményeire vonatkozó adatokat, de mint aláírásnak csak *m*-mel együtt van értelmezhető információtartalma, és az ellenőrzés is csak *m*-mel együtt végezhető el.

Egy Alice által küldött *m* üzenetet és az általa generált *s* aláírást Bob a következőképpen ellenőrizheti:

1. Az Alicehoz tartozó ellenőrzőfüggvénnyel megvizsgálja *m* és *s* kapcsolatát.
2. Elfogadja az aláírást, ha az ellenőrzés sikeres volt, egyébként figyelmen kívül hagyja azt, illetve az ellenőrzés sikertelensége esetén egyéb lépéseket tehet.



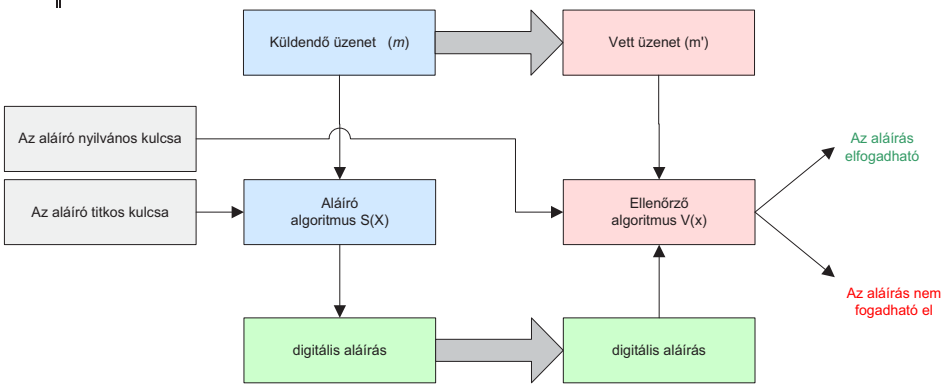
Aláírás művelete | *Ellenőrzés művelete*

46. ábra Digitális aláírás alapvető függvényei

Eddig nem esett szó arról, hogy az aláírás miként biztosítja a hamisíthatatlanságot. Egyik megoldás az lehetne, hogy az aláírófüggvényt titokban kell tartani és egy olyan ellenőrzőfüggvényt kell mellé kitalálni, amiből nem lehet reprodukálni az aláírófüggvényt. Ekkor minden résztvevőnek saját titkos aláíróalgoritmusa van. Ez elég kényelmetlen megoldásnak tűnik, és a gyakorlatban sem ezt használják. Másik lehetőség, hogy nem egyedi, hanem nyilvános és egységes függvényeket használunk. Az aláírásba pedig bevonjuk az aláíró által birtokolt titkos kulcsot, ez garantálja majd az egyediséget. Az ellenőrzéshez szintén szükséges ez a kulcs, ami veszélyezteti az aláíráshoz használt kulcs biztonságát, bizalmasságát. Ha azonban az aláíró- és ellenőrzőalgoritmusokat sikerül nyilvános kulcsú kriptográfiai eszközökkel megvalósítani, akkor aláíráshoz a titkos kulcsot, ellenőrzéshez pedig a nyilvános kulcsot lehetne használni. Ez a megoldás azért szimpatikusabb, mert nem egyedi algoritmusokat kell titokban tartani, csak az aláírókulcsokat, ez a feladat viszont nem újdonság.

A digitális aláírások ezen formája csak nyíltkulcsos kriptorendszerekben valósítható meg, mert csak ezekben a rendszerekben van az aláírónak saját kulcsa. Ez garantálja az aláíró számára, hogy más nem tud az ő nevében aláírni, valamint ez biztosítja a címzett vagy egy harmadik fél számára a letagadhatatlanságot is: ha más nem ismeri a titkos kulcsot, más nem írhat vele alá.

> Az igazsághoz hozzátartozik, hogy a digitális aláírások a fenti gondolatmenet alapján csak indirekt módon bizonyítják az aláíró kilétét, ugyanis a bizonyítási eljárás abból indul ki, hogy a titkos kulcs csak az aláíró birtokában van és nem tulajdonították el tőle, nem vesztette el. Sajnos ez nem minden esetben igaz és további bonyodalmakat vonhat maga után. Ha a kulcsot elveszti birtokosa, akkor az ő nevében aláírás generálható jelenléte vagy jóváhagyó akarata nélkül is. A legbiztonságosabb megoldás, ha az aláíró és az ellenőrző algoritmus valamilyen módon felhasználja az aláíró valamelyik biometrikus azonosítóját (ujjlenyomat, íriszmintázat, stb. pontosabban annak digitális megfelelőjét), amelyet az aláírás pillanatában szolgáltat egy arra alkalmas eszközzel, vagy előre eltárolt minta átadásával. (Bár az utóbbi megoldás visszavezet az eredeti problémára.)



47. ábra Egy üzenet aláírása és ellenőrzése

A titkos kulcs jelenti az iméntiekre az *egyetlen* biztosítékot, így a titkos kulcs elvesztése nemcsak azt eredményezi, hogy a címzettnek szánt üzeneteket illetéktelenek is olvashatják, hanem azt is, hogy a kulcs (volt) tulajdonosának nevében bárki alá tud írni. Ez pedig nem más, mint az aláírás hamisításának elektronikus megfelelője. Ez is az egyik oka, hogy a nyilvános kulcsú rendszereknek lehetőséget kell biztosítani a nyilvános kulcsok érvénytelenné nyilvánítására, visszavonására. Az aláírás dátumának és időpontjának mindenképpen szerepelnie kell az aláírásban, mert ha az aláírás dátuma a kulcspár-visszavonás dátumát követő, az aláírás biztosan hamis, tehát érvénytelen. Ha a támadónak tudomása van a kulcs visszavonásáról, saját órájának vagy naptárának az átállításával – a dokumentum tartalmától függően – elhitetheti, hogy az aláírás még a visszavonás előtt történt. Megoldást az *időbélyeg szolgáltató (TSA)* nyújthat, aki soha nem ad ki igazolást múltbéli vagy jövőbeli időpontról.

> Ha a TSA maga is aláírja a dokumentumot, azt bizonyíthatjuk, hogy a dokumentum az adott időpont előtt már létezett.
>
> Ha a TSA csak egy kis igazolást, egyfajta mikrodokumentot ad ki egy időpontról, amelyet az aláírás előtt csatolunk az aláírandó dokumentumhoz, azt bizonyíthatjuk, hogy az aláírás egy adott időpont után keletkezett.

8.3. ALÁÍRÁS AZ RSA ALGORITMUSSAL

8.3.1. Üzenet kódolása RSA-val

Alice a titkos kulcsával kiszámolja az

$$M=m^d \bmod N$$

üzenetet és elküldi Bobnak (esetleg saját nyilvános kulcsával együtt). Ezután a Bob Alice nyilvános kulcsával megfejt:

$$m'=M^e \bmod N$$

Ha eredményül az üzenetet kapja meg (vagyis *m'* értelmezhető, olvasható), biztos lehet benne, hogy az üzenet Alicetól jött – vagy legalábbis olyan valakitől, aki ismeri Alice titkos kulcsát. Itt titkosítás nem történik, hiszen a nyilvános kulcs ismeretében bárki megfejtheti az üzenetet. Ha Alice elküldés előtt Bob nyilvános kulcsával is kódolja az üzenetet, titkosítás is történik. Ebben a formában az RSA algoritmust nem szabad aláírásra használni, mert a támadó egy rejtjeles blokk elküldésével az aláírás folyamatát megfejtésre változtathatja. (Lásd: „*4.6.1. Néhány RSA elleni egyszerűbb támadás*” alfejezetet)

8.3.2. Kivonat kódolása RSA-val

Gyakran okozhat gondot az üzenet teljes egészének titkosítása, mert a nyíltkulcsos algoritmusok általában lassúak. Másrészt az így aláírt dokumentum olvashatatlan lesz, mindenképpen meg kell fejteni az elolvasás előtt, akkor is, ha az aláírás ténye vagy helyessége az adott pillanatban közömbös. Emiatt nem az egész üzenetet szokás titkosítani, hanem annak csak egy

jellemző, egyedi kivonatát (*message digest*, *MD*, lásd következő fejezetet). Ebben az esetben az aláírás folyamata így alakul:

1. Kiszámol: $s = (MD(m))^d \bmod n$
2. Elküld: $\{ m, s \}$

Vagyis kiszámolunk egy ellenőrző összeget és azt kódoljuk az átvitel előtt, nem pedig a teljes üzenetet. *Az elektronikus dokumentum ellenőrzőösszegének titkos kulccsal kódolt formája az aláírás*, amit a dokumentumhoz csatolva küldünk el. Ezután az ellenőrzés menete:

$V1 = s^e \bmod n$	*ha* $V1 = V2$ → OK, mehet a további feldolgozás, ellenőrzés.
$V2 = MD(m)$	*ha* $V1 \neq V2$ → HIBA, az üzenet (vagy pecsétje) megváltozott.

PGP-ben így néz ki egy aláírt dokumentum (sorszámok nélkül):

```
 1. -----BEGIN PGP SIGNED MESSAGE-----
 2. Hash: SHA1
 3. Sokáig úgy tartották, hogy a gép kódolása feltörhetetlen, azonban
 4. a szövetséges csapatok titkosszolgálatai a lengyelek kutatásai
 5. alapján már a háború korai szakaszában megfejtették a kódolást,
 6. és végül folyékonyan fejtették a rejtjelezett üzeneteket.
 7. -----BEGIN PGP SIGNATURE-----
 8. Version: PGPfreeware 7.0.3 for non-commercial use <http://www.pgp.com>
 9. iQA/AwUBO3Lvq4FLPmKo6RZMEQJRywCgzAJtJwWkeiK22x3lq73xVhtCi6IAoJgS
10. NFH25JsOA8LoxXaXcGjEZYgQ
11. =wmPM
12. -----END PGP SIGNATURE-----
```

Az (1)-(2) sor jelzi, hogy aláírt dokumentum következik, és egyben megadja az aláírás során használt MD algoritmust is. A (3)-(6) sorok magát az üzenetet tartalmazzák. A (7)-(12) sorokban az aláírás van, némi program- és verzióinformációval kiegészítve. A (9)-(11) sorok ákombákomjai logikailag a következő adatokból állnak össze:

$$BASE64(C_d(\text{az üzenet SHA-1 pecsétje} + \text{időpont} + \text{email cím} + \text{név}))$$

> A BASE64 egy olyan kódolási séma, ami lehetővé teszi, hogy bármilyen bináris adatot ASCII7 karakterek sorozatává konvertáljunk. Bővebben lásd a Kislexikonban.

8.4. AZ ALÁÍRÁS TARTALMA ÉS HITELESSÉGE

8.4.1. Mit tartalmaz az aláírás?

Most foglaljuk össze, mi kell egy aláírásba, és hogyan készül el a gyakorlatban.

- Szükség van egy *MD* algoritmusra, ez általában *SHA-1* vagy *MD5*. Miután az algoritmus átrágja magát az aláírandó bitfolyamon, eredményül egy fix hosszúságú bitsorozatot ad. (Ennek hossza *SHA-1* esetén *160 bit, MD5* esetén *128 bit*). Ez a viszonylag rövid bitsorozat képviseli a továbbiakban a dokumentum tartalmát.

- Az *MD* bitsorozathoz hozzáfűzzük a következő kiegészítő információkat:
  - az aláíró nevét vagy más azonosítóját,
  - az aláírás idejét,
  - a használt *MD* algoritmus nevét vagy azonosítóját,
  - az aláírás helyét,
  - egyéb fontosnak tartott adatot, jellemzőt.
- Ezután ezt az egész csomagot az aláíró kódolja a titkos kulcsával, így előáll egy olyan adat, mely csak az aláíró titkos kulcsával készíthető el úgy, hogy a nyilvános kulcsával elolvasható legyen.
- A küldendő dokumentumhoz hozzácsatolja az iménti kódolás eredményét, mint aláírást és a kettőt együtt küldi el. (Arra az esetre, ha a címzett nem ismerné a feladó nyilvános kulcsát, csatolható azt is.)
- A fogadó oldal (vagy bárki, aki kíváncsi az aláírás érvényességére) megfejti csatolt aláírásadatot, kiszámolja a kapott dokumentum *MD* bitsorozatát, amit utána összehasonlít az aláírás megfelelő részével. Ehhez az aláírásban szereplő algoritmust használja. Ha a kettő egyezik, minden rendben van: az aláírás kizárólag a küldő kulcsaival készülhetett és sem a dokumentum, sem az aláírás nem változott meg.

8.4.2. A hitelesség

Már korábban beláttuk, hogy a nyilvános kulcsú technikákon alapuló titkosításoknak és aláírásoknak van egy nagyon sebezhető pontja, nevezetesen: bizonyítani kell, hogy a nyilvános kulcs valóban azé, aki azt állítja, hogy az övé. Amíg ez nem biztosított, addig a „*középen lévő ember*” (*man-in-the-middle, interleaving attack*) típusú támadás sikeres lehet: A támadó beékelődik a feladó és a címzett közé, minden áthaladó kulcsot kicserél, így elhiteti a feladóval, hogy a címzettel kommunikál, a címzett számára pedig feladóként látszik. Ehhez csak két olyan kulcspárra van szükséges, melyekről az áldozatok azt hiszik, valódiak:

- Alice elküldi az aláírt levelet.
- Trudy elkapja, leválasztja Alice aláírását. Kedve szerint megváltoztatja a dokumentumot, majd egy *Alice nevére szóló kulccsal* aláírja azt és továbbküldi Bobnak.
- Ha Bob aláírva válaszol, Trudy hasonlóan járhat el: leszedi Bob aláírását a válaszról és egy *Bob nevére szóló kulccsal* aláírva küldi el azt Alicenak.

Valahogy tehát meg kell győzni mindenkit a kulcs és a felhasználók összetartozásáról. Erre a következő lehetőségek állnak rendelkezésre:

1. A kulcs tulajdonosa maga adja oda a nyilvános kulcsát. Ez az a megoldás, ami általában nem működik, ráadásul az aszimmetrikus technika egyik nagy előnye (a publikálás lehetősége és a bizalmasság mellőzhetősége) is kihasználatlan marad.
2. A kulcs tulajdonosa aláírja a saját nyilvános kulcsát (*self-signed key*). Ez ugyan még nem meggyőző a személyazonosságát illetően, de legalább már az bizonyítható, hogy a titkos kulcs valóban a birtokában van. De mi van, ha egy rosszhiszemű fél más valaki-

nek a nevében generál egy kulcspárt, alá is írja a titkos kulccsal, majd az illető nevében közzé teszi? Akkor sajnos majdnem ugyanott vagyunk, ahonnan elindultunk.

3. A kulcs tulajdonosa megkér valakit az ismerősei közül, hogy írja alá a kulcsát. Ez jelzi, hogy már legalább egy ember van, aki tanúsítja a kulcstulajdonos személyazonosságát. Ha valaki ismeri a felkért tanút, és megbízik benne, valószínűleg az általa aláírt kulcsot is elfogadja hitelesnek. És így tovább. Ezt a „pilótajátékot" nevezzük bizalmi hálónak (*web of trust, distributed trust model*).
4. Az előző önszerveződő „keresztül-kasul" technika nem meggyőző mindenki számára. A bizalom egyébként sem tranzitív, tehát ha én megbízok valakiben, és Te megbízol bennem, még nem jelenti azt, hogy Te is megbízol abban, akiben én. Másrész hiányzik belőle a felelősségvállalás, és az olyan – mindenki számára elfogadható – szabályozás, ami lehetővé tenné a valódi jogkövetkezmények érvényesítését. A hagyományos kézi aláírás rendelkezik ezzel az erővel (részben a hagyomány, részben a törvényi védelem miatt). A szükséges jogi szabályozás más megoldást követel: olyan szervezeteket állít fel, melyek
   - feladata az aláíró személy azonosítása,
   - feladata az aláírások hitelesítése (kulcsok azonosítás utáni aláírása),
   - aláírását mindenki elismeri,
   - tevékenysége ellenőrizhető,
   - mulasztása vagy gondatlansága szankcionálható.

   A kulcs tulajdonosa a kulcsgenerálás után elballag egy ilyen szervezethez, ott hitelesítik a nyilvános kulcsát: a megfelelő adatok ellenőrzése után kap egy – a szervezet saját titkos kulcsával aláírt – igazolást arról, hogy neki mi a nyilvános kulcsa. Ez a szervezet a *Certification Authory,* vagyis a *hitelesítésszolgáltató*, az általa kiállított igazolás a *certficate*, a *hitelességi bizonyítvány*. A nyilvános kulcsokat is tartalmazó bizonyítvány birtokában így módosulhat az aláírt küldemény továbbítása:

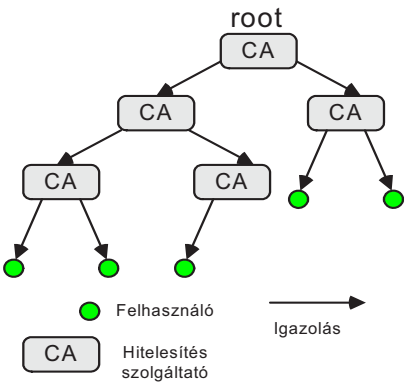
   1. Kiszámol: $s = MD(m)^d \bmod n$
   2. Elküld: $\{ m, s, \text{hitelesített } (e,n) \}$

Bizalmi elvek, bizalmi modellek, hitelesítési kapcsolatok

Az imént pár mondatban megismertük a főbb hitelesítési megoldások alapjait, de lássuk most őket részletesebben. A hitelesítettek és a hitelesítők kapcsolata alapján a következő szerveződések képzelhetők el, melyeket bizalmi elveknek, bizalmi modelleknek nevezünk (*trust models*) [23,25].

Szigorúan hierarchikus viszony

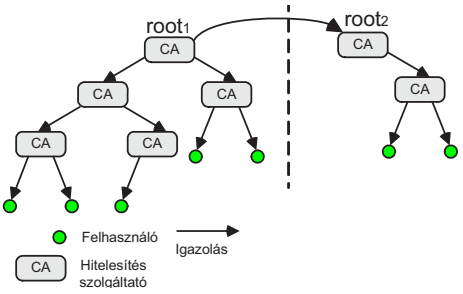
E megoldásban a szereplők között egyértelműen meghatározható alá- vagy fölérendeltség van. A hierarchia csúcsán a *root-CA* áll, akit mindenki hitelesnek ismer el, jóllehet ő maga nincs hitelesítve. Az általa hitelesített hitelesítők (*subordinate CA*) további *CA*-kat illetve kul-



csokat hitelesíthetnek úgy, hogy a hitelesített fél a hierarchia alsóbb szintjén van (kereszt- és egyenrangú hitelesítések nem megengedettek). Az így kialakuló kapcsolatokban egy hitelességi bizonyítvány nyomon követése, a bizonyítványlánc feltárása alulról felfelé (*bottom-to-top*) történhet. A *root-CA* nyilvános kulcsának mindenki számára elérhetőnek kell lennie, hiszen minden bizonyítvány közvetve az ő hitelességén múlik. A modell meglehetősen merev, egy-egy hitelesítő vagy a *root-CA* a kiesése (kompromittálódása) gyakorlatilag a teljes kapcsolatrendszert felborítja. Ezért gyakran előfordul, hogy amint a *root-CA* kiadta a szükséges bizonyítványokat, off-line lesz, gyakorlatilag kikapcsolják, elzárják (titkos kulcsa akár meg is semmisíthető).

Egyenrangú kapcsolatok - kereszthitelesítések

A már megismert „bizalmi háló" modellhez hasonló. Bármelyik *CA* hitelesítheti bármelyik másik *CA*-t kivéve, ha valamilyen megszorítás vagy szabályrendszer ezt nem tiltja. Ez a kereszthitelesítés (*bridge-CA*). A modell a többszörös kapcsolatok révén kevésbé érzékeny a *CA*-k kiesésére, jól alkalmazható gyakran változó környezetben is. Ez főleg akkor igaz, ha a hitelesítési kapcsolatok mindkét irányban léteznek, tehát a hitelesítők kölcsönösen hitelesítik egymást. A bizonyítványok „családfáját" azonban jóval nehezebb feltérképezni a hierarchikus viszonyhoz képest, ott a bizonyítványhoz kikerestük a kibocsátóját, majd a kibocsátó hitelesítőjét és így tovább. Az út végén eljutottunk a *root-CA* „személyéhez", és valószínűleg egyetlen lehetséges utat jártunk be. Most nehezebb dolgunk van, hiszen a „pókhálóban" előfordulhatnak olyan kereszt- vagy kölcsönös hitelesítések, melyek követése szerencsétlen esetben „végtelen ciklust" eredményezhet. (Mivel a gyakorlatban egy kulcsot csak egy CA hitelesít, ilyen ritkán fordul elő.)

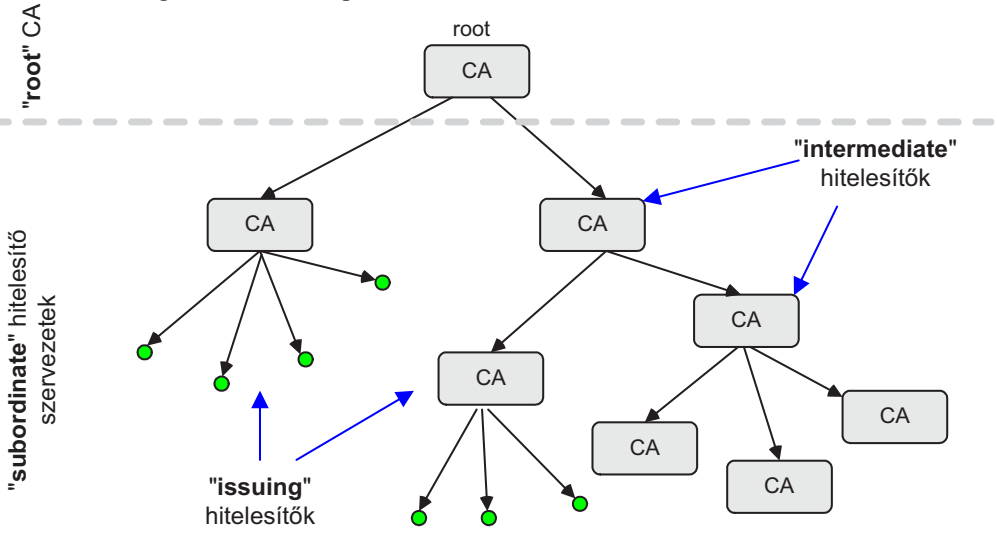


Hibrid megoldások és ki kicsoda?

Ez a kapcsolati modell az iménti két megoldás ötvözése. Tehát:

- több *root-CA* van, de minden nem *root-CA* csak abban a csoportban végezhet hitelesítéseket, melybe maga is tartozik (a fenti ábrán a 3-as *CA* hitelesítheti 1-est és 4-est, de nem hitelesítheti 2-est);
- egy nem *root-CA* csak a saját csoportjában lévő, hierarchiában felette álló hitelesítőtől kaphat bizonyítványt (a fenti ábrán a 3-as CA az 1-estől kaphat bizonyítványt, de a 2-estől nem), de a *root-CA*-k egymás között szabadon hitelesíthetnek az egyenrangú modell szerint.

A hitelesítő szolgáltatók kusza kapcsolatában a következő elnevezésekkel élhetünk:



- *root CA* – szerepét már tisztáztuk, szegényre több szót nem érdemes szánni, már úgyis befalazták...
- *subordinate CA* – olyan *CA*, aki nem *root-CA*. Attól függően, hogy kinek állít ki bizonyítványt, lehet:
  - *issuing CA*: csak végfelhasználókat hitelesít.
  - *intermediate CA*: nemcsak végfelhasználókat hitelesít, hanem más *issuing* és *intermediate* szolgáltatókat is. Az általa kibocsátott bizonyítványok további bizonyítványok kibocsátására jogosíthatnak.

8.5. A HITELESSÉGI BIZONYÍTVÁNY

Többféle formátumú bizonyítvány is kialakult az utóbbi időkben, alapvetően mindegyik a következő adatokat tartalmazza:

1. A kulcs tulajdonosának adatai
   - név vagy azonosító (*subject*),
   - a nyilvános kulcs (*public key*),
   - a kulcs MD-je (*key identifier, fingerprint*)
2. A bizonyítvány adatait
   - kibocsátó neve vagy azonosítója (*issuer*),
   - a bizonyítvány sorszáma, egyéb azonosítója, verziószáma (*serial number, version*),
   - érvényességi ideje (*validity*),
   - az aláíráshoz használt MD algoritmus azonosítója (*signature algorithm*),
   - az aláíráshoz használt algoritmus neve (*signature algorithm*)

3. A bizonyítvány érvényességi köre (*certified usage*)
   - mire használható a tanúsítvány és a nyilvános kulcs? (például csak aláírásra, titkosításra, szerver- vagy kliensszámítógép azonosságára, szoftver eredetének tanúsítására, SSL kapcsolat kiépítésére, stb.)
   - ki használja a tanúsítványt:
     - szervezet vagy szervezeti egység (cégszerű aláírás lehetősége!)
     - szerepköri tanúsítvány, amely az aláíró szervezeti tagságát és betöltött funkcióját tanúsítja.
     - személyes tanúsítvány
     - eszköztanúsítvány

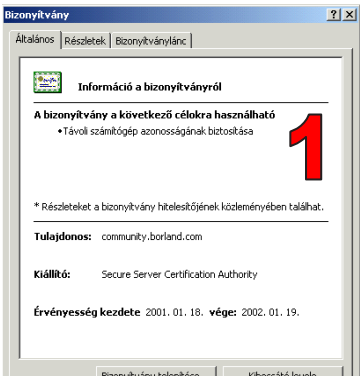
Mindezek bemutatására lássunk egy X.509-es bizonyítványt közelebbről. (Az X.509-es szabvány az ITU-T X.509 nemzetközi szabványon alapuló formátum. [URL54])

```
Certificate:
    Data:
        Version: v3 (0x2)
        Serial Number: 8 (0x8)
        Signature Algorithm: PKCS #1 MD5 With RSA Encryption
        Issuer: CN=Root CA, OU=CIS, O=Structured Arts Computing Corporation, C=US
        Validity:
            Not Before: Fri Dec 5 18:39:01 1997
            Not After: Sat Dec 5 18:39:01 1998
        Subject: CN=Test User, OU=Test Org Unit, O=Test Organization, C=US
        Subject Public Key Info:
            Algorithm: PKCS #1 RSA Encryption
            Public Key:
              Modulus:
                00:c2:29:01:63:a1:fe:32:ae:0c:51:8d:e9:07:6b:02:fe:ec:
                6d:0e:cc:95:4b:dc:0a:4b:0b:31:a3:1a:e1:68:1f:d8:0b:b7:
                91:fb:f7:fd:bd:32:ba:76:01:45:e1:7f:8b:66:cd:7e:79:67:
                8d:48:30:2a:09:48:4c:9b:c7:98:d2:b3:1c:e9:54:2c:3c:0a:
                10:b0:76:ae:06:69:58:ac:e8:d8:4f:37:83:c3:f1:34:02:6d:
                9f:38:60:6f:5e:54:4f:71:c7:92:28:fb:0a:b3:44:f3:1a:a3:
                fe:99:f4:3f:d3:12:e2:f8:3b:03:65:33:88:9b:67:c7:de:88:
                23:90:2b
              Public Exponent: 65537 (0x10001)
        Extensions:
            Identifier: Certificate Type
              Critical: no
              Certified Usage:
                SSL Client
            Identifier: Authority Key Identifier
              Critical: no
              Key Identifier:
                a7:84:21:f4:50:0e:40:0f:53:f2:c5:d0:53:d5:47:56:b7:c5:
                5e:96
    Signature:
        Algorithm: PKCS #1 MD5 With RSA Encryption
        Signature:
            2d:76:3f:49:5b:53:3a:c5:02:06:a3:67:6d:d9:03:50:57:7f:de:a7:a9:
            cd:69:02:97:6f:66:6a:7f:95:ea:89:75:7a:fc:b0:26:81:fc:33:bb:60:
            e8:f7:73:77:37:f8:8a:04:3b:fc:c1:3e:42:40:3d:58:16:17:7e:47:35:
            1c:73:5a:ab:72:33:c3:f5:2b:c6:eb:b5:39:52:82:c6:3e:e1:38:c6:39:
            8b:ee:e3:9f:b3:b9:29:42:0d:11:a5:79:af:6d:3a:f8:a6:ba:d0:9c:55:
            48:0d:75:91:05:0b:47:67:98:32:f3:2d:2e:49:ed:22:ab:28:e8:d6:96:
            a1:9b
```

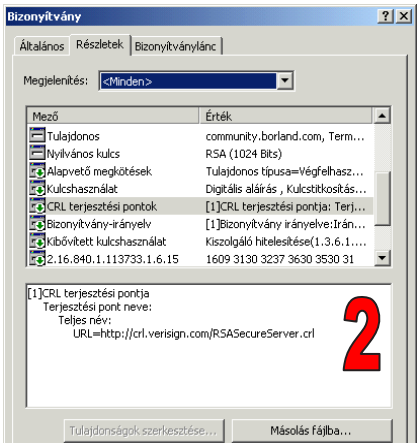
48. ábra Egy X.509 bizonyítvány tartalma

A fenti ASCII dump csak a logikai tartalom bemutatására jó, mert a bizonyítványok gyakorlati megjelenése nem ilyen szöveges, hanem valamilyen bináris (többnyire DER), esetleg Base64(DER) kódolású ASN.1. Mindenesetre megtudhatjuk belőle, hogy olyan 1024 bites RSA kulcsot igazol, ahol e=65537 és amelynek tulajdonosa TEST USER, akinek ezt a SA Computing Co. Root-CA-ja igazolja és SSL kapcsolatok kliensoldalának igazolásához használhatja, egy éven keresztül. A hitelesítő aláírás során az RSA és MD5 algoritmusokat használták, a PKCS#1 szabvány szerint.

A hitelességi bizonyítvány ellenőrzése a bizonyítványon lévő aláírás ellenőrzését jelenti. Ehhez szükség van a hitelesítő bizonyítványára, amelynek ellenőrzéséhez az őt hitelesítő bizonyítványa kell és így tovább egészen a root-CA saját maga által aláírt bizonyítványáig. Hogyan szerezhetők be ezek a tanúsítványok? A kibocsátók bizonyítványlánca általában benne van a tanúsítványban. Ha nincs, rosszabb esetben internetes vagy intranetes csatlakozáson keresztül tölthetők le a hiányzó bizonyítványok vagy jobb esetben (mint a Windows esetében is) az operációs rendszerrel együtt kerülnek a gépünkre. A gyakorlat szempontjából fontos megjegyezni, hogy attól, mert egy bizonyítvány hitelessége nem ellenőrizhető, még maga a bizonyítvány és a benne lévő kulcs funkcionálisan használható!

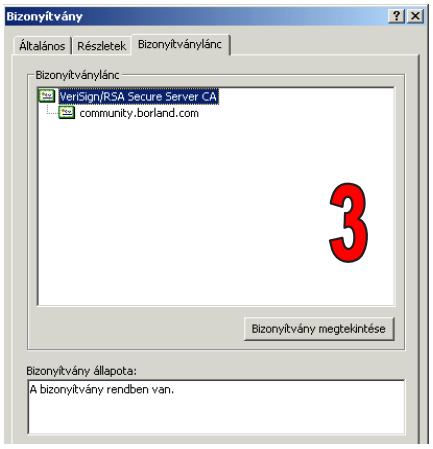


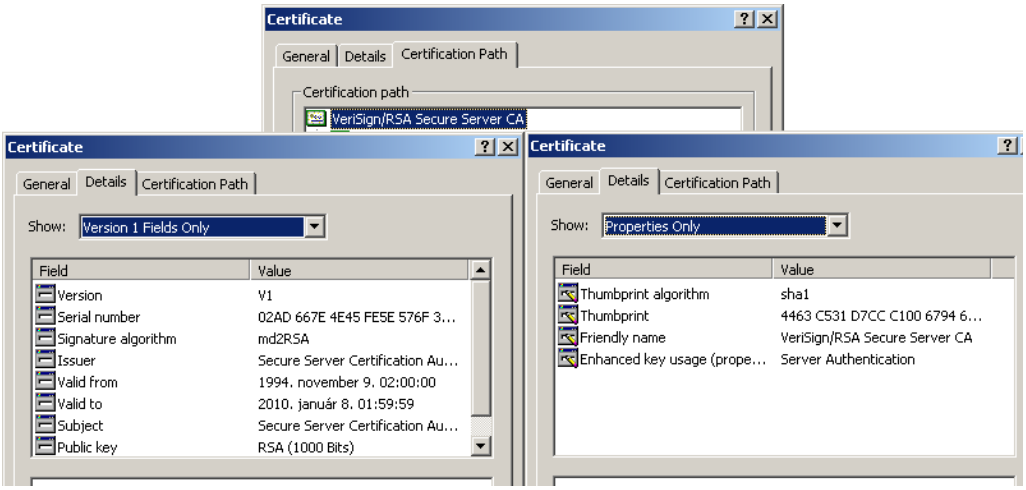
És így néznek ki egy bizonyítvány adatai a Windows megjelenítése szerint. A képeken látható bizonyítvány a Borland egyik szerverének bizonyítványa. Feladata, hogy a távoli szervert biztonságosan azonosítsa például a böngésző SSL-kapcsolatán keresztül. Az első lapon ez látható, valamint a bizonyítvány kiállítójának, tulajdonosának (*Secure Server Certification Authority* és *community.boland.com*) a neve, és itt ellenőrizhető az érvényesség időtartama is.



A második lapon részletes információkat találhatunk a nyilvános kulcsról (amely most egy 1024 bites RSA kulcs), az aláíráshoz használt hashalgoritmusról (amely *SHA-1*, bár épp nem látszik) és más kriptográfiai szempontból fontos jellemzőről. Itt találhatjuk meg a *CRL*-lista címét is, amelynek ellenőrzése adott esetben a böngésző feladata. (A CRL lista a visszavont tanúsítványok adatait tartalmazza.)

A harmadik lapon a bizonyítványlánc tárul elénk. Ez most elég egyszerű, mert a bizonyítványt egy olyan szervezet hitelesítette (a *VeriSign*), amely *root-CA*-ként működik. Ezt a feltevést a következő oldalon lévő ábra bal oldala igazolja, ahol a *subject* és az *issuer* ugyanaz. (A következő lapon láthatjuk majd azt is, hogy a *Secure Server Certification Authority* és a *VeriSign/RSA Secure Server CA* ugyanaz, csak az utóbbi az előbbi „barátságos neve”.)





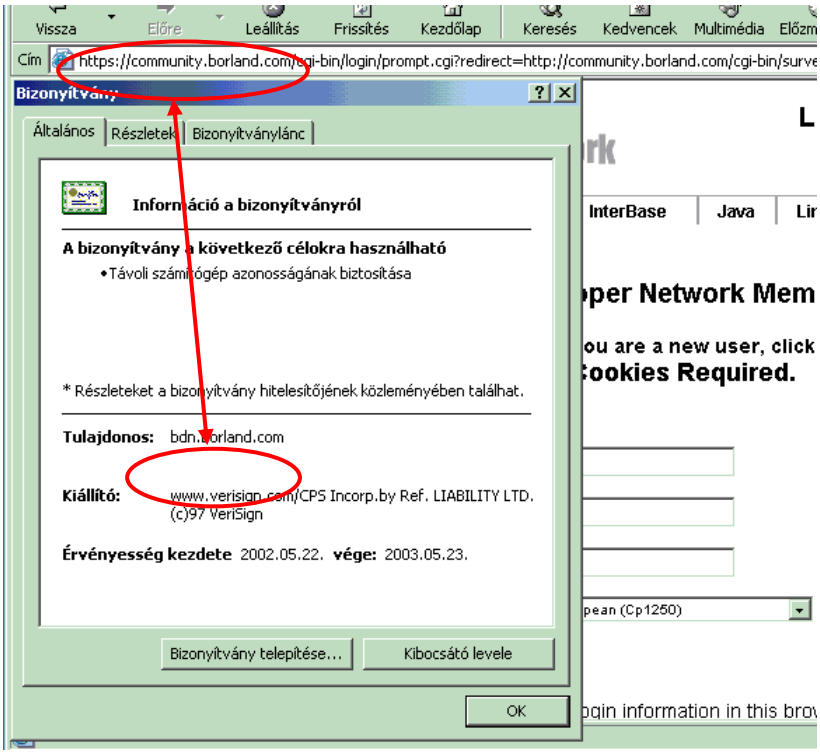
A hármas képen lévő „Bizonyítvány megtekintése” gombra kattintva azt a bizonyítványt láthatjuk, melyet a hitelesítő szervezet a saját igazolására állított ki. (Tulajdonosa és kiállítója egyaránt a *Secure Server Certification Authority*, de csak azért, mert root-CA, így őt nem hitelesíti senki.) Ha ezt a bizonyítványt megszemléljük, érdekes dolgokat láthatunk. Ami elsőre feltűnik, az a szokatlanul hosszú érvényességi időtartam. 1994. november 9. és 2010. január 8. között, akárhogy is számoljuk, körülbelül 15 év van[46]. Ma már az aláíró eljárást is kissé gyengének tartanánk, mert az MD2 algoritmust használja, amit már az RSA sem javasol. További érdekesség az 1000 bites RSA kulcs, legalábbis szokatlan, hogy a kulcs hossza nem kettő vagy 16 hatványa illetve annak többszöröse. Felhívnám a figyelmet még az SHA-1 algoritmusra, melyet a bizonyítvány ujjlenyomat algoritmusaként jelöltek meg. Az SHA algoritmust 1993. májusában fogadták el a *FIPS180*-ban. Az SHA-1 algoritmust az előbbi felülvizsgálataként a *FIPS180-1*-ben fogadták el 1995. áprilisában [15,66]. Akkor most hogyan is jön ide az SHA-1, ha a bizonyítvány 1994. novemberétől érvényes? Némi magyarázatul szolgálhat, hogy az SHA és az SHA-1 megnevezés gyakran összemosódik. Az SHA egy gyakorlatban ritkán használt algoritmus, szinte minden alkalmazásban utódját, az SHA-1-et valósították meg.

Ez a bizonyítvány az előző oldali Borland bizonyítvánnyal együtt ilyen (is lehet) a gyakorlatban: (Base64(DER), borland.cer, részlet)

```
-----BEGIN CERTIFICATE-----
MIIDZjCCAtOgAwIBAgIQRh/oJJ8yM0k4vSOpncI5OjANBgkqhkiG9w0BAQUFADBf
MQswCQYDVQQGEwJVUzEgMB4GA1UEChMXUlNBIERhdGEgU2VjdXJpdHksIEluYy4x
LjAsBgNVBAsTJVNlY3VyZSBTZXJ2ZXIgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkw
HhcNMDEwMTE4MDAwMDAwWhcNMDIwMTE4MjM1OTU5WjCBqzELMAkGA1UEBhMCVVMx
EzARBgNVBAgTCkNhbGlmb3JuaWExHDAaBgNVBAoUE0lucHJpc2UgQ29ycG9yYXRp
b24xFDASBgNVBAsUC2JvcmxhbmQuY29tMTMwMQYDVQQLFCpUZXJtcyBvZiB1c2Ug
YXQgd3d3LnZlcmlzaWduLmNvbS9ycGEgKGMpMDAxHjAcBgNVBAMUFWNvbW11bml0
.....................................
-----END CERTIFICATE-----
```

[46] Találtam ennél hosszabb érvényességi időt is: *Entrust.net Secure Server Certification Authority* saját bizonyítványa, mely „Biztonságos e-mail és Kiszolgáló hitelesítésére” használható, 20 évig és fél óráig érvényes 1999. május 25. 18:09:40 és 2019. május 25. 18:39:40 között. A Windows által készített EFS-bizonyítvány még hosszabb ideig érvényes, majdnem 100 évig (99 év 11 hónap)…

Néha azért a nagyok is bakiznak. Az alábbi képen látható bizonyítványt a böngésző érvénytelennek nyilvánította, bár az egyes részletekben első ránézésre nincs hiba. Viszont nem mindegy, hogy egy bizonyítványt hol használunk fel… Ennek a bizonyítványnak „mindössze" annyi a baja, hogy más szerver számára állították ki (`bdn.borland.com`), mint amelyik használja (`community.borland.com`). A helyzet olyan, mintha valaki egy talált személyi igazolvánnyal akarná igazolni magát. Az irat valódi, csak éppen máshoz tartozik.



49. ábra Borland certificate baki

8.6. DIGITÁLIS ALÁÍRÁS JOGI SZABÁLYOZÁSA MAGYARORSZÁGON

2000. augusztus végén kormányhatározat rendelte el az elektronikus aláírást szabályzó törvény megalkotását, előkészítését. Az előkészített törvénytervezetet 2001. május végén fogadta el az Országgyűlés, és szeptember elején lépett hatályba (*2001.évi XXXV. törvény*, melynek teljes szövege és indoklása letölthető például az [URL20] címről vagy elolvasható a *Magyar Közlöny 2001/65.* számában. Amennyiben valaki a részletes indoklásra is kíváncsi, keresse meg a törvénytervezet szövegét az iménti címen, amelyben az is benne van.)

A törvény az *Európai Parlament és Tanács* elektronikus aláírásokra vonatkozó irányelvein (1999/93/EC, 1999. december 13.) alapszik. Az aláírás jogi szabályozásának az EU irányelvek szerint technológiafüggetlennek kell lennie. Az ilyen szabályozás csak az aláíráslétrehozó eszközzel szembeni követelményeket határozza meg, és nem azt, hogy ezeknek milyen technológiával tegyen eleget. A digitális aláírás előzőekben áttekintett jellemzői közül jogilag is fontosak a következők:

- könnyen létrehozható és ellenőrizhető
- nem hamisítható és letagadhatatlan
- az aláírás hitelesíti a dokumentum tartalmát és az aláíró személyét is
- szavatolni kell a nyilvános kulcs személyhez kötöttségének valódiságát
- meg kell oldani a hagyományos keltezés elektronikus megfelelőjét is

A törvény jelentősége abban áll, hogy – néhány kivételtől eltekintve – az elektronikus aláírást minden szempontból egyenlővé teszi a hagyományossal, annak minden jogkövetkezményével együtt, ha az elektronikus aláírás bizonyos biztonsági feltételeknek eleget tesz. Ez az egyenlőség éppúgy igaz a polgári perrendtartás, a bűnvádi eljárások, mint az államigazgatás és az üzleti élet területén. A törvénytől függetlenül eddig is használhattunk elektronikus aláírást, mert a technológia mind elviekben, mind gyakorlatban már régen rendelkezésre áll, csak éppen a jogi szabályozása nem létezett eddig.

A Függelék jelen fejezethez kapcsolódó alfejezetei

14.4. Moduláris aritmetika nagyon dióhéjban

14.7. Hibrid kriptorendszer digitális aláírással, viszonykulccsal – logikai vázlat

További kiegészítések folyamatosan bővülő helye a http://www.netacademia.net/konyv webcím.

A titkosírások megfejtésének tudománya dedukción és ellenőrzött kísérleteken alapszik. Feltevéseket fogalmaz, ellenőrzi és gyakran elveti őket. Az a szövegrész, amely kiállta az ellenőrzés próbáját, egyre gyarapszik, míg végül eljön a pillanat, mikor a kísérletező szilárd talajt érez a lába alatt: feltevései összefüggést kapnak, és a jelentéstöredékek értelmessé válnak: törik a kód.

John Chadwick (kriptográfus, nyelvész):
A lineáris B megfejtése

9. Üzenetpecsétek

Ha Alice az egész dokumentumot kódolja az aláírás során, Bob azt csak akkor tudja majd elolvasni, ha egyúttal az aláírást is ellenőrzi. Ha az ellenőrzőalgoritmus lassú, kisebb teljesítményű eszközökön a gyakori aláírásellenőrzés gondot jelenthet, főként, ha az aláírás ellenőrzése nem szükséges minden esetben, vagy az ellenőrzés eredménye közömbös. Ezért Alice a gyakorlatban nem az egész üzenetet kódolja a titkos kulcsával, hanem annak csak egy egyedien jellemző lenyomatát, és e kódolás eredményét csatolja, mint aláírást, a dokumentumhoz.

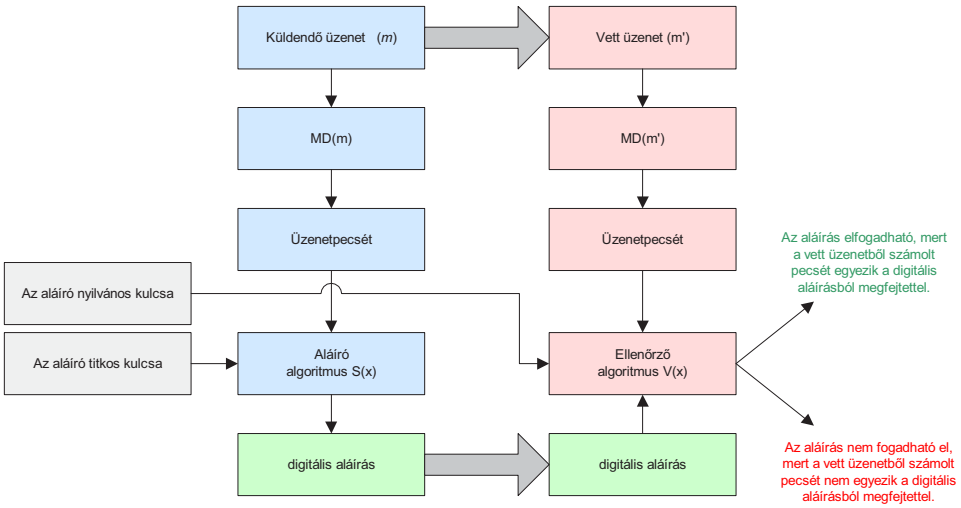
Egy bitsorozatnak is van ujjlenyomata?

Ha annak hívjuk, akkor van. Egy szöveg lenyomata (üzenetpecsét**,** lenyomat, ellenőrzőösszeg, *message digest, MD, hash value*) egy olyan rövid ellenőrzőkód – pontosabban rögzített hosszúságú bitsorozat –, amelyet az eredeti szövegből lehet generálni, és nagyon-nagyon-nagyon-nagyon kicsi a valószínűsége annak, hogy két különböző üzenet ellenőrzőkódja azonos legyen. Erre a feladatra olyan egyszerűbb megoldások is használhatók – bár nem ajánlottak, csak a példa kedvéért – mint a CRC32, vagy az egyszerű ellenőrzőösszegek, például az egyes karakterek bájtjainak összege, vagy a bájtok, bitek közötti XOR műveletek alkalmazása, hasonlóan a paritásbit számolásához. Azonban pont a paritásbit példája jól mutatja e módszerek korlátait: ha egy bit változik meg, a paritásbit jelez, de ha kettő, akkor már nem biztos, hogy fény derül a változásra.

Szerencsére a matematikus és a kriptográfus szakemberek ebben a témában sem tétlenkedtek, és olyan egyirányú függvényeket konstruáltak, melyek igen komoly követelményeket is teljesítenek. Működésük lényege röviden az, hogy a függvény bemenetére az üzenetet képviselő bitsorozatot (*m*) adjuk, a függvény rágódik rajta egy sort, és eredményül egy olyan értéket ad, amely csak *m*-re jellemző. Ennek mérete nem függ *m*-től, csak az értéke, és ezt az értéket *hashérték*nek nevezzük. Az egész folyamat egy nagy darálóhoz hasonlít. Az üzenetpecsétek kiszámolása sokkal kevesebb időt vesz igénybe, mint ugyannak az üzenetnek a nyilvános kulcsú titkosítása. (Ezt az ellenőrzőösszeget kódolja Alice a titkos kulcsával, így egy olyan elektronikus aláírást kap, ami róla, mint feladó személyéről és az üzenetről egyaránt hordoz információt.) Az üzenetpecsétek alkalmazása olyan hitelesítési módszert és integritásvédelmet nyújt, ami titkosítatlan üzenetek küldését is lehetővé teszi, így az üzenet elolvasható az aláírás ellenőrzése nélkül is, de kérésre az ellenőrzés elvégezhető.

Ha valaki a csatolt aláírást elválasztja a dokumentumtól, máshol nem tudja felhasználni azt, hiszen származásából adódóan az aláírt dokumentumot képviseli. Egy kicserélt dokumentum az aláírás számára azt jelenti, hogy megváltozott az eredeti dokumentum tartalma,

tehát az ellenőrzésnek hibát kell jeleznie. Bob megfejti a titkosított lenyomatot Alice nyilvános kulcsával, és összehasonlítja az általa elolvasott üzenet általa számított lenyomatával. Ha a kettő egyezik, valóban ezt az üzenetet és valóban Alice küldte. Ilyen módon nemcsak azt vizsgálhatjuk, hogy kitől származik az üzenet, hanem annak tartalmi eredetisége, változatlansága is ellenőrizhető, és a hashfüggvényeknek ez az egyik fő célja.



50. ábra Aláírt üzenet küldése üzenetpecséttel

Ha a titkosítást csak arra használjuk, hogy védelmet nyújtsunk az üzenet módosítása ellen, akkor titkosítás nélkül is küldhetünk üzenetet, csak az üzenet ellenőrzőösszegét kell rejtjelezni! A címzett ennek megfejtésével ellenőrizni tudja, hogy valóban azt kapta-e, amit a feladó küldött.

A hashfüggvények másik gyakori felhasználási területe, amikor a jelszavak kódolt tárolására van szükség. Nem a jelszót, hanem csak annak hashértékét tárolják le és a jelszóellenőrzés a begépelt jelszó hashértékének és a tárolt érték összehasonlítását jelenti. Így, ha valaki meg is szerzi a felhasználók jelszavainak hashértékét, nem tudja magukat jelszavakat, hiszen nem lehet egy hashből kitalálni, hogy miből készült. Ez a hálózati forgalom lehallgatása ellen is nyújt némi védelmet, mert a jelszavak nem nyíltszövegként közlekednek. Jó identifikációs rendszerrel pedig nem lehet a hashértéket közvetlenül vagy kerülőúton „megetetni”.

A továbbiakban megismerkedünk a hashfüggvények alapvető jellemzőivel és felépítésével. A fejezet végén akár meg is valósíthatjuk az egyik legnépszerűbb hashfüggvényt, az *MD5*-öt.

9.1. TULAJDONSÁGOK

A hashfüggvényeket a következő tulajdonságok jellemzik:

1. Az üzenetpecsét *MD(m)* könnyen kiszámolható.
2. Egy adott véges hosszúságú *m* üzenethez rögzített (általában 128-160 bit) hosszúságú eredményt ad. Ezt a tulajdonságot tömörítésnek is nevezik.
3. Adott *MD(m)*-ből lehetetlen meghatározni *m*-et. Csak a hashérték ismeretében nem számolható ki az eredeti *m*, illetve megkeresése időben lehetetlen feladat. (*preimage resistance*).
4. Egy rögzített *m* üzenethez nem lehet olyan eltérő m_2 üzenetet találni, amely ugyanazt a pecsétet adja (*second preimage resistance*), de legalábbis ennek megkeresése időben lehetetlen feladat.
5. Nem lehet olyan két szabadon választott, különböző üzenetet generálni, ami ugyanazt a pecsétet adja. Az ilyen tulajdonágú függvény *ütközésmentes* (*collision resistance*). Mivel ilyen nem létezik, ezért a „nem lehet" kitétel most is „időben lehetetlen"-re korlátozódik.
6. Ha egy bitet megváltoztatunk az eredeti üzenetben, a pecsét bitjeinek körülbelül a fele változzon meg. Ez a *lavinahatás (avalanche effect).*

A (4) és az (5) megszorításnak sajnos csak elvi jelentősége van, Ugyanis, ha egy 1000 bites üzenetből kell egy 128 bites lenyomatot készíteni, tulajdonképpen egy 2^{1000} elemet tartalmazó üzenethalmazt kell összepárosítani egy 2^{128} elemű halmazzal. Belátható, hogy ha az első halmaz minden eleméhez a második halmazból keresünk párt, a második halmaz páratlan elemei hamar elfogynak, és egyszer csak kénytelenek vagyunk olyan elemet kiválasztani, amit már korábban párosítottunk... Minél hosszabb az üzenetpecsét, annál jobban enyhül a probléma, maximális eredmény akkor érhető el, ha a pecsét hossza megegyezik az üzenet hosszával vagy nagyobb annál, de ezek egyike sem nevezhető hatékony megoldásnak. Emiatt csak az időbeli lehetetlenség biztosítása a cél. Az első három tulajdonság alaptulajdonság, minden függvénynél elvárás. A szakirodalomban a különböző tulajdonságokra vonatkozóan a következő terminológiával is találkozhatunk:

- *preimage resistance = one-way (egyirányú)*
- *2^{nd} preimage resistance = weak collision resistance (gyengén ütközésmentes)*
- *collision resistane = strong collision resistance (erősen ütközésmentes)*

Magukat a függvényeket is meg lehet különböztetni aszerint, hogy az egyes tulajdonságok közül melyikkel bírnak:

- *preimage resistance + 2^{nd} preimage resistance = one way hash function (OWHF, weak one way hash function)*
- *2^{nd} preimage resistance + collision resistance = collision resistance hash function (CRHF, strong one way hash function)*

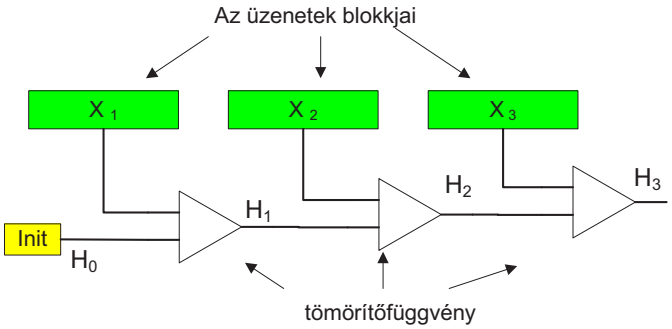
A továbbiakban nem alkalmazok ilyen finom megkülönbözetéseket, mert nem is fogok ilyen részletes tárgyalásba bocsátkozni, de nem árt ismerni a pontos elnevezéseket.

Iteráció: ismétlés a siker anyja?

A hashfüggvények első generációi a DES-hez hasonló blokkos algoritmusokon alapultak. Mivel lassúnak bizonyultak – illetve a későbbi algoritmusok lettek gyorsabbak – végül feledésbe merültek az ilyen megoldások. Napjaink minden hashfüggvénye iterációs eljárással dolgozik, a tetszőleges hosszúságú bemeneti adatot rögzített méretű blokkok sorozataként értelmezi. A bemeneti adatot (X) a blokkméret többszörösére kell kiegészíteni, ha mérete ettől eltér. Így az üzenet feldarabolható lesz t darab egyenlő részre: $X_1 ... X_t$. Ezután a $h(x)$ hashfüggvény a következőképpen írható le:

H_0=*Init_Érték*
$H_i=f(H_{i-1},X_i)$
$h(X)=H_t$

$1 \leq i \leq t$, és $f(x)$ a tömörítőfüggvény (*compression function*) valamint H_i az *i.* és az *i-1.* állapot közötti láncváltozó (*chaining variable*)



Egyes alkalmazásokban kulcs bevonásával egészítik ki a pecsét generálásának folyamatát. Jellemzően a H_0 értéke ekkor nem egy specifikáció szerinti állandó kezdőérték, hanem a kulcstól függ, vagy éppen maga a kulcs. Az ilyen kulcsos pecséteket – megkülönböztetésként az ***MD*** (*message digest*), vagy ***MDC*** (*modification detection code*) pecsétektől, amelyek csak adatintegritás ellenőrzésére alkalmasak – ***MAC*** betűhármassal (*messages authetication code*) jelölik, és egyidejű integritás- és eredetvizsgálatra is alkalmasak. (A kettő közötti átmenet a digitális aláírás, amely a *MD* eredményét egy adott kulccsal titkosítja, de maga az MD algoritmus és annak eredménye nem függ a kulcstól.)

Az egyik legnépszerűbb hashfüggvény, amit nagyon sok helyen használtak, a *Rivest* által tervezett MD4 (1990) volt. Gyors, 32-bites kódra optimalizált. *Rivest* később újratervezte és megerősítette az algoritmust: ez lett az MD5 1992-ben.

MD2: 1989, Rivest. Úgy egészíti ki az üzenetet, hogy annak bájtban mért hossza osztható legyen 16-tal, majd egy 16 bájtos ellenőrzőösszeget tesz a végére. 8 bites processzorokon is jól érzi magát. *Rogier* és *Chauvaud* megmutatta, hogy az ellenőrzőösszeg elhagyásával az algoritmus megtörhető. Más törési módszer vagy eredmény nem ismert.

MD4: 1990, Rivest. Úgy egészíti ki az üzenetet, hogy annak bitben mért hossza + 448 osztható legyen 512-vel. Ezt kiegészíti a 64 biten tárolt üzenethosszal. *Hans Dobbertin* megmutatta, hogy az algoritmus egy tipikus PC-vel perces nagyságrendű idő alatt törhető (*collision* – lásd előző oldal). Főként az ő munkájának köszönhetően az MD4-et ma már feltörtnek tekintjük.

9.2. MD5

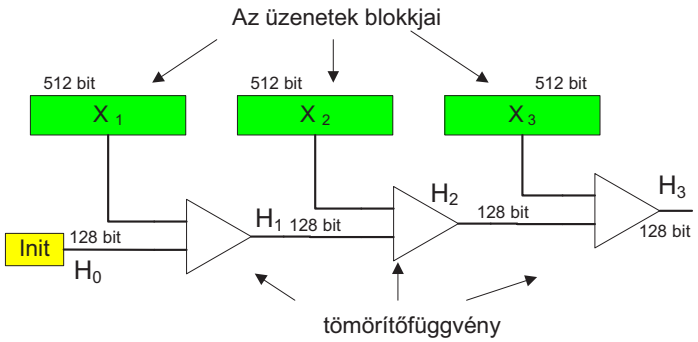
Az MD5 (Message Digest algoritmus 5. változata, *Ron Rivest*, RFC 1321, 1992. április) egy olyan egyirányú hashfüggvény, amely egy tetszőleges hosszúságú üzenetből rögzített 128 bit hosszúságú bitsorozatot generál. 32 bites gépekre optimalizált, minden kimeneti bit értéke függ minden bementi bit értékétől [2,7]. Mielőtt a pecsét kiszámolását megkezdi, az üzenetet kiegészíti egy 100000000…00 sorozattal úgy, hogy a bemeneti üzenet hossza bitekben mérve 448 bit legyen (*mod* 512). A kitöltő bitsorozat hozzáfűzése akkor is megtörténik, ha az üzenet hossza eredetileg is ennyi, vagyis a kitöltés kötelező (*forced padding*). Ezután hozzáfűzi az eredeti üzenet hosszának 64 bites reprezentációját (low-order bájtsorrendben), így a bemeneti üzenet hossza végül 512 bit többszöröse lesz.

```
„abc” = 0x61 0x62 0x63
1100001 1100010 1100011 1 0000000....0000 00011000 ..... 00000
|       Üzenet         | 1 |    423 db    | | 64 bites hossz   |
```

A számolás során egy 128 bites pufferben készül az üzenetpecsét, ezt a puffert egy meghatározott kezdőértékkel (H_0=*0123456789abcdef*) kell indítani. A feldolgozás 512 bites blokkonként (X_1...X_t) történik. Az algoritmus minden „üzenetszeletet” alaposan összekever a 128 bites pufferrel, ráadásul felhasznál egy (szinuszfüggvény értékeiből készített) „random” táblázatot is. Minden bementi blokkon négyszer hajtja végre ezt a speciális keverést.

> Nem azért használja a szinuszfüggvényt, mert az véletlenszerűbb értékeket adna, mint más véletlenszám-generátor, hanem hogy kerülje annak gyanúját, hogy az algoritmusban kiskapu lenne. Az, hogy a DES-ben alkalmazott helyettesítések alapelvét nem hozták nyilvánosságra, sok kényelmetlen feltételezést vont maga után.
>
> Jóllehet, ha Rivest megkérdezett volna egy marketingszakembert, az biztosan azt javasolta volna, hogy ne a szinuszfüggvényt használja, hanem keressen másikat. Így ugyanis egy rosszindulatú szójátékkal elmondható az algoritmusról, hogy a „random” táblázat értékei a bűn szülöttei... (Ugyanis a szinusz függvény jelölésére használt „sin” az angolban értelmes szó és jelentése „bűn”.)

Miután az összes bemeneti blokkot feldolgozta, a 128 bites pufferben lévő érték adja az üzenet pecsétjét. 1996-ban *Hans Dobbertin* vizsgálta az MD5 algoritmust, és bár teljes mértékben nem sikerült feltörnie, rámutatott az algoritmus néhány gyenge pontjára. Az RSA Lab. is ismertet egy lehetséges támadási módszert [50]-ben.



Az MD5 lépései applikációs mélységben

A következőkben a „szó” 32 bites, a bájt pedig hagyományosan 8 bites egységet jelöl. A bitek sorrendje a bájtokban a megszokott „high-order”, tehát az MSB az első (7.) és az LSB az utolsó (0.). A 32 bites szó nem más, mint a 8 bites bájtok sorozata, „low-order” sorrendben, tehát a legkisebb helyértékű bájt van elöl (*Intel* konvenció).

MD5 message digest algorithm

Bemenet: • meghatározatlan, de véges hosszúságú bitsorozat *M*

Kimenet: • 128 bites hashérték, a bemenetre adott bitsorozat MD5 pecsétje

1. *Kitöltőbitek hozzáfűzése*
   A bemeneti üzenetet úgy kell kiegészíteni, hogy bitekben mért hossza 448-at adjon maradékul, ha 512-vel elosztjuk. A kitöltést mindig el kell végezni, *akkor is, ha az eredeti hossz megfelel a fenti feltételnek.* Elsőnek egy darab „1” bitet fűzünk az üzenethez. A kitöltést „0” bitekkel addig folytatjuk, amíg az üzenet hossza 448 bit nem lesz (*mod* 512).

2. *Kitöltés befejezése*
   A kitöltés utolsó lépéseként az üzenet hosszának 64 bites reprezentációját fűzzük az üzenethez low-order bájtsorrendben. Ha az üzenet hosszabb, mint 2^{64} bájt, csak a hossz alsó 64 bitjét kell iderakni. E lépés után az üzenet hossza 512-vel osztható (egy 512 bites blokk 16 darab 32-bites szóból áll). Az üzenet jelölése legyen `M[0..N-1]`, ahol `N` így 16 többszöröse és `M[i]` egy 32 bites szó. (A gyakorlatban a kitöltés nem megelőzi a feldolgozást, hanem az utolsó lépések egyike. Igen kényelmetlen lenne, egy 4 Gbájtos fájlt előre beolvasni, ehelyett amikor az utolsó blokkhoz érkezünk, akkor a fentiek szerint kiegészítjük.)

3. *Előkészítések*
   Négy darab 32 bites puffer (`A,B,C,D`) tartalmazza a számítások részeredményét, és itt keletkezik majd a végeredmény is. A következők szerint kell inicializálni:

```
word A: 01 23 45 67 = 0x67452301 word B: 89 ab cd ef = 0xefcdab89
word C: fe dc ba 98 = 0x98badcfe word D: 76 54 32 10 = 0x10325476
```

   Definiáljuk a következő függvényeket (vagy makrókat), amelyek majd a tömörítőfüggvényhez kellenek:

```
F(X,Y,Z) = (X and Y) or (not(X) and Z)
G(X,Y,Z) = (X and Z) or (Y and not(Z))
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X or not(Z))
```

   Következő lépésben készítsünk egy olyan 64 elemű táblát (`T[1..64]`), melynek elemeit a szinuszfüggvényből számítjuk ki:

```
T[i] = trunc(4294967296 * abs(sin(i)),
```

   ahol a *TRUNC* az egészrész képzést (és a lebegőpontos-egész konverziót) jelenti és a szinusz radiánban értelmezi operandusát.

4. *Üzenet feldolgozása*

```
for i := 0 to N/16-1 do
 { Az i. blokk bemásolása az X munkaváltozóba }
 for j := 0 to 15 do
    X[j]:= M[i*16+j]
 end of j

 {A bufferértékek elmentése }
 AA := A    BB := B    CC := C    DD := D
```

Az MD5 tömörítőfüggvénye

Első kör a négyből, ami az **F** függvényt használja. A [w,x,y,z,k,s,i] jelentése:

w:=x+((w+**F**(x,y,z)+X[k]+T[i]) rol s)

```
[A,B,C,D,  0,  7,  1]
[D,A,B,C,  1, 12,  2]
[C,D,A,B,  2, 17,  3]
[B,C,D,A,  3, 22,  4]
[A,B,C,D,  4,  7,  5]
[D,A,B,C,  5, 12,  6]
[C,D,A,B,  6, 17,  7]
[B,C,D,A,  7, 22,  8]
[A,B,C,D,  8,  7,  9]
[D,A,B,C,  9, 12, 10]
[C,D,A,B, 10, 17, 11]
[B,C,D,A, 11, 22, 12]
[A,B,C,D, 12,  7, 13]
[D,A,B,C, 13, 12, 14]
[C,D,A,B, 14, 17, 15]
[B,C,D,A, 15, 22, 16]
```

Második kör a négyből, ami a **G** függvényt használja. A [w,x,y,z,k,s,i] jelentése:

w:=x+((w+**G**(x,y,z)+X[k]+T[i]) rol s)

```
[A,B,C,D,  1,  5, 17]
[D,A,B,C,  6,  9, 18]
[C,D,A,B, 11, 14, 19]
[B,C,D,A,  0, 20, 20]
[A,B,C,D,  5,  5, 21]
[D,A,B,C, 10,  9, 22]
[C,D,A,B, 15, 14, 23]
[B,C,D,A,  4, 20, 24]
[A,B,C,D,  9,  5, 25]
[D,A,B,C, 14,  9, 26]
[C,D,A,B,  3, 14, 27]
[B,C,D,A,  8, 20, 28]
[A,B,C,D, 13,  5, 29]
[D,A,B,C,  2,  9, 30]
[C,D,A,B,  7, 14, 31]
[B,C,D,A, 12, 20, 32]
```

Harmadik kör a négyből, ami a **H** függvényt használja. A [w,x,y,z,k,s,i] jelentése:

w:=x+((w+**H**(x,y,z)+X[k]+T[i]) rol s)

```
[A,B,C,D,  5,  4, 33]
[D,A,B,C,  8, 11, 34]
[C,D,A,B, 11, 16, 35]
[B,C,D,A, 14, 23, 36]
[A,B,C,D,  1,  4, 37]
[D,A,B,C,  4, 11, 38]
[C,D,A,B,  7, 16, 39]
[B,C,D,A, 10, 23, 40]
[A,B,C,D, 13,  4, 41]
[D,A,B,C,  0, 11, 42]
[C,D,A,B,  3, 16, 43]
[B,C,D,A,  6, 23, 44]
[A,B,C,D,  9,  4, 45]
[D,A,B,C, 12, 11, 46]
[C,D,A,B, 15, 16, 47]
[B,C,D,A,  2, 23, 48]
```

Negyedik kör a négyből, ami az **I** függvényt használja. A [w,x,y,z,k,s,i] jelentése:

w:=x+((w+**I**(x,y,z)+X[k]+T[i]) rol s)

```
[A,B,C,D,  0,  6, 49]
[D,A,B,C,  7, 10, 50]
[C,D,A,B, 14, 15, 51]
[B,C,D,A,  5, 21, 52]
[A,B,C,D, 12,  6, 53]
[D,A,B,C,  3, 10, 54]
[C,D,A,B, 10, 15, 55]
[B,C,D,A,  1, 21, 56]
[A,B,C,D,  8,  6, 57]
[D,A,B,C, 15, 10, 58]
[C,D,A,B,  6, 15, 59]
[B,C,D,A, 13, 21, 60]
[A,B,C,D,  4,  6, 61]
[D,A,B,C, 11, 10, 62]
[C,D,A,B,  2, 15, 63]
[B,C,D,A,  9, 21, 64]
```

```
 {A bufferértékek frissítése }
 A := A + AA    B := B + BB    C := C + CC    D := D + DD
end of i
```

5. *Eredmény*

Az eredmény az `A, B, C, D` regiszterekben van: az „`A`” regiszter alsó bájtján kezdődik és a „`D`” regiszter felső bájtján fejeződik be.

Néhány MD5 tesztvektor az RFC 1321-ből

```
MD5("") = d41d8cd98f00b204e9800998ecf8427e
MD5("a") = 0cc175b9c0f1b6a831c399e269772661
MD5("abc") = 900150983cd24fb0d6963f7d28e17f72
MD5("message digest") = f96b697d7cb7938d525a2f31aaf161d0
MD5("abcdefghijklmnopqrstuvwxyz") = c3fcd3d76192e4007dfb496cca67e13b
```

9.3. SHA-1

Az MD5 kiváltásának egyik lehetséges alternatívája az SHA (*Secure Hash Algorithm, 1993. május*), amit az MD4 alapján az NSA fejlesztett ki, és egy átdolgozott változatát (*SHA-1, 1995. április, FIPS180-1* [15,66]) azóta szabványként is elfogadták. Ma valószínűleg a második leggyakrabban használt hashfüggvény. (Érdekes közbevetés, hogy az eredeti SHA tervezési részletei és problémái, amelyek szükségessé tették az SHA-1 kidolgozását, titkosak…) A világ hashfüggvényre szakosodott kriptográfusai szerint jól megtervezett algoritmus, az elődök legtöbb gyenge pontját kijavította.

Az MD5-höz hasonlóan 512 bites blokkokban dolgozza fel az üzenetet, viszont 160 bites pecsétet generál. A blokkonkénti keverést 80-szor hajtja végre, de minden 20. keverés után megváltoztatja a keverés módját. Az SHA-1 pecsétje 32 bittel hosszabb, mint az MD5-é. A műveletek száma azonban jóval több, így lassabb az MD5-nél, és további esetleges hátrányt jelenthet, hogy a pecsét nem kettőhatvány hosszúságú, valamint az algoritmus az Intel-konvenciótól (és az MD5-től) eltérően „big-endian” számábrázolást használ. További (politikai) különbség, hogy az MD5 egy RFC-ben került publikálásra, míg az SHA-1 kormányzati szabvány (bár 2001-ben végül mégiscsak RFC lett belőle: RFC 3174).

9.3.1. SHA-1 változatok

Bár eddig csak SHA-1-ről beszéltünk, az algoritmusnak további három változata van. Az SHA algoritmuscsalád négy tagjának összehasonlítását mutatja az alábbi táblázat [66]:

Algoritmus	Maximális üzenetméret	Belső blokkméret	Belső szóméret	Pecsét mérete	FIPS szabvány
SHA-1	$< 2^{64}$ bit	512 bit	32 bit	160 bit	180-1
SHA-256	$< 2^{64}$ bit	512 bit	32 bit	256 bit	180-2
SHA-384	$< 2^{128}$ bit	1024 bit	64 bit	384 bit	180-2
SHA-512	$< 2^{128}$ bit	1024 bit	64 bit	512 bit	180-2

A 384 bites változat megegyezik az 512 bites algoritmussal, csak a kezdőértékei mások. Az 512 bites eredmény csonkolásával adja a 384 bites eredményt. Az algoritmus kiválasztásánál további szempont lehet, hogy az SHA-1 és SHA-256 a 32 bites processzorokon érzi jól magát, míg az SHA-384 és SHA-512 a 64 biteseken. Mind a négy algoritmus felépítése hasonló, csak a használt függvények és konstansok mások. További részletek [66]-ban találhatók, itt pedig a következő néhány oldalon az SHA-1 implementációszintű leírása következik.

9.3.2. Az SHA-1 megvalósítása

SHA-1 függvények, műveletek

Az SHA-1 iterációs köreiben az alábbi 80 függvényt ($f_0, f_1, f_2, f_3, ..., f_{79}$) használja. Nem kell megijedni, ez nem nyolcvan különböző függvényt jelent, hanem csak négyet (hármat), az alábbiak szerint:

Első húsz kör:	*f(t,x,y,z)=*	*(x and y) xor (not(x) and z)*	0 ≤ t ≤ 19
Második húsz kör:		*(x xor y xor z)*	20 ≤ t ≤ 39
Harmadik húsz kör:		*(x and y) xor (x and z) xor (y and z)*	40 ≤ t ≤ 59
Negyedik húsz kör:		*(x xor y xor z)*	60 ≤ t ≤ 79

További műveletek

Forgatás balra: `rotate_left(n, b)` – a `b` szót `n` bittel balra forgatja, vagyis a kilépő felső helyértékű bitek az alsó helyértékre lépnek vissza.

Összeadás: $(a+b) \bmod 2^{\text{szó\_méret}}$

SHA-1 konstansok

Az SHA-1 minden iterációs körében egy 32 bites konstanst használ. Szerencsére ez sem nyolcvan darab állandót jelent, csak négyet, az alábbiak szerint:

Első húsz kör:	*K[t] =*	*5a827999*	0 ≤ t ≤ 19
Második húsz kör:		*6ed9eba1*	20 ≤ t ≤ 39
Harmadik húsz kör:		*8f1bbcdc*	40 ≤ t ≤ 59
Negyedik húsz kör:		*ca62c1d6*	60 ≤ t ≤ 79

Előfeldolgozás - padding

Az SHA-1 a feldolgozás előtt kötelező módon kiegészíti az üzenetet úgy, hogy bitekben mért hossza 448-at adjon maradékul, ha 512-vel elosztjuk. A kitöltést mindig el kell végezni, *akkor is, ha az eredeti hossz megfelel a fenti feltételnek*. Elsőnek egy darab „1” bitet fűzünk az üzenethez. A kitöltést „0” bitekkel addig folytatjuk, amíg az üzenet hossza 448 bit nem lesz (*mod* 512). A kitöltés utolsó lépéseként az üzenet hosszának 64 bites reprezentációját fűzzük az üzenethez high-order bájtsorrendben. (Vagyis a legkisebb helyértékű bájt az utolsó bájtba kerül, ez eltérés az MD5-höz képest.)

```
„abc” = 0x61 0x62 0x63
1100001 1100010 1100011 1 0000000....0000 00000000 ... 00011000
|       Üzenet         | 1 |    423 db    | |  64 bites hossz   |
```

A gyakorlatban a kitöltés nem megelőzi a feldolgozást, hanem az utolsó lépések egyike. E lépés után az üzenet hossza 512-vel osztható. Az üzenet jelölése legyen *M[1..N]*, ahol *M[i]* egy 512 bites blokk, és 16 darab 32 bites szóból áll.

Magic numbers

Az MD5-höz hasonlóan ez az algoritmus is néhány varázs-számmal startol (H_0). Ezeket 32 bites számokat a feldolgozás előtt munkaváltozókba töltjük:

```
    a = 67452301   b = efcdab89   c = 98badcfe   d = 10325476   e = c3d2e1f0
```

Számítás

Az üzenet blokkjain végiglépkedve az alábbi számítást végezzük:

```
aa:=a;  bb:=b;  cc:=c;  dd:=d;  ee:=e;

for t:=0 to 79 do
begin
   if t ≤ 15
     then w[t]:=M[i,t]
     else w[t]:=rotate left(1, w[t- 3] xor
                               w[t- 8] xor
                               w[t-14] xor
                               w[t-16] )

   temp := rotate left(5,a) + f(t,b,c,d) + e + K[t] + w[t];
   ee := d;
   dd := c;
   cc := rotate left(30, b);
   bb := a;
   aa := temp;
end;

a:=a+aa;  b:=b+bb;  c:=c+cc;  d:=d+dd;  e:=e+ee;
```

Kimenet

Az SHA-1 algoritmus eredményét az utolsó blokk feldolgozása után a munkaváltozókban találjuk: `output := a || b || c || d || e (munkaváltozók összefűzése: 160 bit)`

9.3.3. Kis indián – nagy indián

A gyakorlati megvalósítás során nagyon fontos figyelni arra, hogy az algoritmus az Intel-konvenciótól eltérően „big-endian” számábrázolásra épít! Ilyenkor a legnagyobb helyiértékű bájt a legkisebb című memóriarekeszbe kerül. Például az `a=67452301` „varázsszám” fizikai tárolása a szabvány elképzelése szerint:

```
0000000: 67 45 23 01
```

Ha Intel alapú platformon dolgozunk, a fenti konstanst `a=$01234567` alakban kell megadni. Sajnos nem elegendő csak a konstansok átalakítása, mert a számábrázolásra a forgatás és az összeadás is érzékeny. (Az RFC3174 minderről mélyen hallgat, hasonlóan a FIPS180-1-hez. A FIPS180-2 már külön felhívja a figyelmet a számábrázolás problémáira.)

9.3.4. A hashalgoritmusok szoftveres megvalósításainak felépítése

Eddig mind az MD5, mind az SHA-1 bemutatásánál megjegyeztem, hogy a kitöltés csak logikailag előzi meg a feldolgozást. Az általánosan használt séma szerint a következő módon történik a számítás:

Általános adatstruktúra

```
work  = record
           total   :word64;     // üzenet hosszát tárolja
           index   :word32;     // index a következő adatbájt
                                // buffer-ben történő tárolásához

           a,b,c,d,e  : word32; // munkaváltozók (→ láncváltozó szerepe)

           case boolean of
                true : (messblock : array[0..15] of word32 );
               false : (buffer    : array[0..63] of byte8  );
         end;
```

Általános eljárások

`HASH_Process( work );`

Nem nyilvános eljárás. Feladata, hogy a munkaváltozók aktuális értéke és az aktuális `messblock` alapján kiszámolja az új munkaváltozóértékeket. (→ $H_i=f(H_{i-1},X_i)$)

`HASH_Init( work );`

A kezdeti értékadások (`a,b,c,d,e` = magic numbers), a hossz-számláló és az index nullázása. (→ H_0=*Init_Érték*)

`HASH_Update( work, data, datalength );`

A `data` által megcímzett területről bemásolja az adatot a `buffer` területére. Közben figyeli a `buffert`, és ha az betelik, egy `HASH_Process` kerül végrehajtásra. Újabb és újabb adatok érkezésekor ezt az eljárást kell újra és újra meghívni.

`HASH_Final( work );`

Elfogytak az adatok, nincs mire meghívni a `HASH_Update`-et. A `buffer` vagy tartalmaz még adatot vagy nem. Tulajdonképpen mindegy, ezt a csonka üzenetblokkot kell kiegészíteni a szabályok szerint az `"100000000....+méret"`-tel. Na erre kell még egy `HASH_Update`!

`HASH_Print( work );`

A munkaváltozókat – a szabvány előírása szerinti formában – kiírja egy sztringbe vagy egyéb adatterületre.

9.4. RIPEMD

Az algoritmus első verziója 128 bites volt; egy RIPE[47] nevű EU projekt keretében készült. Fejlesztői: *Hans Dobbertin, Antoon Bosselaers* és *Bart Preneel*. (Az MD4 és az MD5 algoritmusokat is nevezték a projektbe, azonban pont *Bosselaers és Dobbertin* kutatásai alapján végül nem kerültek be az ajánlott algoritmusok körébe). A 128 bites RIPEMD algoritmust használják néhány európai banki rendszerben, de soha nem lett olyan népszerű, mint fejlesztésének alapja, az MD4. A RIPEMD-128 továbbfejlesztett verziója a 160 bites pecsétet generáló RIPEMD-160, melynek célja, hogy leváltsa 128 bites elődeit: az *MDx* családot és az eredeti RIPEMD-t. A váltást legalább két tényező indokolja:

- A 128 bites hasheredmény nem nyújt már igazán védelmet. Az egyre növekvő számítási teljesítmények miatt egy *brute-force* ütközéskeresés lassan kivitelezhetővé válik. '94-ben *Paul van Oorshot* és *Mike Wiener* megmutatta, hogy a születésnapi paradoxonon alapuló támadás 10.000.000$ befektetésével három hét alatt eredményes (ekkor egy 128 bites pecsét ütköztetéséhez legfeljebb $2^{128/2}=2^{64}$ művelet kell, lásd következő fejezetet). Ha csak 1.000.000$-t engedünk meg, a támadás időigénye körülbelül fél év. (1994-ben, ez a költség vagy idő azonban 18 hónaponként megfeleződik Moore törvényének következtében.)
- *Dobbertin* 1995 elején mind a RIPEMD, mind az MD4 vizsgálatakor ütközést talált az említett algoritmusok pecsétjei illetve tömörítőfüggvényei között. Egy évvel később az MD5 került terítékre, részleges sikerrel. Az RSA Inc. az MD4 használatát ma már nem javasolja és ellenjavallja az MD5 beépítését a jövőbeli alkalmazásokba.

A RIPEMD-160 az eredeti RIPEMD egy megerősített változata, amely 160 bites hasheredményt generál, megfelelő biztonságot nyújtva az elkövetkező legalább 10-15 évre [16]. Létezik a megerősített algoritmusnak 128 bites változata is, de használata az előbbiek miatt nem javasolt, csupán átmenetet kíván nyújtani a régi és az új algoritmus között. A 256 bites és a 320 bites változatok csupán hosszabb eredményt adnak, de nem feltétlenül nagyobb biztonságot. (Például a RIPEMD-256 nem más, mint két párhuzamosan futó RIPEMD-128 más-más kezdőértékkel inicializálva. A részeredményeket a két algoritmus egymás között cserélgeti, a végeredmény pedig a két eredmény összefűzése.) A tervezés során igyekeztek az MD4, az MD5 és a RIPEMD minden jó tulajdonságát megőrizni, és a rosszakat kiküszöbölni. A RIPEMD-160 – elődeihez hasonlóan – 32 bites processzorra optimalizált, és az alkalmazott műveletek sem különböznek jelentősen az ősök műveleteitől:

- forgatás
- *modulo* 2^{32} összeadás
- logikai műveletek (AND, OR, NOT, XOR)

[47] RACE Integrity Primitives Evaluation, 1988-1992. A projekt célja az volt, hogy nyilvánosan hozzáférhető alapalgoritmusokat ajánlásként összegyűjtsön.

A bemeneti bitsorozatot 512 bites egységekre bontja. Az 512 bites blokkokat 32 bites szavakban kezeli. Számábrázolása „*little-endian*”, tehát kisebb helyértékű bájt van elől, ahogy azt már az Intel x86 processzorcsaládot programozók megszokhatták.

	Assembly		C	
	Mbit/s	százalék	Mbit/s	százalék
MD4	165,7	146%	81,4	136%
MD5	***113,5***	***100%***	***59,7***	***100%***
SHA-1	46,5	41%	21,2	35%
RIPEMD-128	63,8	56%	35,6	59%
RIPEMD-160	39,8	35%	19,3	32%

51. ábra Optimalizált MD kódok abszolút és relatív sebessége

Pentium @ 90MHz processzoron

Az iménti táblázat forrása: *H. Dobbertin, A Bosselaers, B. Preneel: RIPEMD-160: A strengthened version of RIPEMD, 1996 April*. Egyébként ez az adatsor is jó példája a bevezetőben említett ellentmondásoknak: ugyanezek a szerzők másfél évvel később ugyanerre a processzorra ugyanebben a táblázatban a következő adatokat adták meg [16]-ban az assembly megvalósításra vonatkozóan (Mbit/s): 190.6, 136.2, 54.9, 77.6, 45.3 – Átlag 18%-kal gyorsult a 90 MHz-es Pentium? Vagy mégsem? A C nyelvre vonatkozó adatok ugyanis változatlanok maradtak.

Ha mély benyomást akarsz tenni egy biztonsági szakértőre, mondd azt, hogy az elmúlt 5 évben csak egyszer törtek be a rendszeredbe. Ha ugyanis azt mondod, hogy egyszer sem, akkor azt fogja hinni, még arra is képtelen vagy, hogy észleld a betöréseket.

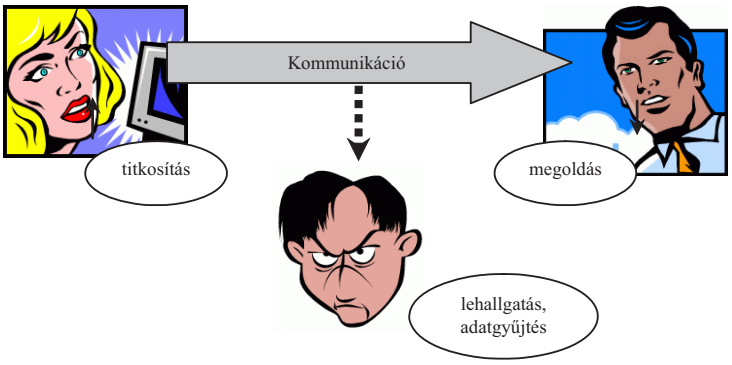
Ismeretlen

10. TÁMADÁSFAJTÁK

A most következő viszonylag rövid fejezetben a támadások főbb alapelveit tekintjük át. Nem konkrét algoritmusok vagy protokollok hackeléséről lesz szó, csak olyan irányelveket fogalmazunk meg, amelyek útmutatást adhatnak egy vélt vagy valós támadó viselkedésére, eredményességére és a támadás veszélyeire. Mindezt technológiafüggetlen módon teszzük, mint ahogy egy aktív vagy passzív támadás lényege sem függ attól, hogy a kommunikáló felek nyíltkulcsos vagy szimmetrikus algoritmust használnak a forgalom védelmére.

10.1. PASSZÍV TÁMADÁS – A NEM KÍVÁNT HALLGATÓSÁG

Egy információs rendszerhez való csatlakozás és a támadó lehetőségei alapján kétféle támadást különböztethetünk meg. Az egyiket passzív, a másikat aktív támadásnak nevezzük. *A **passzív támadás*** (*passive attack, sniffing*) esetében a behatoló hozzájut ugyan az adatokhoz, képes a teljes kommunikáció lehallgatására, adatokat gyűjthet, de ott megváltoztatni semmit sem tud (vagy nem akar) és semmilyen hamis forgalmat nem tud (vagy nem akar) bonyolítani. A támadás lényege az észrevétlen megfigyelés és adatgyűjtés, ami esetleg egy későbbi aktív támadást készít elő. Tradicionálisan a magánszféra megsértésének leggyakoribb problémájának (*privacy problem*) tekinthető.



52. ábra A passzív támadás modellje

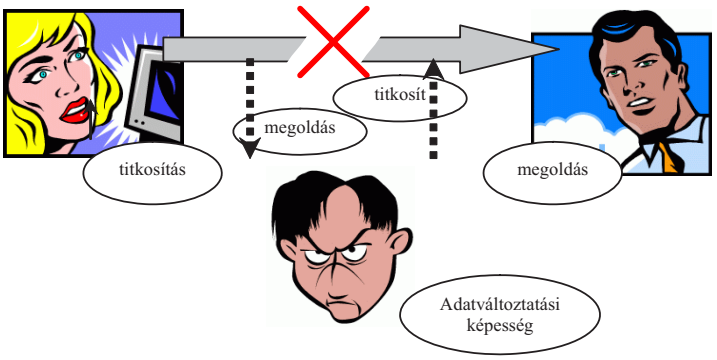
Klasszikus lehallgatás esete

10.2. AKTÍV TÁMADÁS

Az ***aktív támadás*** (*hijack, active attack*, megszemélyesítés) esetében a támadó olyan módon csatlakozik a feltört vagy feltörni kívánt rendszerhez, hogy ott képes adatokat hamisítani és a résztvevők nevében hamis üzeneteket küldeni, tranzakciókat indítani. Az aktív támadást általában passzív megfigyelés előzi meg, és az ott összegyűjtött információk felhasználásával indul az aktív támadás. Ez a támadási mód tudja a legnagyobb anyagi és erkölcsi kárt okozni. A támadó beépül a kommunikációs összeköttetésbe (gyakorlatilag beékelődik Alice és Bob közé), az üzeneteket elnyeli, és az ellenállomások helyett válaszol mindkét irányba. Találóan „*man-in-the-middle*" támadásnak is nevezik (*interleaving attack,* „*középen lévő ember*"). A támadás az alábbi feltételezéseken alapul:

1. a kommunikáló felek a beékelődött támadót partnerként azonosítják,
2. a tőle kapott üzeneteket eredeti, nem hamis üzenetnek fogadják el.

Érdekes veszélyforrás lehet, ha a támadó az egymással kapcsolatban lévő állomásokat sorozatos ismétlésre kényszeríti, esetleg ugyanazon üzenet több különböző titkosított variációját szerzi meg, vagy valamelyik állomásról ismert tartalmú, de titkosított választ kényszerít ki, így megszerezheti a nyílt szöveg titkosított változatát. Előfordulhat, hogy a támadó a továbbított üzenetek tartalmához nem fér hozzá, hamisat sem tud készíteni, mégis rombol: egyszerűen megváltoztatja a továbbított titkos blokkokat, lehetetlenné téve így a megfejtést és az egész kommunikációt.



53. ábra Az aktív támadás modellje

Beékelődés a két fél közé. A támadónak el kell hitetnie a kommunikáló felekkel, hogy mindenki a másikkal beszél. A támadónak teljesen átlátszó módon kell viselkednie.

10.3. BELSŐ TÁMADÁSOK – PROTOKOLLOK KIJÁTSZÁSA

Abban az esetben, ha a támadó fél egyébként jogosult felhasználó, akkor őt **csalónak** (*cheater*) nevezzük és általában több információt kíván megszerezni a szabályok be nem tartása révén, mint amennyit a rendszerben betöltött szerepe alapján kaphatna. A külső támadók és belső csalók közül az utóbbiak jelentik a nagyobb veszélyt egy rendszer biztonságára nézve,

mert ebben az esetben a belső felhasználók illegális tevékenységéről van szó. Rendelkezhetnek olyan információval, melyeket egy külső támadó nem ismer, és sok esetben nem kell számolnia a hozzáférést szabályzó eszközök (tűzfal, RAS, fizikai védelem) kijátszásával sem.

> A legtöbb rendszerben egyébként van olyan résztvevő vagy folyamat (*trusted entity, trusted process*), aki bizalmi pozícióban van. Képes arra, hogy megsértse a rendszer biztonsági előírásait, de magától nem teszi ezt meg, hanem betartja a szabályokat. Ha egy rosszindulatú betörés során a behatoló megszerzi ennek a résztvevőnek az azonosítóit vagy jogosultságát (általában identitását), ő valószínűleg nem lesz ilyen becsületes…

A csalók kiszűrése és megállítása azért is nehezebb a külső támadó tettenérésénél, mert a csalók pont azok a szereplők, akikben megbízunk vagy meg kell bíznunk. A csalók egy része azok közül kerül ki, akik már régóta a rendszer szereplői, de gyakori az új szereplők csalása is, ha azok kifejezetten a támadás szándékával váltak a rendszer szereplőivé. Az ipari kémkedés egyik jól bevált módja, hogy a konkurens cég egyik megbízottja próbaidős felvételt nyer a piac másik szereplőjének egy meghirdetett állására. Az új munkaerő már a belépést követően rendelkezni fog loginnal, bizonyos szintű hálózati és alkalmazáseléréssel. Lehet, hogy ez nem teljes jogú hozzáférés, de sokkal több, mintha egy külső támadónak kellene mindezt megszereznie, nem is beszélve arról, hogy képes kijuttatni olyan információkat, amelyek egy későbbi külső támadás eredményességét elősegítik.

Egy csaló általában ismeri tettének következményeit, rendszerre gyakorolt hatását is. Tudja, hogy tettét ki és mikor fedezheti fel, illetve tettének ki és milyen alapossággal fog utána nézni. Gyakran a rendszer saját erőforrásait és eszközeit használja fel a rendszer ellen. Ma már egy belső támadás valószínűsége kisebb ugyan, mint egy külsőé, de a rendszer sokkal kiszolgáltatottabb egy belső támadónak, mint egy külső behatolónak. Korábban egyfajta axióma volt, hogy a betörések 90%-át belülről hajtják végre. Változnak az idők, az ISBS 2002-re vonatkozó felmérése szerint a biztonsági incidensek már csak 34%-a köthető belső szereplőhöz (*Information Security Breaches Survey*, [URL67]).

10.4. ADATMANIPULÁCIÓ AZ AKTÍV TÁMADÁSBAN

10.4.1. Adatsérülés

Tételezzük fel, hogy egy *m* üzenetet bináris formában tárolunk, és minden bitkombináció érvényes információt jelent. Amennyiben a titkosított blokkok egymástól függetlenek, a támadó számára tálcán kínált lehetőség az adatblokkok megváltoztatása. Ha valaki nem ismeri sem a titkosított információt, sem a kulcsot, akkor is tud kárt okozni, mert egy megváltoztatott rejtjeles üzenet a megfejtés után is érvényes (bár nem feltétlenül értelmes) marad. Hasonló helyzet alakul ki, ha adatátviteli hiba miatt a rejtjeles üzenet megváltozik. Ezért az üzenet kódjait úgy kell megválasztani, hogy legyen köztük érvénytelen kód is: például ha üzenetként csak páros számokat titkosítunk és továbbítunk, a megfejtések eredményeiben felbukkanó páratlan szám valamilyen hibára utal. Az üzenetben lévő redundancia azonban kétélű fegyver. Ha a támadó valamilyen módon rájön, hogy csak a páros számok a jó üzenetek, ezt az információt

fel tudja használni egy megsejtett kulcs helyességének ellenőrzésekor: ha az üzenetből a feltételezett kulcs páratlan számot fejt meg, a kulcs biztosan nem jó. Ha valamilyen okból nem lehetséges a továbbított adatok redundáns kódolása a titkosítás előtt, átmeneti megoldásként valamilyen kiegészítő (elő)feldolgozást, például tömörítést lehet használni, így a vevőoldalon adatsérülés esetén a kicsomagolás hibát fog jelezni. Ez a megoldás csak akkor használható, ha a járulékos feldolgozás megoldható, és annak ideje nem okoz kritikus késleltetést. További hátrányt jelenthet, hogy a tömörítés idejének relatív csökkentéséhez és a tömörítés hatékonyságának növeléséhez minél nagyobb blokkokat kell küldeni. Az adatintegritást ennél sokkal rugalmasabban biztosítják az üzenetpecsét algoritmusok (lásd előző fejezetet), melyek feladata, hogy egy adott üzenetből olyan ellenőrzőösszeget generáljanak, ami az adott üzenetre jellemző. A vételi oldalon a kapott üzenet számolt ellenőrzőösszegét összehasonlítjuk a küldött összeggel, és ha a kettő nem egyezik, akkor vagy az üzenet vagy az ellenőrzőösszeg megváltozott a feladás óta.

Hibatípusok

Ha a bekövetkezett adatsérülés

- a továbbított bitek számának növekedésével (bit beszúrás) vagy csökkenésével (bit kiesés) jár, a hibát *szinkronhibának* nevezzük.
- egy vagy több teljes blokk kiesését vagy ismétlődését jelenti, *blokkszinkron hibáról* beszélünk. Vegyük észre a szinkronhiba és a blokkszinkron hiba közötti különbséget: az utóbbi nem változtatja meg a bitek blokkhatárokhoz viszonyított helyzetét, vagyis olyan szinkronhibának tekinthető, amelyben a kiesett vagy beszúrt bitek száma a blokk méretének egész számú többszöröse.
- a továbbított bitek számát nem, csak a bitek értékét változatja meg, a hibát *bithibának* nevezzük.

10.4.2. Visszajátszás

Az aktív támadás során a támadó képes arra is, hogy korábban elküldött és lehallgatott üzeneteket a számára megfelelő pillanatban vagy kedvező helyzetben újra elküldjön (*reply attack*). Ezt természetesen meg kell akadályozni (*anti-replay systems*). A probléma megoldásának egyik módja, hogy minden üzenetet – természetesen titkosított – időbélyeggel (*time stamp*) látunk el, és csak azokat az üzeneteket fogadjuk el, melyek bélyege valamilyen ésszerű határidőn belüli, a többit eldobjuk vagy visszautasítjuk. Másik lehetséges megoldás, ha minden egyes üzenetnek egyedi azonosítót adunk. Viszont minden egyes, már felhasznált azonosítót el kell tárolni valahol, ami vagy nagy adathalmazt eredményez hosszú távon, vagy ha a tárolóegység megsemmisül, nincs mit ellenőrizni. Ha minden üzenetváltás sorszámot kap, már csak azt kell tárolni, hogy hányadik sorszámnál tartunk, ez pedig nem nagy mennyiségű adat. Újként fogadunk el minden olyan üzenetet, aminek sorszáma *nagyobb* az általunk utoljára fogadotténál, a többit saját belátásunk szerint eldobjuk vagy visszaküldjük (*protocol alert*).

És ide tartozik a *challenge-response* (kérdés-válasz) jellegű azonosítás is, mint az azonosító adatok visszajátszási problémájának egyik gyakori megoldása. Az így működő protokoll szerint a szerver egy véletlen adatot titkosít a kliens számára. Amennyiben a kliens valóban birtokolja a kulcsot, képes az üzenet megfejtésére, vissza tudja küldeni a helyes véletlen adatot és elkezdheti vagy folytathatja a kommunikációt. Mivel a „feladat megoldása" minden alkalommal más és más, egy lehallgatott, rögzített csomag megismétlését a szerver nem fogja elfogadni. Ha a kliens helytelen adatot küld vissza, nem ismeri a helyes kulcsot, vagy csalással próbálkozik. A folyamat során lényegtelen, hogy a titkosítás szimmetrikus vagy aszimmetrikus algoritmussal történik-e, és nincs szükség időbélyeg, azonosító vagy sorszám használatára sem, hiszen a folyamat nem függ sem az időtől, sem korábbi lépésektől.

10.5. KÓDFEJTÉSEK TÍPUSAI

Az előző bekezdésekben említett módszerek elsősorban a kommunikációs csatornát veszélyeztető támadásokba nyújtott betekintést. A fejezet további része a kriptográfiai támadásokra, a nyílt és titkos szöveg kapcsolatára összpontosít. A kriptográfus kódfejtő célja a nyílt szöveg vagy a kulcs meghatározása, összességében a titkosított blokkok megfejtése. Attól függően, hogy eddigi tevékenysége során milyen adatok jutottak a birtokába, a következő helyzetekbe kerülhet a támadó:

- Amikor sok titkos szöveggel, de egyetlen nyílt szöveggel sem rendelkezik, akkor a **csak titkosított szöveg alapú** problémával áll szemben. Gyakori esetnek tekinthető, bár a kódfejtőnek gyakran lehet sejtése az eredeti nyílt szövegre vonatkozóan. (*ciphertext only attack*)
- Amikor néhány nyílt szöveget és azok titkosított párját is ismeri, **ismert nyílt szöveg alapú** támadást hajthat végre. A ma használt algoritmusok nagy része nyilvános, így a támadó maga is állíthat elő ilyen párokat, például egy megsejtett kulcs ellenőrzésére (*know plaintext attack*).
- Amikor a kódtörőnek lehetősége van az ismeretlen kulccsal a saját maga által választott nyílt szöveg kódolására, a támadás **választott nyílt szöveg alapú**. (*choosen plaintext attack*)
- Ha a támadónak nemcsak eseti lehetősége van a kódolásra, hanem újra és újra akárhányszor kérheti egy nyílt szöveg rejtjeles párját, a módszert **adaptív választott nyílt szöveg alapúnak** nevezzük. (*adaptive choosen plaintext attack*)
- Amikor a kódtörő tetszőleges rejtjelezett szöveget megfejthet egy fekete dobozzal, akkor **választott titkosított szöveg alapú** támadásról beszélünk. (*adaptive choosen ciphertext attack*)

Az első két eset szinte bármikor előfordulhat, de az utolsó háromra beékelődéses aktív támadás esetén van a legnagyobb esély. Ha valaki azt gondolja, hogy egy titkosítóalgoritmus biztonságos, ha ellenáll a *csak titkosított szöveg* típusú támadásoknak, sajnos nincs igaza. Sok esetben a kódfejtő jó becslést tud adni a nyílt szöveg egyes részeire. Például egy többfelhasználós rendszer első üzenetei között valószínűleg megjelenik a „login" szó. Egy üzleti üzenet-

ben valószínűleg felbukkan az „ezer” vagy a „millió” szó valamilyen pénznem megnevezésének kíséretében, vagy egy cégnév, ami a kommunikáló felek valamelyikéhez köthető. Néhány nyílt szöveg - titkos szöveg párral pedig a kódtörő munkája egyszerűsödhet.

10.6. KÓDFEJTÉSEK ÉS FELTÖRÉSEK EREDMÉNYESSÉGE

A korábbiakban gyakran használtuk azt a kifejezést, hogy egy algoritmust vagy egy üzenetet feltör a támadó. A feltörés *(code breaking, attack)* az az eljárás, amikor egy titkosított üzenetből a kulcs ismerete nélkül megfejtik az eredeti üzenetet. Elképzelhető az is, hogy a támadó olyan alternatív algoritmust készít, amely a kulcs nélkül is képes a visszafejtésre.

> Ha elfogadjuk, hogy a kulcs nem az algoritmus egyik paramétere, hanem kijelöl egy algoritmust, könnyebben belátható, hogy létezik ilyen alternatív algoritmus. Igaz azonban az is, hogy egy ilyen algoritmus valószínűleg csak az adott kulccsal titkosított blokkok visszafejtésére képes. (lásd: „*3. Szimmetrikus kulcsú módszerek*” fejezet második oldalát!)

Ha a titkosításhoz használt kulcsot megtalálja a támadó, és így a további üzeneteket gond nélkül el tudja olvasni, *teljesen feltörtnek* tekintjük a kommunikációt. Ezek az esetek a következőképpen néznek ki, veszélyességi (csökkenő) sorrendben:

- **Teljes feltörés** (*total break*). A támadó megtalálja a feltöréshez szükséges kulcsot, így minden üzenetet el tud olvasni, sőt újabbak rejtjelezésére is képes: $\boldsymbol{D_K}(M)=m$.
- **Teljes következtetés** (*global deduction*). A támadó nem találja meg a feltöréshez szükséges kulcsot, de egy olyan alternatív algoritmust készít, amellyel el tudja olvasni az adott kulccsal titkosított üzeneteket: $A(M)=D_K(M)=m$. Újabb üzenetek rejtjelezésére valószínűleg nem képes.
- **Egyedi vagy lokális következtetés** (*instance or local deduction*). A támadó nem találja meg a kulcsot, és általános alternatív algoritmust sem talál, de egy bizonyos elfogott üzenetet megfejt. (Ebből az eredményből utána elindulhat a siker felé, de ez most lényegtelen. Legfeljebb felsőbb osztályba lép a támadás.)
- **Informatív következtetés** (*information deduction*). A támadó a kulcsról vagy a titkosítatlan üzenetről részinformációkat szerez (ez lehet a kulcsnak, vagy a nyílt szövegnek egy része, néhány betű vagy bit), de teljes egészében sem a kulcsot, sem az üzenetet nem ismeri meg.

10.7. HASHTÖRÉSEK ALAPJA - A SZÜLETÉSNAPI PARADOXON

A címbeli kifejezés az üzenetpecsétek egyik támadási módját takarja. A születésnapi paradoxonon alapuló támadások matematikai statisztikán alapszanak és azt használják ki, hogy van esélye annak, hogy két kriptografikus művelet eredménye ugyanaz lesz. Igazából nem is paradoxon, csak néha hihetetlennek tűnik a helyes eredmény, ezért hívják így. A probléma eredeti két kérdése a következő:

A1. Hány embernek kell együtt lennie ahhoz, hogy egy kiválasztott ember születésnapja legalább 50%-os valószínűséggel megegyezzen egy adott dátummal?

Mindenkinek 1/365 esélye van, ezért:

$$n * \frac{1}{365} \geq 0{,}5 \qquad \Rightarrow \qquad n \geq 183$$

A2. Hány embernek kell együtt lennie ahhoz, hogy közülük tetszőleges kettőnek legalább 50%-os valószínűséggel megegyezzen a születésnapja?

Minden *párosnak* 1/365 esélye van, ezért:

$$\frac{n*(n-1)}{2} * \frac{1}{365} \geq 0{,}5 \qquad \Rightarrow \qquad n \geq 20$$

A fenti kérdések az üzenetpecsétekre vonatkozóan a következőképpen néznek ki (egy 128 bites üzenetpecsétjét feltételezve):

B1. Ha adott egy X_1 üzenet, hány X_2 üzenetet kell generálni, hogy az X_2 üzenetek között k valószínűséggel legyen legalább egy olyan, aminek pecsétje megegyezik X_1 pecsétjével? (Vagyis egy olyan üzenetet keresünk, melynek pecsétje adott.)

$$n * \frac{1}{2^{128}} \geq k \qquad \Rightarrow \qquad n \geq k * 2^{128}$$

Ennek a keresésnek *k=100%* esetben és 1 millió pecsét/másodperc sebességet feltételezve több, mint 10^{25} év az időigénye.

B2. Néhány esetben egy protokoll támadásához az is elég, ha találunk két olyan X_1 és X_2 közömbös tartalmú üzenetet, melyek kötetlenek, csak pecsétjük legyen azonos. Hány üzenetpárt kell generálni, hogy k valószínűséggel legyen közöttük jó páros?

$$\frac{n*(n-1)}{2} * \frac{1}{2^{128}} \geq k \qquad \Rightarrow \qquad n \geq \sqrt{2 * k * 2^{128}}$$

Ennek a keresésnek *k=100%* esetben és 1 millió pecsétpáros/másodperc sebességet feltételezve már „csak” 584 542 év az időigénye.

Ha a vizsgált üzenetpecsét hossza nem 128 bites, hanem csak 64 bites lenne, a két időtartam B1=584 542 év és B2=72 perc (!!!) lenne. Ez a különbség már jóval érzékelhetőbb, és sokkal veszélyesebb is. A kapott eredmény „formája” miatt „négyzetgyök-támadásnak” (*square root attack*) is nevezik.

S A T O R
A R E P O
T E N E T
O P E R A
R O T A S

11. Eltemetett bitek: Szteganográfia

Van egy másik módszer is a bizalmas üzenetváltásra, amikor nem titkosítunk, csak elrejtünk valamit valamibe. Az eljárás nem újdonság, különböző megvalósításait az ókoriak éppúgy használták, mint a XX. század embere a II. Világháborúban. Azt, hogy hogyan és miként, mennyire hatékonyan, megtudhatjuk ebből a fejezetből. A fejezet nem fog annyi megoldást bemutatni, mint a titkosítással foglalkozó előző rész, de ha megértettük a *rejtett írás, adatrejtés* ötletét, most tényleg csak a képzelőerőnk szab határt [24, 27, 31, 39][48].

11.1. A szteganográfia célja

A görög származású *szteganográfia* szó jelentése: *rejtett írás.* Ez a szó sok eljárást foglal magába, amelyeket a védendő kommunikáció során használhatunk, mint például a láthatatlan tinta, mikroírás, a betűk és szavak eltolásos elhelyezése, szinonim szavak szabályszerű használata, ismert kommunikációs csatornák közé titkosak keverése, szórt spektrumú kommunikáció és így tovább. *Markus Kuhn* 1995-ben ezt írta az eljárásról: *„A szteganográfia a kommunikáció művészete és tudománya, lehetőség magának a kommunikációnak az elrejtésére. Ellentétben a kriptográfiával, ahol a támadó észreveheti, feltörheti, és módosíthatja az üzenetet, a szteganográfia célja, hogy a nyílt szöveget úgy rejtse el a gyanúmentes üzenetbe, hogy a támadó ne is láthassa meg, hogy a továbbított üzenet egy második – esetleg titkosított – üzenetet tartalmaz”.* Vagyis a titkosítás az üzenet értelmét változatja meg, míg az adatrejtés az információcsere tényét igyekszik titkolni, elfedni, egy kívülálló csak az ártalmatlannak tűnő hordozót látja. Egy titkosított üzenet látszólagos zagyvasága elégséges a gyanú felébresztésére, de ha nincs titkosításra utaló jel, a feltörés provokációja is kisebb.

> A szteganográfiát egyes (különösen katonai) irodalmakban átviteli biztonságnak (*transmission security, TRANSEC*) is nevezik. A magyar szóhasználatban a lényeget jól megmutató kifejezés, az adatrejtés (*data hiding*) terjedt el.

11.2. A szteganográfia történelmi előzményei

Áttekintve a történeti emlékeken láthatjuk, hogy az információ elrejtésére számtalan módszert dolgoztak ki az idők folyamán. Az ókori Görögországban viasszal bevont táblákat használtak az írásra. Egy történet szerint, amikor Demeratus figyelmeztetni akarta Spártát, hogy Xerxész Görögország inváziójára tervezi, lekaparta a viaszréteget és üzenetét a csupasz fa ré-

[48] A fejezet előtti "mottó" feloldását lásd a "*14.14. A Sator négyszög*" című fejezetben!

tegre írta fel. Ezután az egészet úgy vonta be ismét viasszal, hogy egyetlen ellenőrzés során sem derült fény a rejtett üzenetre. Egy másik, nem igazán emberbarát módszer volt, hogy egy megbízható rabszolga fejét leborotválták és az üzenetet a csupasz fejbőrre tetoválták. Miután a küldönc haja ismét megnőtt, elindulhatott, és az üzenet mindaddig láthatatlan maradt, amíg meg nem borotválták ismét. A láthatatlan írás egyszerűbb és szokásos módszere volt a láthatatlan tinták használata, amely igen jól helyt állt, a II. Világháborúban is. Egy ártatlannak tűnő levél sorai között sok fontos információt rejthettek el így. A világháború korai szakaszában az alkalmazott szteganográf módszerek köre szinte kizárólag a láthatatlan tinták használatára korlátozódott. Ezek a tinták általában tej, ecet, gyümölcslevek felhasználásával készültek és közös jellemzőjük, hogy száradás után láthatatlanok lesznek, de melegítésre elsötétednek vagy egy másik anyaggal kezelve láthatóvá válnak. A technológia fejlődésével a láthatatlan tinták láthatóvá tétele egyre könnyebb lett, ezért sokan kísérleteztek azzal, hogy egyre többféle kémiai anyag felhasználásával egyre biztonságosabb tintákat állítsanak elő.

A kommunikációs csatornákon a sok titkosított üzenet között gyakran kódolatlan (*null ciphers*) szövegek is közlekedtek, de korántsem biztos, hogy minden az volt, aminek látszott. Az alábbi angol nyelvű szöveg egy időjárással kapcsolatos jelentés:

> `News Eight Weather: Tonight increasing snow. Unexpected precipitation smothers eastern towns. Be extremely cautious and use snowtires especially heading east. The highways are knowingly slippery. Highway evacuation is suspected. Police report emergency situations in downtown ending near Tuesday`[49].

Fondorlatos módon, most olvassuk össze minden szó első betűjét:

> `Newt is upset because he thinks he is President`[50].

A következő üzenetet egy német kém küldte el a II. világháborúban [75]:

> `Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by products, ejecting suets and vegetable oils.`[51]

Ha minden szó második betűjét összeolvassuk, egészen más értelmet kap a szöveg:

> `Pershing sails from NY June 1.` [52]

A könyv készítése előtt, alatt többen jelezték, hogy a fenti két szöveg hibás. Valóban, az elrejtett és a „kihámozott” szöveg között van néhány betűnyi eltérés, de én ezt nem tartom hibának. Véleményem szerint az üzenet elrejtője az emberi kommunikáció redundanciájára alapozott, és bízott benne, hogy az apró hibák ellenére az üzenet érthető marad.

[49] Értelem szerinti fordításban: "Nyolc (órás) hírek időjárás-jelentése: Ma növekszik a havazás. Hirtelen havazásra lehet számítani a keleti városrészben. Óvatosság és hólánc használata ajánlott. Az autópályák csúszósak, lezárásuk várható. A rendőrség nagy fennakadásokról számolt be a belvárosban, és keddig nem várható változás."
[50] "Newt meghibbant, mert azt hiszi, ő az Elnök."
[51] Nyersfordításban: „A semlegesek tiltakozását nyilvánvaló módon teljesen figyelmen kívül hagyták. Isman érzékenyen érintve. A blokád következményei érintik az embargós termékekért emelt kifogásokat is, a faggyú és a növényi olaj visszautasítását.”
[52] Pershing június elsején kihajózik New Yorkból.

Ahogy egyre több módszert dolgoztak ki és alkalmaztak a titkos üzenetet váltani kívánók, a cenzorok (*Office of Censorship*) egyre szélsőségesebb módon próbálták ellátni a feladatukat: tilos volt például olyan házhozszállítási megbízásokat postázni, amelyen (kézbesítési) időpont szerepelt, tilos volt keresztrejtvényeket, rejtvényeket és általában minden olyan dokumentumot postai úton elküldeni, ami titkos üzenetet tartalmazhatott. Ha a cenzor végképp nem tudott „belekötni" egy levél vagy képeslap tartalmába, akkor összecserélgette a bélyegeket vagy a szavak sorrendjét, esetleg egyes szavakat más, rokon értelmű szavakkal helyettesített, vagy éppenséggel újrafogalmazta az egész levelet. Minden új, az üzenet elrejtését célzó módszer már meglévő eszközöket és ötleteket használt fel, és alapvetően nem sokat változott. Az igazi újdonság az volt, amikor a régi módszer alapötletét ötvözték egy újjal: az üzenetet változó vonalak, színek és egyéb képi elemek felhasználásával kódolták.

11.3. A SZTEGANOGRÁFIA MA

Az elektronika, a digitális technika fejlődése új utat nyitott a szteganográfia számára, és ma is aktív kutatási területnek számít, főként a szerzői jogvédelem és a rejtett csatornák alkalmazásának és felderítésének területén. Talán már az eddigiekből is nyilvánvaló, hogy a módszer lényege nagyjából az, hogy nagy tömegű „lényegtelen" információ között rejtjük el a védeni kívánt információt.

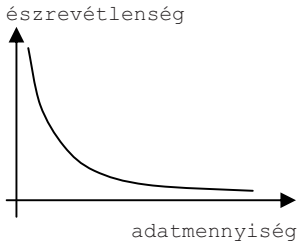
11.3.1. A szteganográfia alapelvei

Terminológia

A továbbiakban azt, amibe beletesszük az információt **hordozónak** vagy fedő információnak (*cover, cover-text, cover-image*) nevezzük, amit beleteszünk, azt egyszerűen **üzenetnek** (*plaintext, embedded data*). Amit eredményül kapunk, az a *stegotext* vagy *stegoimage* (néhány irodalomban, például [51]-ben: *host signal*). Hogy az üzenet nyílt szöveg vagy pedig rejtjelezett, az eljárás szempontjából közömbös. A „digitális szteganográfia" legtöbbször a hordozó egyes bitjeinek, bitcsoportjainak a megváltoztatását jelenti, így mielőtt elkezdenénk, egy feltételt el kell fogadnunk: *csak olyan adat lehet hordozó, melynek* ***értelmezésében*** *nem okoz zavart, ha leírásához használt bájtok egyes bitjeit, bitcsoportjait, (például, de nem kizárólag a legkisebb helyértékű bitjeit) megváltoztatjuk. Az adat* ***feldolgozásának*** *eredményében a megváltozott bitek általában egyfajta zajként jelennek meg: minél több bitet változtatunk meg, annál erősebben*. Egy jó adatrejtő rendszerről is fel lehet tételezni, hogy a támadó – hasonlóan a titkosító rendszerekhez – a rendszer minden elemét ismeri, kivéve a feldolgozást esetleg vezérlő titkos kulcsot. A továbbiakban ezt a tulajdonságot nem veszzük figyelembe, de egy gyakorlati megvalósítás során erre emlékezzünk!

Célok

Alapvetően két fő irányt különbözetünk meg az adatrejtési technikákban. Az egyik esetben nem kritikus a „rejtettség”, viszont fontos, hogy az elhelyezett adat robosztusan, lehetőség szerint eltávolíthatatlanul kerüljön be a hordozóba. Ki kell állnia azt is, ha a stegotextet az általánosságban használt transzformációknak vetik alá. A másik esetben nem annyira fontos a robosztusság, viszont garantálni kell a rejtettséget, az észrevétlen továbbítás lehetőségét. A gyakorlatban ezek mellé még egy jellemző társul: az elrejtendő adat mennyisége. Ahol az eltávolíthatatlanság a fő követelmény, ott általában kevés adatot kell elhelyezni (*vízjelek*), ahol pedig a rejtettség, ott viszonylag sokat (*átvitel*). Némi ellentmondás, hogy minél több adatot kell elrejteni, annál kevésbé valósítható meg a rejtettség. Az egyensúly az adott feladattól és alkalmazástól függ.



Egyéb értékelhető tulajdonságok:

1. Mennyi az elrejthető adatmennyiség (*payload*)
2. Mennyire észrevehetetlen az elrejtett információ?
3. Mennyire megbízható az alkalmazott technika? (*reliability*)
4. Mennyire biztonságos az alkalmazott technika? (*security*)
5. Az elrejtett adat mennyire képes túlélni a behatásokat? (*survivability*)
6. Milyen hatással van a hordozóra az elrejtett adat? (Változik-e például a tömöríthetősége vagy más statisztikai jellemzője?)

Támadásfajták

Az adatrejtés ellen irányuló támadásoknak a kommunikációs csatornák elleni támadásoknál már megismert két fő módszere van:

- az adatrejtés elleni *passzív támadásnak* azt nevezzük, amikor a támadó felfedi az adatrejtés tényét és esetleg az elrejtett adatot is megismeri, de nem változtatja meg vagy nem tudja megváltoztatni azt.
- az adatrejtés elleni *aktív támadásnak* azt nevezzük, amikor a támadó az elrejtett adatot megváltoztatni vagy eltávolítani is képes, függetlenül attól, hogy megismeri-e annak tartalmát, vagy sem.

A két támadásfajta közül az aktív a gyakoribb, sőt vegyük figyelembe azt, hogy a stegotext átvitele során is keletkezhetnek olyan sérülések, amelyek nem szándékos manipuláció eredményei. Az átvitelre egyébként a következő lehetőségek állnak rendelkezésünkre:

<table>
<tr><td rowspan="2">Forrásállomás</td><td>Digitális csatorna
❑ Hálózati csatlakozások
❑ ISDN vonalak
❑ Adathordozók</td><td rowspan="2">Célállomás</td></tr>
<tr><td>Analóg csatorna
❑ Telefonvonalak
❑ nyomtatás
❑ on-air (rádiós) továbbítás</td></tr>
</table>

Az átviteli csatornák is modellezhetik azokat a támadásokat, amelyeket egy stegotextnek el kell viselnie, vagy jó esetben elvisel. Egyébként sem nyújt minden adatrejtési módszer megfelelő biztonságot, de nem is mindegyiknél ez a fő szempont. A következő (csak példaértékű) felsorolás felsorolás a főbb lehetséges behatásokat vagy támadásokat mutatja:

- Veszteséges tömörítés (kép- vagy hangalapú hordozó esetén).
- Szűrések (kép- vagy hangalapú hordozó, de egy analóg átviteli csatornára is jellemző a sávkorlátozás, ami egyidejű alul- és felülátereszto szűrést jelent).
- Editálás (részek kivágása, áthelyezése, átméretezés – hangalapú hordozó esetén erősítés vagy csillapítás stb.).
- Újabb adatrejtési eljárásban a stegotext hordozóként szerepel.
- Járulékos zaj (analóg átviteli csatorna fő jellemzője).
- A/D – D/A átalakítók (az analóg – digitális világok határainak átlépésekor további problémát okoznak a különböző gyakoriságú mintavételezések).
- Nyomtatás, fénymásolás, szkennelés (szöveg- vagy képalapú hordozó esetén).

11.3.2. Példa 1: Teljes spektrumú adás

Alkalmazása ott lehetséges, ahol az adathordozó valamilyen frekvenciájú jel, ez lehet elektromágneses jel (rádiókapcsolatok), de lehet akár a fény is. Osszuk fel a rendelkezésre álló frekvencia-tartományt különböző csatornákra! Az adó és a vevő a kommunikáció során valamilyen módon folyamatosan váltogatja az éppen használt csatornát. A következő csatorna kiválasztása kötődhet valamilyen algoritmushoz, vagy minden üzenetcsomag tartalmazhatja a következő azonosítóját. Az adó az összes többi, pillanatnyilag nem használt csatornán valamilyen zajszerű jelet sugároz. Ha valaki egy ilyen összeköttetést le akar hallgatni, először is olyan sávszélességű vevőre van szüksége, ami a teljes spektrumot képes fogadni. Ehhez egy spektrum-analizátort kell csatlakoztatnia, ami normális esetben megmutatná, hogy éppen melyik frekvenciasávot használják a felek. Azonban a nem használt csatornák is jellel vannak terhelve, így az analizátor képernyője minden frekvenciatartományban mutat valamit. Egyszerű eszközökkel tehát nem lehet eldönteni, hogy melyik csatornán folyik kommunikáció és melyiken nem: a jeltől nem lehet megkülönböztetni a zajt. Ennél egyszerűbb csatornák – például telefonvonalak, rádióadás – is hasonló tulajdonságokkal bírnak: mindig tartalmaznak valamekkora zajt, bár jó esetben a hasznos jelhez viszonyítva keveset. Tekintettel arra, hogy az elrejtett adat valamilyen zajt visz a hordozóba, nem árt, sőt kifejezetten hasznos, ha a hordozó már maga is tartalmaz valamekkora zajt. Ez a zaj részben vagy egészében kicserélhető az elrejtendő információra, és megfelelő eszközökkel elérhető, hogy ez az új zaj megkülönböztethetetlen legyen az eredetitől (*adatrejtés zajba*).

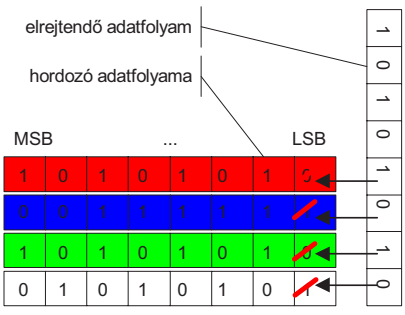
11.3.3. Példa 2: Image steganography - adatrejtés képbe

A mai digitális adattengerben igen sok lehetőség van arra, hogy elrejtsünk valamit valamiben: az ISDN csatornán folyó telefonbeszélgetéstől és adatforgalmaktól kezdve a különböző hang, kép- és videóformátumokig, vagy szinte bárhol, ahol digitális adatok közlekednek. Igazán azok a digitális csatornák alkalmasak, amelyek valamilyen emberi információt továbbíta-

nak: hangot vagy képet, mert ezek – természetükből adódóan – tartalmaznak bizonyos redundanciát, látszólag felesleges adatot. Az ilyen „hordozójelölteket” digitalizálással készítik, ennek lényege, hogy a továbbítandó hangot, fényerőt – vagy egyéb mérhető jelet – úgy alakítunk át, hogy elektromos úton mérhető legyen. Ezután bizonyos időközönként mintát veszünk a jelből és az abban a pillanatban mért értéket digitális eszközökkel valahány biten ábrázoljuk (általában 8-24 biten). És itt máris adódik egy lehetőségünk az adatrejtésre! Ha ugyanis a számábrázolás a kettes számrendszer szabályainak megfelelően történik, a legkisebb helyértékű bitek esetleges megváltozása nem lesz jelentős hatással a rögzített jel rekonstruálására. Gyakorlatilag egy kissé pontatlan mérésnek fog tűnni. Természetesen nem szabad túl sok bitet módosítani, mert a változás előbb-utóbb akkora lesz, hogy az már zavaró és észrevehető. Teljesen azonos elvek szerint helyezhetünk el információt digitalizált fényképekben is, jóllehet azok készítésének folyamata kicsit más. A szteganográfia alkalmazásában azonban mindegy, hogy az adat honnan származik, csak az adatok felépítéséről kell sok mindent tudnunk annak érdekében, hogy az eredeti információ minél kisebb sérülést szenvedjen. Az adatokat általában valamilyen feldolgozás után küldik tovább, ekkor még a feldolgozás előtt el kell rejtenünk az információt. Gondoljunk egy hangfelvételre: sok a szünet, a hosszabb-rövidebb csönd, vagy ha ránézünk egy fényképre igen sok, viszonylag homogén területet találhatunk rajta. A továbbításra alkalmas formátumok túlnyomó többsége eltünteti ezt a redundanciát (tehát tömörít) és olyan formába alakítja a digitalizálás közvetlen eredményeit, amelyek már emberi szemmel, füllel nem értelmezhetőek, és többé-kevésbé érzékenyek az adatsérülésre is. A formátumtól függően elképzelhető, hogy még ekkor is van lehetőség adatmanipulációra, kihasználva az adott formátum specialitásait (például kitöltőrészek, meta-információk, extension tags), bár egy formátumváltásnál sajnos ezek elvesznek.

LSB módszer

A korábbi feltételeknek jellemzően megfelel egy tömörítetlen, true-color BMP formátumú kép. Ez a fájlformátum a képet úgy tárolja, hogy minden egyes mintavétel és mérés eredménye tömörítés vagy más varázslat nélkül kerül az állományba. Gyakorlatilag a mérési eredmények szekvenciális halmaza. Ez a formátum megengedi a pixelek színmintáinak kismértékű megváltoztatását, amit az emberi szem csupán zajként érzékel. A változtatások a BMP formátumot nem teszik tönkre. Ha kevesebb bitet használunk fel a mintákból, a zaj kevésbé lesz észrevehető, ha többet, a zaj erősebb lesz. Általában nem alkalmasak hordozónak a szövegfájlok, alfanumerikus adatokat tartalmazó táblák, vektorizált térképek, tömörített vagy bináris állományok. A fentiek figyelembevételével tekintsünk példaként egy true-color 1024×768 pixel méretű bitképet egyelőre tömörítés, fejléc és más formátum információ nélkül:

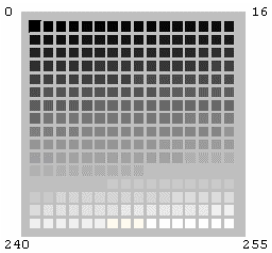


- Minden képpont leírására 3 bájtot használunk, ezek rendre a vörös, zöld, kék színösszetevőket tartalmazzák.
- A kép mérete ekkor: 1024×768×3×8 = 18 874 368 bit. (18Mbit, 2304Kb)

- Ha minden képpontból „ellopjuk” a legalsó bitet, és helyére az üzenetet írjuk, a kép alapvetően nem változik meg, a színek megváltozását emberi szemmel nem lehet észrevenni.
- Egy bit felhasználásával 1024×768×3×1 = 2 359 296 bit (2.25Mbit, 288Kb) adatot tudunk elraktározni a hordozó képbe, annak 12,5%-át felhasználva.
- Lehetőség van arra is, hogy ne 1, hanem 2, 3 vagy 4 bitet „csípjünk” le. Ekkor rendre 576Kb (25%), 864Kb (37,5%) vagy 1152Kb (50%) kapacitást biztosít a hordozó. Minél több bitet használunk el, annál nagyobb lesz a színek torzulása, amit előbb vagy utóbb már emberi szemmel is észre lehet venni. Ezért a 3-4 bit felhasználásának eredményét mindenképpen ellenőrizzük! Figyelhetünk arra is, hogy az emberi szem a legjobban a zöld fényt látja, legkevésbé a kéket.
- Ha az üzenet hosszabb, mint a hordozóból felhasználható méret, akkor vagy másik hordozó képet választunk, vagy addig nagyítjuk, amíg az üzenet végül elfér benne. Vagy egyszerűen feldaraboljuk az üzenetet több részre.

Az ilyen nagyfelbontású, RGB színsémát alkalmazó, tömörítetlen képek egyik legnagyobb baja, hogy hatalmasak. A példában szereplő kép fájlként legalább 2Mbájt, de még egy átlagosnak mondható 640×480×3 tulajdonságokkal rendelkező kép is 900Kbájt. Ha ilyen méretű állományt továbbítani akarunk, *tömörítenünk* kell. Ha magát a manipulált képfájlt, mint bináris állományt veszteségmentesen tömörítjük, elküldjük és a túlsó oldalon pedig kicsomagolják, semmi problémánk nincsen. Ha azonban a képet már eleve valamilyen tömörített formátumban akarjuk tárolni, nem mindegy, hogy milyen formátumot választunk. Egy a lényeges, hogy ne veszteséges (például *JPEG*) formátumot válasszunk, mert sajnos szembe kell néznünk az ilyen tömörítők sajátosságával: adatveszteség lép fel, ráadásul pont a legkisebb bitek helyén. Így tömörítéskor pont az az információ vész el, amit az előbb tuszkoltunk bele a hordozóba. Ezzel szemben nyugodtan használhatunk minden más olyan formátumot, amely ismeri a 24 bites színmélységet, emellett nem okoz adatveszteséget (*TIFF, PNG, JPEG2000*).

Főleg az internetes alkalmazásokhoz szívesen használják a 256 szín kezelésére képes *GIF* formátumot. A formátum jó tömörítést biztosít a kisebb képekhez, és viszonylag egyszerű a kódolása is. A gond csak az, hogy itt egy pixelérték csak közvetetten jelenti a pont színét: csak egy mutató a palettába, ahol végül majd eldől a pont színe. Ez nem a *GIF* sajátossága, szinte minden 256 színű képformátumnál így van. Tehát nem lehetünk biztosak benne, hogy ha megváltoztatunk egy 11110000_2 értékű pixelt mondjuk 11110001_2-re, akkor senki semmit nem fog észre venni, mert lehet, hogy az első érték fekete színre, a második érték pedig fehérre mutatott. Csak akkor alkalmazhatjuk a már vázolt módszerünket, ha a palettában az egymást követő színek hasonló RGB értékekkel rendelkeznek, vagy addig cserélgetjük a pixelértékeket és a hozzájuk tartozó palettabejegyzéseket, amíg hasonlóak nem lesznek. Átmeneti megoldást jelenthet a palettaalapú színábrázolásokban a szürkeskálás képek (*gray-scale*) alkalmazása, ahol mindössze egy bájt ír le egy pixelt, de egy-két bitet már itt is fel tudunk használni. Ezeknél a képeknél szinte kizárólag lineáris megfeleltetés van a 0-255 pixelértékek és a fekete-fehér átmenet között.



A veszteséges formátumok, mint például a JPG, tehát gondot okozhatnak, mert hiába kódoljuk megfelelően a hordozót, ha azt utána konvertáljuk JPG formátumba, sajnos pont az imént elhelyezett adatok fognak elveszni. Itt igazából nincs más lehetőség, csak az, hogy nem a hordozót alakítjuk át JPG-re, hanem a kész JPG-et használjuk fel hordozónak. A legegyszerűbb, ha a formátum fejlécében lévő nem kötött mezőket használjuk fel. A JPG típusú kódolások esetében további lehetőség, hogy figyelembe vesszük azt a tényt, hogy a tárolt képet valamilyen *DCT*, *FFT*[53] eljárásnak vetik alá, az így keletkező együtthatók pedig nem egyformán fontosak: a legmagasabb frekvenciájú tagok együtthatói kis mértékben megváltoztathatók anélkül, hogy az eredeti információ láthatóan (vagy hallhatóan) sérülne. Ezt használják ki jó hatásfokú tömörítés érdekében is: a kép(hang)minőség rovására a magasabb frekvenciájú tagok együtthatóit kinullázzák. Minél több zérusérték van egy adatblokkban, az annál redundánsabb és annál jobban tömöríthető. Ha az együtthatókat akarjuk információtárolásra használni, olyan saját JPG-kódolót kell használnunk, ami megengedi, hogy a DCT és a tömörítés között manipuláljunk az adatokkal: a két művelet között a magasabb frekvenciájú jelek együtthatóit céljainkra felhasználhatjuk. A tömörítés hatékonysága természetesen erősen romlik, de az adataink nem sérülnek meg [URL35].

A képek területén a BMP olyan, mint a hangfelvételek esetében a PCM formátumú hangfájl. Ez a fájlformátum a digitalizált hangot szintén a mérési eredmények időrendes, szekvenciális halmazaként tárolja. (Lényegében ugyanilyen az audió CD formátuma is, csak ott még különböző hibajavító kódok és technikák is szolgálják a felhasználókat.) Ez a „nyers" formátum megengedi a hangminták legkisebb helyértékű bitjeinek megváltoztatását, amit az emberi fül csupán zajként érzékel. A változtatások a PCM formátumot nem teszik tönkre. Ha kevesebb bitet használunk fel a mintákból, a zaj kevésbé lesz észrevehető, ha többet, a zaj erősebb lesz. A PCM kódolású hangfájlok meglepően sok kapacitást képesek biztosítani. Ha az alsó négy vagy akár nyolc(!) bitet felhasználjuk, a WAV fájl méretének 25%-a illetve 50%-a áll rendelkezésre, ami már több megabájt adat továbbítását is lehetővé teszi, például egy audió CD-n. Ha sok bitet használunk el, a hordozó lehetőleg nagy dinamikájú zeneszám legyen kevés szünettel vagy csönddel.

Jóllehet az LSB módszer viszonylag sok adat átvitelét teszi lehetővé (*high payload*), sajnos a módszer – ismertsége miatt – önmagában már nem alkalmas biztonságos adatrejtésre.

11.3.4. A kivétel erősíti a szabályt: szöveges állományok

Sorkiegészítés – white space encoding

Az alapelvek tisztázásakor elfogadtuk, hogy csak olyan adat alkalmas hordozónak, amelyik elviseli, ha egyes bitjeit megváltoztatjuk. Ennek szellemében tekintettük át az előbb a képek és hangok feldolgozásának egy lehetséges módszerét. Azonban más szempont szerint is rejthetünk el információt. Ha azt mondjuk, hogy az alkalmas hordozónak, ami elviseli, hogy az adatok közé pótlólagos információt szúrunk be, akkor lehet, hogy egy szöveges állomány is

[53] Diszkrét Cosinus Transzformáció, Gyors Fourier Transzformáció

alkalmas adatrejtésre. Írjuk fel a rejteni kívánt adatokat egyetlen bitfolyamként! Hordozónak válasszunk egy jó hosszú tördelt szöveget, például egy novellát vagy egy kisregényt! Végezzük el a feldolgozást a következőképpen:

- A hordozó szövegét soronként olvassuk be.
- Ha a sor végén egy vagy több szóköz karakter található, töröljük ki mindet onnan.
- Ezután vegyük a rejteni kívánt bitfolyam soron következő – mondjuk – három bitjét, mint bináris számot.
- Ennek megfelelő mennyiségű szóközt szúrjunk a sor végére: ha például a bitek a $011_2=3_{10}$ számot adják ki, 3 darab szóközt adunk a sorhoz.
- Ezután visszaírhatjuk a sort a merevlemezre.

Minél hosszabb bitcsoportokat használunk, minél több szóközt szúrunk a sor végére, annál nagyobb a „lebukás” veszélye, de annál több információ tárolható egy adott szövegben. Minél kevesebbet, annál nagyobb a biztonság, de ezért cserébe kevesebb a tárolásra használható hely. Számszerű példa: az A4 méretű oldalon átlag 50 sor van, ez az iménti 3 bitet feltételezve 50×3=150 bit=19 bájt elrejtését teszi lehetővé oldalanként. Ez nem sok, de néhány módszer még ennyit sem képes elrejteni. Igaz, ezek másban jeleskednek, például az elrejtett információ nem változtatja meg a hordozó statisztikai jellemzőit, vagy éppen nagyon sok mindent képesek átvészelni.

Leírónyelv-manipuláció

Más jellegű átalakításokkal is érhetünk el eredményeket. Ha megnézzük az olyan szöveges fájlokat, mint a `HTML` vagy az `RTF` formátum[54], egy kis ötlettel ott is rejthetünk el információt úgy, hogy a leírt dokumentum **megjelenítésekor** semmit nem veszünk észre. És nemcsak arról van szó, hogy nem vesszük észre a változásokat, hanem arról, hogy a megjelenített vagy kinyomtatott dokumentum, az eredetivel pontosan egyező lesz! Ugyanis mindkét formátumnak az a közös jellemzője, hogy olyan leírónyelvet használnak, amely angol szavakon vagy azok rövidítésein alapul, és a dokumentum fizikailag tiszta, hétbites `ASCII` formátumban leírható. Mindekét nyelvben egyértelműen elkülöníthetők a formázáshoz használt kulcsszavak (*control-tags, format-tags*) a dokumentum szövegétől vagy egyéb objektumától: a `HTML`-ben „<” és „>” jelek, a `RTF`-ben „{” és „}” jelek határolják őket. Egyik formátum sem érzékeny arra, hogy ezek a határoló egységek kis- vagy nagybetűvel vagy éppen vegyesen vannak írva. Ezért akárhogy írhatjuk őket: ahol egy „1”-es bitet akarunk jelezni oda nagybetűt írunk, ahol „0” a kérdéses bit, oda kisbetűt. Így például a „<HTmL>” tag egy 1101_2 bitsorozatot képviselhet. Sajnos nem egészen ilyen egyszerű a helyzet, mert a `HTML`-ben Javascript is lehet, ami viszont érzékeny a kis és nagybetűk közötti különbségre. Hasonlóan nem írhatjuk át az „<img>”, a „<form>”, a „<link>” és „<a>” tagok fájlhivatkozásait sem, mert az `UNIX`-os szerver esetén katasztrofális lehet. A korlátokat figyelembe véve viszont szabadon garázdálkodhatunk.

[54] *Hypertext markup-language, Rich Text Format*

Ismereteim szerint `RTF`-ben hasonló korlátozások nincsenek, csak a fájl első hat karakterének kell „`{rtf1}`”-nek lennie (kisbetűvel). Az eljárás során a hordozó mérete nem változik meg. Sajnos az ilyen és hasonló módszerek csak elvi jelentőssséggel bírnak, tekintettel arra, hogy a forrásfájl megtekintése (HTML) vagy szerkesztése gyakori művelet lehet.

Sajnos mindkét bemutatott szöveges módszernek (sorkiegészítés és leírónyelv manipuláció) van egy rossz tulajdonsága: a kinyomtatott dokumentumból biztosan nem állítható elő az elrejtett adat, így az kizárólag elektronikus úton terjeszthető.

11.3.5. A jelek és zajok viszonya

Miután kiválasztottuk a képformátum színmélységét (például 3×8 bit) vagy a hangfájl „hangmélységét” (például 16 bit), el kell döntenünk – esetleg próbálgatással – hány bitet használunk fel az üzenet tárolására a hordozóból. Ha minden mintából 2 bitet használunk el, a legrosszabb esetben a képet –36dB, míg a hangot –84dB zaj terheli.

$$Q_z = 20 * \lg \frac{4}{256} < -36\text{dB} \qquad Q_z = 20 * \lg \frac{4}{65536} < -84\text{dB}$$

Ez jóval kevesebb annál, amit észre lehetne venni. Az adat épségben való megérkezésének annál nagyobb a valószínűsége, minél nagyobb a fentebb számított Qz értéke, viszont ez az üzenet könnyebb észrevehetőségét is jelentheti. Az ideális egyensúly megtalálása nem mindig egyszerű feladat. Vegyük figyelembe azt is, hogy az LSB módszer alkalmazásakor átlagosan a felhasznált bitek felét kell megváltoztatni. Kép esetén célszerű „láthatóan” digitalizált fényképet használni, így ha valaki észre is veszi a színátmenetek ingadozását vagy a kép zajosságát, hamarabb fogja a szkenner optikáját vagy CCD-jét hibáztatni, minthogy feltételezne valami turpisságot. Ha nagyon részletdús képet használunk, a zaj el fog bújni a részletekben. Amennyiben színes képet használunk, ugyanekkora zaj fogja terhelni képünk mindhárom színcsatornáját, azonban az emberi szem sokkal inkább a fényerővváltozásra (szürke skála), mint a színek változására érzékeny, így látszólag jobb eredményt érhetünk el. Általánosságban is felírható, hogy ha egy hordozó egy eleme *K* bitet foglal el és ebből *S* bitet szeretnénk felhasználni, ezzel

$$Q_z[\text{dB}] = (S - K) * 20 * \lg 2$$

zaj terheli a hordozót. A tapasztalat szerint a színcsatornánkénti 8 bitből 4-4 nyugodtan felhasználható, ha a hordozó kép elég részletdús, és kevés benne a homogén terület. Ilyenek például a tájképek.

> Az itt bemutatottnál mélyebb kapcsolat is kialakítható a szteganográfia és a kommunikációelmélet között. Ehhez csak azt kell elfogadni, hogy az elrejtendő adat az információtartalommal bíró jel, a hordozó pedig nem más, mint az a csatorna, amelyen a jelet továbbítani fogjuk. Például a kép, amelybe elrejtünk némi adatot, nem más, mint az a csatorna, amely az adat továbbítására szolgál. Az üzenet elhelyezése megváltoztatja a csatorna statisztikai jellemzőit is. E szteganográf csatorna kapacitását az az információmennyiség jellemzi, amely felfedezés nélkül sikeresen átvihető, és sikeresen elő is állítható a vételi oldalon.

A különböző hangformátumokra mindaz igaz, amit eddig elmondtunk a képekről. A legideálisabb formátum a PCM WAV, míg a JPG képformátumhoz a zene területén az MP3 formátum hasonlít. Egy négyperces – átlagos hosszúságú – sztereó WAV 44,1 kHz mintavételezéssel,

16 bites mintákkal 4×60×2×44100 = 21 168 000 mintát = 42 336 000 bájtot jelent. Ha itt elhasználunk 4 bitet – a bátrabbak nyugodtan próbálkozhatnak akár 8 bittel is –, akkor 21 168 000 × 4 = 84 672 000 bit, azaz megközelítőleg 10 Mbájt információt tudunk elrejteni.

A 16 bites mintavételezésű analóg CD-k elméletileg olyan elő- és végerősítőket (általában elemeket) követelnének meg, amiknek a jel/zaj viszonya jobb, mint 96 dB. Azonban ezt a minőségi követelményt az erősítők általában nem teljesítik, emiatt a 16 bites mintákat „túl-mintavételezésnek" is tekinthetjük. Ha a minták valódi 16 bitesek (tehát az LSB bitjük is valódi információt hordoz, nemcsak zajt), a visszajátszás minősége szinte kizárólag a visszajátszó eszközök minőségétől függ.

$$Q_z = 20 * \lg \frac{1}{2^{16}} \approx -96{,}3\text{dB}$$

Hasonló igaz a 16 bites WAV fájlokra is. Most használjuk fel adatrejtésre az alsó 4 bitet! Ekkor –72 dB zajt teszünk az audió információ közé.

$$Q_z = (4-16) * 20 * \lg 2 \approx -72{,}2\text{dB}$$

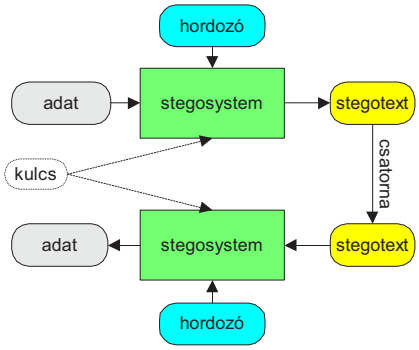
Tekintve, hogy egy átlagos végfok nem garantál 60-80dB-nél jobb jel/zaj viszonyt, a mi –72 dB-es zajunk el fog veszni az audiojelet erősítő, esetleg formáló analóg egységek termikus zajában. Ez azt is jelenti, hogy egy audió CD felét vagy negyedét adatrejtésre is használhatjuk, miközben kedvenc számainkat is meghallgathatjuk.

Mindezek miatt a jó adatrejtési alkalmazás szó szerint a legkisebb részletekre is figyel, és a hordozó minden tulajdonságát igyekszik megtartani. A manipulált hordozó továbbításakor szintén valamilyen zaj terhelheti a kommunikációs csatornát, ezért a kódolás és a dekódolás során szükséges különböző (ön)szinkronizáló és hibajavító eljárások használata, bár ez általában a digitális átvitelt biztosító csatornák és protokollok feladata, nem pedig az őket használó alkalmazásoké (*réteg modell – layer model*).

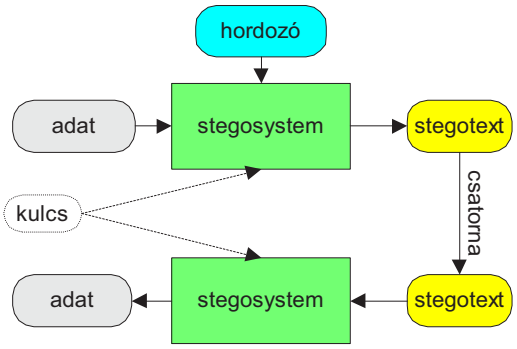
11.4. SZIMMETRIKUS ÉS ASZIMMETRIKUS ADATREJTÉS

Az eddig bemutatott szteganográf eljárásoknál nincs szükség az eredeti hordozóra, mert anélkül is dekódolható az elrejtett adat, így a hordozót nem kell továbbítani, vagy előzetesen egyeztetni. Ezeket az eljárásokat *vak eljárásoknak* nevezzük (*blindly methods*), és a jelenleg használt technikák döntő többsége ilyen.

Egy egyszerű példa a másik megoldásra: az elrejtendő adatbiteket egy képben úgy helyezzük el, hogy az egyes biteket valamilyen szabályszerűség szerint az eredeti kép fölé „terítjük", majd az eredeti pixelértékekhez hozzáadjuk (valamelyik színcsatornához, de akár mindháromhoz egyszerre). A létrejövő színingadozás senkinek sem fog feltűnni. Az elrejtett információ visszaállításakor az eredeti hordozót az stegotext képből ki kell vonni, az így létrejövő különbözeti kép maga az elrejtett adat (a módszert különbségi rejtésnek is hívják). Az ilyen eljárásokat, ahol szükség van az eredeti hordozóra, *letételjárásoknak* nevezzük (*cover escrow*).



Cover escrow rendszer

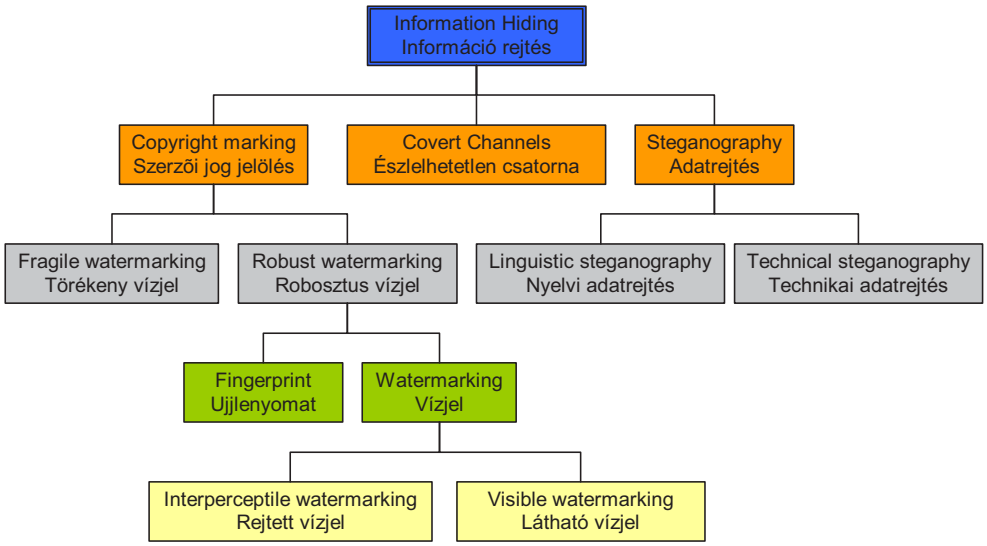


Blind eljárás

Jóllehet ez utóbbi eljárások biztonságosabbak, a gyakorlati megvalósítás során jelentős problémát okozhat az eredeti hordozó biztonságos eljuttatása a címzetthez, vagyis ugyanazok a problémák jelentkeznek ismét, amelyek a szimmetrikus titkosító algoritmusok kulcscsere problémáját is jellemezték. Ezért a letételjárásokat elsősorban látható és láthatatlan vízjelekhez, és nem üzenetváltásra alkalmazzák.

11.5. ALKALMAZÁSI TERÜLETEK

Az alábbi ábra az adatrejtési technikák csoportosítását mutatja be, a felhasználási terület és a megvalósítandó célok figyelembevételével. A következőkben mi is ilyen szempontok szerint ejtünk néhány szót az egyes megoldási lehetőségekről.



11.5.1. Copyright watermarking

A szteganográfia napjaink leggyakoribb felhasználása az elektronikus úton terjesztett publikációk védelme, az elektronikus vízjel- vagy ujjlenyomat alkalmazása. A grafikával foglalkozók egy-egy elkészült képükbe például az *Adobe Photoshop* használatával tudnak egyedi azonosítót tenni. De hasonlóan lehetne minden audió CD sávba is copyright információt rejteni. Akár képről, akár hangról van szó, a módszer biztosítja, hogy az azonosító minden másolatban benne lesz, onnan gyakorlatilag eltávolíthatatlan. Ezt a folyamatot vízjelzésnek nevezzük (*watermarking*), és szinte kizárólag a digitális multimédia területén használják. A folyamat során valamilyen adatot (vízjelet) ágyazunk a megjelölni kívánt hordozóba, amely később a hordozóból kinyerhető, vagy legalább detektálható. A vízjel lehet látható (*visible*) vagy láthatatlan (*invisible*), de ezek a fogalmak nem korlátozódnak a képi hordozókra. Talán helyesebb lenne az *észlelhető* vagy *észlelhetetlen* jelzők használata, de a szakirodalom ebben nem következetes, és általában a *visible/invisible* jelzőket használja. Akárhogyan is nevezzük, a láthatatlan vízjel nagy előnye, hogy a hordozóban elfoglalt helye ismeretlen. Legfeljebb tudjuk, hogy ott van, de nem tudjuk, hol.

Ha copyright jelzésként nem szöveget vagy számot helyezünk el, hanem képet, emblémát vagy hangot (tehát olyan üzeneteket, melyek egyébként is redundánsak), valószínű, hogy még különböző szűrések és „belerajzolások" után is megmarad a készítő eredeti(hez hasonló) azonosítója[55]. Érdemes megemlíteni azt is, hogy ha a hamisító saját védjegyet tesz a „művére" (*re-watermarking*), az nem fogja az eredetit felülírni, hanem mindkettő megmarad. Az előzőekből az is következik, hogy az *„electronic watermark"* a hordozó integritását is ellenőrizheti, hiszen ha a vízjel visszaállítása nem pontosan azt adja vissza, ami eredetileg bele lett rakva, akkor a *stegotext* megváltozott.

Az ilyen jellegű védelmek másik fajtáját, az elektronikus ujjlenyomatot (*fingerprint*) másra használják. Ennek feladata egy-egy másolat útjának nyomon követése, ugyanis az ujjlenyomat egy olyan egyedi azonosító, melyet minden egyes jogszerű másolatban elhelyeznek. Ha később egy illegális másolat felbukkan, a szerzői jog tulajdonosa az ujjlenyomattal azonosíthatja az eredeti másolat tulajdonosát[56] (*traitor-tracing, áruló-követés*). A *watermark*-technikák között tehát két fő feladat oszlik meg:

Fragile watermarking *„törékeny vízjel"*	**Robust watermarking** *„strapabíró vízjel"*
Feladata, hogy biztosítsa a legkisebb módosítás észrevételét is, felfedve a hamisításokat, *változtatásokat*.	Feladata, hogy túlélje a módosításokat, a lehető legtovább megmaradva bizonyítsa a megjelölt hordozó *eredetét*.

> Néhány olvasó, aki a könyvet megjelenése előtt olvasta, feltette azt a gyakorlatias kérdést, hogy a bankók vízjelét vagy a zárjegyeket hova sorolnám? Nos, a zárjegy egyértelműen „törékeny vízjel". A bankók vízjele robosztus vízjel és a papír eredetét igyekszik bizonyítani. Az egyéb ismérvek (UV-reagens szöszök, a nyomdai eljárás, az illeszkedő ábrák, a

[55] Ez különösen a látható vízjelekre igaz.
[56] Ha a vízjel a készítőt azonosítja, *source-based watermark*, ha a felhasználót, *destination-based watermark*. További, főként matematikai háttér található (különösen az árulókövetés problémájához) [76]-ban.

hologramcsík, stb.) pedig a törékeny vízjelekhez tartozhatnak. Általában a legtöbb elhelyezett védjegy (itt most nem a márkanevekre gondolok) törékeny – így akadályozva meg az újrahasznosítást – és egyúttal az eredetiséget is igazolják. Az egyértelmű megkülönböztetés vagy besorolás nehéz, mert legalább két fontos különbség van a fizikai formában megjelenő értékek és a fentebb tárgyalt technikák között:

1. Az elektronikus dokumentumoknál lényegtelen, hogy milyen fizikai formában tárolódnak. Egy bankónál viszont nagyon is fontos, hogy milyen papírból készült, a vízjel klasszikus feladata egyébként pont ez: a papír eredetének bizonyítása.
2. A bankóhamisítás során a különböző ismertetőjegyek elhelyezése a cél (bizonygatva, hogy a bankót az állam készítette), és nem azok eltávolítása vagy lecserélése, ami az eredet letagadását jelentené.

11.5.2. Covert channels

Az adatrejtés támogathatja az ún. észlelhetetlen csatorna (*hidden channel, covert channel*) megvalósítását, illetve biztonsági szempontból felveti ezek keresésének szükségességét. A kommunikáló felek az észlelhetetlen csatorna segítségével kiegészítő információkat küldhetnek egymásnak, például jelezheti a küldő a címzettnek, ha az adott dokumentumot egy kényszerítő harmadik fél hatására írta alá. A legfontosabb következménye a csatorna létének azonban az, hogy egy ilyen eljárást implementáló programozónak, hardvergyártónak lehetősége van arra, hogy az észlelhetetlen csatorna segítségével olyan információkat juttasson ki egy rendszerből, amelyeket nem lenne szabad (például egy kulcspár titkos kulcsát, jelszavakat vagy más érzékeny adatokat, dokumentumokat). Ezért az informatikai rendszerek biztonsági vizsgálatánál az egyik legfontosabb szempont az észlelhetetlen csatornák keresése és megszüntetése.

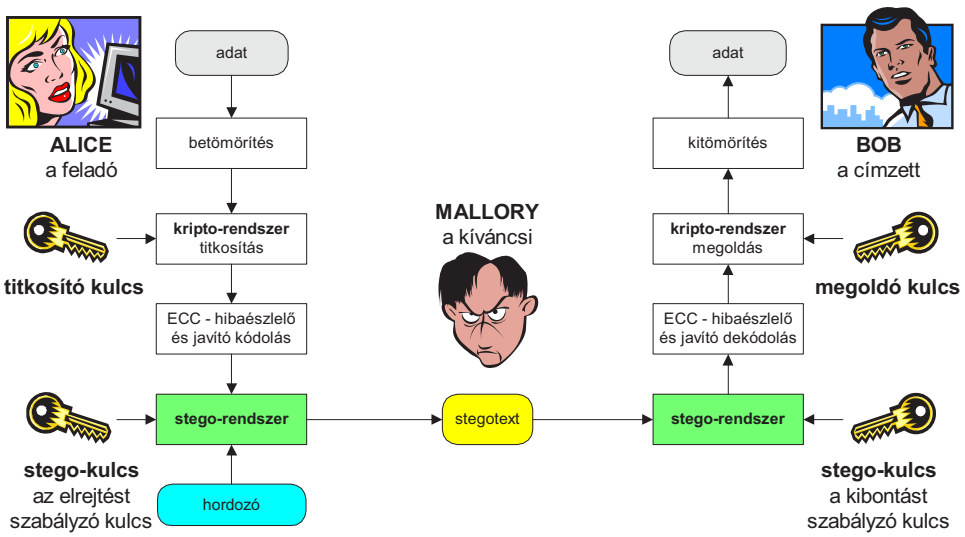
11.5.3. Steganography

Az összefoglaló ábra harmadik főcsoportja azokat a módszereket takarja, amelyekről a fejezet eddig is szólt, illetve a következő fejezet is ilyen célból foglalkozik a szteganográfiával. Ez nem más, mint a *rejtett, tömeges adatátvitel*, a kriptográfia alternatívájaként, vagy azt kiegészítő módszerként. Ne feledjük, az adatátviteli szteganográfia fő célja, hogy úgy rejtse el az üzenetet, hogy a lehetséges támadó észre se vegye az üzenetküldés tényét, csökkentve így annak az esélyét, hogy egyáltalán megpróbálja dekódolni és feltörni az üzenetet. Más szavakkal ez azt jelenti, hogy az üzenet elküldése nem provokálja az üzenet feltörését. Nagy különbség a watermark technikákhoz képest, hogy az adatátviteli adatrejtés során beágyazott üzenetnek és a hordozónak semmiféle logikai kapcsolata nincsen: a hordozónak mindegy, hogy mit ágyaznak bele és az elrejtett adat sem foglalkozik azzal, hogy miben fog utazni.

11.6. TITKOSÍTÓ MÓDSZEREK VS. SZTEGANOGRÁFIA

A szteganográfiai eljárásoknak rengeteg megvalósítása lehetséges. Az eljárás nemcsak digitális feldolgozásban használható, hanem egyéb médiákon is megvalósítható. A szteganográfiának helye van a titkosító rendszerek között, jóllehet azokat nem válthatja fel, de

megvalósíthatja azok célkitűzéseinek egy részét, illetve kiegészítheti azokat. Még vitatott, hogy a szteganográfia egyáltalán titkosító módszernek számít-e, vagy sem. Mindenesetre más az elvük, más a céljuk. Ez a kérdés nemcsak szakmai körökben vetődik fel napról napra, hanem egyes országok titkosítást szabályzó törvényeinek előkészítésénél is. A szteganográfiának nem az a célja (legalábbis nem az a fő célja), hogy az üzenet hozzáférhetetlen legyen illetéktelenek számára, hanem hogy azok észre se vegyék az üzenetküldés tényét. Előnye a kriptográf módszerekkel szemben, hogy a továbbított üzenetnek kisebb esélye van arra, hogy a kódtörő egyáltalán észreveszi. Bár ha egy feltételezetten titkosított kommunikációban a felek csak képeket küldözgetnek egymásnak, az elég gyanús lehet. Ilyenkor célszerű a különböző módszereket keverni, vagy a csatornán valóban titkosított, de tartalom nélküli üzeneteket is küldeni.



54. ábra Adatrejtéssel kiegészített titkosított kommunikáció

Ha a fenti ábra szerint titkosított üzenetet rejtünk el, kettős védelmet kapunk, mert ha a támadó észre is veszi, hogy az üzenetben van valami más is, és sikerül kikódolnia azt a hordozó környezetből, még mindig meg kell fejtenie az elrejtett üzenetet. Ez az elv általánosságban a titkosító módszerekre is igaz: ha kettő vagy több módszert együttesen alkalmazunk, egy jóval erősebb titkosításhoz jutunk, mintha valamelyik módszert (akár többször) önmagában használtuk volna. Az ilyen kódolókat nevezzük produkciós kódolónak (*lásd 18. ábra*).

A technikai bemutatás során nem esett róla szó, de az adatrejtésben is lehet kulcsokat alkalmazni, lehetővé téve, hogy az elrejtett adatot csak az találhassa meg, aki tudja, hol keresse. A titkosítás elvével ellentétben az adatrejtés folyamán a kulcsot nem arra kell használni, hogy az adatot megváltoztassuk, hanem az adatrejtés folyamatát szabályozzuk vele, így a kulcs az üzenet hordozóban elfoglalt helyét, a felhasznált bitek számát és egyéb jellemzőket határozhatja meg.

11.7. GYAKORLATI SZTEGANOGRÁFIA – LSB MÓDSZER

Az fejezet eddigi elméleti fejtegetései után, azok alapján nézzük meg a gyakorlati megvalósítást is. Két egyszerű formátum lesz a vizsgálódás tárgya, de az elv más formátumokon is hasonlóan alkalmazható.

11.7.1. Példa a WAV fájl feldolgozására

A WAV az egyik legrégibb hangtárolásra használt formátum, legalább annyi hátránya van, mint előnye. A tömörítés nélküli, PCM kódolású állományok a legegyszerűbb felépítésűek, de a formátum a tömörítés hiánya miatt zeneszámok archiválásra nem alkalmas. Viszont jól lehet használni olyan alkalmazásokban, ahol a lejátszásra csak igen kis erőforrás áll rendelkezésre, hiszen a lejátszás nem igényel más műveletet, mint a bitminták kiküldését egy D/A átalakítóra. A következő példa során egy WAV fájlba egy tetszőleges adatot vagy állományt teszünk be és veszünk ki onnan.

A WAV fájl formátuma

Egy WAV állomány az alábbi fejléccel kezdődik. Az egyes mezők közül csak azok jelentése van feltüntetve, amelyek a feladat szempontjából szükségesek.

Mezőnév	*Hossz*	*Tartalom*
RIFF	4	„RIFF”
rLen	4	RIFF blokk hossza
WAVE	4	„WAVE”
FMT_	4	„FMT_”
fLen	4	Formátumleíró rész hossza. Ez gyakorlatilag az eddigi 16 bájt.
Format	2	1 = PCM
Channels	2	1 = monó, 2 = sztereó
SamplesPerSec	2	Mintavételi frekvencia
AvgBytesPerSec	2	
BlockAlign	2	
Reserved1	2	
Reserved2	2	
BitsPerSample	2	Felbontás, hány bites minták?
DATA	4	„DATA”
dLen	4	Az ezután következő „DATA” blokk hossza.

A RIFF, WAV és FMT_ mezők tartalma most csak arra fog kelleni, hogy a fájlt „belülről” is azonosítani lehessen, ne kelljen kizárólag a kiterjesztésre hagyatkozni. A fejléc után a hangminták jönnek. A példában csak a 16 bites, PCM kódolású, tömörítetlen állományokat fogjuk feldolgozni. Ha a minták 16 bitesek, akkor az Intel-platformon megszokott formátumban tárolódnak a bájtok. Ha a hangminta sztereó, a bal és jobb csatorna felváltva van tárolva. A legösszetettebb a 16 bites, sztereó minták leírása, ezeket így találjuk a fejléc után:

```
Right-Low8, Right-High8, Left-Low8, Left-High8, stb.
```

A megvalósítás során célszerű a fenti fejlécmezőkkel egy rekordot deklarálni, így egy helyen lesz minden információ, ami a számításokhoz szükséges. Miután a fejléc adataiból megfelelőnek találtuk az állományt, a `dLen`, `BitsPerSample` mezőkből és a felhasználni kívánt bitek (a továbbiakban `uBits`) számából kiszámítható a WAV befogadóképessége:

```
Capacity = dLen*8/BitsPerSample * uBits / 8
```

Ebben a méretben el kell férnie az elrejtendő adatnak, és egy kiegészítő rekordnak is, amelyre két okból van szükség:

1. Az eredeti állomány adatait (név, méret, dátum, és a feldolgozás során használt bitek száma) el kell tárolnunk. A hordozó neve vagy más kiegészítő fájl nem alkalmas a feladat ellátására, mert a hordozó a feldolgozás után semmit nem árulhat el magáról, és a benne lévő adatokról.
2. Ha a kiegészítő rekord az iménti adatokon kívül egy egyedi karaktersorozatot is tartalmaz (mint például a RIFF mező „RIFF”, a WAVE mező „WAVE” tartalma), akkor a rekord visszaolvasása után eldönthető, hogy az érvényes-e egyáltalán, a WAV-ban van-e általunk értelmezhető adat?

A kiegészítő rekord ajánlott, minimális tartalma:

```
TInfoREC = packed record
             id    : array[0..6]of char; // 'STIDV10'
             Fsize : longint;            // fájl mérete
             Bits  : longint;            // a használt bitek
             FName : ShortString;        // a fájl neve
             chks  : longint;            // ellenőrzőösszeg
           end;
```

A rekord igény szerint egyéb jellemezőkkel is kiegészíthető (dátum, más attribútumok, pozíció a hordozóban, stb). Az `id` mező mindig maradjon a legelső mező, így a többi szabadon változtatható, bővíthető. Érdemes az `id` mezőbe egy verziójelzést is beletenni, és azt dekódoláskor figyelembe venni, így a rekord pontos értelmezésére a verziószám adhat útmutatást. A kiegészítő rekordot mindig ugyanannyi `uBits` felhasználással kell kódolni, mert a kiegészítés nem hordozhat saját magára vonatkozó adatokat. A hordozó maximális védelme miatt a rekord tárolásához célszerű mindig egy bitet felhasználni.

A WAV feldolgozása: adatfájl elhelyezése

- WAV fejléc felolvasása, ellenőrzések
- A kiegészítő rekord adatainak feltöltése
- Ciklus *i* 1-től rekordhosszig
  - Ciklus *j* 1-től 8-ig
    - WAV-ból 16 bites adat beolvasása, az alsó egy bit törlése
    - A rekord *i*. bájtjának 0. bitjének leválasztása
    - az előző két lépés között OR művelet
    - az eredmény kiírása a kimeneti fájlba
    - A rekord *i*. bájtjának eltolása jobbra
  - Ciklus vége *j*
- Ciklus vége *i*

- Az adatfájl bevitele a WAV-ba
- Ciklus *i* 1-től adatfájl hosszig
  - *d* = Adatbájt olvasása az adatfájlból
  - Ciklus *j* 1-től 8/*Ubits*-ig
    - WAV-ből 16 bites adat beolvasása, az alsó *Ubits* bit nullázása
    - A *d* alsó *Ubits* bitjének leválasztása
    - az előző két lépés között OR művelet
    - az eredmény kiírása a kimeneti fájlba
    - A *d* eltolása jobbra *Ubits* pozícióval
  - Ciklus vége *j*
- Ciklus vége *i*
- A WAV fájl maradékának átmásolása a kimeneti fájlba
- Feldolgozás vége

Adatfájl kivétele a stegotext WAV-ból

- WAV fejléc felolvasása, ellenőrzések
- Azonosító rekord felolvasása
- Ciklus *i* 1-től rekordhosszig
  - Ciklus *j* 1-től 8-ig
    - WAV-ból 16 bites adat beolvasása, az alsó egy bit leválasztása
    - A rekord *i*. bájtjának (*j-1*). bitjének beállítása az előző lépés alapján
  - Ciklus vége *j*
- Ciklus vége *i*
- Adatfájl visszanyerése
- Ciklus *i* 1-től adatfájl hosszig
  - *d=0*
  - Ciklus *j* 1-től 8/*Ubits*-ig
    - WAV-ből 16 bites adat beolvasása, az alsó *Ubits* bit leválasztása
    - Az eredményt balra toljuk (j-1) lépéssel és OR művelet *d*-vel
    - Ciklus vége *j*
  - Kiírjuk *d*-t a kimeneti fájlba
- Ciklus vége *i*
- Feldolgozás vége

A felhasznált bitek (`Ubits`) értéke tetszőleges, de programozástechnikailag legegyszerűbb az 1-2-4 vagy 8 bitek felhasználása, így minden adatbájt `8/Ubits` lépésben feldolgozható.

Az eredmény értékelése

A fenti algoritmusok alapján megírt program már elvben működik. Figyelni kell azonban arra, hogy a fájlkezelést valamilyen módon pufferelni kell, mert a bájtonként történő olvasás és írás igen lassú művelet lehet. A végrehajtás idejét alapvetően az elrejtendő adatállomány hossza és a felhasznált bitek határozzák meg. Ha `uBits` = 1, nyolc mintát kell kiolvasni a WAV-ból egy adatbájthoz, ha `uBits`=8, akkor csak egyet.

Érdekes lehetőség lenne, ha a rejtett adatot tartalmazó WAV fájlt egy audió CD-re felírnánk, mert a CD meghallgatható lenne bármilyen CD-lejátszóval, és tökéletesen élvezhető még `uBits`=8 esetben is. A probléma azonban az, hogy ha az audió sávot a CD-ről újra lementjük, előfordulhat, hogy nem ugyanazt kapjuk vissza, amit felírtunk, ennek pedig két oka lehet. Az egyik, hogy felíráskor a számok elé néhány kitöltő 0 bájt kerül, és eltolja az eredeti adatblokkok kezdetét. Ez még nem nagy baj, mert a dekódolás során megpróbálkozhatunk az azonosító rekord keresésével is: ha elsőre nem találjuk meg a fejléc után, kihagyunk egy bájtot és újra próbálkozunk. Azonban előfordulhat olyan olvasási (esetleg írási) hiba is, amely csak egy-két mintára terjed ki, így a normál zenehallgatást nem befolyásolja, azonban az adatfeldolgozás szempontjából végzetes bithibát vagy szinkronhibát okozhat[57]. Egy hangfelvételt terhelő járulékos zaj `Ubits` függvényében:

$$Q_z(1) = (1-16) * 20 * \lg 2 \cong -90\text{dB}$$
$$Q_z(2) = (2-16) * 20 * \lg 2 \cong -84\text{dB}$$
$$Q_z(4) = (4-16) * 20 * \lg 2 \cong -72\text{dB}$$
$$Q_z(8) = (8-16) * 20 * \lg 2 \cong -48\text{dB}$$

11.7.2. Példa a BMP fájl feldolgozására

Minden, amit a WAV-ról elmondtunk, igaz a BMP-re is, akár azt is mondhatnánk, hogy a BMP a képtárolás WAV-ja. Mint látni fogjuk, a feldolgozás is hasonló mindkét formátumnál, így a BMP-hez nem is fogok olyan részletes leírást adni, mint a WAV-hoz. Az ismertetett két algoritmus leírásában minden „WAV 16 bit”-et ki kell cserélni „BMP 8 bit”-re, de minden más változatlan.

A BMP fájl formátuma

Egy BMP állomány az alábbi fejléccel kezdődik. Az egyes mezők közül csak azok jelentése van feltüntetve, amelyek a feladat szempontjából szükségesek. A `BM` mező tartalma itt is csak arra kell, hogy a fájlt „belülről” is azonosítani lehessen.

Mezőnév	*Hossz*	*Tartalom*
BM	2	„BM”
Size	4	fájl mérete
Reserved1	2	
Reserved2	2	
OffBits	4	
HeadSize	4	a következő fejléc méret = 40 (bájt)
Width	4	kép szélessége
Height	4	kép magassága
Planes	2	Bitsíkok száma

[57] A gyakorlati próba során a CD-ről újra lementett WAV-ból hiányzott kettő minta, ami 2 bit kiesését eredményezte, ez pedig csúnya szinkronhibát okozott dekódoláskor. Természetesen egy hibajavító vagy –tűrő kódolással a probléma orvosolható.

BPP	2	Bit Per Plane
Compression	4	0=nincs tömörítés
IMGSize	4	a képleíró rész mérete
DPI_X	4	X irányú felbontás (?)
DPI_Y	4	Y irányú felbontás (?)
Reserved3	4	
Reserved4	4	

A fejléc után a pixelértékek jönnek. A példában csak a true-color, 24 bites, tömörítetlen állományokat fogjuk feldolgozni (`Planes=1, BPP=24, Compression=0`). Egy pixelhez három érték (RGB) tartozik, de a feldolgozás szempontjából ez tulajdonképpen közömbös (Esetleg négy, mert van egy alfacsatorna is, de a biztonság kedvéért a kapacitást csak három bájttal számoljuk). Miután a fejléc adataiból megfelelőnek találtuk az állományt, a `Height, Width` mezőkből és az `uBits` értékéből kiszámítható a BMP befogadóképessége:

```
Capacity = Height*Width * 3 * Ubit / 8
```

A feldolgozó algoritmus ugyanaz, mint a WAV algoritmusa, mindössze a WAV-ra vonatkozó 16-bites változókat kell egy 8 bitesre kicserélni. Kiegészítő rekordot itt is használni kell, a már megismert okokból.

Az eredmény

Az elkészült BMP kép az eredetihez hasonlóan megnézhető, kezelhető. Ha figyelembe vesszük, hogy a fejléc utáni bájtok a képernyő jobb alsó sarkától kezdik leírni a képet (jobbról balra, alulról felfelé), és ha a lehetséges kapacitásnak csak kevesebb, mint 30-40%-át használjuk ki, a kép felső felét akár össze is firkálhatjuk. Amit nem tehetünk: színkonverzió, összemosás, élesítés és egyéb teljes képre vonatkozó effektek, kicsinyítés, nagyítás és formátumváltás (kivéve, ha az új formátum veszteségmentes és azonos színmélységet biztosít). Érdemes kipróbálni, hogy az eredeti és a feldolgozott képet egy könyvtárba rakjuk és egy képnézegetővel teljes képernyős megjelenítésben ide-oda lapozgatunk a két kép között. Vajon mennyire fog szembetűnni a két kép közötti különbség?

> A témával foglalkozó irodalmak ilyenkor mindig két képet tesznek ide: az egyik az eredeti, a másikban adat lapul. Látszólag egyformák. Jó adatrejtés esetén monitoron, szemgúvasztva sem látszik a különbség, hát még nyomtatásban. Épp' ezért megkímélem az Olvasókat ettől...
>
> Az Interneten elérhető kész alkalmazás az MP3-as fájlok kezelésére, lásd [URL35].

„Nem titok az, akit sok ember tud."

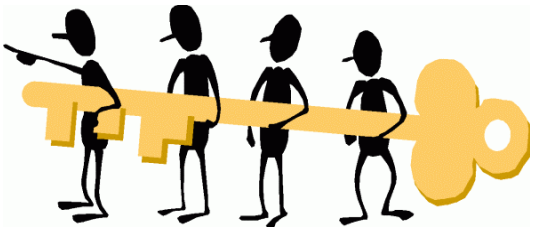
gr. Zrínyi Miklós

12. Titokmegosztás

Lássuk most azokat a módszereket, amelyek lehetővé teszik, hogy egy titkot úgy oszszunk meg a résztvevő felek között, hogy azt csak együtt tudják rekonstruálni. És vajon 1/3 = 0 mikor teljesül?

A titokmegosztás az eddigiekhez hasonlóan nem új dolog. A feladatra jó példa, amikor három kalóz elássa a rablott kincset, majd a kincs környezetéről egy térképet készít. A térképet három részre vágják, és mindenki eltesz egy darabot. Utódaik öröklik a térképdarabokat. Ha az utódok meg akarják keresni a kincset, a következő problémákkal kerülhetnek szembe:

- Nem találják meg egymást, vagy valamelyik térképdarab elvész. Kétséges, hogy a maradék darabokból a kincs helye meghatározható lenne, hiszen a három kalóz éppen azért (és úgy) darabolta fel a térképet, hogy egymásra legyenek utalva. Valószínűleg mindegyik térképdarab hordoz olyan információt a kincs helyéről, ami a többi darabon nincs rajta, ha nem így van, az adott darab egyszerűen felesleges.
- Elképzelhető az is, hogy egyetlen térképdarab is ad annyi információt, ami lehetővé teszi egy kincskereső expedíció elindulását, vagyis a fennmaradó lehetőségek kipróbálását idő, pénz vagy más erőforrás befektetésével.



55. ábra A titokmegosztás lényege: csak több ember együtt

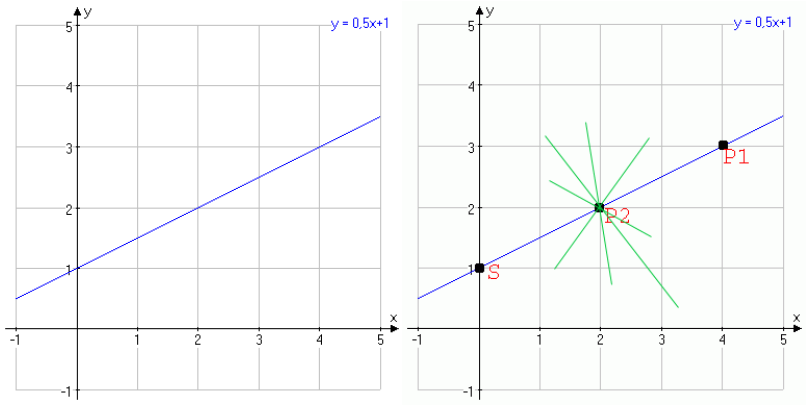
Látható tehát, hogy a fenti „ősi” módszerrel két probléma van: egy szereplő kiesése, egy titokdarab (árnyék vagy *share*) elvesztése meghiúsíthatja a titok teljes rekonstrukcióját. Ha pont ez volt a cél, ez természetesen nem baj, de valamilyen S.O.S helyzetben sem könyörül meg a próbálkozón. A másik probléma viszont éppen az, hogy a maradék titokdarabok már adhatnak annyi információt, amennyi a fennmaradó lehetőségeket esetleg kipróbálhatóvá teszi. A titok értéke dönti el, hogy megéri-e végrehajtani a próbálkozásokat, vagy sem.

12.1. MATEMATIKAI MODELLEK

12.1.1. A többismeretlenes egyenletrendszer

A titokmegosztás matematikai módszerei mindkét problémát megoldják. A legismertebb módszer általánosított változata a *Langrange*-féle polinominterpoláción alapul. Nem túlságosan bonyolult az elve, akár a középiskolában vagy a gimnáziumban is taníthatnák. Vajon hányszor hangzott el matekórán az a kérdés, hogy hány pont határoz meg egyértelműen egy egyenest vagy egy parabolát? A módszer általánosítva ugyan, de ezen a kérdésen alapul. (Sokkal általánosabb és jóval több matematikai háttérrel, mégis egy viszonylag egyszerű megközelítést találhatunk [41]-ben, ami végső soron az egyenletrendszeres modellen alapul.)

Egy *k*-ad fokú polinom együtthatóit vissza tudjuk állítani, ha ismerjük a *k+1* különböző helyen felvett értékét. Legyen a titok az $x=0$ helyen felvett érték (vagyis a konstans együttható), a többi együttható véletlenül választott! Ha *n* részre akarjuk bontani a titkot, legyenek a titokdarabok a polinom $x=x_1,x_2,...,x_n$ helyen felvett $y=y_1,y_2,...,y_n$ értékei. Ezek az értékpárosok a polinom által meghatározott görbe pontjai. Ha az *n* érték közül tudunk *k+1* darab (x_i,y_i) értékpárost, a polinom együtthatói visszaállíthatók, így a 0. együttható is, ami maga a titok.



Ha a jobb oldali ábrán csak P2 pontot ismerjük, akkor az ábra közepén lévő „napsugár” azon egyesekből mutat néhányat, amelyeket P1 ismerete nélkül behúzhatunk. Ha P1 pont koordinátái (x_1,y_1) és P2 koordinátái pedig (x_2,y_2), akkor az $S(0,s)$ a következőképpen számolható ki:

$$s = y_2 - (y_1-y_2) / (x_1-x_2) * x_2$$

Első fokú polinom	$y=a_1*x^1+a_0$	*egyenes*	2 pont
Másodfokú polinom	$y=a_2*x^2+a_1*x^1+a_0$	*parabola*	3 pont
...	...		...
k-ad fokú polinom	$y=a_k*x^k+a_{k-1}*x^{k-1}+...+a_0$		k+1 pont

56. ábra Polinomok és a meghatározásukhoz szükséges pontok száma

Shamir – akinek a nevéhez az eredeti ötlet is kapcsolódik – kicsit másként írta fel a függvényt. Később látni fogjuk, hogy a séma alkalmazásakor felbukkanó számok igen nagyok is lehetnek, ezért is hasznos a Shamir-féle formula, ami a moduláris aritmetika keretein értelmezi az iménti függvényeket:

$$y \equiv (a_k * x^k + a_{k-1} * x^{k-1} + \ldots + a_0) \mod p$$

A gyakorlati megvalósítás során ezt vegyük figyelembe, de itt és most eltekintünk a moduláris aritmetika alkalmazásától. Az alapelv megértését ez nem befolyásolja, de amíg az első verziót egyenesekkel és parabolákkal lehet szemléltetni, addig a moduláris változatot képtelenség vizuálisan alátámasztani.

Az olyan sémákat, ahol az összes titokbirtokos (legyen a számuk *n*-nel jelölve) közül bármely *t* vissza tudja állítani a titkot, (*t,n*) küszöbsémáknak nevezzük. Ha a megadott *t* résztvevőnél kevesebb t_1 darab titokbirtokos nem tudja előállítani a titkot, valamint összes információjuk pontosan annyi, mintha nem is rendelkeznének titokdarabokkal, akkor a (*t,n*) sémát egyúttal tökéletes sémának nevezzük. Ha a rekonstruáláshoz szükséges résztvevők száma pontosan annyi, mint ahány titokdarab van ($t=n$), akkor **titokszétvágásról**, ha kevesebb ($t<n$), **titokmegosztásról** beszélünk.

Nézzük meg egy példán, hogy mit tud nyújtani ez a módszer? Tegyük fel, hogy egy számkódos zár kódját szeretnénk felosztani. Meghatározhatjuk, hogy hány személy között akarjuk a titkot szétosztani, legyen ez mondjuk 5. Meghatározhatjuk azt is, hogy az 5 ember közül minimálisan hány szükséges a kód visszaállításához. Legyen ez 3, vagyis ez egy (3,5) küszöbséma (titokmegosztás). Ekkor az 5 résztvevő közül bármelyik 3 ki tudja nyitni a zárat, és legfeljebb két titokdarab elvesztése vagy megrongálódása esetén is visszaállítható a kód.

Legyen a felosztandó kód *1974*. Azt akarjuk, hogy 3 ember már elő tudja állítani a kódot, ezért egy másodfokú egyenletet kell felírni:

$$y=a_2*x^2+a_1*x+a_0$$

Válasszuk meg az együtthatókat :

a_2= véletlenül választott, legyen 23

a_1= véletlenül választott, legyen 5

a_0= 1974

Ekkor az

$$y=23*x^2+5*x+1974$$

másodfokú egyenlet adja munkák alapját. Mivel 5 ember között osztjuk szét a kódot, 5 számpárost kell generálnunk szabadon választott *x* értékekkel:

A résztvevő			Az ő számpárosa:
E1:	x=12	y=5346	(12; 5346)
E2:	x=7	y=3136	(7; 3136)
E3:	x=22	y=13216	(22; 13216)
E4:	x=6	y=2832	(6; 2832)
E5:	x=19	y=10372	(19; 10372)

A számpárokat kiosztjuk a résztvevő felek között, majd az eredeti együtthatókat megsemmisítjük. Ezután állítsa vissza a kódot mondjuk E1, E3 és E5 (többen is lehetnek, de elég csak hármat figyelembe venni)! Az ő számpárosuk rendre *(12;5346)*, *(22;13216)*, *(19;10372)*. Ezekkel a következő három egyenlet írható fel:

$$a_2*144 + a_1*12 + a_0 = 5346$$
$$a_2*484 + a_1*22 + a_0 = 13216$$
$$a_2*361 + a_1*19 + a_0 = 10372$$

Ez pedig nem más, mint egy lineáris, háromismeretlenes egyenletrendszer. Három független egyenletünk van, ezért az egyenletrendszer megoldható, sőt minket nem is érdekel minden ismeretlen, csak a_0. Az egyenletrendszert megoldva $a_0 = 1974$ lesz. (Aki nem hiszi, számolja ki!)

A modell tulajdonságai

- Látható, hogy ha 3-nál több számpáros (titokdarab) áll rendelkezésre, akkor is elég a számoláshoz a három adat. Ha viszont háromnál kevesebb számpárosunk van, az egyenletrendszer nem oldható meg, mert három ismeretlenünk van, de csak kettő független egyenletet tudunk felírni. Így egy vagy két titokbirtokosnak ugyanannyi információja van a titokról, mint egy olyan kívülálló embernek, akinek egy számpárja sincsen. (Mindenkinek 10000 próbálkozása marad.) *Ezért a séma tökéletes*. Bizonyítható (és első-, másodfokú polinomok esetében akár koordinátageometriával is belátható), hogy bármely két titokdarab együttesen is nulla ismeretet hordoz a titokról, vagyis két titokdarab ismerete esetén éppen annyi a szóba jöhető kódok száma, mintha nem tudnánk semmit.
- Ha új fél csatlakozik a csoporthoz (de a visszaállításhoz szükséges résztvevők száma nem változik), egyszerűen kiszámolunk neki is egy értékpárt. A már meglévő titokdarabokon nem kell változtatni. Mivel az eredeti együtthatók megsemmisítésre kerültek, *ideiglenesen szükség van a titok rekonstruálására*.
- Ha a titokdarab-hordozók nem egyformán fontosak, a modell azt is tudja kezelni. Osszuk az „árnyékok hordozóit" két csoportra, az első csoport tagjai kapjanak egy titokdarabot, míg a második csoporthoz tartozók kettőt. Ha a példabeli három titokdarab szükséges a visszaállításhoz, akkor a második csoportból bármely kettő, az első csoportból bármely három személy visszaállíthatja a titkot, illetve a harmadik lehetőség az, hogy az első csoportból egy és a második csoportból is egy titokdarab-birtokosra van szükség a visszaállításhoz. (A lényeg, hogy a visszaállításban részt vevő tagok összesen legalább a szükséges számú titokdarabot birtokolják.) Általában igaz az, hogy ha meghatározzuk azokat a priorizált vagy egyenrangú részhalmazokat, amelyek jogosultak a titok visszaállítására, készíthető olyan megosztási séma, amely teljesíti a feltételeket.
- Viszonylag egyszerű matematikai problémán alapul és nem egy (még) megoldatlan problémán.

- Ha azt akarjuk, hogy a titokról a megosztásban résztvevők ne tudjanak semmit, szükség lehet egy megbízható harmadik félre (*trusted third party*), aki a számításokat elvégzi.

Igaz az is, hogy ha a séma tökéletes, akkor a titokdarabok mérete elméleti szempontból legalább akkora, mint maga a titok. Ha két titokdarab-birtokos összesen is nulla információval rendelkezik a titokról, magának a harmadik titokdarabnak legalább annyi információt kell hordoznia, mint maga a titok.

12.1.2. Logikai műveletek

Lehetőség van egyszerűbben kivitelezhető algoritmus megvalósítására is XOR művelet használatával. Az egyszerűség főként egy esetleges célhardver megvalósítás miatt lehet fontos.

1. Legyen a megosztandó információ bináris reprezentációja S. A megbízható harmadik fél eggyel kevesebb véletlen bitsorozatot generál, mint ahány résztvevő van (n), de ezek hossza S hosszával megegyező.
2. Legyenek ezek a bitsorozatok $R_1, R_2, ..., R_{n-1}$.
3. Ezután kiszámolja a $C=S \oplus R_1 \oplus R_2 \oplus ... \oplus R_{n-1}$ értékét és megsemmisíti S-t valamint szétosztja a felek között $R_1, R_2, ..., R_{n-1}, C$ értékeket (mindenkinek egyet).
4. A visszaállításhoz az $S=C \oplus R_1 \oplus R_2 \oplus ... \oplus R_{n-1}$ értékét kell kiszámolni. A visszaállításhoz minden darabra szükség van, ezért ez egy (n,n) küszöbséma (titokszétvágás).

Egy példa három résztvevőre:

Generálás

```
S  = 00100100111001101010
R1 = 10011100011100101101
R2 = 01010101001101001001
C  = 11101101101000001110 kiosztandó: R1, R2, C
```

Visszaállítás

```
R1 = 10011100011100101101
R2 = 01010101001101001001
C  = 11101101101000001110
S  = 00100100111001101010
```

Egyszerűsített XOR művelet: ha az egymás felett lévő „1” bitek száma páros, az eredmény „0” ha pedig páratlan, az eredmény „1”.

A modell tulajdonságai

- Ha a szükséges n titokdarabnál kevesebb áll rendelkezésre, (a többi megsemmisült vagy elveszett), a titok nem állítható vissza. Éppen ezért ezt a módszert nem titokmegosztásnak, hanem titokszétvágásnak nevezzük. A szétvágás kényes a titokdarabok meglétére, ezért csak indokolt esetben használjuk, ha tudomásul vettük, hogy a szétvágás nem tudja biztosítani a megosztás azon lehetőségét, hogy a titkot tárolásbiztonsági okokból is megoszthatjuk.
- Az előbbi pont miatt ez a séma tökéletes.

- A visszaállításhoz minden darabra szükség van, ezért ez egy (*n,n*) küszöbséma.
- A titokdarabok fizikai mérete megegyezik a titok méretével.
- Ha új fél csatlakozik a csoporthoz, az csak akkor kaphat titokdarabot, ha a titkot időlegesen egyesítjük, és *C* értékét újraszámoljuk az új *R* figyelembe vételével. A már kiosztott *R* darabokon azonban nem kell változtatni, de akinél *C* volt, az új darabot kap. Célszerűnek látszik, hogy a titokdarabok ne legyenek egymástól megkülönböztetve, így viszont az sem mondható meg, hogy kinél van a *C* darab, ezért az egész szétvágási procedúrát elölről kell kezdeni. A polinomalapú sémánál egyetlen már meglévő együtthatót sem kellett eldobni.
- Ez a séma nem tudja figyelembe venni a résztvevők eltérő fontosságát: mindenkire szükség van.
- Ha azt akarjuk, hogy az információról a megosztásban résztvevők ne tudjanak semmit, szükség van egy megbízható harmadik félre (*trusted third party*), aki a számításokat elvégzi.

12.2. EGY GEOMETRIAI MODELL

Blakley másképp közelített a problémához. Legyen a titok reprezentációja egy pont (*x,y,z*) a háromdimenziós térben. Legyenek a titokdarabok a pont vetületei az (*x,y*), (*x,z*), (*y,z*) tengely által meghatározott síkokra. (Ez az eredete az árnyék elnevezésnek?) Egy ilyen vetület egy egyenest határoz meg, amelyen a titoknak rajta kell lennie. Bármely két titokbirtokos vissza tudja állítani a titkot – ami az egyenesek metszéspontja –, ezért ez egy (2,3) küszöbséma. Ez a séma tökéletes vagy sem? A kívülálló számára az egész térben bárhol lehet a pont, az árnyék számára azonban már adott egy egyenes, bár mindkettőnek végtelen sok lehetősége van. De tételezzük fel, hogy létezik egy olyan, a titok jellegéből meghatározható *h*, melyre egyszerre teljesül:

$$|X| \le h \qquad |Y| \le h \qquad |Z| \le h$$

Ekkor egy kívülálló számára már csak egy térrész (a kocka belseje) marad véges sok ponttal, egy árnyéknak pedig csak egy szakasz. E két ponthalmaz már jóval kézzelfoghatóbb: a kívülállónak $8 \times h^3$, az árnyék tulajdonosának $2 \times h$ pontja van a próbálkozáshoz. Tehát ez *nem tökéletes séma*. (A parabolás, számzáras példánál is meghatározható egy *h=9999,* azonban a titokbirtokos saját darabjának a birtokában sem tudja szűkíteni a szükséges próbálkozások számát: neki is 10000 kódot kellene kipróbálnia.)

12.3. PROBLÉMÁK

12.3.1. Jogosult és jogosulatlan résztvevők

Sajnos az itt bemutatott módszereknek van egy közös problémája: a visszaállításkor a titokdarabok egzakt módon nem ellenőrizhetők le:

- Ha a visszaállítás a minimálisan szükséges résztvevőkkel történik, egy rosszindulatú résztvevő hamis titokdarab megadásával megakadályozhatja a rekonstruálást és személyét nem lehet kideríteni.
- Ha a minimálisnál több személy vesz részt a visszaállításban, előfordulhat, hogy valaki csak színleli a titokdarab felfedését (esetleg nincs is titokdarabja) és mégis megismeri a titkot.

Viszonylag egyszerű megoldás lehet valamilyen – aláírásra alkalmas – nyilvános kulcsú algoritmus kiegészítő használata. Ha ugyanis a megbízható harmadik fél (akit egyébként a szakirodalom gyakran *dealer*-nek nevez) a titokdarabok kiosztása előtt aláírja azokat a saját privát kulcsával, visszaállításkor pedig a résztvevők az aláírt darabot adják a közösbe, mindkét eset felfedhető. Senki sem ismeri *dealer* titkos kulcsát, ezért egyik résztvevő sem tudja a saját titokdarabját meghamisítani.

12.3.2. Hallgatózó illetéktelenek

Nemcsak belülről érheti támadás a titokmegosztást alkalmazó protokollt, hanem kívülről is. Amennyiben a titokdarabok felfedésének kommunikációját illetéktelenek lehallgatják, megszerzik az összes titokdarabot. Ha ismerik a titok rekonstruálásához szükséges algoritmust, akkor a résztvevőkkel párhuzamosan képesek előállítani titkot is. Első gondolatra a titokdarabok titkosítása tűnik megoldásnak, azonban a titok előállítását a titkosításhoz használt kulcs hiánya meghiúsíthatja, így a titkosító kulcs birtokosa mindenképpen szükséges résztvevővé válik. Éppen ezért nem a titokdarabokat kell titkosítani, hanem a titokdarabok kiosztásához vagy begyűjtéséhez használt kommunikációs csatornát kell védeni!

12.4. FELHASZNÁLÁSI TERÜLETEK

A fentebb vázolt módszereket abban az esetben használhatjuk, ha valamilyen információt (jelszót, mesterkulcsot vagy éppen jogosultságot) úgy kell megosztani a résztvevők között, hogy azok mindegyikének (titokszétvágás) vagy egy meghatározott csoportjának (titokmegosztás) jelen kell lennie az információ rekonstruálásakor.

Az ilyen célú megosztás *lehetőség a kulcs biztonságos tárolására* is. Ha ugyanis a részeket különböző helyeken, különböző személyek felügyelete alatt őrizzük, nem fordulhat elő, hogy az információ megsemmisül. Ez titokszétvágással nem, csak megosztási séma használatával valósítható meg biztonságosan.

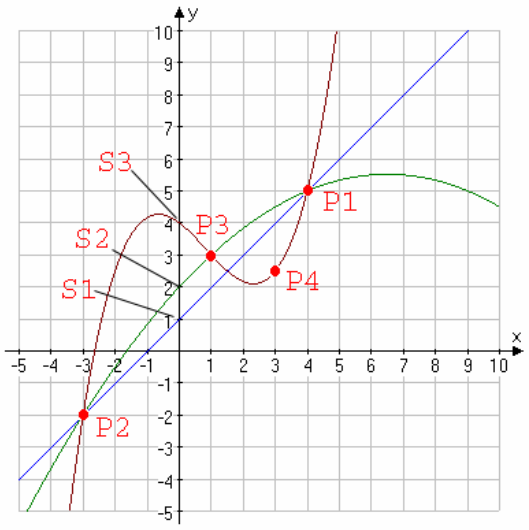
Megoldható egy digitális aláírás jogosultságának szétosztása is, például egy bankban egy nagy értékű átutalást bármely két alkalmazott csak együttesen írhasson alá. Ez a tehát a titokmegosztás másik felhasználási területe: *egy fontos információt (vagy jogosultságot) úgy bontunk fel, hogy a részek külön-külön semmire nem jók*.

Hozzáférési szintek szabályzása

Hozzáférési szintek szabályozása is elvégezhető az eddig bemutatott eszközökkel. Vegyük a következő egyszerű példát!

- Legyen a felhasználók besorolása Level-1, Level-2 és Level-3.
- Azt kívánjuk elérni, hogy
  1. aki Level-1 jogosultsággal bír, csak a Level-1 szintű titokhoz férjen hozzá.
  2. aki Level-2 jogosultsággal rendelkezik, az a Level-1 és Level-2 szintű titok rekonstruálásra egyaránt képes legyen.
  3. aki Level-3 jogosultságot kapott, az Level-1, Level-2 és Level-3 szintű információkhoz is hozzáférhessen.

A következő ábrán az egyes titkokat S1, S2 és S3 pontok képviselik, míg a titokdarabok jelölése P1, P2, P3 és P4. A pontok meghatározásának menete a következő:



1. S1-P1 egyenesen (tetszőleges helyen) kijelölhető egy P2, amely P1-el együtt S1 védelmét látja el.

2. Majd P1-P2-S2 ismeretében meghatározható egy olyan parabola, amely áthalad mindhárom ponton. A parabolán kijelölhető tetszőleges P3, amely P1-gyel és P2-vel együtt S2 titkot védi.

3. Végül P1, P2, P3, S3 pontokkal kijelölhető egy harmadfokú görbe, amin egy tetszőleges pontot kijelölve P4-et kapjuk.

Ezután azok a résztvevők, akik birtokolják

- P1 és P2 titokdarabokat, képesek S1 előállítására,
- P1, P2 és P3 darabokat, S2 és S1 előállítására egyaránt képesek és végül
- P1, P2, P3 és P4 darabok birtokosai S3, S2 és S1 előállítására képesek.

Mivel P1 és P2 pont mindhárom szint hozzáféréséhez szükséges és nélkülük egyetlen titokdarab sem állítható elő, ezeket *aktiváló pontoknak* is nevezik. A titokdarabok egy lehetséges elosztását és a résztvevők hozzáféréseit a következő táblázat szemlélteti:

	Résztvevő1	**Résztvevő2**	**S1**	**S2**	**S3**
Level-1 jogosultságú páros:	P1	P2	✓	×	×
Level-2 jogosultságú páros:	P1, P3	P2	✓	✓	×
Level-3 jogosultságú páros:	P1, P3	P2, P4	✓	✓	✓

A titokszétvágásnak létezhet egy más megközelítése is: gondoljunk csak a nyíltkulcsú titkosításokra! A hagyományos titkosításnál a közös titok egyeztetése volt a fő probléma, amelyet ma a PKI rendszerek eszközei hidalnak át. Azonban ha az egyik legismertebb kulcscsere protokollt, a Diffie-Hellmant szemügyre vesszük, az sem tesz mást, mint a közös titkos kulcsot felbontja két részre, melyeket a kommunikáló felek kicserélnek, így képesek előállítani egy olyan kulcsot, amit más nem ismerhet meg. Igaz, hogy nem egy meghatározott titkot bont a protokoll ketté, hanem két részből rak össze „valamit”, de bizonyos nézőpontból ez is titokmegosztás. Hasonló elv mutatható meg a legtöbb nyíltkulcsos algoritmus működésében is.

Your password must be at least 18770 characters and cannot repeat any of your previous 30689 passwords. Please type a different password.

Egy – azóta már kijavított – Windows 2000 hibaüzenet,
amely igen maximalista hozzáállást tükröz... (Q276304)

13. A WINDOWS FŐBB KRIPTOGRÁFIAI SZOLGÁLTATÁSAI

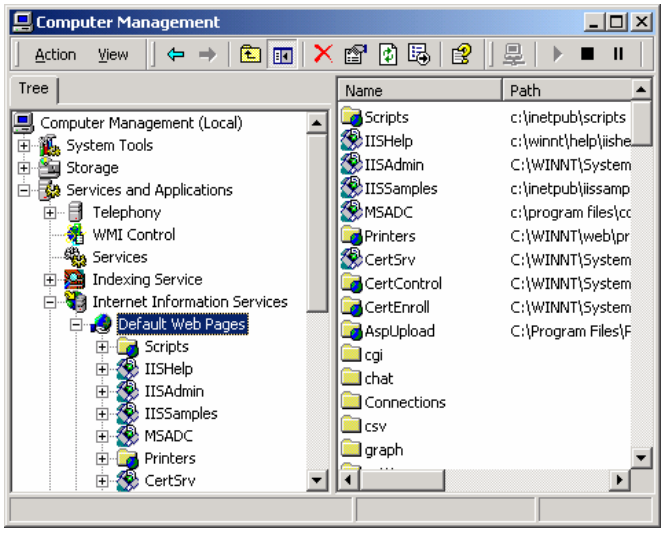
A Windows számtalan kriptográfiai eszköz használatát teszi lehetővé, néha a felhasználó számára rejtett módon. A PKI Windows környezetben való használatához [54]-ben találunk sokkal részletesebb útmutatást, szinte az egész könyv ezzel foglalkozik, mi csak néhány példát érintünk: egy webszerver SSL-re való felkészítését, a titkosított fájlrendszer működését és a biztonságos kulcstároló eszközök felhasználását.

13.1. AZ IIS 6.0 BIZONYÍTVÁNYA

Ha valaki webszervert üzemeltet, felmerülhet az igénye, hogy a szervere képes legyen igazolni magát, sőt a korábban bemutatott SSL-kapcsolaton keresztül titkosított, biztonságos kommunikációt tudjon biztosítani a kapcsolódó ügyfél részére. Vagyis azt szeretné, hogy a 

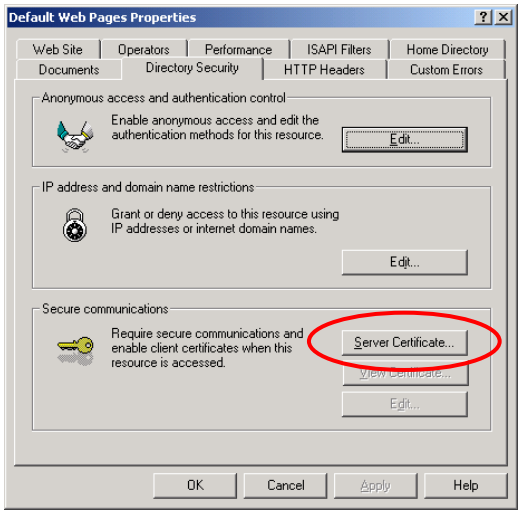
böngésző a szokásos kis lakatikon megjelenítésével jelezze a felhasználó felé a szerver biztonságosságát. Mit is kell ehhez tennie?

Windows 2003 alatt az Internet Information Server 6.0 (IIS6) telepítésével ez nem is olyan nehéz. Az IIS6 komponenst a Windows 2003 telepítőkészlete alapból tartalmazza, csak ki kell választani[58], de telepítés után a webszerver még nem kész az SSL-alapú kapcsolatok kiszolgálására. A bizonyítvány telepítését a *Computer Management* → *Internet Information Services* pontján kezdjük.

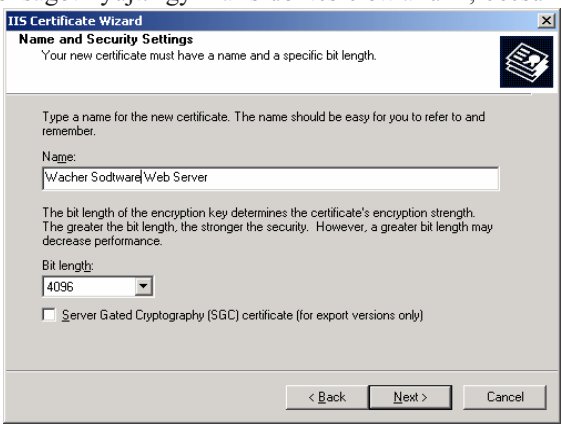


A *Default Web Pages* tulajdonságainál kattintsunk a *Direcytory Security* fülre. Ezen a lapon most a legalsó rész (*Secure communication*) az érdekes.

[58] Windows 2000 – IIS5 alatt pontosan ugyanezeket a lépéseket kell elvégezni és minden ismertetett funkció ott is elérhető..



Ha a mellékelt ábrához hasonlóan a három gombból csak egy érhető el, a szerver még nem rendelkezik bizonyítvánnyal, így bátran nekiállhatunk a telepítésnek. Bökjünk is egyből a *Server Certificate* gombra, hogy az *IIS Certificate Wizard* üdvözölhessen bennünket. Itt válasszuk a „*Create new certificate*” opciót, majd a *Next* gomb után a *„Prepare the request now, but send it later”* vagyis *„Kérés előkészítése, küldés később*” opciót. Mit is fogunk most csinálni? Megadjuk a szerverünk minden fontos adatát, generálunk hozzá egy kulcspárt és a nyilvános kulcsunkat az adatokkal együtt elküldjük a *CA*-nak, aki majd ellenőrzi adatok helyességét és valódiságát. Mindent, amit kapott, a saját kulcsával aláírva visszaküldi nekünk. Ez az a bizonyítvány, amire szükségünk van. Tehát *Next*. A következő ablakban adjunk valamilyen nevet a bizonyítványnak, célszerűen olyat, ami a bizonyítvány felhasználására és a szerverre egyaránt jellemző. Válasszunk kulcsméretet is! Minél nagyobb a kulcs (legfeljebb 4096 bit) annál biztonságosabb, de egyúttal lassabb is a rendszer[59]. Minél kisebb (legalább 512 bit), annál gyorsabb, de kvázi kisebb biztonságot nyújt. Így máris döntés előtt állunk, becsüljük fel mire van szükségünk. Gyakori megoldás az is, hogy kisméretű kulcsokat használnak, de a bizonyítványt gyakran cserélik (tipikusan 1-10 nap). Ilyenkor az üzemeltetőnek egy saját *issuing-CA*-ja van. Ha nem akarunk napi rutinként ezzel bajlódni és az automatizálásra sincs lehetőség, válaszszunk nagyobb kulcsot, ami hosszabb időtartamra garantálja a biztonságot. Ne feledjük azt sem, hogy ha a bizonyítványt piaci szolgáltatótól szerezzük be, a bizonyítványok kiállítása pénzbe kerül, ha nagyon gyakran cserélgetjük, nem is kevésbe. Ha döntöttünk, *Next*.



[59] Egyébként kapható „eCommerce gyorsítókártya” is, bár kissé borsos áron (kb. 2000$-tól). A kártyák átveszik a biztonsági szerverek processzoraitól az SSL/TLS, WAP/WTLS, S/MIME, IKE/IPSec, stb. protokollokhoz szükséges számításokat.

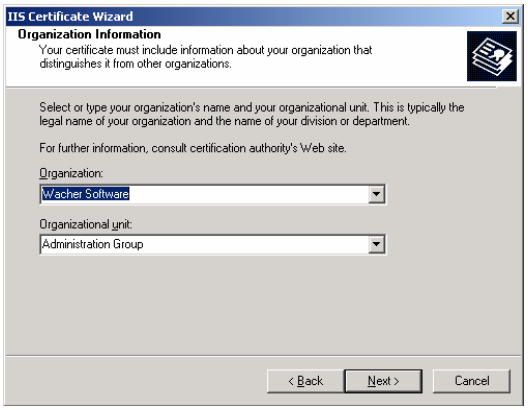
Intranetes környezetben lehetőség van arra, hogy egy Windows 2003 szerver legyen a root-CA. Ezt az opciót külön kell telepíteni és konfigurálni a szerverre. Azonban nem javaslom, hogy egy belső portált ilyen „házi" bizonyítvánnyal lássunk el, legfeljebb a kísérletezés kedvéért. Miért? Mert a mi kis házi CA-nk nem szerepel a Windows „megbízhatónak tartott root-CA" listáján, így a kedves felhasználót minden egyes alkalommal megkérdi a böngésző, hogy mit csináljon egy olyan bizonyítvánnyal, ami ugyan jó is meg érvényes is, meg az használja, akinek szól, csak éppen a hitelesítő szervezet nem szerepel a nyilvántartásban. Ügyesebb felhasználók meg tudják oldani a problémát, és telepítik a hitelesítő szervezet bizonyítványát a Trusted Root tárolóba (ha a kérdéses bizonyítvány tartalmazza azt). Aki kevésbé bátor, az vagy egy bizonytalan „tovább"-ot nyom, vagy szól rendszergazdának. Az arány a felhasználók számától függ…

A megoldást egyébként két dolog jelentheti:

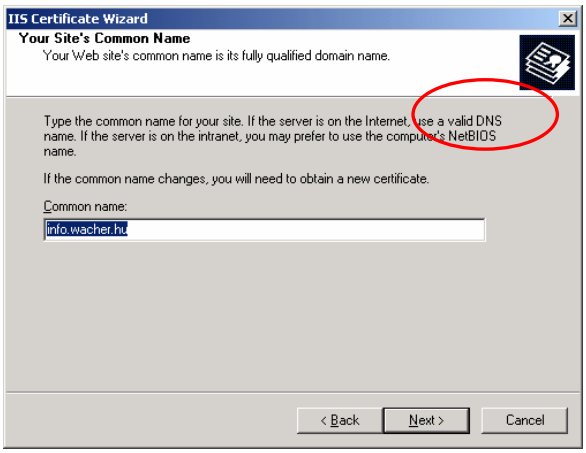
1. A házi CA-t nem root-CA-ként üzemeltetjük, hanem subordiante CA-ként, egy létező bejegyzett CA által hitelesítve.
2. A webkiszolgálón beállítjuk a CTL listát (Certificate trust list), amely a megbízhatónak tartott CA-kat tartalmazza. (Ez azonban egyben korlátozás is, mert innentől csak ezek lesznek megbízhatóak.)

Más kérdés, hogy az operációs rendszer telepítésekor több mint 100 olyan bizonyítvány is telepítésre kerül, amelyek kibocsátóiról a legtöbben még nem is hallottak (lásd Függelék, „*14.11. Alapértelmezésben telepített bizonyítványok*"). Vajon ezekben valóban megbízunk, vagy csak ismét elfogadunk egy mások által meghozott döntést? Ne hamarkodjuk el a választ! Merthogy az is igaz, ha ezek a bizonyítványok nem lennének telepítve, a felhasználó számára kész rémálom lenne a tanúsítványok használata, hiszen a rendszer lépten-nyomon megkérdezné valamelyik kibocsátóról, hogy megbízik-e benne a felhasználó vagy sem. Csakhogy a fenti helyzet állna elő: a felhasználó a szervezetek nagy részéről azt sem tudná, hogy léteznek egyáltalán, hiszen nem ismer(het)i a (világ)piac összes szereplőjét. Azonban ha használni szeretné azt a szolgáltatást, amihez az adott tanúsítvány kapcsolódik, igenis el fogja fogadni annak kibocsátóját, mint megbízható kibocsátót – még akkor is, ha esetleg nem az.

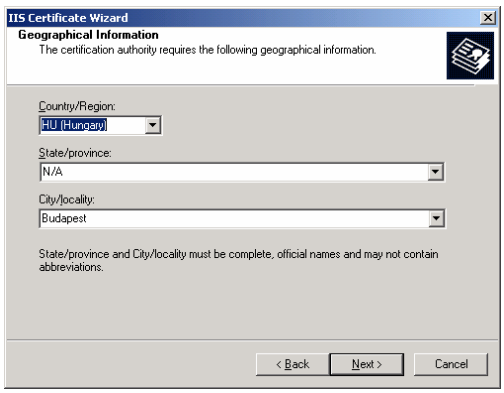
Ha eldöntöttük, hogy mekkora kulcsot kérünk, a következő ablakban megadhatjuk saját adatainkat. A példában az „*Organization*" = **Wacher Software**, a „*unit*" = **Administration Group**. Az ezt követő ablakban adjuk meg a szerverünk nevét. Itt érdemes megtartani a wizard ajánlását, és a gép azon nevét megadni, ahogyan a felhasználók hivatkoznak rá (vagyis egy érvényes DNS névnek kell lennie). Ha nem így teszünk, a felhasználó minden egyes látogatásakor figyelmeztető üzenetet fog kapni, hogy a bizonyítványt nem az használja, akinek ki lett állítva. (Például a gép neve ugyan `srv-wacher`, de a DNS-ben van egy `info.wacher.hu` alias, és az intranetes felhasználók azt használják, ez utóbbi számára kell kiállítani a bizonyítványt. Internetes környezetben a gép fizikai nevét nem is látja a látogató, úgyhogy ott egyértelműen a regisztrált domainnévre kell kiállítani a bizonyítványt. Bizonyos szempontból ez az egyik legfontosabb adat a bizonyítvány kérésekor. Ha ezt elrontjuk, a látogató böngészője garantáltan érvénytelennek

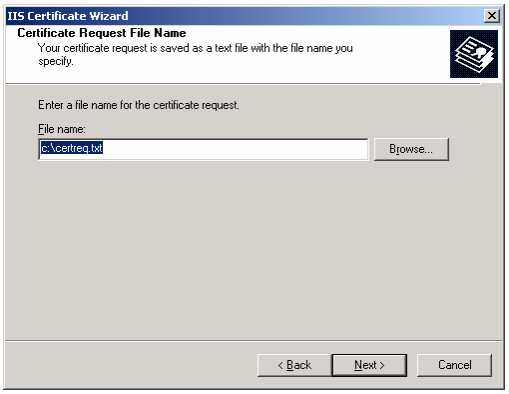


fogja nyilvánítani a bizonyítványunkat. Ne feledjük: a bizonyítvány azért készül, hogy biztonságot és bizalmat adjon! De ha a felhasználó ehelyett egy biztonsági figyelmeztetést kap...?

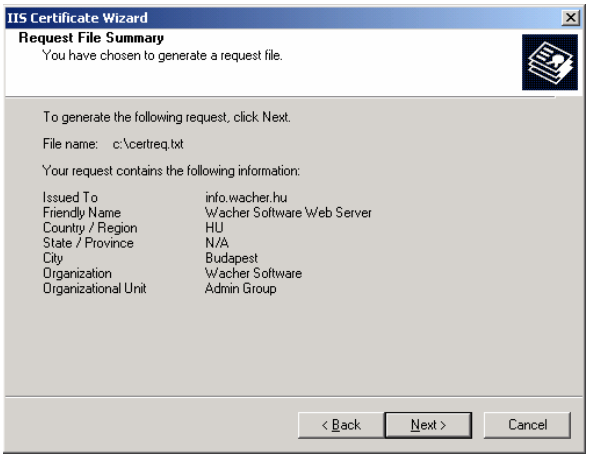


Ezután már csak két képernyőn kell túllenni: az egyiken a néhány földrajzi adatot kell megadni (ország, város), a másikon pedig azt a fájlrendszerbeli útvonalat, ahova a varázsló elkészíti a bizonyítványkérelmünket. Ezt a fájlt kell továbbítani majd a hitelesítő szervezet felé, ami egy BASE64-DER kódolású bizonyítványkérelem lesz, és tartalmazni fog minden eddig megadott adatot, valamint a most generálásra kerülő kulcspár publikus tagját.





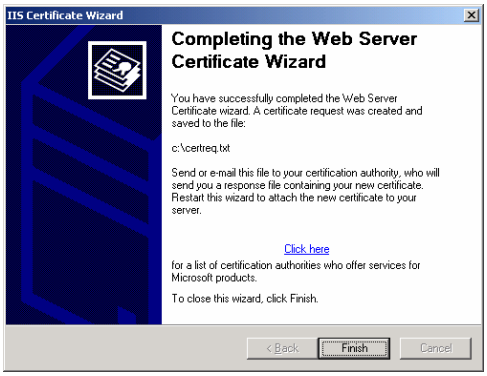
Végül a kulcspárgenerálás megkezdése előtt az alábbi összefoglaló képernyőt láthatjuk. Itt minden adatot „visszaolvas” a Windows, és csak ezután kezdi majd el a kulcsok és a bizonyítványkérelem készítését.



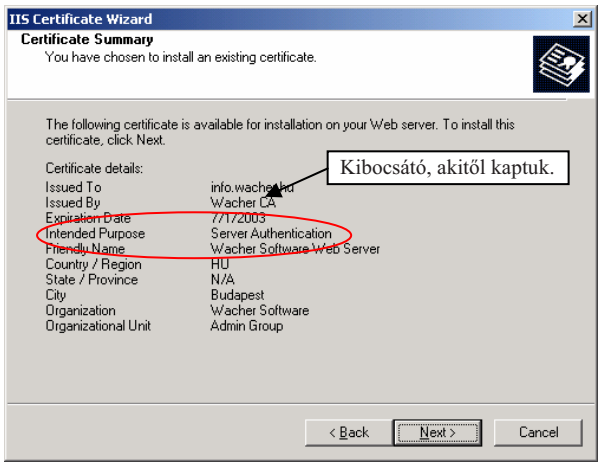
Összefoglaló képernyő a bizonyítvány kérelem előtt

Ha eddig eljutottunk, már félig kész is vagyunk. Az adatok ellenőrzése után nyomjuk meg a *Next* gombot! A következő művelet (vagyis kulcspár és bizonyítványkérelem elkészítése) a gép sebességétől és a választott kulcsmérettől függően 5-30 másodpercig tart.

Ha megkaptuk a lenti ablakot, kész a bizonyítványkérelmünk, vagyis a fentebb megadott helyre elkészült egy körülbelül 2kb méretű BASE64 kódolású szöveges állomány! Ezt az állományt kell eljuttatnunk a hitelesítést végző szervezethez. (Ez általában titkosított weboldalon keresztül történik: vagy egy adott szövegbeviteli mezőbe kell *copy+paste* módon beilleszteni a fájl tartalmát, vagy fájlfeltöltéssel magát az állományt kell elküldeni. Ezután a hitelesítő szervezet (előbb vagy utóbb) megkeres bennünket, hogy adatokat egyeztessen és ellenőrizzen. Az adategyeztetés „intenzitása” attól függ, milyen felhasználási célra igényeljük a bizonyítványt.



Ha minden jól megy, visszakapunk egy *.cer állományt. Ennek telepítéséhez menjünk a *Computer Management* már ismert helyére (*IIS* → *Web pages* → *Properties* → *Directory Security* → *Server Certificate*) és válasszuk a "*Process pending request…*" kezdetű menüpontot. Keressük meg a CER fájlunkat, majd örüljünk a megjelenő összefoglaló adatoknak! Itt az általunk korábban megadott adatokon kívül láthatjuk azt is, hogy meddig érvényes a bizonyítványunk, és mire használható. Mi egy szerver számára kértük, így „*Server Authentication*" lesz felhasználási terület.



Kattintsunk néhányszor a *Next* gombra és kész! Mostantól szerverünk alkalmas biztonságos, SSL-alapú kommunikációra a látogatókkal, pontosabban azok böngészőivel. Ha a már ismert *Directory Security* fülnél a *Secure Communications* szekcióban az *Edit* gombra kattintunk, a *Require secure channel* (SSL) opció kiválasztásával előírhatjuk, hogy csak SSL-képes böngészővel lehet az oldalunkat látogatni, vagyis csak `https://` protokollon keresztül. Ha ez a választónégyzet nincs bejelölve, a kliensek a `http://` és `https://` protokollokkal egyaránt csatlakozhatnak a szerverhez. De nemcsak a szerver rendelkezhet tanúsítvánnyal, hanem a felhasználó is. Sőt elő is írhatjuk, hogy a felhasználó csak akkor tudja az oldalt meglátogatni, ha maga is rendelkezik erre alkalmas bizonyítvánnyal. Ezzel a védett oldalak szokásos jelszóalapú azonosítását a sokkal biztonságosabb tanúsítványalapúra cserélhetjük.

13.2. ENCRYPTING FILE SYSTEM - EFS

13.2.1. Az egyik probléma: OS-szintű hozzáférés-vezérlés

Azok az operációs rendszerek, amelyek képesek fájlrendszer-szintű jogosultságkezelésre, általában autentikáció alapján teszik ezt. Vagyis a felhasználó bejelentkezéskor azonosítja magát (*identifikáció*) és az operációs rendszer bizonyos ACL-ek (*access control list,* vagyis *hozzáférést vezérlő lista*) alapján eldönti, hogy mihez van joga, mit érhet el, és mit nem. Sajnos ez csak akkor nyújthat megfelelő védelmet, ha nem indítható el a gépen más operációs rendszer,

ami esetleg képes az eredeti jogosultságokat megkerülni. Gondoljunk csak arra a viszonylag egyszerű esetre, hogy az adott gép merevlemezét egy másik gépbe tesszük. Ha ennek a másik a gépnek az operációs rendszere képes olvasni az eredeti merevlemezt anélkül, hogy az ottani ACL-eket figyelembe venné, bármit el tud olvasni.

Ez viszont mindenképpen megoldható, ha másként nem, a lemez szektoronkénti olvasásával. Aki ismeri az eredeti fájlrendszer felépítését, az eredeti tartalmat korlátozás nélkül rekonstruálni tudja. A megoldás az lenne, ha a fájltartalom titkosítással kerülne a merevlemezre, így azt *értelmezni* csak az lenne képes, akinek birtokában van az ehhez szükséges kulcs is.

13.2.2. A másik probléma: maga a felhasználó

Ha egy olyan programot használunk, amely egy fájlt titkosít vagy megfejt, de ezt a felhasználó utasítására teszi, alkalmazásszintű szolgáltatásról beszélünk. Vagyis a felhasználó kérésére a titkosított fájlból nem titkosított lesz, amivel utána dolgozhat, de neki kell gondoskodni a fájl újbóli titkosításáról vagy a titkosítatlan példány megsemmisítéséről is. Ennek a megoldásnak számtalan hátránya van:

- Manuális titkosítás és megfejtés a fájl minden használatakor.
- A fájl bezárása nem jelenti egyúttal a plaintext példány megsemmisítését.
- Esetleges ideiglenes fájlok létrehozása, amelyek nem titkosított adatot is tartalmazhatnak.
- Jelszóalapú védelem. A felhasználónak egy újabb jelszót kell választania, megjegyeznie. Persze használhatja a már jól bevált jelszavát is, de ha a támadó már megismerte azt, az új fájl sem lesz számára titok. A jelszavak feltörésére használt módszerek (*social engineering*, *dictionary attack*) itt is használhatók.
- Ha a jelszót elfelejti a felhasználó, a titkosított adat a legtöbb esetben nem állítható vissza. Egy szervezeti környezetben az „elfelejtés” jelentheti azt is, hogy a jelszót ismerő felhasználó kilép a szervezetből. Az elfelejtés másik tipikus esete, amikor az alkalmazások egy része felkínálja a „Jelszó megjegyzése” opciót, a felhasználó él vele, aztán amikor egy újratelepítés után először megjelenik az üres „Password” beviteli mező, na akkor kezd el aggódni…

Még egyszer a jelszavakról

Már több jellemző is ismertté vált előttünk, amik alapján el lehet dönteni, hogy egy jelszó jó vagy sem. Szóval mikor is jó egy jelszó?

- *Ha elég hosszú.* Minél rövidebb egy jelszó, annál könnyebb próbálkozással kitalálni. Ajánlott a legalább 8 karakteres jelszó választása. (Egy nem hivatalos felmérés szerint a felhasználói jelszavak átlagos hossza mindössze 5.6 karakter, bár ez valószínűleg függ a vizsgált nyelvtől is…[22,URL38]) A folyamatos próbálkozások ellen sok rendszer a következő egyszerű, de hatásos módszerek egyikével (vagy mindegyikével) védekezik:

1. Néhány (általában 3-5) hibás próbálkozás után egyszerűen kizár a rendszerből, és ha esetleg kitalálnánk a helyes jelszót, nem megyünk vele semmire, mert a rendszer be sem enged.
2. Az első hibás próbálkozás után 3 másodpercet vár a beléptető rendszer, mielőtt újra lehetőséget adna a belépésre, a második után 6 másodpercet, a harmadik után 12 másodpercet, a negyedik után 24 másodpercet és így tovább. A tizedik próbálkozás után – ha egyáltalán eljutunk odáig – majd' fél órát kell várni. (Utána meg egyet, kettőt, négyet…)
3. Nem pontosan idevágó „szokás", de érdemes megemlíteni, hogy a szerverek nagy része nem árulja el, hogy egy esetleges sikertelen identifikáció mi miatt hiúsult meg: a jelszó volt rossz, vagy a megadott felhasználónév nem létezik a rendszerben. Sok szerver a felhasználónév megadása után kiírja, hogy „*XY user okay, need password*" függetlenül attól, hogy „XY" a rendszer nyilvántartásában létezik-e vagy sem.

Azonban ezek a módszerek olyan rendszerekben nem megnyugtatóak, ahol az autentikációs adatok viszonylag könnyen hozzáférhető helyen vannak, még akkor sem, ha kódolva, titkosítva kerülnek tárolásra. A Unix egy jó példa erre: egy szöveges fájlban tárolja a felhasználói neveket és jelszavakat – utóbbiakat DES titkosítással. Akinek egyszer is sikerül *root* jogokat magához kaparintani, az le tudja másolni ezt a fájlt, és később próbálgatással meg tudja fejteni a jelszavakat, immáron az eredeti rendszertől teljesen függetlenül, saját gépének teljes kapacitását felhasználva.

- *Nem csak betűkből áll*, pláne nem csak kis betűkből (és főleg nem csak számokból!). Az ilyen jelszavak legtöbbje kereszt- vagy becenév, állatnév esetleg valamilyen cég- vagy márkanév. Ha valaki csak kisbetűket használ és 5 karakteres a jelszava, a legrosszabb esetben is 26^5 próbálkozással kitalálható a jelszava. Ha nagybetűket is használ, máris 32-szer több, vagyis 52^5 a próbálkozások száma. Ha meg még számokat is kever bele, az a legjobb. Általánosan ajánlható, hogy a második és a hatodik pozíció között legalább egy helyen valamilyen szokatlan karaktert is tartalmazzon.

> „[…] Az első 100 leggyakoribb jelszó pedig mintegy 1100 felhasználói azonosítót fed le! […] A leggyakoribb jelszavakat a feltörők listákban tárolják, így ez alapján a pár száz legismertebb jelszó feltörését egy közepes PC segítségével és a kódolt jelszavak ismeretében kb. egy perc alatt végezhetik el.
>
> Közismert, hogy évekkel ezelőtt a híres Internet Worm is részben úgy terjedt, hogy a magában hordozott 432 db legegyszerűbb jelszót próbálgatta ki a különböző rendszerekben. Megvizsgáltuk az Elender jelszófájlját is ezzel a 432 db angol jelszóval, és ebből összesen 64 db felhasználói azonosító/jelszó párost találtunk. Ezeknek a jelszavaknak a használata lehet talán a leggyengébbnek nevezhető, mert ezeket nemzetközi szinten sorolhatjuk a leggyengébbek közé." [22]

Ez idáig már majdnem jó, de itt jön az egyik legnehezebb része a dolognak. A felhasználót arra még rá lehet venni, hogy viszonylag hosszú jelszót használjon, és időnként cserélje is le, sőt arra is rá lehet beszélni, hogy számokat is tegyen bele, és ekkor általában valami ilyen jelszavak születnek: TAMAS74 meg DOV269, C3P0R2D2. Az ilyen tulajdonságú jelszavak főleg intranetes környezetben rosszak, mert ott a felhasználók több-kevesebb személyes információval is rendelkeznek egymásról. És akkor mi a megoldás? Egyesek szerint tuti módszer,

ha az ember választ egy könnyen megjegyezhető versikét és a versike szavainak kezdőbetűit használja jelszóként. (például *Boci boci tarka, se füle se farka* → *Bbtsfsf*) De ebben meg nem nagyon lesz számjegy, és általában nagy kezdőbetűvel indul. Van egyáltalán tökéletes gyakorlati megoldás, amit bármelyik felhasználó kezébe lehet adni?

> Két informatikus beszélget:
> - Te milyen jelszót választottál?
> - A kutyám nevét adtam.
> - De hát azt könnyű kitalálni!
> - Kitalálni?! Én magam is alig tudom megjegyezni, hogy Y2k89mzper7!

Sajnos a megjegyezhető jelszavak nem biztonságosak, a biztonságosak viszont nem megjegyezhetők. Amennyiben kellően biztonságos (következésképp nem, vagy nehezen megjegyezhető) jelszót választ a felhasználó, azt valószínűleg le fogja írni valahova: a billentyűzet aljára vagy egyszerűen egy Post-It segítségével a monitorra. Korábban láttuk, hogy a legtöbb autentikáció azon alapszik, hogy a felhasználó tud valamit és rendelkezik valamivel. De akkor miért baj az, hogy leírja a jelszavát egy darab papírra? Az autentikációs modell látszólag nem sokat változik, csak most már nem arról lesz szó, hogy tud valamit, hanem rendelkezik valamivel... A probléma ott van, hogy a papír által tárolt érzékeny információt illetéktelen személyek is értelmezni tudják, a felhasználónév pedig gyakorlatilag nyilvános...

A jelszavaknak van egy másik problémája is, amely elsőre nem nyilvánvaló, de a gyakorlati alkalmazás során hamar előjön. Képzeljünk el egy felhasználót, aki lelkesen használja a kibertér lehetőségeit vagy egyszerűen munkájához az kell, hogy több rendszerbe is bejelentkezzen. Idővel egyre több felhasználónév – jelszó párost kell megjegyeznie a különböző szolgáltatásokhoz:

- operációs rendszer belépési jelszava (esetleg több operációs rendszerhez is)
- alkalmazásokhoz kötődő belépési jelszavak
- előfizetéshez vagy regisztrációhoz kötött szolgáltatások (kereskedelmi vagy ingyenes levelező rendszerek jelszavai, levelező listák, e-boltok, fórumok bejelentkezései stb.)

A felhasználók döntő többsége a következő evolúciós lépcsőn megy végig:

1. Minden rendszerhez, minden belépéshez egyedi felhasználónevet és biztonságos egyedi jelszót választ.
2. Törekszik az első pontban leírtakra.
3. Idővel feladja, és elkezdi mindenhol ugyanazt az azonosítót és jelszót használni.

A probléma nem is azokkal a jelszavakkal és azonosítókkal van, amelyeket naponta használ, hanem azokkal, amelyekre ritkábban, csak eseti jelleggel van szüksége. (Például levelezőlisták adminisztrációs felületei, ritkán igénybevett szolgáltatások.) Miért? Ha elég ritkán használja ezeket, egyszerűen elfelejti, hogy mi volt a jelszava. A sok kényelmetlen helyzet után (és azok megelőzésére) végül mindenhol ugyanazokat az autentikációs adatokat adja meg, ami magában hordozza annak a veszélyét, hogy a felhasználónév – jelszó páros kompromittálódása a felhasználó összes loginjának kompromittálódását jelenti.

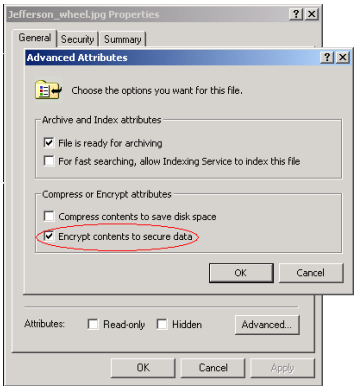
Nemcsak a felhasználó törekedhet saját életének egyszerűsítésére, hanem gyakran az alkalmazott informatikai rendszerek együttműködése is biztosítja ezt a lehetőséget. Ilyenkor egyetlen bejelentkezéssel mindent elér a felhasználó (persze csak amire joga van): Windows

munkaállomást, domain tagsághoz kötött szolgáltatásokat, Novell hálózatot, UNIX fájlrendszereket Samba segítségével, postafiókot és ezt „Single Sign-On”-nak hívjuk. A módszer előnye nyilvánvaló, mint ahogy az is, milyen előnyökhöz jut, aki a jelszót megszerzi.

13.2.3. A megoldás: integrált szolgáltatás – EFS

A Windows 2000-től kezdve – az NTFS-be szorosan integrálva – a fenti problémákra nyújt megoldást az EFS (*Encrypting File System*), elrejtve a felhasználó elől az adatvédelem folyamatait. Az EFS a kernelben van, és a nem lapozható memóriakészletet használja a kulcsok tárolására, így azok nem kerülnek a lapozófájlba sem. (A kernel számára egyébként is előírható, hogy teljes mértékben „maradjon” a fizikai memóriában, ne lapozódjon ki.)

> Megjegyzendő, hogy az EFS távoli fájlkiszolgálókon is működik, csakhogy a hálózati forgalomba a plaintext adat kerül, ezért hálózati forgalomfigyeléssel és lehallgatással szemben nem nyújt védelmet! A hálózati forgalom titkosításához *SSL* vagy *IPSec* protokoll használata szükséges! (*storage security* ≠ *traffic security*)



A felhasználó az EFS alapértelmezett beállításaival további konfigurálás vagy rendszergazdai beavatkozás nélkül titkosíthat fájlokat[60]. Ha a felhasználónak nincs még kulcspárja, amivel az EFS-t használhatná, az operációs rendszer automatikusan generál neki egyet. A titkosítás kérhető egyedi fájlokra vagy teljes könyvtárra is. Utóbbi esetben a könyvtárban lévő összes meglévő, újonnan létrehozott fájlra valamint alkönyvtárra vonatkozik a titkosítás. A titkosítás be- illetve kikapcsolása igen egyszerű: a fájl vagy könyvtártulajdonságok között lehet ki- bekapcsolni az ábrán is látható választónégyzettel, de egy fájl nem lehet egyszerre tömörített és titkosított[61]. A titkosítási vagy megfejtési folyamat a felhasználó beavatkozása nélkül, a diszk adatforgalma közben, automatikusan történik. Az EFS felismeri a titkosított fájlokat, és a rendszer kulcstárából kikeresi a szükséges kulcsokat. Az EFS kulcskezelése a CryptoAPI-ra épül, annak legtöbb szolgáltatását használni tudja.

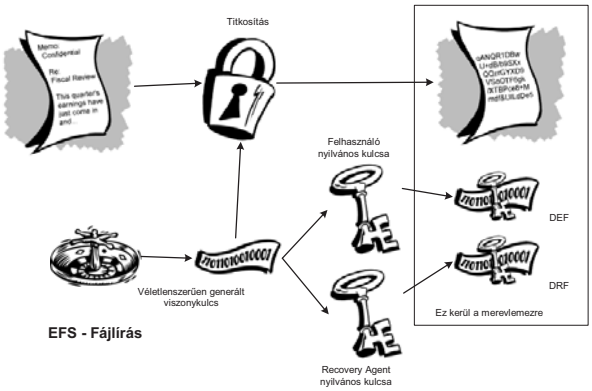
Az EFS működése

Az EFS működése hibrid kriptorendszeren alapul vagyis egyszerre használ szimmetrikus és nyilvános kulcsú algoritmust. Ezt részben a jobb teljesítmény érdekében teszi, részben pedig azért, hogy megoldást nyújtson a visszaállíthatóság problémájára, mert e nélkül egyfelhasználós lenne a kriptorendszer, vagyis csak az érhetné el a fájlt, akinek birtokában van a titkos kulcs. A Windows által alkalmazott megoldással azonban akárhány felhasználó hozzá-

[60] Egyetlen ajánlott feltétel, hogy legyen visszaállító ügynök (*recovery agent*), de ez alapértelmezésben mindig van: a lokális vagy a tartományi rendszergazda. Sajnos ez csak Windows 2000 alatt igaz, XP alatt nincs alapértelmezett RA!!!
[61] Az EFS szolgáltatásai parancssorból is elérhetők a CIPHER.EXE program használatával.

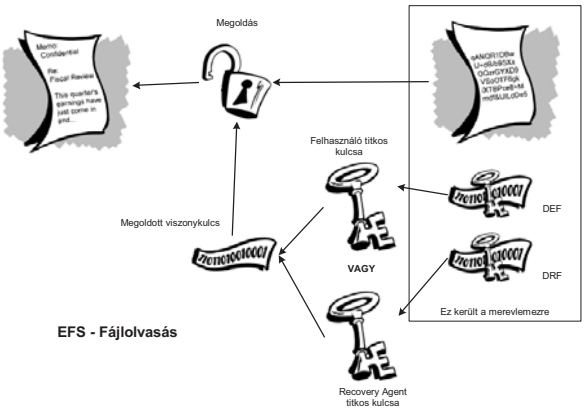
férhet a titkosított fájlhoz, mindenki a saját kulcsával[62]. A titkosítás a következő elemekkel és szereplőkkel történik:

- felhasználó
- a felhasználó RSA kulcspárja
- dokumentum
- titkosító kulcs (DES, DESX, AES algoritmusokhoz)
- visszaállító ügynök (aki az *„Encrypted Data Recovery Agents”* tagja a Security Policyban)
- a visszaállító ügynök RSA kulcspárja



Ezután, ha a felhasználó bekapcsolja egy fájl titkosítását, a Windows generál neki egy véletlenszerű titkosítókulcsot, majd azzal titkosítja a fájlt. A fájl mellé elhelyez még két adatot: a felhasználó nyilvános kulcsával titkosított DES-kulcsot és az *RA* nyilvános kulcsával titkosított DES-kulcsot. (A Windows terminológia ez előbbit DDF-nek vagy *Data Decryption Field-nek*, míg az utóbbit DRF-nek vagyis *Data Recovery Field-nek* hívja. A DES-kulcs pedig a FEK, vagyis *File Encryption Key*) Ezután csak azok férhetnek hozzá az állomány eredeti tartalmához, akik ki tudják nyitni e két mező valamelyikét: vagyis a tulajdonos és az *RA*. A többiek „*Access denied*” üzenetet kapnak.

Egészen addig, amíg az *RA* privát kulcsa megtalálható a rendszerben, ő is el tudja olvasni a fájlokat. Ez nem túl szép, úgyhogy a visszaállító ügynökök privát kulcsát „illik” kiexportálni úgy, hogy az exportálás után a privát kulcs törlését kérjük. Az exportált állományt utána (több) biztonságos adathordozón el kell zárni, és csak indokolt esetben szabad használni. A helyreállító ügynök – az esetleges visszaállítás során – csak a fájl véletlenszerűen generált DES-kulcsához férhet hozzá, a felhasználó kulcspárjához – és más azzal titkosított adathoz – természetesen nem. Amikor egy felhasználó kilép a

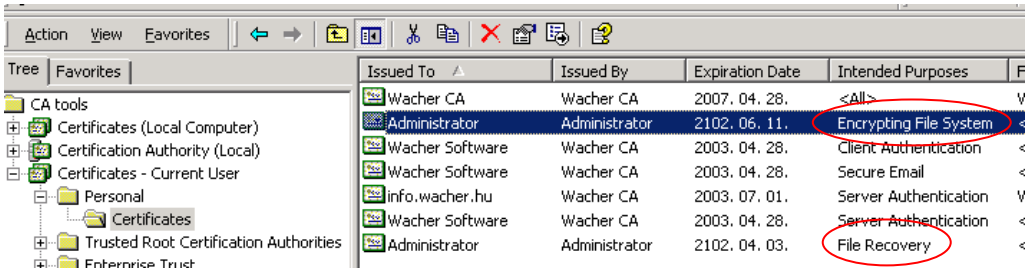


[62] Igazság szerint a Windows használhatna csak szimmetrikus titkosítást is, de ez elég sok kérdést felvetne...

szervezetből, de a titkos kulcsát nem törölte le, egyszerűbb az ő nevében bejelentkezni és úgy megszüntetni a tikosítást. Ez csak akkor járható, ha ismert a felhasználó jelszava, ugyanis a jelszó erőszakos felülírása (rendszergazdai joggal „Set password"), elérhetetlenné teszi az áhított RSA-kulcsokat... Amennyiben a felhasználó a titkos kulcsát elvitte magával, vagy elvesztette a biztonságos kulcstároló eszközét, akkor kell az *RA* titkos kulcsát importálni a rendszerbe, és így elvégezhető a fájlok visszaállítása. A titkosított fájlok másolásánál figyelni kell arra, hogy a másolat titkosítatlan lesz, amennyiben a cél fájlrendszere nem NTFS. *Másik fontos dolog, hogy* ***semmilyen rendszerfájlt sem szabad titkosítani****!!!*

A kulcsok helye

Gondolom másban is felmerült a kérdés, hogy a Windows hol tárolja a felhasználók kulcsait? Első ránézésre nem a legbiztonságosabb helyen: a felhasználói profilban. Erről magunk is meggyőződhetünk, feltéve ha van már EFS-kulcspárunk. (Ha legalább egy fájl titkosítását kértük már, akkor van.) A képen egy adminisztrátor szerepkörű felhasználó bizonyítványait láthatjuk, aki mellesleg *RA* is. Ezért a tanúsítványai között megtaláljuk mindkét tanúsítványt: a titkosításhoz és a visszaállításhoz szükségeset is.



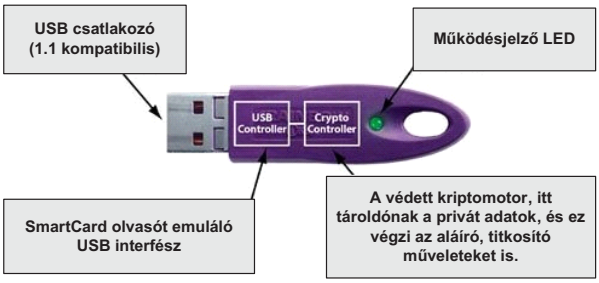
57. ábra A Windows tanúsítványtárolója

A tanúsítványok fizikai helye a registry-ben:
HKCU\Software\Microsoft\SystemCertificates és HKLM\Software\Microsoft\SystemCertificates

Az EFS bizonyítvány titkos és nyilvános kulcsot egyaránt tartalmaz, míg az FR csak nyilvánosat (mert az előző bekezdés szerint kiexportáltuk a titkosat...). Nyissuk meg az EFS bizonyítványt, és láthatjuk, hogy ott a privát kulcsunk is! Amennyiben kiexportáljuk és töröljük a titkos kulcsunkat, tartsuk szem előtt, hogy később minden esetben szükség lesz rá, ha egy általunk titkosított fájl tartalmát olvasni akarjuk! Felhasználók esetén okozhat némi bonyodalmat, hogy a Windows – valamilyen okból – szigorúan értelmezi a „*ha nincs neki, akkor generálunk*" elvet. Tételezzük fel, hogy a felhasználó egy fájl titkosítását kérte, de még nem volt neki EFS-kulcspárja! A Windows azonnal készít neki egyet, és titkosítja a fájlt. Ha a felhasználó kiexportálja és törli a privát kulcsát a rendszerből, a fájl nem nyitható meg többet. Amennyiben egy újabb fájl titkosítását kéri, a Windows nem kéri be a már létező a titkos kulcsot, nem kéri annak importálását sem, hanem generál egy új kulcspárt a felhasználó számára, hiszen nincs neki… Csakhogy az új kulcspárral nem nyitható meg az előző fájl, és az első kulcspárral sem nyitható meg az utóbbi! Bármelyik fájl megnyitásakor rendelkezésre kell állnia a hozzá tartozó kulcspárnak!

13.3. BIZTONSÁGOS KULCSTÁROLÓ ESZKÖZÖK

A kulcsok tárolására *biztonságos* kulcstároló eszköz kell. Mitől biztonságos egy ilyen eszköz? Röviden attól, hogy a benne lévő privát kulcsot semmilyen körülmények között sem lehet kiimádkozni belőle. Persze ezek az eszközök nemcsak makacsok, hanem okosak is. A képen látható USB eszköz egy *Rainbow iKey 2032* a következő feladatok elvégzésére képes:



- kulcsgenerálás (*on-board key generation*)[63]
- kulcstárolás (PGP kulcspárokat is képes kezelni)
- X.509 bizonyítványtárolás (összesen 32KB memóriában, a kulcsokkal együtt)
- titkosítás (DES CBC és ECB módban, DESX és TripleDES, RC2, RC4, RC5)
- digitális aláírás (SHA-1 és MD5 hashalgoritmusokkal, DSA szabvány szerint)
- *valódi* véletlenszám generálás (A FIPS142-2-nek megfelelő módon.)

Az eszköz lelkét egy Philips mikrokontroller adja (P8WE5032), ami a következő műveleti időket biztosítja:

- 3DES < 200µs
- DES < 100µs
- RSA-1024 tipikus műveleti idő: ~400ms

A mikrokontroller az alábbi legfontosabb PKI rendszereket támogatja[64]:

- RSA
- ElGamal
- DSS
- Diffie-Hellman
- Guillou-Quisquater
- ECC (elliptic curve)

Létezik FIPS142-1-level2 minősítésű változata is, amely arról ismerszik meg, hogy fekete színű. Az eszköz használatához különösebb bűvészkedés nem szükséges, mert virtuális SmartCard-olvasóként települ a számítógépünkre. A Windows is ilyen eszköznek látja a továbbiakban és minden olyan helyzetben használható, amelyben az intelligens kártyák. Drivere

[63] Az eszköz „lelke", a mikrokontroller legfeljebb 4096 bites RSA kulcsot tud kezelni, de a token csak 1024 biteset. A 2032 SmartCard változata támogatja a 2048 bites RSA kulcsot is.
[64] További részletek és még több technikai adat [49]-ben található, illetve a http://www.elp-internet.com linken.

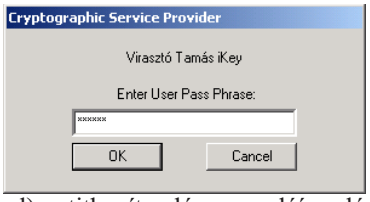
a CryptoAPI-n és a Smart Card API-n alapul. A kulcspárok és bizonyítványok eszközre juttatásának két módja van:

1. Vagy a kulcson generáljuk a kulcspárunkat
2. Vagy pkcs#12 fájlból importáljuk be.

Ha a kulcstároló eszközön generáltuk a kulcspárunkat, és a bizonyítványt utólag kapjuk meg, azt előbb az operációs rendszer bizonyítványtárolójába kell importálni, majd a kulcs használatát segítő (*token utility*) eszközzel kell a kulcstároló eszközre tenni. Lehetőség van egyébként a már az eszközön lévő bizonyítványok gépre való importálására is. Az ebben nyújtott szolgáltatások egyébként a token utility verziójától függnek.

> Egyébként is ajánlott a különböző verziójú meghajtóprogramokat és eszközöket az éles felhasználás előtt kipróbálni és összehangolni, mert a gyakorlati próba során volt olyan helyzet, hogy az egyik verziójú token utility nem tudta olvasni a másik verziójú szoftver által elhelyezett kulcsokat és bizonyítványokat…

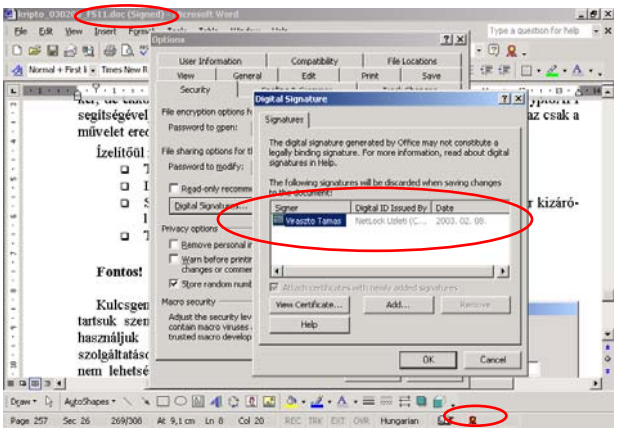
Mire használhatjuk?



Kulcstároló eszközünkkel lehetőségünk van minden olyan PKI szolgáltatást igénybe venni, amit a Windows támogat. Ne feledjük: a titkos kulcsot nem az operációs rendszer tárolja, hanem az eszköz! Minden alkalommal, amikor az adott alkalmazásnak szüksége lenne a privát kulcsra az eszköz jelszót kér, de ekkor sem adja ki a kulcsot! Az adott alkalmazásnak kell (a CryptoAPI segítségével) a titkosítandó vagy aláírandó adatot az eszközön áttuszkolni, és az csak a művelet eredményét adja vissza. Ízelítőül néhány szolgáltatás:

- Titkosított vagy digitálisan aláírt levél küldése.
- Digitálisan aláírt (Office) dokumentumok.
- Smart Card Login → tanúsítványalapú belépés Windowsba (akár kizárólagosan) Smart Card használatával.
- Tanúsítványalapú kliensazonosítás webhelyekhez.

A tanúsítvány használata során tartsuk szem előtt a jelszó bekérésének tényét, így ne használjuk a kulcstároló eszközt olyan PKI-szolgáltatásokhoz, ahol az emberi beavatkozás nem lehetséges: vagyis ne használjuk IPSec-nél, vagy például egy szerver SSL-bizonyítványához se!



A szabadság ára a szüntelen éberség.

Thomas Jefferson

(Ezékiel 3:17 alapján)

14. FÜGGELÉK

14.1. A TITKOSÍTÁS ÉRTÉKELÉSE ÉS ALAPVETŐ FELADATAI

14.1.1. Értékelési szempontok

A titkosság mértéke (Level of security)

Nehezen számszerűsíthető, de a legfontosabb tulajdonság. Általában a pillanatnyilag legjobbnak elfogadott támadási módszer műveleteinek számával jellemzik. Cél a minél nehezebben megfejthető üzenetek generálása. A titkosság szükséges mértékére a felhasználás helye ad útmutatást. Belátható, hogy egy baráti levelezés titkosításánál nem nagy baj, ha a használt algoritmus nem „bombabiztos”, viszont egy üzleti levelezésben vagy egy elektronikus üzleti, banki műveletnél minél jobb algoritmus használata a cél. Ez azonban nem jelenti azt, hogy ha a védendő adatok, információk nem „annyira” fontosak, arra kötelezően gyengébb algoritmust kellene használni. Napjaink nyilvánosan hozzáférhető algoritmusai is közelítik a megfejthetetlenség határát. Ezeket – megfelelő implementáció birtokában – bárki használhatja. Jelenleg egyetlen algoritmus biztonsága bizonyított, ez a *one-time-pad*. Az összes többiről csak feltételezzük, hogy biztonságos egészen addig, amíg az ellenkezője ki nem derül.

Teljesítmény (Performance)

A módszerek sebességére vonatkozó elvárásokat a felhasználás területe adja meg. Ha csak egy levelet vagy egyéb kisebb adatállományt szeretnénk titkosítani, az esetek legnagyobb részében mindegy, hogy a program fél vagy kettő másodpercen belül végez, a lényeg, hogy az eredmény jó legyen. Ha azonban egy merevlemez-partíciót vagy egy logikai meghajtót titkosítunk, igencsak tartania kell a titkosító algoritmusnak az előírt válaszidőt, bár az esetleges késlekedésből alapvetően nem származik probléma. Viszont ha egy telefonvonalat vagy hálózati kapcsolatot titkosítunk, erősen zavaró lehet, ha a titkosítást ellátó végberendezések áteresztő képessége, sebessége a megszokott, esetleg előírt mértéket nem éri el. Mérése általában *bit/sec* mértékegységgel történik.

Könnyű megvalósíthatóság (Easy implementation)

Ha egy algoritmus annyira bonyolult, hogy csak kis teljesítményre képes, a gyakorlati felhasználásban sem fog igazán elterjedni. Azonban meg kell különböztetni azt is, hogy hardver- vagy szoftver-környezetben implementálunk egy algoritmust, mert ez sem mindegy. Például a DES szoftveres megvalósításai sokkal lassabbak, mint a hardveres megoldások. Ez általában természetes, de lehetnek elvi okai is. A DES tele van bitszintű műveletekkel (bitcserékkel), ami szoftverben több órajeles memóriaciklusokat eredményez, míg hardverben huzalozással egyszerűen kivitelezhető, és egy bit megcserélése egy másikkal egyetlen órajelet sem igényel.

Az algoritmus komplexitása és az alkalmazott műveletek döntik el, hogy inkább szoftveres vagy inkább hardveres megvalósításai terjednek el. Ráadásul az egyszerűbb felépítésű algoritmusokat vizsgálni is könnyebb, így egy biztonságosnak elfogadott egyszerű felépítésű módszer feltehetően népszerűbb lesz, mint egy bonyolult.

Működési módok (Modes of operation)

Jelentősen befolyásolja egy algoritmus erejét az, hogy milyen módban működik, vagyis mi történik a nyílt vagy titkosított szöveggel a művelet előtt vagy után: például egy új titkosításhoz felhasználja az előző eredményét. Erről részletesen lásd „*7. A blokkos rejtjelezők működési módjai, és a folyamtitkosítók világa*” című fejezetet.

14.1.2. A titkosítás alapvető feladatai

Üzenet rejtjelezése, megfejtése (Encryption, decryption)

Olyan módon kell átalakítani az üzenetet (*rejtjelezés*), hogy annak információtartalmát csak egy kiegészítő információ birtokában lehessen megismerni (*megfejtés*). Ha a megfejtés a plusz információ nélkül történik, visszafejtésről vagy feltörésről beszélünk.

Digitális aláírások, időpecsétek (Digital signature, time stamp)

A hagyományos aláírás digitális megfelelője. A digitális aláírás egy olyan üzenet összeállítását jelenti, ami általában tartalmazza az aláírt adat jellemzőit, valamilyen ellenőrzésre alkalmas kiegészítést és az aláírás idejét, esetleg helyét, valamint az aláíró nevét. Így nemcsak az aláírót azonosítja, hanem hitelesítő eljárásokkal kiegészítve az aláírt adatokat is védi hamisítás és ismétlés ellen. További feladata a „letagadhatatlanság” (*nonrepudation*) biztosítása is: vagyis nem tagadhatja le később az aláíró az aláírás tényét. Ez a bizonyítás általában indirekt, mert azon alapul, hogy az aláírást csak az hozhatta létre, akinek birtokában van az aláíráshoz szükséges kulcs. Ennek a feltételezésnek igen jelentős következményei lehetnek, ezért kell komolyan venni például a kulcs érvénytelenítését, ha az elveszik.

Titokmegosztás, titokszétvágás (Secret sharing, secret splitting)

Valamilyen információt (jelszót, kulcsot) vagy jogosultságot úgy kell megosztani a résztvevők között, hogy azok mindegyikének (titokszétvágás) vagy egy meghatározott csoportjának (titokmegosztás) jelen kell lennie az információ rekonstruálásakor. A megosztás további célja lehet az információ biztonságos tárolása is, mert ha egy-egy titokdarab elveszik, az eredeti információ még visszaállítható. Ha az „elveszett” darabot valaki megtalálja, semmire sem tudja felhasználni. Bizonyos módszerekkel hozzáférési szintek szabályozása is megoldható.

Hitelesítés (Certification)

Az adatokat védi hamisítás, módosítás, üzenetkivonás vagy járulékos üzenet beiktatása ellen. Az eljárás igyekszik bizonyítani, hogy a tárolt adatok keletkezésük óta nem változtak meg, és a kapott információ minden eleme (tartalma, feladója, feladás ideje, stb.) hitelesnek tekinthető.

Partnerazonosítás – Identifikáció (Identification)

Alice és Bob kölcsönösen bemutatkoznak egymásnak, ezt valamilyen azonosító megadásával teszik, ami lehet név, felhasználói név vagy tetszőleges jelsorozat, de lehet akár valamilyen biometrikus azonosító is. Ez a pillanat még nem bizonyít semmit, csupán a „*Ki vagy?*” kérdésre adott válasznak felel meg. Ehhez kötődik az autentikáció folyamata, amelynek során majd a személyazonosság bizonyítást nyer.

Azonosító hitelesítése – Autentikáció (Authentication)

Alice és Bob kölcsönösen bizonyítják egymásnak kilétüket. A kommunikáló feleknek meg kell győződniük partnerük személyazonosságáról, meg kell teremteni a biztos kapcsolatot a partner valódi – a partnerazonosítás során tetszőleges azonosítóval azonosított – és virtuális lénye között. E megfeleltetésnek egyértelműnek kell lennie, mert csak ez lehet az alapja az egyéni felelősségre vonhatóságnak (*individual accountability*). E folyamat során a felek különböző eszközökkel, például hatósági igazolásokkal, okmányokkal, tanúsítványokkal megpróbálják igazolni kilétüket. Az autentikáció tehát az a folyamat, amely az azonosítás állítását bebizonyítja, és általában azon alapszik, hogy valaki ***tud valamit*** (melyik jelszót, PIN kódot kell használnia) ***és rendelkezik valamivel*** (kulccsal vagy felhasználói névvel, kártyával, stb.). Ezt kéttényezős autentikációnak (*two-factor authentication*) is hívják.

Jogosultság kiosztása és felhatalmazás, tulajdonságbirtoklás – Autorizálás (Authorization, attribute ownership)

Hozzáférések, jogosultságok felhasználóval történő összerendelése. Legalább kétféle értelmezése van, az egyik nézőpont szerint megelőzi az autentikációt, másik esetben követi azt.

1. Az első esetben a jogosultságok kiosztása *megelőzi* az azonosítási-hitelesítési folyamatot és mint ilyen, a felhasználó *rendszerbeli* entitásának jogokkal való felruházását jelenti. A partnerazonosítás és hitelesítés folyamata kapcsolja össze a valódi partnert a rendszerbeli „csontvázzal”, így a virtuális partner jogosultságai a kapcsolódást követően a valódi partner által testesülnek meg (de már korábban meghatározásra kerültek!).
2. A másik értelmezés szerint viszont az autorizáció *követi* a kapcsolat-felvételi folyamatot és nem megelőzi azt, mert az első pont szerint „előkészített jogosultságokat” csak a kapcsolódás után lehet a felhasználóhoz rendelni.

A lényegi eltérés a két szempont között, hogy az első a jogosultságok előkészítését tekinti a felhasználó felhatalmazásának (és az előkészített jogosultsággal valaki majd fel lesz ruházva), míg a második szempont azt a pillanatot tekinti felhatalmazásnak, amikor a felhasználó lehetőséget kap jogai gyakorlására.

Hozzáférés-védelem (Access-control)

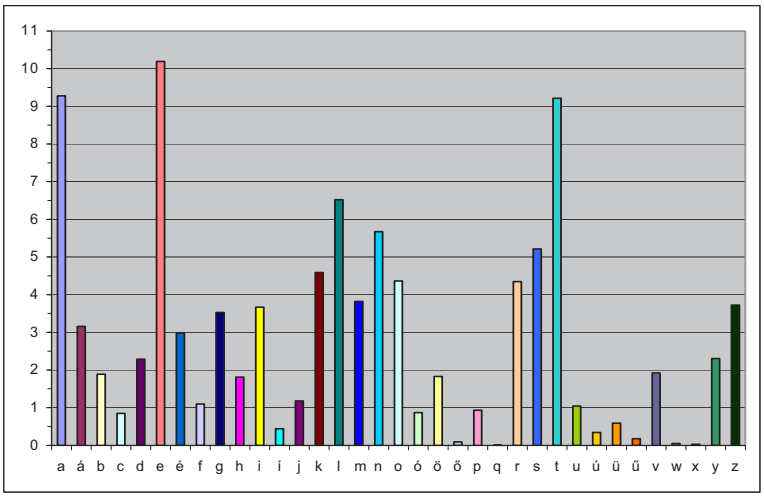
Garantálja, hogy egy adott erőforráshoz vagy adatokhoz csak a jogosult felhasználók férhetnek hozzá, illetve azon módosító műveleteket csak jogosult felhasználók végezhetnek. A jogosultságok halmazát az autorizáció határozza meg. Általában jelszavakat menedzselő, ellenőrző és hozzáférési jogosultságot kezelő részekből áll. Erős kriptográfia alkalmazása mellett megoldható, hogy mindenki mindenhez hozzáfér, legfeljebb kulcs hiányában nem érti meg a kapott

adatokat, illetve nem tud értelmezhető adatokat, utasításokat küldeni egy erőforrásnak. Ez a megoldás jelentősen leegyszerűsíti a jogosultságok kezelését, bár nagyobb technológiai követelményeket támaszt a megvalósításokkal szemben, mint a hagyományos, jelszóalapú rendszerek esetében. A jogosultságok ellenőrzésekor biztosítandó, hogy a hozzáférés ténye, módja és ideje a jogosultság esetleges változásával együtt naplózva legyen (vagy naplózható legyen), mert ez feltétele az egyéni felelősségre vonhatóságnak. Gyakran kapcsolódik a partnerazonosításhoz és azonosító-hitelesítéshez, hiszen amíg a partnerek be nem bizonyították egymásnak kilétüket, nincs értelme vizsgálni azt sem, hogy kinek mihez van joga.

14.2. BETŰELOSZLÁS EGY MAGYAR SZÖVEGBEN

Az alábbi táblázat alapján készült az *7. ábra* grafikonja. A statisztikához tizenegy magyar nyelvű regény elektronikus változata szolgált mintaként. A feldolgozás előtt a szövegekből eltávolítottam a számokat, az írásjeleket, a szóközöket. Így mintegy 4 500 000 karakter maradt. A táblázatban szereplő számok százalékot jelentenek.

q	0,003	ó	0,871	d	2,284	o	4,364
x	0,027	p	0,934	y	2,305	k	4,586
w	0,047	u	1,043	é	2,979	s	5,212
ő	0,092	f	1,098	á	3,159	n	5,671
ű	0,177	j	1,179	g	3,527	l	6,523
ú	0,343	h	1,808	i	3,667	t	9,213
í	0,439	ö	1,830	z	3,721	a	9,278
ü	0,589	b	1,887	m	3,819	e	10,189
c	0,849	v	1,922	r	4,346		



58. ábra ABC rendes betűgyakoriság magyar szövegben

14.3. ENIGMA-, ÉS HAGELIN-MÚZEUM

14.3.1. Enigma

Bal felső kép:

Az *Enigma G-312*

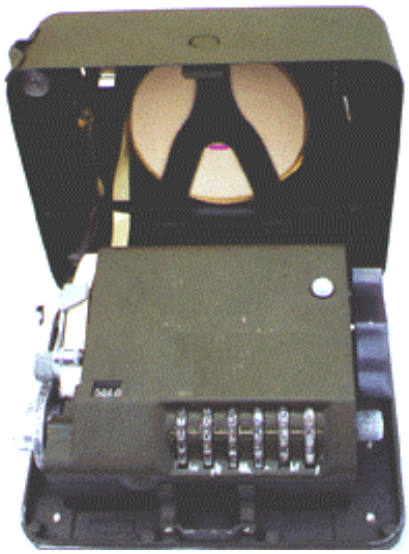
Jobb felső kép:

Az *Enigma G-312*
billentyűzete és lámpái

Jobb alsó kép:

Az *Enigma G-312*
egyik rotorja

14.3.2. Hagelin M209



Bal felső kép:

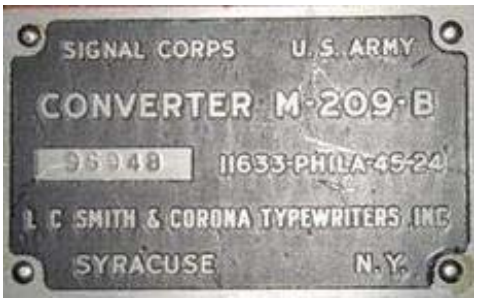
M209 használat közben

Jobb felső kép:

M209 beállításkor

Jobb alsó kép:

Az egyszerűen csak „konverternek” nevezett M209 adatlapja



14.3.3. Enigma vs. Hagelin

A második fejezetben mindkét eszközt hordozhatónak jellemeztem, noha jelentősen eltérnek a méreteik, és határozottan nem esenek egy súlycsoportba. Mégis mekkora az eltérés? Ekkora:

A kép lelőhelye: http://www.laugle.com/Crypto.html

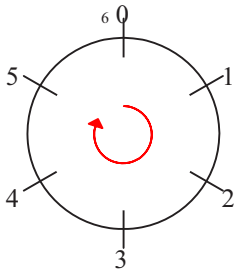
14.4. MODULÁRIS ARITMETIKA NAGYON DIÓHÉJBAN

A matematikai teljesség (és bizonyítás) igénye nélkül álljon itt néhány mondat a moduláris aritmetikáról azoknak, akik még nem hallottak róla, amolyan Móricka-féle gyorstalpaló tanfolyamként.

14.4.1. Moduláris aritmetika

A moduláris aritmetika során nem magukkal a (természetes) számokkal, hanem azok egy adott számmal (modulussal) történő osztási maradékaikkal számolunk. Megszoktuk, hogy a természetes számok a számegyenesen helyezkednek el, nincs legkisebb, nincs legnagyobb, de van negatív és van pozitív szám.

Nos, a moduláris aritmetika számai között nincs negatív szám és nincs végtelen sem, de van legnagyobb és legkisebb. Mi lenne, ha ennek a jól megszokott egyenesnek egy részét, mondjuk a [0..6[közötti szakaszát – a 0 véget beleértve, de a 6-ot nem – kivágnánk és a szakasz két végét „összefogva” egy kört formáznánk belőle?



1 + 2 = 3
3 + 3 = ?
4 + 5 = ?
4 + 11 = ?

A számegyenesen a 3+3-at szépen ki lehet számolni: ráállunk a 3-ast jelölő osztásra és a „+” jel miatt jobbra elindulunk, és lépünk 3 osztást. Eredményül a 6-os osztáson állunk meg, ami helyes eredmény. Most álljunk meg a számgyűrű alsó részén, a 3-as osztásnál és lépjünk három osztásnyit az nyíllal jelzett irányba! Hol álltunk meg? A 0-t jelölő osztáson? Helyes. Akkor mennyi itt most 4+5 ? Három? Helyes! És mennyi 4+11? Az is három!

Vegyük észre, hogy minden egyes alkalommal, amikor a 0-ra lépünk, az aktuális részeredményünk 6-tal csökken, és az eredmény minden egyes alkalommal [0..5] számok valamelyike! Vagyis minden eredmény a „hagyományos” eredmény hattal történő osztásának maradéka.

3 + 3 = 6 → 6/6=1 maradék 0
4 + 5 = 9 → 9/6=1 maradék 3
4 +11= 15 → 15/6=2 maradék 3

Ezeket a számításokat a következőképpen jelöljük (mod = moduló, a 6-ot modulusnak hívjuk):

$3 + 3 \bmod 6 = 6 \bmod 6 = 0$

$4 + 5 \bmod 6 = 9 \bmod 6 = 3$

$4 + 11 \bmod 6 = 15 \bmod 6 = 3$

Hasonlóan végezhető el például

- a kivonás – *a-b mod m* (ellentétes irányban haladva),
- a szorzás – *a×b mod m* (visszavezetve több összeadásra),
- az osztás – *a/b mod m* (az osztás kivonásra „szokás" visszavezetni, de itt és most akár szorzásra is visszavezethetjük: $\frac{a}{b} \bmod m = ab^{-1} \bmod m$ [65])
- a hatványozás – a^b *mod m* (szorzásra visszavezetve).

A moduláris aritmetikát néha egyszerűen óraszámtannak is hívják. Ha a fenti példában a körnek 12 osztása lenne, egy óra számlapját látnánk viszont, és egyértelmű lenne, hogy a 3 óra és a 15 óra ugyanannyi, legalábbis abban az értelemben, hogy mindkét esetben ugyanúgy állnak a mutatók. A mindennapi életben is mondjuk, hogy „3-kor találkozunk", és egyértelmű, hogy a találkozó időpontja 15:00. (Feltéve, ha nem éjszakai túrára készülünk, ebben az esetben a hajnali időpont a magától értetődő...) Az órán alapuló hasonlat segít megérteni azokat a tulajdonságokat, amelyek elsőre furcsának tűnhetnek. Órában és percben gondolkodva akárhány nap múlva (vagyis *n×24* óra múlva) ugyanennyi lesz az idő, mint most. És akárhány óra múlva (vagyis *n×60* perc múlva) ugyanannyi perc lesz, mint most.

A moduláris aritmetikában a számok nem nőnek a végtelenségig, sőt ez utóbbi fogalomnak nincs is értelme. A maradékképzés egyszerűen „lekaszálja" a műveletek eredményét, ha az túl nagyra nőtt. A végeredmény szempontjából a legtöbb esetben mindegy, hogy a maradékképzést a végeredményen vagy az operandusokon végezzük. Egy órás példa:

- 16 óra + 10 óra + 8 óra = (26 + 8) mod 24 = 34 mod 24 = 10 óra.
- 16 óra + 10 óra + 8 óra = 26 mod 24 + 8 mod 24 = 2 + 8 = 10 óra.

Néhány azonosság a moduláris aritmetikában:

1. $(a \pm b) \bmod m = (a \bmod m \pm b \bmod m) \bmod m$
2. $(a \times b) \bmod m = b \times (a \bmod m) \bmod m = (b \bmod m) \times (a \bmod m) \bmod m$
3. $(a \times a^{-1}) \bmod m = 1 \bmod m$
4. $(a^b) \bmod m = a \times (a^{b-1}) \bmod m = (a \bmod m) \times ((a^{b-1}) \bmod m) \bmod m$

[65] Itt b^{-1} nem más, mint *b multiplikatív inverze m*-re nézve. Lásd picit később: *14.6. Euler-féle φ függvény*

Érdekes módon a maradékok megőrzik az eredeti számok néhány tulajdonságát, de van azért jó néhány különbség is a megszokottakhoz képest, például:

- A 3. azonosságban a^{-1} csak akkor létezik, ha a és m relatív prímek (az a^{-1} becsületes neve „*a multiplikatív inverze m-re nézve*").
- A 4. azonosságban lévő hatványt kiszámolni nem nehéz, de ha $c=(a^b) \bmod m$ kifejezésből ki akarnánk számolni a kitevő b értékét, akkor $b=\log_a c \bmod m$ kifejezést kellene kiszámolni, ami már korántsem olyan „hétköznapi" feladat.
- Ha memóriában el kell tárolni egy egész számot, annak véges, tipikusan 1-4 bájt területet tartunk fenn. Mire elég ez az egész számok körében? Nem sokra, kb. 0-4 milliárd közé eső számokat tárolhatunk. Ha ennél nagyobb számokkal kell dolgozni, nagyobb tárterületet kell lefoglalni. És ha még nagyobbakkal, még többet. Persze ez nem mehet a végtelenségig így, ráadásul végzett műveletek közben túlcsordulás következhet be, amikor is a fenntartott tárterület nem elég, az eredmény tárolásához még több memória kellene. A moduláris aritmetikában minden szám tárolására legfeljebb akkora hely kell, mint a modulus tárolására. Ha a modulus 2 valamelyik hatványa, a modulus képzése egyszerűen AND művelettel végezhető.
- Nincs negatív szám, van viszont legkisebb (*nulla*) és legnagyobb (*modulus-1*).
- A természetes számok körében az 1,2,3,4,5,6,7,8,9,10,11,12,… sorozat a monoton növekvő, de ugyanez *mod 6*-ban már nem monoton és nem is mindig növekvő: 1,2,3,4,5,0,1,2,3,4,5,0,…

14.4.2. Kongruencia

Két egész számot (a és b) egymással kongruensnek mondunk egy m modulusra nézve, ha mindkettőt elosztva m-el ugyanazt a maradékot adják. Ennek jelölése:

$a \equiv b \pmod m$ szavakkal: „a kongruens b-vel az m modulusra nézve"
$9 \equiv 15 \pmod 6$ szavakkal: „9 kongruens 15-el a 6 modulusra nézve"

Az m-mel való osztáskor azonos maradékot adó valamennyi szám ugyanabba a *maradékosztály*ba tartozik. A *mod m* kongruencia valamennyi természetes számot besorolja a lehetséges m darab maradékosztály egyikébe. Az $m=2$ modulus például az összes természetes számot két csoportra osztja: párosakra és páratlanokra. Más szavakkal mondva a *mod m* kongruencia a természetes számok megszámlálhatóan végtelen halmazát m darab halmazra képzi le.

Megjegyzendő, hogy két szám kongruenciája *nem művelet*, hanem a két szám *egymáshoz való viszonyát fejezi ki*, reláció. Ráadásul ún. ekvivalencia típusú reláció, vagyis:

(1) reflexív: $a \equiv a \pmod m$,
(2) szimmetrikus: ha $a \equiv b \pmod m$, akkor $b \equiv a \pmod m$,
(3) tranzitív: ha $a \equiv b \pmod m$ és $b \equiv c \pmod m$, akkor $a \equiv c \pmod m$.

14.5. A KIS FERMAT-TÉTEL BIZONYÍTÁSA

Tétel: *Ha p prímszám, és nem osztója egy a egésznek, akkor* $a^{p-1}-1$ *osztható p-vel.*

$$a^{p-1}-1 \equiv 0 \pmod p$$
$$a^{p-1} \equiv 1 \pmod p$$

És a bizonyítása

Néhány matematikai bizonyításnál felmerül a kérdés: „miért?" Miért tesszük ezt vagy azt, miért pont azt és akkor? Miért pont így vagy úgy indulunk el? Erre a legegyszerűbb válasz, hogy nem kívánjuk megkeresni a bizonyításhoz vezető utat, csak meg szeretnénk ismerni azt. Az az ember, aki bizonyítást adott egy tételre vagy sejtésre, talán ötletes volt, és hamar rájött a megoldásra, de az is lehet, hogy bejárta a matematika labirintusát, mire a helyes utat megtalálta. Vagyis azért kell néha látszólag értelmetlen lépést tennünk, mert ma már tudjuk, hogy az elvezet a végső megoldáshoz is. Ennek szellemében vágjuk bele a kis Fermat-tétel bizonyításába, amelynek rengeteg verzója létezik, de most csak egyet mutatok be!

I. Írjuk fel az *a* szám alábbi szorzatait!

```
1a mod p,
2a mod p,
3a mod p,
4a mod p,
 ...,
(p-1)a mod p
```

II. Lássuk be, hogy nincs két egyforma eleme a fenti listának!

Mikor lenne két egyforma? Ha találnánk két, egymástól eltérő *x*-et és *y*-t úgy, hogy:

```
    xa ≡ ya  (mod p)
 xa-ya ≡ 0   (mod p)
a(x-y) ≡ 0   (mod p)
```

Mikor lehet ez igaz?

1. Ha a szorzat eredménye *p* többszöröse. Csakhogy ilyen eset nem lehet, mert az eredmény prímtényezős bontásában nincs kettő vagy több olyan prím, aminek szorzata *p*-t adná, hiszen ő maga is prím. A tényezők egyike sem *p* többszöröse: az *a* szám a feltétel szerint nem az, *x* és *y* pedig kisebb *p*-nél.
2. Ha az egyik tényező zérus. Ha az *a* szám nulla, akkor a tétel belátása nagyon egyszerű, hiszen nulla bármelyik hatványa 1. Ha az *a* szám nem nulla, akkor (*x*-*y*)-nak kell nullának lennie.

Vagyis *xa* csak akkor lehet egyenlő (kongruens) *ya*-val, ha *x* és *y* egyenlő. Mivel a fenti sorozatban az *a* szorzója minden esetben más és más (*1*-től *p-1*-ig), így minden szorzatnak különbözőnek kell lennie.

III. Lássuk be, hogy a nulla nincs az elemek között!

Ha lenne közöttük nulla, akkor lenne olyan *p-1*-nél kisebb *k* szám, amivel *a*-t megszorozva *p* többszörösét kapnák:

```
ka ≡ 0 (mod p)
```

Csakhogy ilyen eset nem lehet, mert az eredmény prímtényezős bontásában nincs kettő vagy több olyan prím, aminek szorzata *p*-t adná, hiszen ő maga is prím. A tényezők egyike sem *p* többszöröse: az *a* szám a feltétel szerint nem az, *k* pedig kisebb *p*-nél.

IV. Lássuk be, hogy 1-től p-1-ig minden elem előfordul!

Mivel *p-1* darab egymástól különböző eredményünk van, valamint nincs köztük a nulla és a legnagyobb *p-1*, így a fenti *a*-szorzatok maradékai az *1, 2, 3, 4, 5, ..., (p-1)* számok közül kerülnek ki valamilyen sorrendben.

V. És az utolsó lépések...

Most szorozzuk össze az egyenletek bal oldalait (az *a*-szorzatokat), majd a jobb oldalait (vagyis az *1,2,3,...,(p-1)* eredményeket):

```
1a*2a*3a*4a*5a*...(p-1)*a      ≡ 1*2*3*4*5*...*(p-1) (mod p)
1*2*3*4*5*...*(p-1) * a^(p-1) ≡ 1*2*3*4*5*...*(p-1) (mod p)
```

Most osszuk el mindkét oldalt 1-gyel, majd 2-vel, 3-mal, ..., (p-1)-gyel stb. (Ezt megtehetjük, mert mindegyikük reltív prím *p*-hez, így mindegyiknek van inverze *p*-re nézve) És íme, ami marad:

```
a^(p-1) ≡ 1 (mod p)
```

14.6. EULER-FÉLE φ FÜGGVÉNY

Az Euler-féle φ függvény minden *n* természetes számhoz a nála kisebb, hozzá relatív prím természetes számok számát rendeli, $\varphi(n)$-nel jelöljük. Ha egy *n* szám prímtényezős felbontása:

$$n = p_1^{k_1} * p_2^{k_2} * p_3^{k_3} * ... * p_m^{k_m}, \text{ akkor}$$

$$\varphi(n) = n * \left(1 - \frac{1}{p_1}\right) * \left(1 - \frac{1}{p_2}\right) * \left(1 - \frac{1}{p_3}\right) * ... * \left(1 - \frac{1}{p_m}\right)$$

Euler kongruenciatétele: $a^{\varphi(n)} \equiv 1 \bmod n$ *ha* $(a,n)=1$

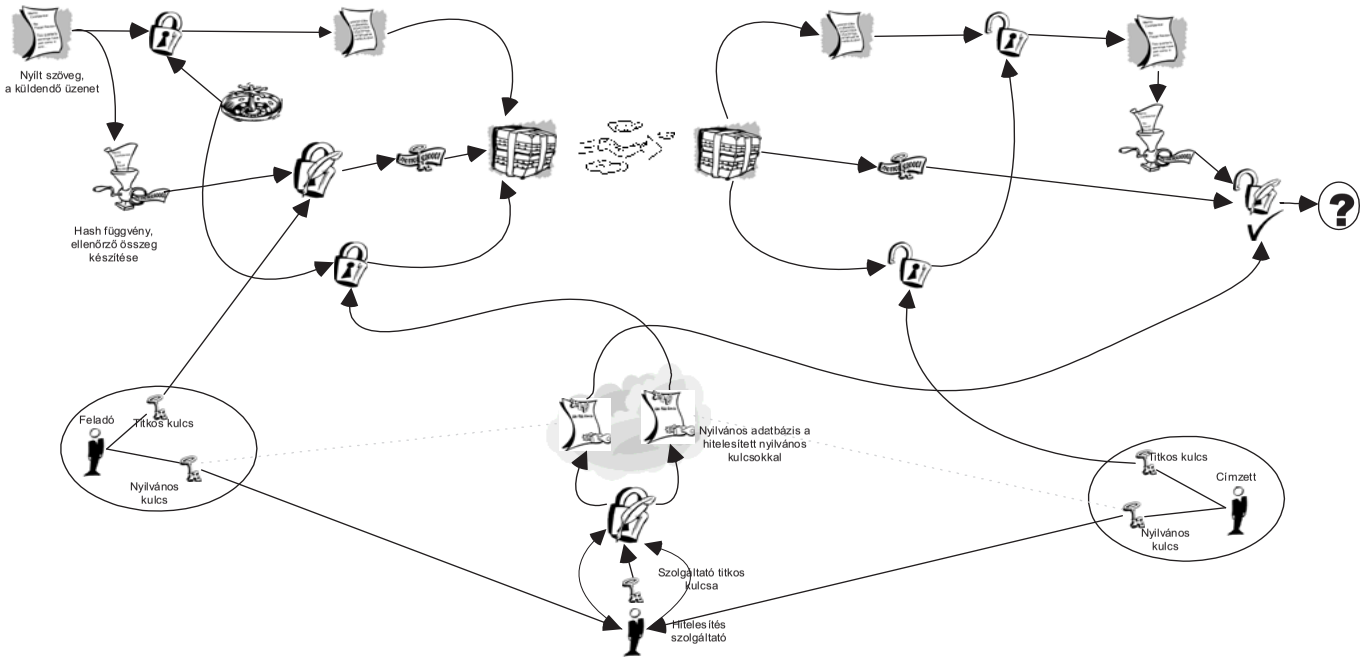
Modulo inverz számítása: $a^{-1} = a^{\varphi(n)-1} \bmod n$

Euler φ függvényének multiplikatív tulajdonsága:

$\varphi(nm) = \varphi(n) * \varphi(m)$, ha $(n,m)=1$, tehát *n* és *m* relatív prímek

Euler tételének van egy speciális esete, ami már ismerős: ha *n* prím. Ekkor $\varphi(n)=\varphi(p)=p-1$. Ha a tétel képletébe helyettesítünk, akkor $a^{\varphi(n)}=a^{p-1} \equiv 1 \bmod n$, ami a kis *Fermat*-tételhez vezet.

14.7. HIBRID KRIPTORENDSZER DIGITÁLIS ALÁÍRÁSSAL, VISZONYKULCCSAL – LOGIKAI VÁZLAT



14.8. SZABVÁNYOK ÖSSZEFOGLALÓ TÁBLÁZATA

Kategória	Név	RFC	ANSI	ITU	IEEE	FIPS	PKCS	ISO
Symmetric	DES		X3.92			46-2		
Symmetric	DES modes of operation		X3.106			81		
Symmetric	TDES modes of operation		X9.52					
Symmetric	AES					197		
Symmetric	CAST 128	2144						
Symmetric	CAST 256	2612						
Symmetric	RC2	2268						
Symmetric	RC5	2040						
Asymmetric	RSA	(2313), 2437	X9.44		P1363		#1	
Asymmetric	Elliptic Curve Cryptography		X9.62,X9.63		P1363		#13	
Asymmetric	Diffie-Hellman	2631	X9.30,X9.42		P1363		#3	
Asymmetric	ECDH		X9.63					
Certificate	Framework (X.509)		X9.45	X.509			(#6, #12)	
Certificate	Request format	2314					#10	
Certificate	Coding – ASN.1			X.680				8824-*
Certificate	Coding – BER, CER, DER			X.660,X.690				8825-*
HASH	MD2	1319						
HASH	MD4	(1186), 1320						
HASH	MD5	1321						
HASH	SHA-1	3174	X9.30			180-1		
Digital signature	ECDSA		X9.62,X9.25		P1363	186-2		14888-3
Digital signature	RSA		X9.31				#1	
Digital signature	DSS/DSA		X9.30			186-2		
Other	Crypto messages format	2315,2630(?)					#7	
Other	Password base encryption	2898					#5	

Név	PKCS leírása
PKCS #1	Az RSA alapú titkosítás és digitális aláírás szabványa
PKCS #2	(nem létezik, bekerült #1-be)
PKCS #3	A Diffie-Hellman kulcscserélő algoritmus szabványa
PKCS #4	(nem létezik, bekerült #1-be)
PKCS #5	Jelszóalapú kulcsgenerálás (password-base encryption)
PKCS #6	Tanúsítványok szabványa (az X.509 átvette)
PKCS #7	A titkosított üzenet szabványa
PKCS #8	A magánkulcs tárolásának szabványa
PKCS #9	A többi PKCS-szabvány attribútumainak gyűjteménye
PKCS #10	A tanúsítványigénylés szabványos formátuma
PKCS #11	Smart Card API
PKCS #12	Bizalmas adatok, kulcsok és tanúsítványok tárolási formátuma
PKCS #13	Az EC titkosítási algoritmus szabványa (lényegében elhalt?)

Rengeteg további kriptográfiai algoritmus adatait találhatjuk meg itt: [URL42]. (Kié az algoritmus szabadalma, mikor és hol publikálták? Kulcs- és blokkhosszadatok és sok más...)

14.9. POLLARD-ρ ALGORITMUS – UBASIC IMPLEMENTÁCIÓ

```
 10   'RHO version 2.3
 20   '
 30   '  Factorization with rho method - UBASIC implementation [66]
 40   '
 50   word -120:point -2
 60   print "Pseudo prime decomposition (RHO method)"
 70   '
 80   '  for each number
 90   '
100   print:input "Input an integer =";N
110   if N=0 then end
120   gosub *Factoring_RHo(N)
130   goto 100
140   '
150   '  main routine
160   '
170   *Factoring_Rho(N)
180   dim SS(20),PD(20):'for intermediate divisors and prime factors
190   NN=N:S1%=1:SP%=0:DP%=0
200   '
210   '  divide by small primes
220   '
230   D=prmdiv(N):if D=0 goto 320
240   if N=D goto 270
250   print D;"*";:N=N\D:S1%=0
260   goto 230
270   if S1% then print N;"is a prime":goto 740
280   print N:goto 740
290   '
300   '  if N has a big factor
310   '
320   print
330   '
340   '  pseudo primality check
350   '
360   if fnPrpChk(N)=0 then clr S1%:goto 470
370   '
380   '  store in PD if pseudo prime
390   '
400   PD(DP%)=N:inc DP%
410   if SP% then dec SP%:N=SS(SP%):goto 360
420   goto 530
430   '
440   '  if composite then find a factor
450   '  with rho method
460   '
470   N1=fnRHO(N):N2=N\N1
480   if N1>N2 then swap N1,N2
490   SS(SP%)=N2:inc SP%:N=N1:goto 360
```

[66] Az UBASIC egy olyan BASIC variáns, amely vígan számol 7-8000 bites számokkal is. A rengeteg implementált matematikai függvény és algoritmus mellett ismeri a komplex számokat is. Bővebb infó [URL74]-en található.

```
  510   '  display results
  520   '
  530   if S1% then print N;"is a pseudo prime":goto 740
  540   '
  550   '  display small factors
  560   '
  570   N=NN:print:print N;"="
  580   D=prmdiv(N):if D=0 goto 610
  590   print D;"*";:N=N\D
  600   goto 580
  610   if DP%=1 then print PD(0):goto 740
  620   '
  630   '  display big factors after re_ordering
  640   '
  650   clr SW%
  660   for I=0 to DP%-2
  670   if PD(I)>PD(I+1) then swap PD(I),PD(I+1):SW%=1
  680   next
  690   if SW% goto 650
  700   for I=0 to DP%-2
  710   print PD(I);"*";
  720   next
  730   print PD(DP%-1)
  740   return
  750   '
  760   ' pseudo primality test w.r.t. 2,3,5,7
  770   '
  780   fnPrpChk(N)
  790   local OK%
  800    if or{(modpow(2,(N-1)\2,N)-kro(2,N))@N,
  810         :(modpow(3,(N-1)\2,N)-kro(3,N))@N,
  820         :(modpow(5,(N-1)\2,N)-kro(5,N))@N,
  830         :(modpow(7,(N-1)\2,N)-kro(7,N))@N}
  840    :then OK%=0
  850    :else OK%=1
  860   return(OK%)
  870   '
  880   'RHO METHOD ver 1.2 (function version)
  890   ' originally  by J.M.Pollard, inplemented by Y.KIDA
  910   '
  920   fnRho(N)
  930   local A,X1,X2,I%,GD,Ct
  940    repeat
  950       inc A
  960       X1=6:X2=(X1^2+A)@N
  970       repeat
  980          inc Ct:print chr(13);Ct;
  990          GD=1
 1000          for I%=1 to 100
 1010             GD=GD*(X2-X1)@N:X1=(X1^2+A)@N:X2=((X2^2@N+A)^2+A)@N
 1020          next
 1030          GD=gcd(GD,N)
 1040       until GD>1
 1050    until GD<N
 1060   return(GD)
 1070   '                end of program
```

14.10. A5/1 – GSM TITKOSÍTÁS – C IMPLEMENTÁCIÓ

```
/* A pedagogical implementation of A5/1.
 * Copyright (C) 1998-1999: Marc Briceno, Ian Goldberg, and David Wagner
 *
 * The source code below is optimized for instructional value and clarity.
 * Performance will be terrible, but that's not the point.
 * The algorithm is written in the C programming language to avoid ambiguities
 * inherent to the English language. Complain to the 9th Circuit of Appeals
 * if you have a problem with that.
 *
 * This software may be export-controlled by US law.
 *
 * This software is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.
 * Copyright remains the authors' and as such any Copyright notices in
 * the code are not to be removed.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
 * FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE
 * AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * The license and distribution terms for any publicly available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution license
 * [including the GNU Public License.]
 *
 * Background: The Global System for Mobile communications is the most widely
 * deployed cellular telephony system in the world. GSM makes use of
 * four core cryptographic algorithms, neither of which has been published by
 * the GSM MOU. This failure to subject the algorithms to public review is all
 * the more puzzling given that over 100 million GSM
 * subscribers are expected to rely on the claimed security of the system.
 *
 * The four core GSM algorithms are:
 * A3           authentication algorithm
 * A5/1         "strong" over-the-air voice-privacy algorithm
 * A5/2         "weak" over-the-air voice-privacy algorithm
 * A8           voice-privacy key generation algorithm
 *
```

```
 * In April of 1998, our group showed that COMP128, the algorithm used by the
 * overwhelming majority of GSM providers for both A3 and A8 functionality was
 * fatally flawed and allowed for cloning of GSM mobile phones.
 * Furthermore, we demonstrated that all A8 implementations we could locate,
 * including the few that did not use COMP128 for key generation, had been
 * deliberately weakened by reducing the keyspace from 64 bits to 54 bits.
 * The remaining 10 bits are simply set to zero!
 *
 * See http://www.scard.org/gsm for additional information.
 *
 * The question so far unanswered is if A5/1, the "stronger" of the two
 * widely deployed voice-privacy algorithm is at least as strong as the
 * key. Meaning: "Does A5/1 have a work factor of at least 54 bits"?
 * Absent a publicly available A5/1 reference implementation, this question
 * could not be answered. We hope that our reference implementation below,
 * which has been verified against official A5/1 test vectors, will provide
 * the cryptographic community with the base on which to construct the
 * answer to this important question.
 *
 * Initial indications about the strength of A5/1 are not encouraging.
 * A variant of A5, while not A5/1 itself, has been estimated to have a
 * work factor of well below 54 bits. See http://jya.com/crack-a5.htm for
 * background information and references.
 *
 * With COMP128 broken and A5/1 published below, we will now turn our attention
 * to A5/2. The latter has been acknowledged by the GSM community to have
 * been specifically designed by intelligence agencies for lack of security.
 *
 * We hope to publish A5/2 later this year.
 *
 * -- Marc Briceno      <marc@scard.org>
 *    Voice:            +1 (925) 798-4042
 */

#include <stdio.h>

/* Masks for the three shift registers */
#define R1MASK  0x07FFFF /* 19 bits, numbered 0..18 */
#define R2MASK  0x3FFFFF /* 22 bits, numbered 0..21 */
#define R3MASK  0x7FFFFF /* 23 bits, numbered 0..22 */

/* Middle bit of each of the three shift registers, for clock control */
#define R1MID   0x000100 /* bit 8 */
#define R2MID   0x000400 /* bit 10 */
#define R3MID   0x000400 /* bit 10 */

/* Feedback taps, for clocking the shift registers. These correspond to the
 * primitive polynomials */
#define R1TAPS  0x072000 /* bits 18,17,16,13 */
#define R2TAPS  0x300000 /* bits 21,20 */
#define R3TAPS  0x700080 /* bits 22,21,20,7 */

/* Output taps, for output generation */
#define R1OUT   0x040000 /* bit 18 (the high bit) */
#define R2OUT   0x200000 /* bit 21 (the high bit) */
#define R3OUT   0x400000 /* bit 22 (the high bit) */
```

```
typedef unsigned char byte;
typedef unsigned long word;
typedef word bit;

/* Calculate the parity of a 32-bit word, i.e. the sum of its bits modulo 2 */
bit parity(word x) {
        x ^= x>>16;
        x ^= x>>8;
        x ^= x>>4;
        x ^= x>>2;
        x ^= x>>1;
        return x&1;
}

/* Clock one shift register */
word clockone(word reg, word mask, word taps) {
        word t = reg & taps;
        reg = (reg << 1) & mask;
        reg |= parity(t);
        return reg;
}

/* The three shift registers.  They're in global variables to make the code
 * easier to understand.
 * A better implementation would not use global variables. */
word R1, R2, R3;

/* Look at the middle bits of R1,R2,R3, take a vote, and
 * return the majority value of those 3 bits. */
bit majority() {
        int sum;
        sum = parity(R1&R1MID) + parity(R2&R2MID) + parity(R3&R3MID);
        if (sum >= 2)
                return 1;
        else
                return 0;
}

/* Clock two or three of R1,R2,R3, with clock control according to their
 * middle bits. Specifically, we clock Ri whenever Ri's middle bit
 * agrees with the majority value of the three middle bits.*/
void clock() {
        bit maj = majority();
        if (((R1&R1MID)!=0) == maj) R1 = clockone(R1, R1MASK, R1TAPS);
        if (((R2&R2MID)!=0) == maj) R2 = clockone(R2, R2MASK, R2TAPS);
        if (((R3&R3MID)!=0) == maj) R3 = clockone(R3, R3MASK, R3TAPS);
}

/* Clock all three of R1,R2,R3, ignoring their middle bits.
 * This is only used for key setup. */
void clockallthree() {
        R1 = clockone(R1, R1MASK, R1TAPS);
        R2 = clockone(R2, R2MASK, R2TAPS);
        R3 = clockone(R3, R3MASK, R3TAPS);
}
```

```
/* Generate an output bit from the current state.
 * You grab a bit from each register via the output generation taps;
 * then you XOR the resulting three bits. */
bit getbit() {
        return parity(R1&R1OUT)^parity(R2&R2OUT)^parity(R3&R3OUT);
}

/* Do the A5/1 key setup.
 * This routine accepts a 64-bit key and a 22-bit frame number. */

void keysetup(byte key[8], word frame) {
        int i;
        bit keybit, framebit;

        /* Zero out the shift registers. */
        R1 = R2 = R3 = 0;

        /* Load the key into the shift registers, LSB of first byte of key
         * array first, clocking each register once for every key bit loaded.
         * (The usual clock control rule is temporarily disabled.) */
        for (i=0; i<64; i++) {
                /* always clock */
                clockallthree();

                /* The i-th bit of the key */
                keybit = (key[i/8] >> (i&7)) & 1;

                R1 ^= keybit; R2 ^= keybit; R3 ^= keybit;
        }

        /* Load the frame number into the shift registers, LSB first, clocking
         * each register once for every key bit loaded.  (The usual clock
         * control rule is still disabled.) */
        for (i=0; i<22; i++) {
                /* always clock */
                clockallthree();

                /* The i-th bit of the frame # */
                framebit = (frame >> i) & 1;

               R1 ^= framebit; R2 ^= framebit; R3 ^= framebit;
        }


        /* Run the shift registers for 100 clocks to mix the keying material
         * and frame number together with output generation disabled, so that
         * there is sufficient avalanche. We re-enable the majority-based clock
         * control rule from now on. */
        for (i=0; i<100; i++) {
                clock();
        }

        /* Now the key is properly set up. */
}
```

```
/* Generate output.  We generate 228 bits of keystream output.
 * The first 114 bits is for the A->B frame; the next 114 bits is for the
 * B->A frame.  You allocate a 15-byte buffer for each direction, and this
 * function fills it in. */
void run(byte AtoBkeystream[], byte BtoAkeystream[]) {
        int i;

        /* Zero out the output buffers. */
        for (i=0; i<=113/8; i++)
                AtoBkeystream[i] = BtoAkeystream[i] = 0;

        /* Generate 114 bits of keystream for the
         * A->B direction.  Store it, MSB first. */
        for (i=0; i<114; i++) {
                clock();
                AtoBkeystream[i/8] |= getbit() << (7-(i&7));
        }

        /* Generate 114 bits of keystream for the
         * B->A direction.  Store it, MSB first. */
        for (i=0; i<114; i++) {
                clock();
                BtoAkeystream[i/8] |= getbit() << (7-(i&7));
        }
}

/* Test the code by comparing it against a known-good test vector. */
void test() {
        byte key[8] = {0x12, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF};
        word frame = 0x134;
        byte goodAtoB[15] = { 0x53, 0x4E, 0xAA, 0x58, 0x2F, 0xE8, 0x15,
                              0x1A, 0xB6, 0xE1, 0x85, 0x5A, 0x72, 0x8C, 0x00 };
        byte goodBtoA[15] = { 0x24, 0xFD, 0x35, 0xA3, 0x5D, 0x5F, 0xB6,
                              0x52, 0x6D, 0x32, 0xF9, 0x06, 0xDF, 0x1A, 0xC0 };
        byte AtoB[15], BtoA[15];
        int i, failed=0;

        keysetup(key, frame);
        run(AtoB, BtoA);

        /* Compare against the test vector. */
        for (i=0; i<15; i++)
                if (AtoB[i] != goodAtoB[i]) failed = 1;
        for (i=0; i<15; i++)
                if (BtoA[i] != goodBtoA[i]) failed = 1;

        /* Print some debugging output. */
        printf("key: 0x");
        for (i=0; i<8; i++) printf("%02X", key[i]);
        printf("\n");
        printf("frame number: 0x%06X\n", (unsigned int)frame);
        printf("known good output:\n");
        printf(" A->B: 0x");
        for (i=0; i<15; i++) printf("%02X", goodAtoB[i]);
        printf("  B->A: 0x");
        for (i=0; i<15; i++) printf("%02X", goodBtoA[i]);
```

```
        printf("\n");
        printf("observed output:\n");
        printf(" A->B: 0x");
        for (i=0; i<15; i++) printf("%02X", AtoB[i]);
        printf("  B->A: 0x");
        for (i=0; i<15; i++) printf("%02X", BtoA[i]);
        printf("\n");

        if (!failed) {
                printf("Self-check succeeded: everything looks ok.\n");
                return;
        } else {
                /* Problems! The test vectors didn't compare*/
                printf("\nI don't know why this broke contact the authors.\n");
                exit(1);
        }
}

int main(void) {
        test();
        return 0;
}
```

14.11. ALAPÉRTELMEZÉSBEN TELEPÍTETT BIZONYÍTVÁNYOK

Egy *Windows 2000 Server+SP2* installálásakor a következő bizonyítványok kerülnek telepítésre alapértelmezésben (vagyis a „felhasználó tudta nélkül”)

ROOT Certificates

```
ABA.ECOM Root CA
Autoridad Certificadora de la Asociacion Nacional del Notariado Mexicano, A.C.
Autoridad Certificadora del Colegio Nacional de Correduria Publica Mexicana, A.C.
Baltimore EZ by DST
Belgacom E-Trust Primary CA
C&W HKT SecureNet CA Class A
C&W HKT SecureNet CA Class B
C&W HKT SecureNet CA Root
C&W HKT SecureNet CA SGC Root
CA 1
Certiposte Classe A Personne
Certiposte Serveur
Certisign - Autoridade Certificadora - AC2
Certisign - Autoridade Certificadora - AC4
Certisign Autoridade Certificadora AC1S
Certisign Autoridade Certificadora AC3S
Class 1 Primary CA
Class 1 Public Primary Certification Authority
Class 2 Primary CA
Class 2 Public Primary Certification Authority
Class 3 Primary CA
Class 3 Public Primary Certification Authority
Class 3P Primary CA
Class 3TS Primary CA
```

```
Copyright (c) 1997 Microsoft Corp.
DST (ANX Network) CA
DST (NRF) RootCA
DST (UPS) RootCA
DST RootCA X1
DST RootCA X2
DST-Entrust GTI CA
DSTCA E1
DSTCA E2
Deutsche Telekom Root CA 1
Deutsche Telekom Root CA 2
EUnet International Root CA
Entrust.net Secure Server Certification Authority
Equifax Secure Certificate Authority
Equifax Secure Global eBusiness CA-1
Equifax Secure eBusiness CA-1
Equifax Secure eBusiness CA-2
FESTE, Public Notary Certs
FESTE, Verified Certs
FNMT Clase 2 CA
First Data Digital Certificates Inc. Certification Authority
GTE CyberTrust Global Root
GTE CyberTrust Root
GlobalSign Root CA
IPS SERVIDORES
Microsoft Authenticode(tm) Root Authority
Microsoft Root Authority
NO LIABILITY ACCEPTED, (c)97 VeriSign, Inc.
NetLock Expressz (Class C) Tanusitvanykiado
NetLock Kozjegyzoi (Class A) Tanusitvanykiado
NetLock Uzleti (Class B) Tanusitvanykiado
PTT Post Root CA
SERVICIOS DE CERTIFICACION - A.N.C.
SIA Secure Client CA
SIA Secure Server CA
Saunalahden Serveri CA
Secure Server Certification Authority
SecureNet CA Class A
SecureNet CA Class B
SecureNet CA Root
SecureNet CA SGC Root
SecureSign RootCA1
SecureSign RootCA2
SecureSign RootCA3
Swisskey Root CA
TC TrustCenter Class 1 CA
TC TrustCenter Class 2 CA
TC TrustCenter Class 3 CA
TC TrustCenter Class 4 CA
TC TrustCenter Time Stamping CA
Thawte Personal Basic CA
Thawte Personal Freemail CA
Thawte Personal Premium CA
Thawte Premium Server CA
Thawte Server CA
Thawte Timestamping CA
UTN - DATACorp SGC
```

```
UTN-USERFirst-Client Authentication and Email
UTN-USERFirst-Hardware
UTN-USERFirst-Network Applications
UTN-USERFirst-Object
VeriSign Commercial Software Publishers CA
VeriSign Individual Software Publishers CA
VeriSign Trust Network
Xcert EZ by DST
http://www.valicert.com/
```

INTERMEDIATE Certificates

```
GTE CyberTrust Root
GlobalSign Root CA
MS SGC Authority
Microsoft Windows Hardware Compatibility
SecureNet CA SGC Root
Thawte Premium Server CA
Thawte Server CA
UTN - DATACorp SGC
VeriSign Class 1 CA Individual Subscriber-Persona Not Validated
VeriSign Class 2 CA - Individual Subscriber
www.verisign.com/CPS Incorp.by Ref. LIABILITY LTD.(c)97 VeriSign
```

14.12. NÉHÁNY SZÁM ÉS NAGYSÁGREND

Álljon itt néhány szám az összehasonlíthatóság kedvéért, hogy lássuk, milyen nagy a nagy, milyen sok a sok, és milyen kevés a kevés[67]:

Amíg a Napból nova lesz	2^{30} év	10^{9} év
A Föld becsült életkora	2^{32} év	$4{,}5\times10^{9}$ év
A Naprendszer becsült életkora	2^{32} év	6×10^{9} év
A világegyetem becsült életkora	2^{33} év	10^{10} év
A Föld atomjainak becsült száma	2^{166}	10^{50}
Planck idő	2^{-142}	10^{-43} sec
Egy 500Mhz-es processzor óraciklusainak száma évente	2^{54}	$1{,}57\times10^{16}$
Meddig lehet elszámolni 64 biten? (18446744073709551615-ig)	2^{64}	$1{,}8\times10^{19}$
Meddig lehet elszámolni 128 biten? (340282366920938463463374607431768211455-ig)	2^{128}	$3{,}4\times10^{38}$
Meddig lehet elszámolni 256 biten? (115792089237316195423570985008687907853269984665640564039457584007913129639935-ig)	2^{256}	$1{,}2\times10^{77}$

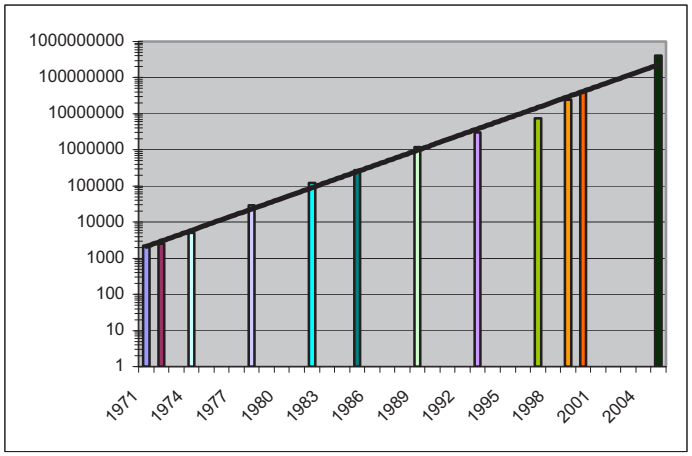
[67] A kettőhatványok felírásánál szándékosan törekedtem a 2^x formára, ezért az így kifejezett adatok pontatlanok lehetnek.

Egy nap	86 400	2^{16}, $8{,}6\times10^{4}$ másodpercből áll
Egy hét	604 800	2^{19}, $6{,}0\times10^{5}$ másodpercből áll
Egy hónap	2 592 000	2^{21}, $2{,}6\times10^{6}$ másodpercből áll
Egy év	31 557 600	2^{25}, $3{,}1\times10^{7}$ másodpercből áll
Fénysebesség (m/s)	299 792 458	2^{28}, $3{,}0\times10^{8}$
Fényév (m)		2^{53}, $9{,}4\times10^{15}$

Egy számszerű példa: 0-tól 2^{256}-ig egyesével elszámolni $1{,}1\times10^{34}$ másodpercbe telne, ha egy lépés ideje a Planck-idő lenne (ez a legkisebb időtartam, amelynek még van valami jelentése a fizikában). Csakhogy ez az idő több, mint a Világegyetem becsült életkorának a kőbe...

14.13. MOORE TÖRVÉNYE

A számítási kapacitások, pontosabban a processzorok teljesítménynövekedésére és felépítésére vonatkozó állítás[68]: ***az integrált áramköri lapkákon elhelyezett tranzisztorok száma (→ a teljesítménye) 18 hónap alatt megduplázódik***. *Gordon Moore* (Intel alapítótag) „törvénye" – melyet 1965-ben, néhány évvel az első integrált áramköri lapka megjelenése után publikált [45] – egyelőre jól megállja a helyét, és egyesek szerint ez a növekedés a „végtelenségig" fog tartani. Mások, például *Stan Williams* (a HP vezető tervezője) szerint ez az ütem a közeljövőben lassulni fog, mert az egyre kisebb áramköri lapkákat eredményező szilíciumalapú MOS-FET technológia már nincs messze a kvantummechanika által állított határoktól. A GaAs (gallium-arzenid) alapú technológia is hasonló problémákkal küzd. (A GaAs technológia, mely a szuperszámítógépek egyik jellemző technológiája, a Si alapú lapkákhoz viszonyítva körülbelül 20-szor gyorsabb áramköröket eredményezhet, ugyanannyi tranzisztorhoz azonban 20-szor nagyobb felület kell.) [10]



[68] Egyesek *tapasztalati* törvénynek tartják Moore kijelentését, de ez nem (egészen) igaz! Moore 1965-ben, gyakorlatilag az első IC lapkák megjelenése után jelentette ki azt, amit ma Moore törvényének ismerünk, így mindössze néhány év állt rendelkezésre a „tapasztalatszerzéshez". Csakhogy '65-ben Moore már '75-re vonatkozó jóslatokat is megfogalmazott. Ekkor az első mikroprocesszor még tervekben sem élt.

2000. végén az Intel bemutatta a közeljövőben bevezetésre kerülő 0,070 mikronos technológiához kifejlesztett CMOS tranzisztorokat: a 70 nm-es tranzisztor hatásos kapuhossza 3 nm, míg a szigetelőréteg néhány atom vastagságú. 2003 végén elkészültek az első 0,065 mikronos SRAM chipek is. Ilyen eszközökkel (és ilyen fejlődéssel) a tervek szerint néhány éven belül olyan lapkák fejleszthetők ki, melyeken 500 millió tranzisztor van (e processzorok várható kódneve: Madison), akár 10 GHz órajel is hajthatja őket, és mindemellett kevesebb, mint 1 V tápfeszültséget igényelnek[69]. Ez a futurisztikus fejlődés még engedelmeskedik Moore törvényének, de vajon hol a határ? Néhány számszerű múltbéli adat látható az alábbi táblázatban (és a fenti grafikonon) az Intel processzorok fejlődési adatait figyelembe véve. [URL31]

Processzor	Kibocsátás éve	Jellemző órajel	Tranzisztorok (ezer db)	
4004	1971	108 kHz	2,2	Busicom
8008	1972	4 Mhz	2,5	Mark-8
8080	1974	5 MHz – 10 Mhz	5,0	Altair
8086-8088	1978	5 MHz – 10 Mhz	29,0	IBM PC
286	1982	6 Mhz – 12 Mhz	120,0	
Intel386™ processzor	1985	16 MHz – 33 MHz	275,0	32bit, multitasking
Intel486™ DX processzor	1989	25 Mhz – 100 MHz	1 180,0	math co-processor
Pentium® processzor	1993	60 MHz – 233 MHz	3 100,0	
Pentium® Pro processzor	1995	60 MHz – 233 MHz	5 500,0	
Pentium II processzor	1997	200 MHz – 400 MHz	7 500,0	
Pentium III processzor	1999	450 MHz – 1,3GHz	9 500,0	0,25μm
Pentium 4 processzor	2000	1,5 GHz – 2GHz	42 000,0	0,18μm

A kezdetek

Az Intel 1969-ben – alapítása után mindössze egy évvel – annak a projektnek a keretében fogott hozzá az első mikroprocesszor megépítéséhez, amelynek eredeti célja a japán számológépgyártó Busicom új, programozható termékcsaládjához szánt lapkakészlet kifejlesztése volt. Az Intel egyik mérnöke, Ted Hoff egy jobb és hatékonyabb működést ígérő, általános rendeltetésű logikai eszközt tervezett a feladatra. Hoff ötlete alapján készítették el a világ első mikroprocesszorát. Hatvanezer dollárért először a Busicom szerezte meg a mikroprocesszor szabadalmi jogát, ám az újításban rejlő lehetőségeket felismerve, az Intel hamarosan visszavásárolta azt.

A vállalat 1971. november közepén lépett piacra 4004-es kódszámú első processzormodelljével, melynek egységára 200 dollár volt. Sikerének titka az eszköz programozhatóságában rejlett, hiszen azelőtt a lapkákat meghatározott, konkrét feladatok végrehajtására tervezték. A 2250 tranzisztorból álló, alig körömnyi 4004 számítási teljesítménye megegyezett a világ első, szobányi méretű elektronikus komputere, az 1946-ban megépített ENIAC kapacitásával. A 4004 megjelenése számos vállalat mérnökeinek érdeklődését felkeltette, ám ekkor még nem körvonalazódtak világosan a mikroprocesszor felhasználási lehetőségei. A korai alkalmazások között különféle gyakorlati megoldások (pl. közlekedésilámpa-vezérlő készülékek, mintaelemző műszerek) és különleges, nem mindennapi eszközök egyaránt szerepeltek (például a NASA 29 évvel ezelőtt útjára bocsátott, ma is működő Pioneer 10 űrszondája, amely 2001. szeptember 1-én 11,8 milliárd kilométer távolságra volt a Földtől).

[69] Az Intel már a THz nagyságrendű órajellel hajtott tranzisztorokat vizsgálja.

14.14. A SATOR NÉGYSZÖG

Mint azt a figyelmes olvasó már észrevette, a 11. fejezet előtti mottó a többitől jelentősen eltér. Egyfajta rejtvénynek tűnik, és az is volt egy ideig a történészeknek éppúgy, mint a régészek vagy a kriptográfusok számára.

S A T O R
A R E P O
T E N E T
O P E R A
R O T A S

1748-ban kezdték meg Pompei város feltárását, amelyet i.sz. 79. évben a Vezúv kitörése pusztított el. A feltárás során számos házban vagy házon találtak ilyen 5×5-ös betűnégyzeteket, de Hadrianus császár i.sz. 2. század elején épült aquincumi palotájának egyik téglájában is találtak ilyen jelet. A lelet nem volt egészen ismeretlen, mert német és spanyol parasztok ezzel a bűvösnek tartott négyzettel védekeztek a gonosz lelkek ellen, sőt 1743-ban egy szász hercegségben rendelet kötelezte a házak tulajdonosait, hogy ilyen feliratú tányérokat tartsanak – ha a házukban tűz ütött ki, egyet a lángok közé kellett dobni. A négyzet feloldása mégis rejtély maradt. Sokféle megfejtési próbálkozás látott napvilágot, újabb és újabb négyzetek kerültek elő (főleg az őskeresztény templomokból), míg végül előkerült a négyszög eredetije is, amely nem volt más, mint az őskeresztények titkos jelképe, jelvénye.

A latin nyelvű Miatyánk első két szava alkotja a kereszt szárát, kiegészítve a görög ábécé első és utolsó betűjével, az alfával és az ómegával, melyek Krisztus mindenhatóságát (a Kezdet és a Vég) jelképezik. A kereszt birtokában már könnyű volt értelmezni az eredeti SATOR négyszöget, amelynek több érdekesnek tulajdonsága is van: például a *P* betűből kiindulva lóugrásban haladva (egyedül) a PATER szó rakható ki és az egész négyzet végül is egy nagy palindrom. Olvashatjuk balról jobbra fentről lefelé, de olvashatjuk fentről lefelé balról jobbra, sőt jobbról balra alulról felfelé vagy éppen alulról felfelé jobbról balra, mindig ugyanazt kapjuk. Ha az olvasást nem az *S* betűnél, hanem a sarokban lévő *R* betűnél kezdjük, csak a szavak sorrendje változik meg. Akár az *S*, akár az *R* betűnél kezdjük az olvasást, a kapott mondatot akár meg is fordíthatjuk – visszafelé olvashatjuk – ugyanazt fogjuk kapni. Mind az öt szó értelmes, és a mondat szó szerint lefordítva körülbelül így hangzik: *Arepo földműves tartja a munkákat, kerekeket*.

A
+
P
A
T
E
R
A + P A T E R N O S T E R + O
O
S
T
E
R
+
O

A számítástechnikai fejlődés alapfeltétele:

mindig van szűk keresztmetszet, avagy az igény mindig a lehetőség előtt jár.

Ismeretlen

15. KISLEXIKON

A

A5/1: A GSM szolgáltatók által használt titkosítási algoritmus, amely a kapcsolat során átvitt digitális jelek védelméért felelős. Három változata ismert: az A5/1 és az A5/2, előbbi az erősebb, utóbbi a gyengébb. 2002. júliusában jelentették be az új A5/3-at.

ACL, access control list: → hozzáférést vezérlő lista

adat: Az információ nem értelmezett, de értelmezhető formája.

adatbiztonság: Az adatok jogosulatlan megszerzése, módosítása és tönkretétele elleni műszaki és szervezési intézkedések és eljárások együttes rendszere.

AES (*Advanced Encryption Standard*): 1997-ben pályázatot írtak ki a DES leváltására. A győztes algoritmust pedig AES-nek keresztelték. A 2000. októberében lezárt pályázat győztese a Rijndael nevű – belga – algoritmus lett.

aktív támadás (*active attack*): A támadás során a támadó képes megváltoztatni a lehallgatott kommunikáció tartalmát, függetlenül attól, hogy meg tudja-e fejteni azt vagy sem. → *passzív támadás, támadás, behatolás teszt, támadó*

algoritmus (*algorithm*): Matematikai és adatkezelési szabályok halmaza, amely egyértelműen és reprodukálhatóan meghatározza a kívánt eredmény eléréséhez szükséges lépéseket.

assurance level: → biztonsági szint

aszimmetrikus titkosítás (*asymmetric encryption*): A rendszer lényege, hogy a kommunikáló feleknek nem kell előre megegyezniük egy közös titkos kulcsban, hanem mindenkinek két kulcsa van: egy titkos és egy nyilvános. A titkosat csak tulajdonosa ismeri, a nyilvánost bárkinek átadhatja. Ezután a feladó a címzett nyilvános kulcsával egy olyan rejtjelezett szöveget tud generálni, amit csak és kizárólag a címzett titkos kulcsa tud megfejteni. → *szimmetrikus titkosítás*

attacker: → támadó

audit: → felülvizsgálat

B

backdoor (*hátsó ajtó*): Az egyszer gépen hagyott, nehezen észrevehető, esetleg rejtett szolgáltatás, ami a géphez később teljes jogú hozzáférést biztosít. Például ~ lehet egy otthagyott CGI szkript, ami speciális paraméter hatására a többi paramétert, mint parancsot rootként lefuttatja. Vagy backdoor lehet valamilyen nem használt portra rakott program, ami egy „kliensprogram” számára gyakorlatilag távszerviz lehetőséget bizto-

sít (például *Sub7, Back Orifice*). A ~ speciális fajtáját jelentik azok a speciális utasítások vagy paraméterek, amelyek könnyebb karbantartást, illetve fejlesztést tesznek lehetővé (*maintnance hooks*). Tervezési időben nincsenek pontosan definiálva és gyakran dokumentálva sem. Ezekkel lehetőség nyílik a kód szokatlan részeken való végrehajtására vagy az adat szokásos ellenőrzések nélküli hozzáférésére, így biztonsági szempontból komoly veszélyt jelentenek.

Base-64: Olyan kódolási technika, ami a 6 bites értékek ($0..63_{10}$, $00..3F_{16}$) és a minden számítógép által elfogadott szöveges szimbólumok (kis és nagy betűk az angol ábécéből, számjegyek) közötti megfeleltetésen alapul. Feladata, hogy az adatot (bitfolyamot) úgy alakítsa át, hogy az akár ASCII7 textfájlként is kezelhető, továbbítható legyen. Így lehetőség van az adatot olyan kommunikációs csatornákon is továbbítani, amelyek csak 7 bites szavak továbbítására képesek (sőt még Kerckhoffs 4. követelménye is teljesíthető). Az eredeti bájtokat 6 bitesével csoportosítjuk és a 6 bites értékeknek megfelelő karakterrel helyettesítjük. Dekódoláskor a vett karaktereket alakítjuk bitfolyammá, és nyolcasával előállnak bájtok. Egy BASE-64 üzenet 4 karakteres egységekből áll, ami 3 bájtos egységnek felel meg. (4*6bit = 24 bit → 3*8bit = 24bit)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
2	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
3	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Ha a kódolandó üzenet elfogy mielőtt a kódolás véget érne, képzeletbeli nullákkal egészítjük ki, amíg el nem éri a 24 bites hosszúságot. Azokat a „0” sorozatokat, melyek részben sem tartoznak az eredeti üzenethez nem „A” betűvel, hanem „=”-lel kódoljuk. (pl: 0x80 → gA==, 0x8080 → gIA=, 0x808080 → gICA)

behatolás teszt (*penetration test*): (1) Biztonsági tesztelés, amelyben a tesztelők megkísérlik kijátszani a biztonsági rendszert, annak vázlatának és implementációjának értelmezésén keresztül. (2) A tesztelő ellenőrzi, hogy az ismert sebezhető pontok kihasználhatók-e a gyakorlatban. (3) Biztonsági tesztelés, amelyben a tesztelők megkísérlik kijátszani a biztonsági rendszert. A tesztelőkről feltételezhető, hogy használják a rendszer leírását, ami tartalmazhatja a forráskódot, kapcsolási rajzokat, stb. → *aktív támadás, passzív támadás, támadó, behatolás*

biztonság (*security*): → *security*

biztonsági rés (*security hole*): Egy biztonsági rendszer azon eleme vagy pontja, ahonnan (adott események vagy feltételek esetén) információ szivároghat ki vagy más jellegű támadásnak (pl. → DoS) megteremti a feltételeit.

biztonsági szint (*assurance level*): Az egyre fokozódó biztonsági szinteket ábrázoló hierarchikus skálán azt a szintet jelöli, amellyel a rendszer biztonsága egyenértékű vagyis a rendszer megfelel e szint előírásainak.

blokktitkosítók (*block ciphers*): Rögzített méretű (általában 64, 128, 256 bites) adatblokkokat titkosító eszközök. A nyílt szöveg és a titkosított szöveg egyformán ilyen hosszú. Ezzel szemben folyamtitkosítók (*stream ciphers*) esetében a blokkok bitekből (vagy bi-

tek kis számú csoportjából) állnak. A legtöbb blokktitkosító átalakítható folyamtitkosítóvá.

bonyolultság (*complexity*): Az egyes algoritmusok végrehajtásához idő és háttértár szükséges. Azon algoritmusoknak, amelyeknek sok háttértárra van szükségük, nagy a tárolási bonyolultságuk (*space complexity*), míg azok, amelyek végrehajtása nagyon sok lépésből áll, nagy az időbonyolultságuk (*time complexity*). Általában – de nem feltétlenül – a kétféle bonyolultság „fordítottan arányos" egymással: azok az algoritmusok, melyek nagy tárolási bonyolultságúak, viszonylag kevés lépésből állnak és azok az algoritmusok, amelyek sok lépésből állnak, kevés tárhelyet igényelnek. Gyakran lehetőség van az átjárásra: néha a tárkapacitás igény csökkenthető a műveletszám rovására és viszont. Az idő és/vagy tárolási bonyolultság becslése vagy számítása általában a legrosszabb, a legjobb és az átlagos esetre történik.

brute-force: → *teljes kipróbálás*

C

certificate: → *tanúsítvány*

Certificate Authory: → *hitelesítés szolgáltató*

csaló (*cheater*)**:** Abban az esetben, ha a támadó fél egyébként jogosult felhasználó, akkor őt **csalónak** nevezzük és általában több információt kíván megszerezni a protokollszabályok be nem tartása révén, mint amennyit a rendszerben betöltött szerepe alapján kaphatna.

cipher: Titkosító eszköz vagy algoritmus.

Certificate Revocation List, CRL: az a folyamatosan karbantartott lista, amely a visszavont tanúsítványok adatait tartalmazza. A visszavonás oka lehet: lejárt érvényesség, kompromittálódás vagy egyéb ok. Nem minden webböngésző és levelezőkliens ellenőrzi alapértelmezésben a CRL listát!

challenge-response authentication: Ha egy hagyományos jelszóalapú azonosítás során a belépni kívánó ügyfél a szervernek elküldi jelszavát, számolnia kell a lehallgatás veszélyével. A titkosítva küldött jelszó sem nyújt igazán megoldást, mert a lehallgató támadó ugyan nem tudja meg a valódi jelszót, de a titkosított jelszó újraküldésével ugyanazt az eredményt érheti el. Általánosan használt megoldás a *challenge-response* protokoll. Ekkor a szerver egy egyedi azonosítót vagy véletlen számot generál, majd azt a felhasználónévhez tartozó jelszóval titkosítja. Az eredményt elküldi a belépni kívánó ügyfélnek. Ha annak birtokában van a helyes jelszó, ki tudja bontani a kapott csomagot és a helyes azonosítót vissza tudja küldeni. Ennek lehallgatása nem ér semmit, mert az azonosító egyedi, a jelszó pedig semmilyen formában nem jelenik meg a hálózati forgalomban.

confidentality: → titkosság, bizalmasság

D

DES (*Data Encryption Standard*): Szimmetrikus kulcsú titkosítási eljárás, 64 bites blokkos rejtjelezés, vagyis a nyílt szöveg egy 64 bit méretű blokkjához egy ugyanekkora rejtjeles blokkot rendel. Az algoritmust 1977-ben hozták nyilvánosságra. Bár a DES kifejlesztése óta több mint 20 év telt el, ma is élő, még engedélyezett szabvány, széles körben használják a polgári élet minden területén. A DES elérte életciklusának végét, de kiváltására már több alternatíva is van: a háromszor egymás után alkalmazott DES, vagy más alternatív algoritmusok (*AES, IDEA, CAST, BLOWFISH*). A hivatalos utódja a belga Rijndael lett. → *AES*

dictionary attack: → *Teljes kipróbálás*

digitális aláírás (*digital signature*): Személyek vagy digitális adatok hitelesítésére alkalmas módszer, amely nyilvános kulcsú kriptográfiával valósítható meg. Segítségével az üzenet címzettje vagy egy harmadik személy képes megbizonyosodni a feladó személyéről valamint az üzenet sértetlenségéről. A feladó a digitális aláírást úgy hozza létre, hogy a az aláírandó információt (vagy annak lenyomatát → *hashfüggvény*) kódolja a saját titkos kulcsával. A digitálist aláírást a feladó nyilvános kulcsával lehet ellenőrizni. Az aláírás általában tartalmazza az aláírás időpontját, illetve más kiegészítő információt is.
→ *elektronikus aláírás*

DoS attack: *Denial of Service* támadás lényege, hogy egy adott szolgáltatást nyújtó végpontot összehangolt támadás révén annyi hamis, érdektelen kéréssel árasztanak el egyszerre, hogy az végül nem bírja kiszolgálni a kéréseket, így nem tudja a kívánt szolgáltatást nyújtani sem. A végpont jó esetben egy bizonyos mértékű kérésgyakoriság felett nem fogad újabb kéréseket és igyekszik kiszolgálni a már folyamatban lévőket, emellett a gyanús kapcsolatokat egyszerűen megszakítja. Rosszabb esetben a végpont erőforrásai elfogynak és a továbbiakban egyetlen kérés kiszolgálására sem képes (esetleg összeomlik). A rengeteg kérés „özönvízszerű" érkezését jól jellemzi, hogy a DoS támadások egyes típusait „*flood attack*"-nak is nevezik (flood=áradás, özön, árvíz). Mindazonáltal a DoS támadás nem minősül behatolásnak, hiszen a megtámadott rendszer adatait nem érinti, illetve adatlopásra sincs lehetőség. Kivéve persze, ha a megtámadott rendszer a védelmi rendszer, ami ezután képtelen ellátni a feladatát és zöld utat enged minden látogatónak! Ettől még a DoS támadás nem elhanyagolható, mert egy „mission-critical" vagy „life-critical" rendszer megtámadása nemcsak az üzemeltetők vagy a tulajdonos létét kockáztatja, hanem adott esetben a szolgáltatás igénybevevőjét is: mi történne, ha valaki a 104 vagy 105 telefonszámokat tenné elérhetetlenné?

E

előkészítő számítás (*precomputation*): Azokat a műveleteket nevezzük előkészítő számításnak, amelyeket a tényleges titkosítás vagy megfejtés előtt el kell végezni. Az előkészítő számításokra szükség lehet a folyamat gyorsításához is: tipikus eset az iteratív kódolók (*iterated block ciphers*) esete. Elvileg lehetőség lenne a körkulcsok menet köz-

beni (*on fly*) számítására is, azonban a járulékos adminisztráció és egy feladat többszöri elvégzése jelentősen csökkentené a teljesítményt. Az igény szerinti számítások (*on demand*) sem jelentenek segítséget, mert általában minden specifikált részeredményre vagy körkulcsra szükség van, így előbb vagy utóbb úgyis mindent ki kell számítani.

elektronikus aláírás: Gyakorlatilag minden elektronikus formában megjelenő aláírás. Természetesen más a megbízhatósága az email végére írt vezeték- és keresztnévnek vagy egy kedves kis ASCII ábrának, mint egy kriptográfia háttérrel rendelkező aláírásnak. Általában a kriptográfiai hátterű aláírásokat (ha eleget tesznek néhány más követelménynek is) *digitális* (vagy *minősített*) aláírásnak nevezzük → *digitális aláírás*

F

fehérzaj (*white noise*): Olyan véletlenszerű jel, amely minden frekvencián egyforma energiájú jeleket sugároz. A fehérzaj ilyen értelemben hasonló a fehér fényhez, mert az minden lehetséges színt tartalmaz. → *rózsaszín zaj*

feltörés (*code breaking, attack*): Az az eljárás, amikor egy titkosított üzenetből a kulcs ismerete nélkül megfejtik az eredeti szöveget, vagy ami ezzel egyenértékű, a titkosításhoz használt kulcsot. Elképzelhető az is, hogy a támadó olyan alternatív algoritmust készít, amely a kulcs nélkül is képes a visszafejtésre. Ha a titkosító kulcsot megtalálja a támadó és így a további üzeneteket gond nélkül olvasni tudja, *teljesen feltörtnek* tekintjük a kommunikációt.

felülvizsgálat (*audit*): Független fél felülvizsgálata az adatok kezelésére, tevékenységekre, folyamatokra vonatkozóan. Ennek célja, hogy a vizsgálat megállapítsa, mennyire teljesülnek az előírt magatartásformák, technológiák, illetve az hogy, felderítse a termékek, eszközök olyan nem megfelelő jellemzőit, amelyek a rendszer biztonságát fenyegetik.

FIPS (*Federal Information Processing Standard*): Az amerikai kormányzati szabványok sorozata, a NIST gondozza és publikálja őket. A legtöbb az [URL26] címen érhető el. A szabványok az információfeldolgozás és információtechnológia egyes területei számára biztosítanak egységes módszereket.

fizikai biztonság (*physical security*): Fizikai akadályok és korlátozások alkalmazása, mint megelőző intézkedések az erőforrások és az érzékeny információk védelme érdekében. Ahhoz, hogy egy csatorna biztonságos (*secure*) és ne csak biztosított (*secured*) legyen, szükség van fizikai védelemre is. Abszolút biztonságos csatorna (*Shannon féle csatorna*) csak elméletileg létezik.

folyamtitkosítók (*stream ciphers*): A →*blokktitkosítókkal* ellentétben a folyamtitkosítók bitenként vagy bájtonként dolgozzák fel az üzenetet. A folyamtitkosítók a gyakorlati megvalósítás során egy kulcsfolyam-generátort tartalmaznak, amely a titkosítás kulcsától függő kulcsfolyamot generál. Az aktuális kulcsbit(ek) és az adatfolyam-bit(ek) között általában XOR műveletet végzünk, ennek eredménye adja a titkosított bite(ke)t.

H

hashfüggvény (*hash function, one-way function*): Olyan egyirányú függvény, amely egy hosszú, tetszőleges bitsorozat adott hosszúságú ellenőrző összegét készíti el. A lenyomat fix hosszúságú bitsorozat, amely jellemző az eredeti bitsorozatra abban az értelemben, hogy más szöveghez szinte biztosan más hashérték tartozik. Nevezik „*message digest*"-nek is, a gyakran a digitális aláírások alkotórésze. A hashfüggvények egyik támadási alapját a „születésnapi paradoxon" néven ismert probléma adja. (Amely igazából nem is paradoxon...)

hibrid kriptorendszer (*hybrid cryptosystem*): Olyan titkosítási rendszer, amely egyaránt használja a szimmetrikus és az aszimmetrikus algoritmusokat, ötvözve azok jó tulajdonságait.

hitelesítés szolgáltató (*certificate authory*): Olyan szervezet, amely a megbízható harmadik fél szerepében tanúsítványt bocsát ki, amellyel igazolja a nyilvános kulcs (és más jellemzők) egy adott személyhez (vagy általában entitáshoz) való tartozását, kapcsolatát. → *tanúsítvány*

hozzáférést vezérlő lista (*access control list*): Olyan lista, amely a partnerazonosítás (identifikáció) után szabályozza az erőforráshoz való hozzáférés tényét vagy módját, illetve implicit módon a jogosult felhasználókat is meghatározza.

I

időbonyolultság: → bonyolultság

ikerprímek: Két prímszámot akkor nevezünk ikerprímnek, ha különbségük kettő.

információ: Olyan jelentéssel bíró szimbólumok összessége, amelyek adatokat tartalmaznak és olyan új ismereteket szolgáltatnak az értelmező számára, hogy annak valamilyen bizonytalanságát megszüntetik és célirányos cselekvésre ösztönzik.

IDEA (*International Data Encryption Algorithm*): 64 bites blokkmérettel, 128 bites kulcscsal dolgozó, blokkos titkosító algoritmus. Svájcban fejlesztették ki a '90-es évek elején. Kifejezetten adatátvitelhez tervezték, beleértve a digitalizált hang és kép valós idejű titkosítását is. Szabadalmi bejegyzése van, és így (üzleti) felhasználásához licenszdíjat kell fizetni. Egy ideig a DES utódjának tűnt, de ma már csak egy algoritmus a sok közül.

Időbélyegző szolgáltató: Az időbélyegző szolgáltatót mindenki hiteles óraként fogadja el. Az általa kibocsátott időbélyegző nem más, mint a szolgáltató órája szerinti pontos időt tartalmazó elektronikus dokumentum, a szolgáltató titkos kulcsával aláírva. A szolgáltató soha nem ad ki igazolást múltbéli vagy jövőbeli időpontról.

IPSEC (*Internet Protocol Security*): Titkosítási, jogosultságvizsgálati és eredetigazolási eljárások megvalósítása az IP protokollréteg szintjén. Opcionális az IPv4-ben, de kötelező az IPv6-ban. Eltérően az SSL-től nemcsak egy-egy kapcsolat titkosítását teszi lehetővé, hanem az egész hálózat védelmét garantálja. Véd a csomagismétlések ellen és lehetővé teszi a jogosultságalapú hálózathasználatot (forgalom, sávszélesség, stb.) is.

A következő négy fő komponenssel egészül ki az IPv4:

1. *Authentication Header (AH)* leírása az RFC2402-ben található, tartalma az aktuálisan használt hashalgoritmustól függ. Ez lehet *SHA-1* vagy *MD5*.
2. *Encapsulation Security Payload (ESP)*: leírása az RFC2406-ban található, tartalma az aktuálisan használt titkosító algoritmustól függ. Ez lehet *DES*, *3DES*, *RC4*, *IDEA* vagy *BLOWFISH*.
3. *Security Association (SA)*: A következőket definiálja: autentikációs algoritmus AH és ESP mezőkhöz, titkosító algoritmus ESP mezőhöz, titkosítási és viszonykulcsok élettartama, csomag sorszáma, stb.
4. *Internet Key Exchange (IKE)*: leírása az RFC2409-ben található és a viszonykulcsok előállításának eljárását definiálja. *(Diffie-Hellman* vagy *RSA)*

K

kétkulcsos titkosítás: → *aszimmetrikus titkosítás*.

kéttényezős autentikáció: Azon alapszik, hogy *valaki tud valamit* (melyik jelszót, PIN kódot kell használnia) és *rendelkezik valamivel* (kulccsal vagy felhasználói névvel, kártyával, stb.).

kockázat elemzés: Az az eljárás, amely során meghatározzuk, hogy egy esetleges támadás sikerének mekkora a valószínűsége, mekkora a várható kár, illetve a támadó részére a nyereség. Csak ez alapján lehet megtervezni az ésszerű védelmi rendszert és szabályokat (védelmi profilt). A hangsúly az „ésszerű" jelzőn van, mert a végtelen biztonságnak végtelen ára van...→ *maradvány kockázat*

kompromittálás (*compromise*): A biztonsági rendszer megsértése, amelynek következtében érzékeny információ illetéktelen kezekbe kerül és annak felhatalmazás nélküli felfedése követezik be.

kriptoanalízis: A kriptográfia ellentéte, adatvédelmi rendszerek támadásával, feltörésével foglalkozó tudomány. Matematikai eszközökkel igyekszik feltörni a titkosított adatokat. A kriptoanalízis olyan alapokból táplálkozik, mint a valószínűség számítás, a számelmélet, a statisztika, az algebra, a topológia, a káosz elmélet, a matematikai játékelmélet, a mátrixok, a nemlineáris differenciálszámítás és a hírközlési nyelvek elmélete.

kriptográfia (*cryptography*): Az információ algoritmikus módszerekkel történő védelmének tudománya. Matematikai eszközökkel titkosítja a nyílt- és megfejti a titkosított adatokat. Lehetővé teszi, hogy bizalmas adatokat olyan kommunikációs csatornán továbbítsunk a címzettnek, amely eredetileg nem lenne alkalmas erre a feladatra. Görög eredetű szó: kryptos graphos = titkos írás. → *rejtjelezés*

kriptográfiai protokoll: Kriptográfiai protokollnak nevezzük azt a protokollt, amely a szabályok betartatását és a csalók leleplezését kriptográfia eszközökkel (értsd algoritmusokkal) valósítja meg.

kriptológia (*cryptology*): Az algoritmikus információvédelem és ezen módszerek támadásának tudománya, a kriptográfia és a kriptoanalízis összessége.

kulcs (*key*): Az az információ, ami nélkül nem lehet a titkosított üzenet tartalmához hozzáférni. Ha mindkét félnek ugyanarra a kulcsra van szüksége, szimmetrikus kulcsú titkosításról beszélünk. Ha a titkosításhoz más kulcs szükséges, mint a megfejtéshez, aszimmetrikus- vagy nyilvános kulcsú titkosításról beszélünk. Jó algoritmus esetén a kulcs által biztosított védelem exponenciálisan arányos a kulcs (bitekben mért) hosszával, de általánosítva nem igaz, hogy egy hosszabb kulcs egyúttal nagyobb védelmet is jelent.

kulcsletét (*key escrow*): Kormányzati törekvés, amely nemzetbiztonsági érdekekre hivatkozva a polgári titkosítást csak úgy engedélyezi, ha a felhasználó által (megfejtéshez) használt kulcsról másolat van egy központi nyilvántartásban. Innen bírói végzéssel a kulcs kikérhető, így a megfigyelt személy titkosított kommunikációja lehallgatható. Megvalósítása etikai és technikai problémákat egyaránt felvet, bár volt már rá példa (*Clipper chip*). →*lehallgatás*

kulcs-szervezés *(key scheduling)*: A titkosítás folyamán felhasznált – az eredeti kulcsból származtatott – értékek un. alkulcsok (ritkábban körkulcsok) elkészítésének folyamata. A felhasználó által használt kulcsforma, például 1×128 bit, nem biztos, hogy illeszkedik egy adott algoritmus felépítéséhez, működéséhez. Ezért különböző módszerekkel a felhasználó kulcsát átalakítják olyan formátumúra, ami az algoritmus számára szükséges. Ezek az átalakítások általában darabolások és bővítések. Az alkulcsok összmérete bitben mérve mindig nagyobb, mint az eredeti kulcs mérete, viszont entrópiája általában változatlan. → *precomuptation, előkészítő számítások*

L

lehallgatás (*wiretrapping*): A közvetített adatok valós idejű gyűjtése, mint például: tárcsázott számjegyek, beszélgetések, vezetékes vagy vezeték nélküli számítógépes hálózat adatforgalmának lehallgatása, tárolása. → *kulcsletét*

M

maradvány kockázat: Az a kockázati szint, amely egy adott rendszer meghatározott módon történő védelme után még fennáll és a rendszer tulajdonosa ezt a kockázatot tudatosan elfogadja. → *kockázat elemzés*

megbízható eljárás (*trusted process*): Olyan folyamat, amelyet megvizsgáltak a biztonsági szabályok betartására vonatkozóan. Nem tartalmaz olyan kódokat vagy lépéseket, amelyek a biztonsági előírások megsértéséhez vezethetnek.

megszemélyesítés *(spoofing, masquerading)*: Olyan rendszerhozzáférés, amely során a jogtalan felhasználó úgy tesz, mintha ő egy jogos volna. (megszemélyesítés, álcázás, utánzás) Általában aktív támadást és beékelődést jelent a kommunikáló felek közé. → *aktív támadás*

Message Authentication Code, MAC: Olyan MDC érték, amelynek kiszámolása csak egy kulcs ismeretében végezhető el.

Modification Detection Code, MDC: Az üzenettel együtt elküldött hashérték (→ *hashfüggvény*) vagy ellenőrző összeg. A vett üzenetre kiszámított MDC érték és a vett MDC érték egyenlősége vagy egyenlőtlensége jelzi az üzenet sértetlenségét vagy éppen módosulását.

Moore törvénye (*Moore's law*): → *14.13. Moore törvénye* fejezet.

műszaki sebezhetőség (*technical vulnerability*): Egy olyan hardver- vagy szoftverhiba, amely lehetőséget nyújt egy számítógépes rendszer potenciális kihasználásra vagy támadására a rendszeren kívül vagy belül. A hiba veszélyezteti a rendszer működőképességét és közvetve tulajdonosát, felhasználóját.

N, Ny

nyílt szöveg (*plaintext*): Az a szöveg vagy információ, amit rejtjelezni akarunk. Algoritmikusan átalakított párja a rejtjeles szöveg (*cipher text*).

nyíltkulcsos titkosítás: → *aszimmetrikus titkosítás*

O

orange book: Az Amerikai Egyesült Államok Védelmi Hivatalának (US Department of Defensive) 1983-as biztonságtechnikai ajánlása. Eredeti neve a „DoD Trusted Computer Security Evaluation Criteria", ami alapján néha egyszerűen „Evaluation Criteria"-nak is hívják.

OTP (*one time pad*): Egyszer használt bitminta. A titkosítandó adat elemeihez hozzáadunk egy véletlenszerűen generált másik adatsort, mint kulcsot. Megfejtéskor a kulcsot kivonjuk a kapott elemekből és így kapjuk vissza az eredeti adatfolyamot. A legfontosabb, hogy a kulcsot mindig véletlenszerűen kell generálni és minden kulcsot csak egyszer szabad felhasználni → *2.6. Az egyszer használt bitminta*

P

pass phrase: Jelmondat. Szerepe a jelszóhoz hasonlóan az, hogy valakit azonosítson. A jelszóval szemben – ami gyakorlatilag mindig egy szó –, a jelmondat jellemzően több szóból, akár több mondatból áll. Elvileg biztonságosnak tekinthető, mert hossza miatt a *brute-force* támadásokat eleve kizárja. Valódi biztonsága a megvalósító rendszertől függ, sok esetben a jelmondat hashértéke lesz a kulcs. → *hashfüggvény*

passzív támadás (*passive attack*): A támadás során a támadó nem képes (vagy nem akarja) megváltoztatni a kommunikáció tartalmát, csak lehallgatja azt. → *aktív támadás, behatolás teszt, támadó*

peer-to-peer: Egyenrangú kapcsolat vagy egyenrangú kapcsolaton alapuló valami.

perfect forward secrecy: Tökéletes biztonság. Ha egy tökéletes biztonságot nyújtó titkosító algoritmust használunk, akkor a támadó – akármennyi üzenetet fog el és azok töréséhez akármennyi erőforrást igénybe tud venni –, csak a nyílt szöveg hosszát tudja meghatározni, de a nyílt szövegről és a kulcsról semmilyen más információ birtokába sem jut.

PGP (*Pretty Good Privacy*): 1991-ben *Philip Zimmermann* olyan programot írt, amely erős kriptográfia alkalmazásával lehetővé tette az emailek és a fájlok titkosítását. Ez lett a PGP. Csakhogy ebben az időben (az USA-ban) fegyverexportnak minősült az erős kriptográf eszközök országból való kivitele. A PGP terjesztése az interneten kezdődött, így Zimmermann elvileg megsértette az idevonatkozó fegyverexport-szabályzást. A program másik nagy eredménye az volt, hogy először került a magánszféra kezébe katonai szintű védelemre is képes eszköz. Mindezek miatt a program igen sok vitát kavart kormányzati szinten, és igen sokat zaklatták Zimmermannt is.

piggyback: Egy rendszerhez jogtalan hozzáférést szerez valaki egy másik felhasználó szabályszerű kapcsolatán keresztül. („a hátára kapaszkodva")

PKI - public key infrastructure: Nyíltkulcsú infrastruktúra alatt az olyan rendszert vagy rendszereket értjük, amelyeknek szolgáltatásai – és eszközei – az aszimmetrikus kriptográfián alapulnak.

precomputation: → *előkészítő számítás*

protokoll (*protocol*): Szabályok és formátumok szemantikai és szintaktikai összessége, amely lehetővé teszi a résztvevők közti az információcserét. Másrészről a protokoll egy szabályhalmaz, amely a felek közötti kapcsolatot szabályozza, és amelynek be nem tartása általában a kommunikációból való kizárással jár.

R

recovery: Az a folyamat, amely során egy rendszer az abnormális leállást követően visszanyeri működőképes állapotát: adatok, adatállományok, rendszerkapcsolatok helyreállítása, kijavítása vagy visszatöltése.

rejtjelezés (*encyption, ciphering*): Minden olyan tevékenység, eljárás, amelynek során valamely adatot abból a célból alakítanak át, hogy annak eredeti tartalma az illetéktelenek számára rejtve maradjon. A rejtjelezés részét képezi a rejtjelezett adat eredetivé való visszaállítása is. A magyar jogszabályokban „Nem minősül rejtjelzésnek a személyi vagy objektumazonosítás céljaira szolgáló kódolás, továbbá az automatikus rendszerek mérési vagy vezérlési adatainak kódolása akkor sem, ha az védelmi célokat szolgál." (Magyar Köztársaság 43/1994. kormányrendelete) → *kriptográfia*

rejtjelezett szöveg (*cipher text*): A nyílt szöveg rejtjelezett (titkosított) párja. Megfejtéséhez a kulcsra, vagy alternatív algoritmusra van szükség (de ez már nem megfejtés, hanem visszafejtés vagy törés). → *rejtjelezés*

rózsaszín zaj (*pink noise*): Olyan véletlenszerű jel, amely az egyes frekvenciákon a frekvencia reciprokával (általánosan inverzével) arányos energiájú jelet sugároz. Az eredő jelben így igen sok az alacsony frekvenciás összetevő (vagy igen kevés a magas). Ezzel

ellentétben a fehérzaj *(white noise)* amplitúdója minden frekvencián azonos. (Van barna zaj is, de ne kérdezze meg senki, hogy az mit tartalmaz…) → *fehér zaj*

RSA: *Ron **R**ivest*, *Adi **S**hamir* és *Leonard **A**dleman* által 1977-ben kidolgozott nyilvános kulcsú algoritmus, amely a nagy számok modulo aritmetika szabályai szerinti hatványozásán alapul. Nagy szám alatt extrém nagy számokat (1024-4096 biteseket, 300-1200 decimális jegyűeket) kell érteni. A három kutató alapító tagja az RSA Data Security Inc. cégnek *(1982, http://www.rsa.com)*, mely a nyolcvanas évek közepén még a csőd szélén állt, ma már több tízmillió dolláros forgalmat bonyolít. → *aszimmetrikus titkosítás, szimmetrikus titkosítás*.

S

security (*biztonság*): Kontrolálatlan veszteségek és hatások elleni védelem minősége. Abszolút biztonságot tulajdonképpen lehetetlen elérni; azaz a biztonság relatív fogalom. Már csak azért is, mert a felhasználók nem költenek többet a biztonsági rendszerekre, mint amit a védendő információ ér (→*maradvány kockázat*). Másrészről a biztonságtechnika területén elfogadott az a kijelentés, miszerint a végtelen (tökéletes) biztonságnak végtelen ára van. A ~ *policy* azon szabályok és eljárások gyűjteménye, amely a védendő információ kezelését továbbítását, terjesztését (általánosságban egy rendszer védelmét) szabályozza.

session key: → *viszonykulcs*

SET (*Secure Electronic Transaction*): A SET egy olyan elfogadott, nyílt elektronikus fizetési protokoll, amelyet a *Visa* és a *Mastercard* közösen javasolt, és fejlesztésében többek között részt vett a *Microsoft*, a *Netscape*, az *IBM*, a *VeriSign.* A protokoll főbb feladatai:

1. *A kommunikáló felek azonosítása.*
2. *A kommunikáció titkosítása.*
3. *Az adatok szelektált továbbítása, minden résztvevő csak a számára szükséges információhoz férhet hozzá.*
4. *A pénzügyi és kereskedelmi adatok szükség esetén újra összeilleszthetők.*

Egyes irodalmakban a SET-et, mint titkosítási algoritmust emlegetik, azonban ez helytelen, a SET nem titkosítás, hanem egy protokoll. → *protokoll*

Shannon-féle csatorna: Abszolút biztonságos csatorna, de sajnos csak elméletileg létezik. → *fizikai biztonság, perfect forward secrecy*

siffre, sifrírozás: Rejtjel, rejtjelezés a francia *chiffre* szóból.

Skipjack: Egy 64-bites blokkos, szimmetrikus algoritmus. 80 bites kulcsot használ (a DES csak 56-biteset) és 32-szeres iterációs körrel rendelkezik (DES: 16 kör). A közelmúltban oldották fel, ma már nyilvános algoritmus.

Smart Card: Intelligens kártya. Egy pici, egylapkás, önálló áramforrással nem rendelkező számítógép. Kiválóan alkalmas azonosításra, de a jól ismert telefonkártya és mobilok való SIM kártyája is egy Smart Card.

sniffer: A hálózat forgalmát lehallgató és elemző program vagy hardvereszköz. Használatának célja lehet valamilyen hálózati probléma diagnosztizálása vagy éppen a hálózati forgalom lehallgatása. Bár a „gyári" változatok gyakran jelzik jelenlétüket a hálózaton (bizonyos csomagok szórásával), a kevésbé hivatalos programok ezt általában nem teszik, így a szimatoló kiszűrése, észrevétele szinte lehetetlen feladat is lehet. (Sőt maga az UTP kábel is megbarkácsolható úgy – a Tx szál elvágásával –, hogy a számítógép *fizikailag* képtelen a hálózatba adni, csak hallgatózik, így más megfigyelő számára valóban láthatatlan marad.)

space complexity: → *bonyolultság*.

SSL (*secure socket layer*): Az SSL (újabb nevén *TLS - transport layer security*) napjaink igen elterjedt, titkosított kommunikációt biztosító protokollja. Eltérően az olyan protokolloktól, mint például az *IPSec*, amely az egész hálózatot teszi biztonságossá, az *SSL* csak egy-egy kommunikációs csatornát biztosít (kapcsolatorientált). Az SSL a protokollstackben az alkalmazásréteg alá és a szállítási réteg fölé került, ily módon önmaga is egy protokoll réteget alkot.

SSLeay: Az SSL ingyenes megvalósítása, amely *Eric Young* (és mások) nevéhez fűződik. Ausztráliában fejlesztették, ezért nem esik (esett) az USA exportkorlátozása alá.

szimmetrikus titkosítás: A klasszikus titkosítási séma. Az üzenetet váltó felek ugyanazt a kulcsot birtokolják. A titkosításhoz és a megfejtéshez ugyanazt a kulcsot kell használni. Az ilyen rendszer fő problémája, hogy az üzenetcsere előtt valahogy egyeztetni kell a résztvevők kulcsait. Az ehhez használt csatorna viszont valamilyen értelemben biztosan drága vagy nem állandóan biztonságos, ellenkező esetben nem kellene rejtjelezést használni, hanem ezen lehetne a védendő üzenetet is elküldeni. További probléma, hogy sokrésztvevős rendszerben rengeteg – a résztvevők számának négyzetével arányos mennyiségű – kulcsot kell kezelni, ha mindenki mindenkivel titkosítottan (a többiek előtt védetten) kíván kommunikálni. →*aszimmetrikus titkosítás*

szótáras támadás (*dictionary attack*): → *teljes kipróbálás*

SWOT analízis: Egy megoldási javaslat vagy lehetőség értékelési módszere, amely alapvetően négy összetevőből áll. Az adott megoldás erősségeinek (***s**trenghts*) és gyengeségeinek (***w**eaknesses*) valamint lehetőségeinek (***o**pportunities*) és veszélyeinek (***t**hreats*) mérlegelését jelenti.

T

tanúsítvány (*certificate*): Egy adott partnerhez tartozó igazolás, ami a partner nyilvános kulcsát, nevét, lejárati időt stb. tartalmazza, és amit egy erre felhatalmazott, megbízhatónak tekintett fél (*Certification Authory, CA*) a saját titkos kulcsával aláírt. Ezzel az adott partner és a nyilvános kulcs összetartozását mindenki számára ellenőrizhető módon hitelesítette. További problémákat vet fel a *CA* aláírásának hitelessége: neki is rendelkeznie kell egy tanúsítvánnyal, amit valaki hitelesített és így tovább. E hierarchia csúcsán áll a *root-CA*.

támadó (*attacker*): A kommunikáció résztvevői közötti információcsere titkosságát támadó harmadik személy. Az általa elkövetett cselekedet a biztonsági rendszerek kiiktatására irányul. A támadás lehet aktív vagy passzív, ennek megfelelően megkülönböztetünk aktív támadót és passzív támadót. A behatolás ténye nem jelenti szükségszerűen a támadás sikerességét. A sikeresség foka függ a rendszer sebezhetőségének fokától és a meglévő óvintézkedések hatékonyságától. → *behatolás teszt, aktív támadás, passzív támadás, maradvány kockázat, biztonság*

tárolási bonyolultság (*space complexity*): → *bonyolultság*

teljes kipróbálás (*exhaustive search, brute-force*): A titkosító rendszerekkel szemben alkalmazott támadási mód, ami elvileg mindig eredményes. Lényege, hogy a rejtjelező rendszer ismeretében az összes lehetséges kulcsot kipróbálva határozzák meg az alkalmazott kulcsot. Természetesen a módszer csak akkor végrehajtható, ha a kulcstér véges, illetve egy bizonyos méretnél kisebb. Minél hosszabb a kulcs, annál több időbe telik a kipróbálás. Amennyiben a kulcsokat hamarabb lecseréljük, mint a *brute-force* támadáshoz szükséges átlagos idő, a támadót állandó kulcskeresésre kényszeríthetjük. Ha a keresett kulcs egy jelszó, akkor a *brute-force* első lépése általában a szótáras támadás (*dictionary attack*), melynek keretében először néhány gyakori jelszóval, majd értelmes szavakkal, nevekkel, becenevekkel próbálkoznak.

tempest tampering: Műszaki berendezések (például számítógépek, főként monitorok) által keltett elektromos jelek, elektromágneses sugárzások lehallgatása ellenőrzés, elemzés vagy adatlopás céljából. Akár elhisszük, akár nem, de megfelelő eszközzel monitorunkat a szomszéd is olvashatja... Ugyanígy tempestnek minősül a hardver keyloggerek alkalmazása, ami általában egy kicsi, saját memóriával rendelkező henger és a billentyűzet vezetéke köré rakják (kísértetiesen emlékeztet a zavarszűrő ferrithengerre). A képen egy „active-ghost” látható, ami nem a billentyűzet vezetékének elektromágneses lehallgatásán alapszik (passzív támadás) hanem kábeltoldalékként a gép és a billentyűzet közé ékelődik (aktív támadás). Általában a gép háta mögött van, ezért a legtöbb felhasználó észre sem veszi, hogy egy pici eszköz minden leütését rögzíti... (Ezek az eszközök 100 ezer – 2 millió leütés tárolására képesek.)

time complexity (*időbonyolultság*): → *bonyolultság*

timestamp authory: → *időbélyegző szolgáltató*

titkos kulcsú titkosítás: → *szimmetrikus titkosítás*

titkosítás (*encryption*): → *rejtjelezés*

titkosság (*confidentality*): (1) Annak bizonyossága, hogy az információ nemkívánatos résztvevő előtt nem kerül felfedésre. (2) Olyan tulajdonság, amely következtében az információ hozzáférhetetlen jogtalan résztvevők számára. (3) Olyan törekvés, amelyben arra összpontosítanak, hogy az érzékeny információt biztonságosan kezeljék illetve, hogy korlátozott számú, hozzáférésre jogosult résztvevő férhessen csak hozzá.

titokmegosztás (*secret sharing*): A titokmegosztás feladata, hogy adott titkot, információt úgy osszunk fel *n* résztvevő között, hogy közülük bármely *k* (*k*<*n*) képes legyen a titkot visszaállítani, de bármely *k-1* még nulla ismerettel rendelkezzen a titokról. Ezzel szemben a titok szétvágásakor *(secret splitting)* minden résztvevőre szükség van, ha egy is hiányzik, a titkot nem lehet rekonstruálni.

TripleDES: Három darab DES egymás utáni használta egy adatblokkon. Erősebb titkosítást biztosít, mint a szimpla DES, ez elsősorban a háromszor olyan hosszú kulcsnak köszönhető. Ha a három titkosító egység kettő különböző kulcsot használ (az első és a harmadik algoritmus ugyanazt használja), *3DES*-ről beszélünk. Ha három különböző kulcsot használ a lánc, *TripleDES*-nek hívjuk. Sebessége elvileg elég kedvezőtlen, hiszen háromszor lassabb, mint az egyszeres DES.

trusted process: → *megbízható eljárás*

two-factor authentication: → *kéttényezős autentikáció*

V

vak aláírás (*blind signature*): Olyan aláírási eljárás, amikor az aláíró fél nem ismeri meg az aláírt dokumentum tartalmát. Az aláírása azt bizonyítja, hogy adott időpont óta a dokumentum nem változott meg. A hétköznapi gyakorlatban vegyünk egy indigót, tegyük az aláírandó dokumentumra, és az egészet tegyük egy borítékba. Az aláíró féllel pedig írassuk alá a borítékot a megfelelő helyen. Aláírása így szerepelni fog a dokumentumon is, jóllehet annak tartalmát nem ismerte meg.

véletlen szám (*random number*): Olyan számsorozat egyik tagja, amely sorozat viselkedése nem kiszámítható. A sorozat korábbi elemeinek birtokában sem jósolható meg vagy számítható ki, hogy mi lesz a sorozat következő eleme. Valódi véletlen számokat csak analóg eszközökkel állíthatunk elő vagyis algoritmikusan nem.

viszonykulcs (*session key*): Olyan titkosító kulcs, amely minden egyes kommunikációs folyamatban egyedi és véletlenszerű. Ha új folyamat kezdődik, ahhoz új kulcsot kell generálni és az új kulcsot kell használni a kommunikációs folyamat titkosításához.

X

XOR művelet: Bitszintű művelet, másik neve „*kizáró vagy*”. Értéke akkor „1”, ha a bitek különbözőek, egyébként „0”. A kriptográfiában inkább azt a tulajdonságát használják ki, hogy egy $A=B \oplus C$ kifejezésből bármelyik változót ki lehet számolni a másik kettő ismeretében ugyanennek a műveletnek a felhasználásával, **vagyis a művelet önmaga inverze**. ($B=A \oplus C$ és $C=A \oplus B$. Ha a $\oplus$ műveletet összeadásnak tekintjük, akkor a kivonást szintén a $\oplus$ művelettel kell elvégezni!) A hagyományos aritmetikában ez általában műveletváltással jár, lásd összeadás-kivonás, osztás-szorzás művelet párokat. A XOR művelet egyébként *mod 2* összeadásként is értelmezhető.

0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0

16. FELHASZNÁLT ÉS AJÁNLOTT FORRÁSOK

Irodalomjegyzék

A könyv következő fejezetei – átdolgozott vagy változatlan formában – a *tech.net* magazin alábbi számaiban is megjelentek:

- 11. Eltemetett bitek: Szteganográfia — 2003. március
- 12. Titokmegosztás — 2003. június
- 5. Elliptikus görbék — 2003. augusztus - szeptember

[1] *Othmar Kyas*: Számítógépes hálózatok biztonságtechnikája (2000. Kossuth Kiadó)

[2] *Andrew S. Tanenbaum*: Számítógép-hálózatok (1999. Panem-Prentice Hall, 616. – 661. oldal)

[3] *Ködmön József*: Kriptográfia (1999. Computerbooks, 113. – 201. oldal)

[4] *D.E. Knuth*: A számítógép-programozás művészete 2. kötet (1995. Műszaki Könyvkiadó, második kiadás)

[5] *Cetin Kaya Koc*: High-Speed RSA Implementation (1994, RSA Report)

[6] *Lukács – Tarján*: Játékos matematika

[7] *Ron Rivest*: The MD5 message digest algorithm (1992 April, RFC 1321)

[8] Informatikai Koordinációs Iroda – Informatikai biztonsági módszertani kézikönyv (1994, Miniszterelnöki Hivatal, ITB 8. számú ajánlás)

[9] *Galántai Zoltán*: A nagy adatrablás (1998, Kossuth kiadó)

[10] *Robert D. Silverman*: A cost-based security analysis of symmetric and asymmetric key lengths (2000. April, RSA Laboratories, Bulletin #13)

[11] *Menezes, Oorshot, Vanstone*: Handbook of Applied Cryptography (1996, CRC Press)

[12] CryptoBytes 1998 Summer (Technical Newsletter of RSA Laboratories)

[13] CryptoBytes 1999 Winter (Technical Newsletter of RSA Laboratories)

[14] Data Encryption Standard (Federal Information Processing Stanrds Publication, FIPS PUB 46-3 1999. October – National Institute of Standards and Technology)

[15] Secure Hash Standard (Federal Information Processing Stanrds Publication, FIPS PUB 180-1 1995. April – National Institute of Standards and Technology; Supersedes FIPS PUB 180, 1993 May)

[16] CryptoBytes 1997 Autum (Technical Newsletter of RSA Laboratories)

[17] *Eli Biham, Adi Shamir*: Differential Cryptanalysis of the Full 16-round DES (1998, Springer-Verlag)

[18] *Eli Biham, Adi Shamir*: Differential Cryptanalysis of DES-like Cryptosystems (1991, Journal of Cryptology, Vol. 4. No. 1.)

[19] *Stephen W. Hawking*: Az Idő rövid története (1995, Maecenas Kiadó, 4. kiadás)

[20] *Dan Froomkin, Amy Branson*: Deciphering Encryption (1998, May 8. Washington Post.com, Encryption Special Report)

[21] *Elizabeth Corcoran*: U.S. to relax encryption limits (1998, Sep 17. Washington Post.com, Encryption Special Report)

[22] *Vajda István, Bencsáth Boldizsár, Bognár Attila*: Tanulmány a napvilágra került Elender jelszavakról – URL[38] (2000, február)

[23] *Philip Zimmermann*: An intoduction to cryptography (PGP documentation)

[24] *Bruce Schneier*: Applied cryptography (1996, John Wiley & Sons, Inc., 2nd edition)

[25] *John Linn*: Trust models and management in Public-key infrastructures (2000. november, RSA technical report)

[26] *Roland R. Rivest, Robert D. Silverman*: Are "Strong" primes need for RSA? (1998. december)

[27] *Christian Cachin*: An information-theoretic model for steganography (2000. June)

[28] *David Wagner, Bruce Schneier*: Analysis of SSL 3.0 protocol (technical report)

[29] *Michael J. Wiener*: Efficient DES key search (1993. August)

[30] *Joan Daemen, Vincent Rijmen*: The Rijndael Block Cipher (1999. March document version 2)

[31] *Lisa M. Marvel*: Image steganography for hidden communications (1999. University of Delarware, dissertation for the degree of Doctor of Philosophy in Electrical Engineering)

[32] Electronic Messaging Association. Numbers. (1999. January, Time Magazine, page 23)

[33] Az Infostrázsa (Kürt Rt. kiadványa)

[34] *Simon Singh*: Kódkönyv - A rejtjelezés és a rejtjelfejtés története. (2001. Park Könyvkiadó)

[35] Tech.net magazin (2000. November, 18. oldal)

[36] *Richard Bondi*: Cryptography for Visual Basic - a programmer's guide to the Microsoft CryptoAPI (2000. John Willey & Sons, Inc.)

[37] *Dr. Tóth Mihály*: DES (Data Encription Standard) - Segédlet az Információ-technika c. tárgy Kriptográfia fejezetéhez (2000. február, BMF-KKVK jegyzet)

[38] *Pethő Balázs és tsai*: Security Lexikon – ELTE TTK Security anyagából (1999. június)

[39] *Fabien A. P. Petitcolas, Ross J. Anderson and Markus G. Kuhn*: Information Hiding - A Survey (1999. július)

[40] *M.Joye, J.-J. Quisquarter*: Faulty RSA encryption – UCL Crypto Group Technical Report Series CG-1997/8

[41] *Josh Benaloh*: General Linear Secret Sharing – Extended Abstract (1996. január)

[42] *Geoffrey C. Grabow*: Life after DES (CCE newsletter 1999 Spring, PriceWaterhouseCoopers)

[43] *Egerszegi Krisztián, Erdősi Péter*: Elektronikus aláírás, PKI rendszerek (BME jegyzet, 2002. május)

[44] *Bruce Schneier*: Secrets and Lies – Digital security in a networked world (2000, John Wiley & Sons, Inc.)

[45] *Gordon E. Moore*: Cramming more components onto integrated circuits (1965, Electronics, Volume 38, Number 8, April 19)

[46] 1992. évi LXIII. törvény a személyes adatok védelméről és a közérdekű adatok nyilvánosságáról, Magyar Köztársaság

[47] 43/1994. (III.29.) Korm. rendelet a rejtjeltevékenységről, Magyar Köztársaság

[48] 2001. évi XXXV. törvény az elektronikus aláírásról, Magyar Köztársaság

[49] P8WE5032 Secure 8-bit Smart Card Controller (Philips Semiconductors, Short Form Specification 2000 July)

[50] CryptoBytes 1996 Summer (Technical Newsletter of RSA Laboratories)

[51] *W. Bender, D. Gruhl, N. Morimoto, A. Lu*: Techniques for data hiding (1996, IBM system journal, VOL35)

[52] *Dr. Tóth Mihály*: Szteganográfia avagy adatrejtés és elektronikus vízjelek (KKVK SzGTI jegyzet)

[53] *Saraju P. Mohanty*: Digital Watermarking: A tutorial review

[54] *Almási János*: Elektronikus aláírás és társai (2002, Sans Serif kiadó)

[55] *Joan Daemen, Vincent Rijmen*: The Design of Rijndael (2001, Springer-Verlag)

[56] *Erik De Win, Bart Preneel*: Elliptic Curve Public Key Cryptosystems – an introduction (2001)

[57] Advanced Encryption Standard (Federal Information Processing Stanrds Publication, FIPS PUB 197 2001. November – National Institute of Standards and Technology)

[58] *Amos Beimel, Benny Chor*: Secret Sharing with Public Reconstruction (IEEE Transactions on Information Theory Vol.44 No.5. September 1998)

[59] *Ahmet M. Eskicioglu*: A Key Transport Protocol Based on Secret Sharing – An Application to Conditional Access Systems (Thomson Multimedia, technical report, 2000)

[60] *Elisabeth Oswald*: Introduction to Elliptic Cryptography (2002. July 3, Institute for Applied Information Processing and Communication, Austria)

[61] *Henna Pietilainen*: Elliptic Curve Cryptography on smart cards. (master's thesis, 2000. october 30. Helsinki University of Technology)

[62] *Neal Koblitz, Alfred Menezes, Scott Vanstone:* The state of elliptic curve cryptography (Designs, Code and Cryptography no.19., 2000, Kluwer Academic Publishers, Boston.)

[63] Crypto-gram Newsletter, 1999. November 15.

[64] IEEE P1363 Draft 13 – Standard specifications for Public-Key Cryptography (November 12, 1999)

[65] ANSI X9.62 draft, September 20, 1998

[66] Secure Hash Standard (Federal Information Processing Stanrds Publication, FIPS PUB 180-2 2002. August)

[67] *Francois Koeune, Jean-Jacques Quisquater*: A timing attack against Rijndael (1999, technical riport CG-1999/1, Université catolique de Louvain)

[68] *Nicolas T. Courtois és Josef Pieprzyk:* Cryptanalysis of Block Ciphers with Overdefined Systems of Equations

[69] *Rick Wash*: Lecture notes on stream ciphers and RC4

[70] *Palash Sarkar, Subhamoy Maitra*: Low Cost Hardware Architecture for Secure Stream Cipher Cryptosystems, 2001(?)

[71] *Adi Shamir, Alex Buryukov, David Wagner*: Real Time Cryptanalysis of A5/1 on a PC, 2000(?)

[72] *Charles Brookson*: GSM (and PCN) Security and Ecryption, 1994, GSM Security Group

[73] *Neil F. Johnson*: An introduction to watermark recovery from images 1999, technical report

[74] *Neil F. Johnson, Sushil Jajodia*: Steganalysis: the investigation of hidden information 1999, technical report

[75] *Duncan Stellars*: An introduction to steganography 1999. may 9, technical report

[76] *Mátray Péter*: A szerzői jog védelme a matematika eszközeivel, 2003. ELTE szakdolgozat

Linkek

Lehet, hogy a lenti linkek közül néhány már nem működik, de érdemes őket meglátogatni, sok helyen van linkgyűjtemény is, azokat sem szabad kihagyni!

[URL00] http://www.cryptox.hu
[URL01] http://forum.index.hu/forum.cgi?a=t&t=9037717
[URL02] http://www.szgti.bmf.hu/~mtoth
[URL03] http://web.interware.hu/stef/security.html
[URL04] http://www.io.com/~ritter/CRYPHTML.HTM
[URL05] http://cacr.math.uwaterloo.ca/hac/
[URL06] http://www.iif.hu/~pasztor/feny.html
[URL07] http://ipisun.jpte.hu/~uhi/kurzus/informatika/titok.htm
[URL08] http://www.comptech.hu/VI16/titkositas.html
[URL09] http://www.itb.hu/ajanlasok/a8/html/a8_m3_3-4.htm
[URL10] http://www.crypto.com/
[URL11] http://www.biharinfo.elender.hu/vancsod/cikk2.htm
[URL12] http://tai2.szif.hu/FreeBSD_doc/handbook/handbook68.html
[URL13] http://www.obh.hu/adatved/magyar/index.htm
[URL14] http://www.jya.com/crack-a5.htm
[URL15] http://www.scard.org/gsm/
[URL16] http://www.counterpane.com/crypto-gram.html
[URL17] http://www.cert.hu/irod.html
[URL18] http://www.scramdisk.clara.net/
[URL19] http://www.nist.gov/aes
[URL20] http://www.mkogy.hu
[URL21] http://www.deja.com
[URL22] http://www.fjhirsch.com/~fhirsch/SSL/
[URL23] http://www.eff.org/DESCracker
[URL24] http://msdn.microft.com
[URL25] http://www.dice.ucl.ac.be/crypto

[URL26] http://csrc.nist.gov/publications/fips/
[URL27] http://www.bs7799.hu/
[URL28] http://world.std.com/~dpj/elliptic.html
[URL29] http://csrc.nist.gov/cryptval/
[URL30] http://www.ihm.hu/miniszterium/jogszabalyok/
[URL31] http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/hof/hof_main.htm
[URL32] http://www.semiconductors.philips.com/markets/identification/products/8_bit/we/index.html
[URL33] http://www.certicom.com
[URL34] http://www.cacr.math.uwaterloo.ca/~ajmeneze/misc/cryptogram-article.html
[URL35] http://www.cl.cam.ac.uk/~fapp2/steganography/mp3stego/index.html
[URL36] http://www.certicom.com/research/ch1.html
[URL37] http://catb.org/~esr/faqs/hacker-howto.html
[URL38] http://ebizlab.hit.bme.hu/pub/lrpasswd.html
[URL39] http://www.rsa.com
[URL40] http://www.frontiernet.net/~imaging/graph_my_equation.html
[URL41] http://yoyo.cc.monash.edu.au/~bunyip/primes/index.html
[URL42] http://www.users.zetnet.co.uk/hopwood/crypto/scan/index.html
[URL43] http://andrewsoft.hypermart.net
[URL44] http://www.cryptomathic.com/labs/index.html
[URL45] http://csrc.nist.gov/csrc/fedstandards.html
[URL46] http://www.secg.org/
[URL47] http://www.certicom.com/research/ecc_chal_contents.html
[URL48] http://jf.morreeuw.free.fr/vigenere/vigenere.html (francia nyelvű oldal)
[URL49] http://www.cryptool.com/
[URL50] http://webstore.ansi.org/ansidocstore/dept.asp?dept_id=80
[URL51] http://www.certicom.com/resources/ecc_chall/curves.html
[URL52] http://www.cryptography.com/resources/whitepapers/DES.html
[URL53] http://www.mit.bme.hu/~csilla/
[URL54] http://www.itu.int/itudoc/itu-t/rec/x/x500up/x509.html
[URL55] http://nws.iif.hu/ncd2002/docs/ehu/65/index.html
[URL56] http://nws.iif.hu/ncd2002/docs/ehu/15/index.html
[URL57] http://www.esat.kuleuven.ac.be/~rijmen/rijndael/index.html
[URL58] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/security/security/cryptography_portal.asp
[URL59] http://www.minrank.org/aes/
[URL60] http://www.counterpane.com/crypto-gram-0209.html
[URL61] http://www.origo.hu/mindentudasegyeteme/lovasz/20030630lovasz22.html
[URL62] http://www.mirrors.wiretapped.net/security/cryptography/algorithms/gsm/a5-1-2.c
[URL63] http://www.counterpane.com/twofish.html

[URL64] http://www.counterpane.com/blowfish.html
[URL65] http://frode.home.cern.ch/frode/crypto/simula/index.html
[URL66] http://www.xat.nl/enigma/index.htm
[URL67] http://www.security-survey.gov.uk
[URL68] http://www.afn.org/~afn21533/tinyaes.zip
[URL69] http://support.microsoft.com/support/kb/articles/Q276/3/04.ASP
[URL70] http://www.usembassy.hu/alkotm.htm
[URL71] http://www.microsoft.com/hun/biztonsag/default.mspx
[URL72] http://www.informatik.uni-bonn.de/~adrian/index.html
[URL73] http://www.mersenne.org
[URL74] http://www.rkmath.rikkyo.ac.jp/~kida/ubasic.htm

Stephen W. Hawking, "Az idő rövid története" című könyvében fogadalmat tett, hogy egyetlen képlet alkalmazása nélkül megpróbálja elmagyarázni az "idő" fogalmát és a jelenlegi téridő elméleteket. Végül Einstein $E=mc^2$ egyenletével kivételt tett, de több képlet valóban nem szerepel a könyvében, mégis érthető. Persze, lehet, hogy egy párszor el kell olvasni, de ez nem biztos, hogy csak az ő hibája... Ez a könyv ebből a szempontból hasonló: a lehető legkevesebb képlettel jut el az ősi Caesar-kódtól a DES-en, RSA-n keresztül a mai AES-ig (leánykori nevén: Rijndael). A figyelmes olvasó választ kap alapvető kérdéseire, sőt ujjgyakorlatként a mai rettegett algoritmusok némelyikét is leprogramozhatja a kedvenc programnyelvén...

Ami ennél is fontosabb, az a titkosítás mögött rejlő csodálatos világ, amely az idők folyamán számtalan technológia alkalmazásával próbálta elérni mindig ugyanazt. Ma már gyakran mosolygunk a régi módszerek némelyikén, pedig az, hogy ma AES-t, TWOFISHT, RC4-et és RSA-t meg ECC-t használunk, ezeknek az eljárásoknak köszönhető. Ma már ismerjük a régi (és a jelenlegi) algoritmusok gyenge pontjait is, tudjuk, hol és miben hibáztak elődeink. Valóban tudjuk? Valóban mindenki tudja, akit a téma érdekel? Valóban mindenki, aki érintett lehet a témában tudja, mi az a titkosítás, mikor erős vagy gyenge egy módszer?

Virasztó Tamás

A könyv olvasása során választ kaptam minden olyan kérdésemre, amiket eddig nem mertem feltenni, még magamnak sem. A könyv nagy segítség a témában jártas és kevésbé jártas olvasóknak egyaránt, hiszen segít feltenni, megválaszolni az olykor nehéznek hitt kérdéseket. Nincs más dolgunk, mint e remekművet kézbe venni és megismerni a titkosítás titokzatosnak hitt világát.

Harmath Zoltán
Microsoft Magyarország

ISBN 963 214 253 5
9 789632 142531

Ára: 4 900 Ft