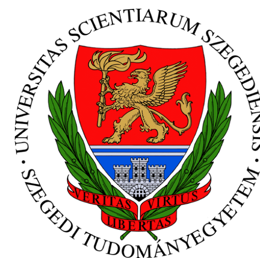
Információs rendszerek biztonságtechnikája

Vassányi István, Dávid Ákos, Smidla József,
Süle Zoltán



2014

A tananyag a TÁMOP-4.1.2.A/1-11/1-2011-0104 "A felsőfokú informatikai oktatás minőségének fejlesztése, modernizációja" c. projekt keretében a Pannon Egyetem és a Szegedi Tudományegyetem együttműködésében készült.

1. rész: Kriptográfia

TARTALOM

1. Történeti áttekintés ... 5
2. A modern kriptográfia alapjai ... 10
3. Modern blokkos kriptorendszerek ... 14
4. Modern folyam-elvű kriptorendszerek ... 17
5. Nyilvános kulcsú kriptorendszerek és alkalmazásaik ... 20
6. Üzenetpecsétek ... 25
7. A kriptográfia jövője ... 28

Irodalom ... 34

I.MELLÉKLET: AZ AES KRIPTOGRÁFIAI ALGORITMUS ... 35

Bevezetés ... 35

Matematikai alapok ... 36

A kulcs kiterjesztése ... 40

Kódolás ... 40

Dekódolás ... 44

II. MELLÉKLET BLOKK KÓDOLÓK ÜZEMMÓDJAI ... 48

Elektronikus kódkönyv (ECB) ... 48

Titkosított blokkok láncolása (CBC) ... 50

Kimenet visszacsatolása (OFB) ... 52

Kódolt üzenet visszacsatolása (CFB) ... 54

Számláló mód (CTR) ... 56

III. MELLÉKLET AZ RSA ALGORITMUS ... 57

Matematikai alapok ... 57

Az RSA algoritmusa ... 62

Az RSA alkalmazása digitális aláírásra ... 64

IV. MELLÉKLET A SZÜLETÉSNAP-TÁMADÁS MATEMATIKAI HÁTTERE ... 68

V.MELLÉKLET AZ SHA-1 ÜZENETPECSÉT ... 71

Alapműveletek ... 71
Az SHA-1 működése ... 72
VI. MELLÉKLET A GRAIN-128 FOLYAMTITKOSÍTÓ ... 76
VII. MELLÉKLET A RABBIT FOLYAMTITKOSÍTÓ ... 79

1. Történeti áttekintés

A szteganográfia

A kriptográfia nagyon régi tudomány, melynek alkalmazásai az ókorba nyúlnak vissza. A legrégebbi időkben nem a ma megszokott, titkos vagy nyilvános kulcsokra alapozott **kriptográfiai** módszerek, hanem a **szteganográfia**, azaz a titkolni kívánt üzenet elrejtése vagy álcázása volt használatban. Néhány klasszikus példa:

- az i.e. 480-ban lezajlott szalamiszi csata, mely a perzsa hajóhad döntő vereségével végződött, illetve Xerxes egész görög hadjárata is másképp végződhetett volna, ha a hadjárat előtt egy Perzsiába száműzött görög nem figyelmezteti honfitársait a közelgő támadásra egy rejtett üzenettel, melyet egy viasszal bevont, üresnek látszó írótáblán, a viaszréteg alatt helyezett el. Az „üres” írótábla gond nélkül eljutott Perzsiából a címzettekhez.
- a diplomáciai levelezésben alkalmazták a küldönc fejére írt, lábbelijébe rejtett stb. üzeneteket, melyekről maga a küldönc sem tudott.
- elhalványuló, de valamilyen kezelésre megjelenő tintával (pl. tej, citromlé, pitypang-nedv stb.), vagy nem látható helyekre (postabélyeg alá, kemény tojás héja alá stb.) írtak üzeneteket.

A fenti szteganografikus módszerek közös jellemzője, hogy a feladónak és a címzettnek előzetesen és titokban meg kellett állapodni a rejtés módjában, és ha ezt a támadó megtudta, akkor a rejtett üzenetet könnyen el tudta olvasni, akár meg is tudta hamisítani. Mivel pedig a titkos módszerek előbb-utóbb kitudódnak, ezért a szteganográfia alkalmazása általában egyedi és erősen korlátozott. Ennek ellenére még a II. világháborúban is alkalmazta pl. a francia ellenállás. Napjainkban pedig a különféle digitális tartalmak, elsősorban kép- és hangfájlok, de akár teljes relációs adatbázisok digitális vízjelzésére (digital watermark) is alkalmazzák a szteganográfia elvét. A cél lehet a szerzői jogi információk elrejtése vagy a zajcsökkentés is. A leggyakrabban azonban az üzenet elrejtését nem önmagában, hanem valamilyen kriptográfiai módszerrel kombinálva alkalmazzák. A szteganográfia és a kriptográfia közötti átmenetre példa az üzenet betűinek összekeverése valamilyen előre megállapodott séma szerint. Néhány példa:

- A sokszögletű pálca, melyet már az ókorban is alkalmaztak. A pálcára spirálisan egy szíjat tekertek, majd az üzenetet a szíjra, a pálca oldalai mentén írták fel. Letekerés után a betűk összekeveredtek, pl. ötszög keresztmetszetű pálca esetén ötös csoportokban. A címzettnek ugyanolyan pálcával kellett rendelkeznie, mint a feladónak, melyre a szíjat feltekerve az üzenet elolvashatóvá vált.
- A Cardano-rejtjelező egy rács, melyet a négyzetrácsos cellákra osztott titkos üzenet fölé helyeztek, és a megjelölt cellák betűit olvasták össze. A rács 90 fokos elforgatásával és ismételt alkalmazásával akár hosszabb szövegek betűit is össze lehet keverni.

Az első példában a sokszögletű pálca, a másodikban a rács tekinthető egyfajta titkos kulcsnak, amely megkönnyíti az üzenetmegfejtését. A betűk összekeverésének, **permutálásának** elvét **P-doboz** néven a modern kriptográfia is alkalmazza.

A klasszikus kriptorendszerek

A történelem megmutatta, hogy a támadó általában előbb-utóbb megismeri a titkosan kommunikáló felek által használt módszert, sőt általában a titkos üzenet nyelvét és témáját is. Olyan módszert kellett tehát találni, melynek feltöréséhez ezeken kívül még egy titkos kulcs is szükséges. Előnyös, ha a kulcsot nehéz kitalálni, és ha a felek időről időre új kulcsokat alkalmaznak. Néhány klasszikus példa:

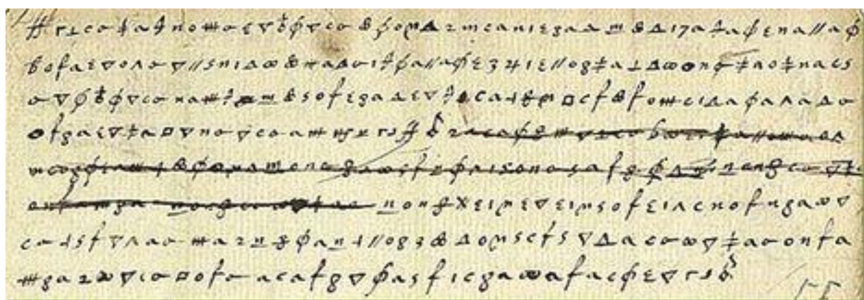
- A Caesar-kód: minden betű helyett az utána következő n-edik betűt írjuk. A lehetséges kulcsok száma így (26 jegyű ABC-t használva) mindössze 26. A Római Birodalomban alkalmazták.
- Ennek továbbfejlesztett változata minden betű szisztematikus helyettesítése egy másik betűvel. Így már 26! kulcs lehetséges, azonban a kulcs egy táblázat, amit nehezebb titkosan kezelni. Ezt a műveletet **S-doboz** néven ismeri a modern kriptográfia.

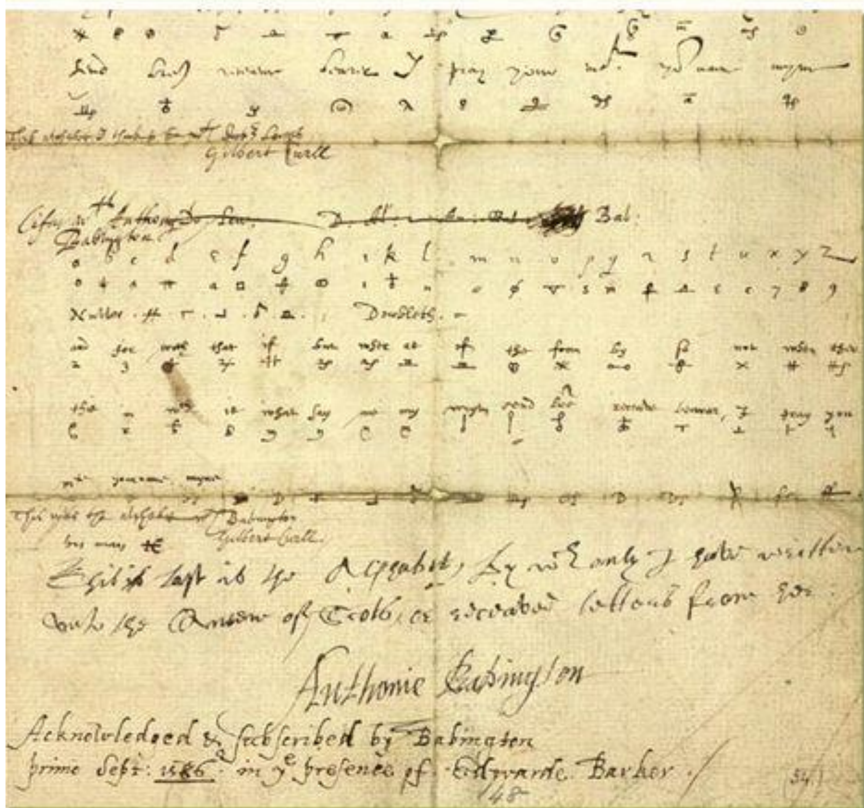
A fenti módszert **egyszerű monoalfabetikus** (betűhelyettesítéses) kódnak nevezzük. Ennek törésére a IX. századig kellett várni, amikor arab tudósok a Korán kéziratainak eredetiségét vizsgálva feltalálták a **betűgyakoriság-statisztikákra** alapozott elemzést. Az üzenet nyelvének és témájának ismeretében a betűpárok könnyűszerrel felderíthetők a természetes nyelvek redundanciáját felhasználva, feltéve, hogy elegendően hosszú üzenet áll rendelkezésre[1]. Ennek ellenére a monoalfabetikus kriptorendszereket, időnként a virágnyelvvel kombinálva, Európa-szerte alkalmazták (és rutinszerűen törték) a diplomáciában. A támadások megnehezítésére a következő javításokat alkalmazták:

- egyes jelek betűket, mások szótagokat vagy egész szavakat jelentettek
- más jelek nem jelentettek semmit (*nullitások*), vagy duplázták, esetleg törölték az előttük álló jelet

Így már olyan mértékben el lehetett bonyolítani a rendszert, hogy például az 1626 után XIII. és XIV. Lajos udvarában a Rossignolok által tervezett, diplomáciai titkok lejegyzésére használt „Nagy Kód”-ot csak 200 évvel később tudták megfejteni. Azonban nem mindig vizsgázott a kód ilyen jól: egy hasonló kódot, melyet a börtönben lévő Mária skót királynő használt az angol királyné ellen összeesküvő katolikus angol nemesekkel való kapcsolattartásra, könnyedén feltört Erzsébet angol királynő titoknoka, sőt a levelezést meghamisítva még a lázadó urak neveit is kicsalta. Valószínű, hogy a lázadók nem írták volna meg a neveket, ha nem hisznek vakon a kód biztonságában, tehát a kriptográfia alkalmazása ebben az esetben kifejezetten káros, sőt végzetes volt a számukra. Ez a jelenség számtalanszor ismétlődött már a történelem során, és a „**korhadt kilátókorlát jelenség**” néven ismeretes. A gyenge kód Mária királynő és az angol nemesek kivégzéséhez vezetett 1586-ban. Alább látható a titoknok által Mary leveléhez hamisított rész, melyben Mary kéri szövetségeseit, írják meg a nevüket (felül), alatta pedig a levelezéshez használt betű- és szókódok.

[1] lásd http://www.unsecure.co.uk/attackingmonoalphabeticciphers.asp





A monoalfabetikus kód legkomolyabb továbbfejlesztése azonban a **homofonikus** kód feltalálása volt: ebben a gyakori betűknek több jel felelt meg, melyek közül véletlenszerűen választottak. Ez is törhető statisztikai elemzéssel, azonban az elemzést a betűkapcsolatokra kell kiterjeszteni, tehát lényegesen nagyobb erőfeszítést és hosszabb titkos üzenetet igényel a sikeres támadás. Sikeres homofonikus monoalfabetikus kódra példa a XVII. századi ún. Copiale-cipher, melyet modern informatikai módszerekkel is csak 2011-ben tudtak megfejteni[2].

A kriptográfia következő nagy ugrása a **polialfabetikus** kód feltalálása volt, ami Vigenére-kód, vagy „a feltörhetetlen kód” néven vált ismertté. A sors iróniája, hogy publikálása éppen 1586-ra esik, tehát megmenthette volna Mária királyné életét. A módszer lényege, hogy nem egy, hanem több kód-ABC-t (betűhelyettesítési táblát) használnak, melyek között betűnként váltanak. Ha például 5 táblánk van, akkor az 1., 6., 11. stb. betű helyettesítéséhez használjuk az első táblát, a 2., 7., 12. stb. betűhöz a második táblát és így tovább. Az eredeti (legegyszerűbb) esetben a helyettesítési táblák az eredeti ABC

[2] lásd http://itcafe.hu/hir/copiale_cipher_rejtjel_titkosiras_megfejtes.html

ciklikus eltoltjai valahány pozícióval, például a 2-es számú tábla a B-t D-re cseréli. Ebben az esetben a lehetséges táblák száma: az eredeti ABC betűinek a száma - 1. A kulcs a felhasznált táblák sorszáma, pl. 21, 2, 5, 7, 2 egy öt hosszú kulcs esetén. A módszer továbbfejleszthető tetszőleges betű-összerendelést megengedő táblák használatával, ami a lehetséges táblák számát jelentősen megnöveli (44 jegyű ABC esetén 44!). Ekkor viszont a titkos csatornán előzetesen megosztandó titok is sokkal nagyobb méretű lesz, hiszen meg kell állapodni az egyes táblák tartalmában.

Az egyszerű betűgyakoriság-statisztikák alkalmazása ennél a módszernél azért nem működik, mert az egyes betűk pozíciójától függően változik a helyettesítési szabály. 1854-ig kellett várni, míg egy angol matematikus-filozófus, Charles Babbage, sikerrel járt. A törés alapgondolata az, hogy először a kulcs hosszát, tehát az alkalmazott táblák számát kell megállapítani, utána az egyes táblák tartalmát a szokásos betűgyakoriság-statisztikai módszerrel ki lehet deríteni. Például ha a kulcs hossza 6, akkor a második tábla felderítéséhez csak a 2., 8., 14. stb. pozíciókon álló betűkből készítünk statisztikát. Természetesen a támadónak hosszabb (jelen esetben 6-szor olyan hosszú) rejtett szövegre van szüksége a sikerhez, mint az egyszerű monoalfabetikus esetben. A kulcs hossza pedig abból határozható meg, hogy minden nyelvben vannak nagyon gyakori betű-kapcsolatok, például az angolban ilyen a th. Ha két th távolsága a nyílt szövegben éppen a kulcs hossza, vagy annak egész számú többszöröse, akkor a rejtett szövegben is ugyanaz a két betű fog tartozni hozzájuk. Tehát a rejtett szövegben ismétlődő betűcsoportokat keresünk, ezek távolságainak pedig megkeressük a legnagyobb közös osztóját. Valószínű, hogy ez lesz a kulcs hossza.

A polialfabetikus kriptorendszerek további fejlődésének az I. és különösen a II. világháború adott nagy lendületet. A táblázatok váltogatását írógép-szerű titkosítógépek végezték. Az I. világháború kimenetét alapvetően meghatározta a korábban németbarát külpolitikát folytató USA belépése Németország ellen, ami egy titkosított diplomáciai távirat (az ún. Zimmermann-féle távirat) sikeres feltörésének volt köszönhető az angol titkosszolgálat részéről. A II. világháborúra a németek kifejlesztették az **Enigmát**, egy rotoros titkosítógépet, amelynek feltörhetetlenségében a német vezetés az intő jelek ellenére vakon hitt. Azonban az angol titkosszolgálatnak A. Turing közreműködésével sikerült a törés, ezért a szövetségesek a németek tudta nélkül éveken keresztül meg tudták fejteni a német hadvezetés által kiadott parancsok és egyéb üzenetek nagy részét. Ez akkora előnyhöz juttatta a szövetségeseket, amely nélkül a világháború kimenetele egészen másmilyen is lehetett volna, de a történészek szerint legalábbis minimum 5-7 évvel tovább tartott volna a harc. Talán ez a legnagyobb történelmi jelentőségű példája a „korhadt kilátókorlátba kapaszkodás” veszélyeinek.

Az Enigma, Arthur Scherbius találmánya, valóban tökélyre fejlesztette a nagyon nagy számú kód-ABC közötti váltogatást. Egymás után elhelyezett forgó tárcsákon lévő huzal-darabok (a rotorok) egy áramkört zártak, amely a billentyűzeten leütött betűhöz a tárcsák pillanatnyi állása szerint meggyújtott egy lámpát valamelyik betűnél. Tehát a nyílt szöveg begépelése során a lámpákról betűnként le lehetett olvasni a rejtett szöveg betűit és viszont. A tárcsák minden leütés után fordultak egyet, ezért a kód-ABC gyakorlatilag minden betű esetén más volt, betűgyakoriság-elemzést tehát nem lehetett alkalmazni. A kulcs a tárcsák induló helyzetéből, sorrendjéből és egy betűcserélő tábla (plugboard) beállításából állott, ezekkel a lehetséges kulcsok száma több, mint 10^{16} volt. Az angolok rendelkeztek az Enigma egy példányával, azonban így is több évi erőfeszítés és Alan Turing zsenialitása kellett hozzá, hogy a német

vezetés által naponta cserélt kulcsot néhány óra alatt megtalálják egy speciális gép segítségével, amely a „Turing-bomba” néven vált ismertté. Az alábbi kép egy katonai célú Enigma berendezést mutat.

Még 1918-ban szabadalmaztatták az Egyesült Államokban az ún. **Vernam-titkosítót**, amely „**one-time pad**” (eldobó kulcsú titkosítás) néven is ismert. Ez a módszer tömeges adatcserére nem alkalmas, de a diplomáciában a mai napig alkalmazzák, később részletesen tárgyaljuk.

Egyedi megoldások

A fent említett alapvető módszereken kívül még számtalan egyedi megoldást használtak a történelem során, több-kevesebb sikerrel. Néhány példa:

- **a szótár-módszer**: mindkét félnél van egy hosszabb szöveg, például egy könyv (szótár) ugyanabban a kiadásban. Az üzenet minden betűjéhez keresnek a könyvben is egy olyan betűt, az üzenetbe aztán azt írják le, hogy hányadik oldalon, hányadik sorban, hányadik betűt kell venni. Ez a módszer természetesen nem alkalmas tömeges kommunikációra, viszont a szótár nélkül gyakorlatilag megfejthetetlen, ezért gyakran alkalmazták a világháborúkban, sőt, még egy 1820 körül elrejtett, húszmillió dollár értékű kincs rejtekhelyét is egy (feltehetőleg) ilyen módon titkosított irat őrzi, a modern kriptoanalitikusok nem kis bosszúságára.
- **a természetes nyelvek**: köztudomású, hogy egy ismeretlen és teljesen idegen nyelvnek még a szavait, hangjait sem tudjuk elkülöníteni egymástól. Ez adta az ötletet, hogy az amerikai hadseregben a II. világháborúban navaho indiánokat alkalmazzanak titkosítási célokra a távol-keleti hadszíntéren. A navaho kódbeszélők egyszerűen elmondták a parancsot navaho nyelven, amit csak egy másik navaho kódbeszélő értett meg. Természetesen baj lett volna, ha az ellenség is szert tesz akár egyetlen navaho katonára, ez azonban nem történt meg, így mindmáig ez az egyetlen olyan, széles körben alkalmazott kriptorendszer, amit soha sem tudtak feltörni.

A kriptográfia története iránt mélyebben érdeklődő olvasóknak ajánljuk Simon Singh: Kódkönyv című, élvezetes könyvét [1].

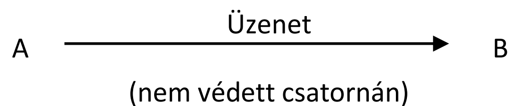
2. A modern kriptográfia alapjai

Alapfogalmak

A modern kriptográfiában használt, a korábbi gyakorlati eredményeket, módszereket rendszerező elmélet csak a XX. században született meg, és jórészt C.E. Shannon érdeme. A kriptográfia csak egy eleme az adatbiztonságnak, ami nagyon tág problémakör. Felöleli a védendő információt tartalmazó és közvetítő berendezések fizikai védelmét, a vállalat szervezeti felépítését és emberi erőforrásait, és az alkalmazott számítógépes információs rendszerek, nem utolsó sorban a kriptorendszerek biztonságát. Bármelyik területen is van hiányosság, az adatbiztonság veszélybe kerül. Ezen problémakörök közül itt a kriptorendszerek minőségi jellemzőivel, lehetséges megoldásaival foglalkozunk. A valóságban használt kriptorendszerek tervezésének alapelvei:

- a rendszer feltörésének a költsége legyen nagyobb, mint az információ aktuális maximális[3] piaci értéke, vagy
- a feltöréshez a jelenlegi eszközökkel szükséges idő alatt a titok évüljön el (veszítse el az értékét).

Zárt rendszerek védelme általában megfelelő fizikai, szervezeti és humán tervezéssel megoldható. A kriptográfiát érdeklő probléma az, hogy a külvilággal kommunikáló, *nyílt rendszert* hogyan lehet a rosszindulatú behatolás ellen megvédeni, illetve még inkább, hogy az ellenséges (nem védett) csatornán keresztül hogyan tud két fél, vagy két információs rendszer biztonságosan kommunikálni. Jó tervezés esetén a szervezeti, fizikai és informatikai rendszer-határ egybeesik. A kriptorendszerek alapsémája szerint az üzenetnek egy feladótól (A) kell eljutnia a címzetthez (B):



A kriptográfiai irodalomban szokás A-t Alice-nek, B-t Bobnak nevezni, a hallgatózó támadót pedig Eve-nek (az angol eavesdropper kifejezés nyomán). Az üzenetet a csatornán titkosított (rejtett) formában továbbítjuk. Az üzenetet eredeti formájában **nyílt szövegnek** nevezzük (jele X, plaintext), titkosítva **rejtett szövegnek** (jele Y, ciphertext). A rejtett szöveget a nyíltból a rejtés (encryption, E) révén kapjuk meg, melynek a nyílt szövegen kívül általában egy K **kulcs** is a paramétere. A vevőoldalon a fejtés (decryption, D) állítja vissza ugyanazon kulcs és a rejtett szöveg alapján a nyílt szöveget. Összegezve:

$$Y = E_K(X)$$
$$X = D_K(Y)$$

Feltehető, hogy a támadó fél ismeri az üzenet nyelvét és a titkosítás módját (E és D algoritmusát), de nem ismeri az éppen használt K kulcsot, ezért nem tudja kitalálni X-et. Az ilyen alapsémájú

[3] „maximális” alatt azt értjük, amennyit az információ a számunkra legkellemetlenebb támadónak ér. Háborúban ez az ellenfél, az üzleti világban a konkurencia, más esetben egy napilap, vagy az adóhatóság.

kriptorendszereket **titkos kulcsú** rendszereknek nevezzük. A kriptorendszert támadó külső fél a következő alapvető módszereket használhatja:

- Hallgatózás, vagy „rejtett szöveg” típusú támadás. A támadó a csatorna forgalmának a figyelésével, az üzenet nyelvének, esetleg témájának az ismeretében próbálja megfejteni az aktuális üzenetet vagy kitalálni a K kulcsot.
- „Nyílt szöveg” típusú támadás. A támadó ismeri egy korábbi üzenet nyílt és rejtett formáját is, ebből próbálja kitalálni a kulcsot, melyet későbbi üzenetek megfejtésére használhat.
- „Választott nyílt szöveg” típusú támadás. C valahogyan rá tudta venni A-t, hogy egy általa választott nyílt szöveget rejtsen, és küldjön el B-nek. A nyílt és rejtett üzenet-párból könnyebb következtetni a kulcsra.
- „Választott rejtett szöveg” típusú támadás. C valahogyan rá tudta venni A-t (például ideiglenesen hozzáfért a fejtő berendezéshez), hogy egy általa konstruált rejtett szöveghez a nyílt szöveget állítsa elő K használatával, és ezt a nyílt szöveget is megszerezte. A rejtett szöveg ügyes konstrukciójával a nyílt és rejtett üzenet-párból könnyebb következtetni a kulcsra.

Ezek a támadások alapvetően a kulcs megszerzésére irányulnak. A támadó azonban már akkor is jelentős kárt tud okozni, ha manipulálni tudja az A-t B-vel összekötő hálózati elemeket:

- Közbeékelődés („**man in the middle**” támadás). A valójában nem B-vel, hanem C-vel van összeköttetésben., B valójában nem A-tól, hanem C-től kapja az üzenetet, de ezt A és B nem veszi észre. Egy hosszú üzenetváltás-sorozat során C hamis identitással mindvégig A és B közé ékelődik, minden üzenetet megfejt, és a megfelelő pillanatban leadja a számára előnyös üzenetet. Hasonló mesterkedés okozta Mária királynő vesztét 1586-ban.
- Átírás (**forging**). Az üzenet C-n keresztül halad, aki elfogja azt, és bár megfejteni nem tudja, egyes részeit megváltoztatja. Például korábban megfigyelt rejtett-nyílt üzenetpárok alapján egy korábbi rejtett üzenetből egy ismert értelmű darabot betesz egy rejtett üzenet megfelelő darabja helyére („**cut-and-paste**” támadás). Esetleg egy korábban megfigyelt ismert tartalmú teljes üzenetet küld el („**replay attack**”), ami C-nek előnyös reakciót vált ki B-ből.

A kriptográfia szempontjából az alapprobléma a rejtett szöveg típusú támadás. Lehet-e, és ha igen, hogyan, olyan „tökéletes” kriptorendszert készíteni, amelyben a kulcs ismerete nélkül *lehetetlen* a nyílt szöveg meghatározása a rejtett szöveg megfigyelésével?

A tökéletes titkosítás

Tökéletesnek nevezzük azt a titkosítási rendszert, amelyben a rejtett szöveg átlagosan semmi információt nem árul el a nyílt szöveggel kapcsolatban, vagyis amelyre

$$I(X,Y)=0$$

Megmutatható, hogy ez akkor és csak akkor teljesül, ha

$$H(X\mid Y)=H(X) \text{ és } H(Y\mid X)=H(Y), \text{ azaz ha}$$
$$p_{i|j}=p_i \text{ és } p_{j|i}=p_j \ \ \forall i,j-\text{re}$$

Itt $H(X|Y)$ a nyílt szövegnek a rejtett szövegre vonatkozó feltételes entrópiája, $p_{i|j}$ pedig az i-edik nyílt szöveg valószínűsége, feltéve, hogy a csatornán a j-edik rejtett szöveg lett elküldve. Eszerint egy adott rejtett szöveghez ugyanakkora eséllyel kell tartoznia bármelyik nyílt szövegnek (és fordítva). Mivel a kettőt a kulcs választása rendeli össze, és a lehetséges rejtett szövegek számának legalább akkorának kell lennie, mint a lehetséges nyílt szövegek számának, ezért a kölcsönös információ hiányára vonatkozó elvárásunkat úgy és csak úgy tudjuk teljesíteni, ha

- $|X| = |Y| = |K|$, tehát a lehetséges kulcsok, nyílt és rejtett szövegek száma egyenlő, és
- $P(K_i) = \frac{1}{|K|}$, tehát a kulcsok közül minden egyes üzenet rejtésekor véletlenszerűen választunk.

Ezek nehezen teljesíthető elvárásoknak tűnnek, azonban a már említett **one-time pad** (eldobó kulcsú titkosítás) esetén teljesülnek. E módszer szerint a nyílt szöveget bitenként egy kétbemenetű XOR kapura vezetjük, a másik bemenetre pedig egy, a nyílt szöveg hosszával egyező hosszúságú véletlen bitsorozatot vezetünk (ez a kulcs). A rejtett szöveg a kapu kimenete. A vevőoldalon ugyanezt a műveletet hajtjuk végre a rejtett szöveggel és ugyanazzal a kulccsal, így visszakapjuk a nyílt szöveget. Látható, hogy a fenti feltételek teljesülnek a sorozatok egyenlő hosszúsága miatt.

Azt, hogy ekkor nincs a nyílt és rejtett szöveg között kölcsönös információ, a fenti tételtől függetlenül is könnyű belátni. Annak a valószínűsége ugyanis, hogy a rejtett szöveg egy bitje például „0”,

$$P(Y=0) = P(K=0)P(X=0) + P(K=1)P(X=1) = \frac{1}{2}(P(X=0) + P(X=1)) = \frac{1}{2},$$

függetlenül a nyílt szövegtől, tehát az X és Y források függetlenek.

Fontos viszont megjegyezni, hogy mihelyt egy már használt kulcsot még egyszer használunk egy másik szöveg rejtésére, tökéletes titkosításunk igen sebezhetővé válik a nyílt szöveg típusú támadással szemben, mivel a kulcs meghatározása egy rejtett-nyílt üzenetpárból triviális. Tehát minden egyes üzenethez egy, a rejtett szöveg méretével egyező méretű, új véletlen kulcsot kell generálni, és azt biztonságosan eljuttatni a vevőhöz a kommunikáció kezdete előtt[4]. Ez a gyakorlati probléma igen erősen korlátozza a módszer használhatóságát. Ezért csak különleges esetekben és elsősorban a diplomáciában alkalmazzák, mint például az amerikai és orosz elnököt összekötő telefonvonal, a „forró drót”.

Általában véve, a titkos kulcsú kriptorendszerek egyik nagy problémája a titkos kulcsok biztonságos eljuttatása az összes résztvevőhöz, amit **kulcskezelési problémának** (key management problem) neveznek. A gyakorlatban használt kriptorendszerek ezért nem tökéletesek, hanem a fejezet

[4] Ha a kulcsokat újra felhasználják, akkor a one-time pad már rejtett szöveg típusú támadással is törhető statisztikai módszerekkel. A szovjet nagykövetségek is elkövették ezt a hibát a hidegháború éveiben, ennek következtében sok diplomáciai üzenetet meg tudott fejteni az amerikai titkosszolgálat. Az eredmény: számos szovjet ügynököt lepleztek le az USA-ban, a Rosenberg-házaspárt pedig 1953-ban kivégezték az atomtitok szovjet kézre játszásáért.

elején ismertetett alapelvek szerint tervezik őket. A kriptográfiában az elv alkalmazását a **számítási teljesítményre alapozott biztonságnak** (computational secrecy) nevezik.

A nyers erő módszere

A titkos kulcsú kriptorendszerek elleni legegyszerűbb támadás az összes lehetséges kulcs végigpróbálgatása, amit **a nyers erő** (brute force) módszerének is neveznek. Például egy 3 számjegyű bőrönd-számzár esetén erre akár fél óra is elég lehet. A megfelelő kombinációnál kinyílik a bőrönd, illetve az üzenek nyelvének és témájának ismeretében észrevesszük, hogy egy adott K kulcs esetén a $D_K(Y)$ értelmes üzenetet eredményez. A nyers erő alkalmazása természetesen *nem praktikus*, ha a lehetséges kulcsok száma túl nagy, illetve a kulcsok számától függetlenül *értelmetlen*, ha a one-time pad módszert alkalmazták. Ekkor ugyanis az $|X|=|Y|=|K|$ feltétel miatt a támadó az összes lehetséges kulcs végigpróbálgatása során *az összes lehetséges nyílt szöveget* állítja elő, melyek mindegyike egyenlő valószínű, és természetesen több értelmes üzenet is van közöttük. Tehát ezek közül a támadó nem tud választani. Az alábbi példában a kulcs1 és a kulcs2 éppen ellentétes értelmű nyílt szövegeket produkál[5]:

rejtett:	`PEFOGJ`		`PEFOGJ`
*kulcs*1:	`PLMOEZ`	*kulcs*2:	`MAAKTG`
nyílt1:	`ATTACK`	nyílt2:	`DEFEND` (mindkettő értelmes!)

Ezért állíthatjuk azt, hogy a one-time pad a rendelkezésre álló számítástechnikai erőforrásoktól és időtől függetlenül is feltörhetetlen.

A nyelvi entrópia

Nem tökéletes kriptorendszer alkalmazása esetén, ha a nyílt és a rejtett szöveg, vagy kulcs és a rejtett szöveg egyes betűi között közvetlen kapcsolat van, mint például az S-doboz vagy az eltolás esetén, akkor betűgyakoriság-statisztikák alapján a támadó könnyen le tudja szűkíteni a valószínű kulcsok körét. Sőt, nem is kell az összes valószínű kulcsot végigpróbálni, hiszen például az egyszerű monoalfabetikus kód esetén a kulcs darabjait külön-külön is ki lehet találni. Ha például a levél első szavából már megvan annyi, hogy „ked*es", akkor a hiányzó betű valószínűleg v, ez alapján pedig a kulcs negyedik betűje próbák nélkül is meghatározható. Ezekben a támadásokban a természetes nyelvek „beépített" redundanciáját lehet kihasználni.

A természetes nyelvek ugyanis nem tekinthetők emlékezet nélküli forrásnak, amennyiben egy forrásszimbólumnak egy betűt tekintünk, mivel egy-egy betű (és szó) előfordulási valószínűsége erősen függ a környezetétől, tehát a szöveg távolról sem véletlenszerű betűhalmaz. Ezt a jelenséget számszerűen a **nyelvi entrópiával**, vagyis a végtelen mértékben kiterjesztett forrás esetén az egy betűre jutó átlagos információmennyiséggel (entrópiával) lehet kifejezni:

$$H_L = \lim_{n\to\infty} \frac{H(A_1, A_2, \ldots A_n)}{n}$$

[5] A példában a rejtett szöveg minden betűje a nyílt szöveg és a kulcs azonos pozíción lévő betűjének mod 26 összege, a 26 karakteres angol ABC használatával.

ahol $H(A_1, A_2, \ldots A_n)$ az n-szeresen kiterjesztett forrás, amelyben tehát egy betű-n-est tekintünk egy forrásszimbólumnak. A természetes nyelvekre a szókészlet korlátozottsága miatt mindig $H(A_1, A_2, \ldots A_n) < n \cdot H(A)$. Például az angol nyelvre statisztikai vizsgálatokkal megállapított értékek:

n	$\frac{H(A_1, A_2, \ldots A_n)}{n}$
1	4 bit
2	3.56 bit
3	3.30 bit

A tényleges nyelvi entrópiát angolra 1.5 és 1 bit közé lehet tenni—az egy angol betű által ténylegesen hordozott átlagos információ tehát lényegesen kevesebb, mint a betűnkénti feldolgozás alapján várnánk. Ennek az az oka, hogy a természetes nyelv zajos közegben való kommunikációra készült, és nyelvbe épült redundanciát folyamatosan használjuk a zaj által okozott hibák javítására. A nyelvi entrópia alapján definiálhatjuk a **nyelvi redundanciát**:

$$R_L = 1 - \frac{H_L}{\log |A|},$$

amely az angol nyelvre 0.75-re adódik, vagyis a szöveg körülbelül 75%-a redundáns. Ezt a redundanciát lehet a támadás során kihasználni olyan módon, hogy egyszerre *sok* lehetséges kulcsot zárunk ki egyetlen próba során, egy kulcsrészlettel megfejtett üzenet-részlet képtelensége alapján. Például szinte biztos, hogy az üzenet nem kezdődik három Q betűvel. Megmutatható, hogy a nyelvi redundancia miatt a lehetséges kulcsok száma 0-ra csökken, vagyis a titkosítás statisztikai értelemben fel van törve, ha elegendően hosszú rejtett szöveget van alkalmunk megfigyelni. A feltöréshez szükséges betűk száma:

$$n_0 = \frac{\log |K|}{R_L \log |A|},$$

amely az egyszerű módszerek esetén meglepően alacsony érték[6]. A nyelvi redundanciára alapozott támadásokat természetesen ki lehetne védeni forráskiterjesztéssel, például *n* > 10 esetén már eléggé megközelíthető a nyelvi entrópia. A kiterjesztéssel azonban a szimbólumok száma (a forrás-ABC) és vele együtt a kulcs mérete is kezelhetetlenül nagyra nő. Hasonló okokból nem jó megoldás az sem, hogy a nyelv szavait, esetleg szócsoportjait tekintsük forrásszimbólumoknak.

3. Modern blokkos kriptorendszerek

Shannon javaslata nyomán olyan titkos kulcsú kriptorendszereket terveztek, melyek ugyan nem tökéletesek, mivel |K|<<|X|, de a rejtett szöveg és a kulcs közti statisztikai kapcsolat elmosásának köszönhetően az $E_K(X)$ függvény véletlenszerű leképezésnek tekinthető a kulcs ismerete nélkül, és ha csak egyetlen bit is hibás a kipróbált kulcsban, a megfejtett szöveg tökéletesen értelmetlen lesz. Ezáltal a támadónak nem marad más választása, mint a nyers erő alkalmazása, ami idő- és energiaigényes művelet. Ha pedig a lehetséges kulcsok számát elegendően nagyra választjuk, akkor a kriptorendszer—

[6] angol nyelvre az egyszerű monoalfabetikus betűhelyettesítéses kód esetén n_0 = 25.

bár nem tökéletes—gyakorlatilag nem támadható, biztonságos. Ez a számítási teljesítményre alapozott biztonság, a modern kriptográfia alapja. Kérdés azonban, hogy mekkora |K|-t válasszunk? Ha a kulcstér túl nagy, lassú és drága lesz a titkosítás, ha túl kicsi, veszélybe kerül a biztonság. Természetesen mindig ki lehet indulni a technika mai állásából, a jövőt azonban nehéz megjósolni.

A termodinamikai korlát

A modern kriptográfia talán legismertebb módszerét, a DES-t (Data Encryption Standard) 1977-ben vezették be[7]. A DES-re nem találtak érdemi algoritmikus törést, és úgy tűnt, hogy az 56 bites titkos kulcstér, mely 2^{56} lehetséges kulcsot jelent, örök időkre védelmet jelent majd a nyers erő típusú támadással szemben, legalábbis a polgári életben. A számítástechnika azonban a Moore-szabály[8] szerint fejlődött, és 1998-ban egy masszívan párhuzamos, 210 ezer dollárból épített számítógépen[9] maximum 186 óra alatt már az összes kulcsot végig lehetett próbálni. Ma pedig a DES töréséhez nincs szükség szuperszámítógépre. A DES tehát elavult. Ezért a kulcstér nagyságának a megítéléséhez a modern kriptográfia már nem a próbálgatás időszükségletét használja, hiszen ennek jövőbeli fejlődése ismeretlen, hanem az energiaszükségletből indul ki, mivel erre termodinamikai alapon alsó korlátot tudunk adni. Ha egy kulcs kipróbálásához akár egyetlen 0/1 átmenet elég is egy szuperszámítógépen, akkor is szükséges ehhez az átmenethez legalább egy energia-kvantum[10]. Ez alapján alsó korlátot lehet adni a nyers erő alkalmazásának az energiaszükségletére. Ha például a kulcs 192 bites (a lehetséges kulcsok száma 2^{192}), akkor a feltöréshez szükséges energia nagyobb, mint a Nap által egy év alatt kisugárzott összes energia, 256 bit esetén nagyobb, mint a Nap teljes élettartama alatt kisugárzott energia. Nem valószínű, hogy a támadónk ennyi energiával rendelkezik. A feltörési energia alsó korlátját **termodinamikai korlátnak** nevezzük.

Blokkos kriptorendszerek alapelvei

A feladat tehát az, hogy a támadót rákényszerítsük a kulcsok próbálgatására. A modern kriptorendszerek Shannon eredeti javaslata alapján ezt a következőképpen érik el:

1. A nyílt szöveget egyenlő méretű blokkokra osztják. A blokk mérete a titkos kulcs méretének a nagyságrendjébe esik.
2. A rejtést a blokkokra külön-külön végzik el, ugyanazt a titkos kulcsot használva[11]. A vevőoldalon is blokkonként fejtik meg és rakják össze az üzenetet. Ezt a működési elvet **blokkos kriptorendszernek** nevezzük. Általában a rejtési és a fejtési algoritmus ugyanazt a kulcsot

---

[7] A kriptográfia a katonai és diplomáciai alkalmazások miatt kevésbé nyilvános tudomány, ennek ellenére léteznek széles körben alkalmazott, szabványos algoritmusai és protokolljai. Ezek jó része, köztük a DES, AES, RSA, SHA stb. az amerikai National Institute of Standards and Technology (NIST) szabványa, azonban egyre erősebben jelentkeznek az EU ajánlásai is, lásd www.ecrypt.eu.org

[8] http://en.wikipedia.org/wiki/Moore%27s_law

[9] AWT Deep Crack, egy chipen 24 mag, 64x28 chip (43008 mag) 40 MHz-en, egy kulcsot 16 óraciklus (0.4 µs) alatt vizsgált meg, ezért maximum 186 óra (7.7 nap) alatt találta meg a kulcsot. A kulcs-teszt egyszerűen az első 3 megfejtett byte-ot ellenőrizte: ha mind a 3 alfanumerikus volt, megtaláltuk a kulcsot.

[10] legalábbis akkor, ha a ma ismert és működő digitális számítási architektúrákat vesszük figyelembe. A kvantum-számítógéppel ez természetesen változhat.

[11] mivel ugyanazt a titkos kulcsot több nyíltszöveg-blokk rejtés során is felhasználják, ezért a titkosítás természetesen nem tökéletes.

használja, és a rejtési és fejtési algoritmus is lényegében ugyanaz (tehát a kulcs ismételt alkalmazása a fejtés során mintegy „megszünteti” a kulcs hatását), ezért a kriptorendszert **szimmetrikusnak**[12] nevezzük. Ennek előnye, hogy a szükséges hardver illetve kódméret fele a nem szimmetrikus megoldásának.

3. A rejtési/fejtési műveletet több *iterációban* végzik el. Egy iterációs lépés bemenete egy szövegblokk és egy iterációs kulcs, kimenete egy szövegblokk. Az egyes iterációkhoz használt kulcsokat az eredeti titkos kulcsból állítják elő különféle keverési műveletekkel. A legelső iteráció szöveg-bemenete a nyíltszöveg-blokk, a további iterációk bemenete az előző iteráció kimenete. A legutolsó iteráció kimenete pedig a rejtettszöveg-blokk. A közbülső szövegblokkokat gyakran állapotnak (state) nevezik.
4. Egy iteráció tartalma eltolás és/vagy keverés (P-doboz) és nemlineáris függvény alkalmazása (ami általában helyettesítés, S-doboz) a bemeneti szövegblokkra az iterációs kulcstól függetlenül, majd az iterációs kulccsal való rejtés (általában XOR művelet). Az ilyen sémájú kriptorendszert gyakran **produkciós** rendszernek (product cipher) is nevezik.

A fenti elvű kriptorendszereket a DES alapjául szolgáló IBM Lucifer tervezője, Horst Feistel német származású amerikai kriptográfus után **Feistel-titkosítónak**, vagy Feistel-hálózatnak is nevezik.

A Feistel-titkosítóra jellemző, és a blokkos kriptorendszerektől elvárt az ún. **lavinahatás**:, tehát hogy a bemeneti blokk vagy a kulcs-blokk egy bitjének megváltoztatása 50% valószínűséggel változtasson meg minden bitet a kimeneti blokkban.

Bár jelenleg (2012-ben) még sok helyen alkalmazzák a DES-t, illetve ennek 2 vagy 3 kulccsal működő, 112 bitesre növelt kulcsú változatát, a TDES-t, a titkos kulcsú kriptorendszerek területén az AES nevű, 2000-ben bevezetett rendszeré a jövő, mivel flexibilisebb és hatékonyabb szoftver/hardver megvalósítást tesz lehetővé. A hatékonyságára jellemző, hogy x86-ra 457 byte-on is leprogramozták, és 700 Mbit/s feldolgozási sebességet értek el 2 GHz-en. Az AES működését és egyéb részleteit lásd az I. Mellékletben.

Minden alap-üzemmódban dolgozó blokkos titkosító támadható a rejtettszöveg-blokkok cseréjén alapuló kivágás-beillesztéses (cut-and-paste) támadással. Ennek kivédésére az egyes blokkok rejtéséhez felhasználják a korábbi rejtettszöveg-blokkokat is a titkosító láncolt üzemmódjában. A szabványos ECB, CBC, OFB, CTR üzemmódok leírását lásd a II. Mellékletben.

A véletlen kulcs

A titkos kulcsú kriptorendszerek alapkérdése a jó minőségű, tehát valóban véletlenszerű titkos kulcs készítése. Hiába elegendően nagy a kulcstér és hiába véletlenszerű a támadó szemszögéből az $E_K(X)$ leképezés, ha a kulcstérben egyes kulcsok valószínűbbek másoknál, vagy egyenesen kizárhatók a támadó számára. A titkos kulcs tehát egy véletlen bitsorozat. „Igazi” véletlen sorozathoz azonban csak valamilyen természeti folyamat mérése révén lehet hozzájutni, mint például a radioaktív sugárzás, az ellenálláson keletkező termikus zaj, a kozmikus háttérsugárzás, a gerjedő oszcillátor, a turbulens áramlás

---

[12] a szakirodalomban gyakran szimmetrikusnak neveznek minden titkos kulcsú kriptorendszert, és aszimmetrikusnak minden nyilvános kulcsú rendszert.

stb. Ha gyakran és nagy mennyiségben van szükség új kulcsokra, akkor az ilyen mérések költsége túl nagy lehet (lásd a szovjet titkosszolgálat esetét a one-time paddel). Gyakran felhasználják a felhasználói interakciót (például a leütések közti szüneteket vagy az egérmozgást) és a különféle számítástechnikai rendszerparaméterek (például a nyitott fájlok száma stb.) kombinációját is, bár ezek már kevésbé véletlenszerűek. A leggyakrabban alkalmazott megoldás azonban a különféle véletlenszám-generátorok alkalmazása. A generált sorozat minőségének megítéléséhez lényeges, hogy a sorozatról el tudjuk dönteni, „mennyire véletlen". Erre a kriptográfiában gyakran alkalmazzák a Samuel W. **Golomb**-tól származó következő három **kritériumot**:

1. A sorozatban az egyesek és nullák száma legyen közel egyenlő.
2. A homogén szakaszok relatív gyakorisága exponenciálisan csökkenjen a sorozat hosszával. Homogén szakasznak a csupa azonos szimbólumból álló sorozatot nevezzük, például a '011110' egy 4 hosszú 1-es sorozat, az '10001' pedig egy 3 hosszú 0-ás sorozat. A kritérium azt írja elő, hogy az *n* hosszú homogén szakaszok száma az egész álvéletlen bitsorozatban körülbelül a fele legyen az *n*-1 hosszú szakaszok számának.
3. A sorozat autokorreláció-függvénye minden pontban közel legyen a nullához. Az autokorreláció-függvényt az *x* pontban úgy képezzük, hogy a sorozatot *x* pozícióval ciklikusan eltoljuk, majd az eredeti és eltolt sorozat bitenkénti XOR függvényét képezzük. A XOR-olt sorozat egy adott pozícióján pontosan akkor lesz 1, ha az ezen a pozíción lévő bitek különböznek az eredeti és az eltolt sorozatban. Az autokorreláció-függvény *x*-beli értéke a XOR-olt sorozatban lévő 1-esek és 0-k számának a különbsége, osztva a sorozat hosszával. Ez a kritérium azt biztosítja, hogy az álvéletlen sorozat egy részletéből ne lehessen következtetni a hiányzó részekre.

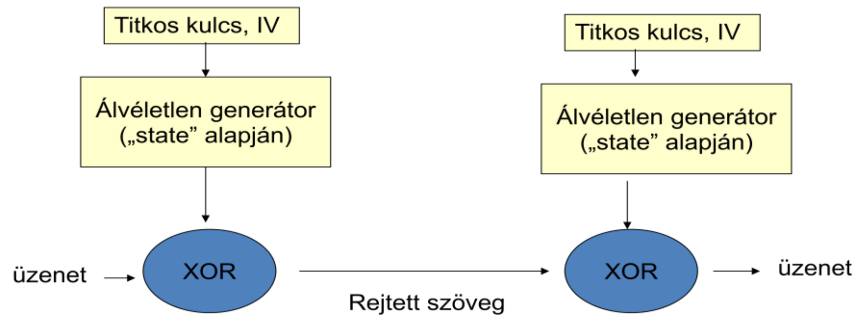
Ha sikerült egy titkos kulcsot minden résztvevőhöz eljuttatni, akkor azt csak korlátozott ideig lehet biztonságosan használni. Szoftver és hardver hibák, vírusok és egyéb támadások vagy emberi korrupció mindenhol előfordul, ezért a titkos kulcsot le kell cserélni. A gyors kulcs-cserélgetés nagyobb biztonságot, és ugyanakkor lassabb működést eredményez. Az 'igazi' titkos kulcsot nem lehet óránként fizikailag védett csatornán (páncélautóval) eljuttatni a felekhez, viszont egy titkos kulcsból sok kulcsot tud készíteni minden résztvevő:

- a titkos kulcsot egy álvéletlen generátor magjaként (seed) alkalmazva a generátor kimenete kulcsok gyakorlatilag végtelen sorát adja. Álvéletlen generátorként használható minden folyam- és blokktitkosító is, utóbbiak az OFB üzemmódban (lásd a II. Mellékletben).
- a titkos kulcsot lehet úgy is alkalmazni, hogy nem a valódi kommunikációra használják, hanem csak arra, hogy az egyik fél által generált rövid életű, véletlen kulcsot (session key) átvigyék. Az ilyen módon használt titkos kulcsot kulcsrejtő kulcsnak (**key encrypting key**, KEK) nevezik.

4. Modern folyam-elvű kriptorendszerek

A folyam-titkosítók előnye a blokkos titkosítókkal szemben a nagyobb működési sebesség, alacsonyabb ár és félvezető-felület, ugyanakkor a más működési elv miatt másmilyen jellegű támadásokra érzékenyek. A folyam-titkosító lényegében egy álvéletlen generátor, melyet a titkos kulccsal, mint

maggal egyszer inicializálnak[13], utána pedig a kimeneti álvéletlen bitfolyamot úgy használják fel, mint egy gyakorlatilag végtelen hosszú one-time pad kulcsot, tehát az adónál bitenként vagy byte-onként XOR-olják a nyílt szöveggel, a vevőnél pedig a kapott rejtett szöveggel. A „blokkméret" tehát itt 1 bit (vagy 1 byte), és a véletlenszám-generátornak van egy belső állapota (state) is, amely a következő kimeneti bitet (byte-ot) és a következő állapotot is meghatározza.



A folyamtitkosítókkal szemben támasztott 3 fő követelmény:

1. Nyílt szöveg típusú támadással az álvéletlen bitfolyamot ne lehessen visszafejteni a state-re (vagy csak a kimerítő keresésnél több művelettel). Ha a state ismert, akkor a támadó is tudja generálni az álvéletlen bitfolyamot.
2. A state ismeretében a titkos kulcsot ne lehessen visszafejteni
3. A kimenetet ne lehessen megkülönböztetni egy valódi véletlen sorozattól (a Golomb-kritériumok szerint)

Az 1. követelmény kielégítése és a state méretezése szempontjából fontos az ún. Babbage-Golic támadás.

A Babbage-Golic támadás

A Babbage-Golic támadás a Hellman-féle általános idő-tárhely ekvivalencia támadás[14] továbbfejlesztett változata. A támadás alapfeltevése az, hogy egy adott state után következő kimeneti bitfolyam a state nem invertálható véletlenszerű függvénye. Legyen *N* a lehetséges state-ek száma. A támadás lépései (dőlt betűvel a paraméterek):

1. a folyam-titkosító módszer ismeretében *előfeldolgozás*: egy táblázatba feljegyzik *M* darab véletlenül választott state-ből indított kimeneti bitfolyam első *b* bitjét ($b = \log N$).
2. nyílt szöveg típusú támadás, vagyis a rejtett és a nyílt szövegből meghatározzák az álvéletlen bitfolyam egy darabját, ez alapján próbálnak következtetni a state-re. Legyen a megfigyelt bitfolyam hossza $D+b$-1, ekkor ugyanis ez tartalmaz *D* darab *b* hosszúságú rész-sorozatot, melyek mindegyike egy másik state-ből indult el. A törés sikeres, ha ezen *D* state közül valamelyiket ki

---

[13] az inicializáláshoz a titkos kulcson kívül általában felhasználnak egy úgynevezett inicializációs vektort (IV) is. Ez a támadó számára is ismert lehet, gyakran egy véletlen érték, vagy például a rendszerparaméterek, rendszeridő függvénye. A célja az, hogy ugyanazzal a titkos kulccsal inicializálva ne kapjuk kétszer ugyanazt a state-et.

[14] Ezt az elvet alkalmazzák az üzenetpecsétek elleni szivárványtáblás támadás során is, lásd később.

tudjuk találni. Ezért minden rész-sorozatot megnézünk az előfeldolgozás során készített táblázatban.

Kérdés, hogy *M* és *D* milyen értékei mellett találunk meg legalább 0.5 valószínűséggel egy sorozatot a táblázatban? A születésnap-paradoxon szerint egy *N* elemű halmazból kivett *A* és *B* elemű részhalmazoknak várhatóan lesz közös eleme, ha $AB \geq N$. *N* a lehetséges state-ek száma, tehát például 128 bites state esetén $N = 2^{128}$, a részhalmazok pedig legyenek a state-ekhez tartozó b hosszúságú bitsorozatok. Várhatóan sikeres a támadás, ha $DM \geq N$. Jelölje *T* a próbálkozások számát, tehát a nyílt szöveg típusú támadás időszükségletét, és tegyük fel, hogy ugyanakkora energiát fordítunk az 1. és a 2. lépésre is, vagyis $T = M$. Mivel a próbálkozások száma megegyezik *D*-vel, ezért végül is azt kapjuk, hogy $T^2 \geq N$. Tehát a fenti előfeldolgozás esetén például a 128 bites state-tel rendelkező folyamtitkosító 2^{64} lépésben törhető. Ha ugyanennek a folyamtitkosítónak az inicializálás során felhasznált titkos kulcsa is 128 bites lenne, akkor a nyílt szöveg típusú támadás esetén a titkos kulcs teljes keresésénél (ami 2^{128} lépés) lényegesen egyszerűbb a state támadása, tehát a state mérete a titkos kulcshoz képest túl kicsi. Konklúzióképpen elmondható, hogy a folyamtitkosítók esetén *a state méretét legalább a titkos kulcs méretének kétszeresére* célszerű választani.

Néhány elterjedt folyam-elvű kriptorendszer

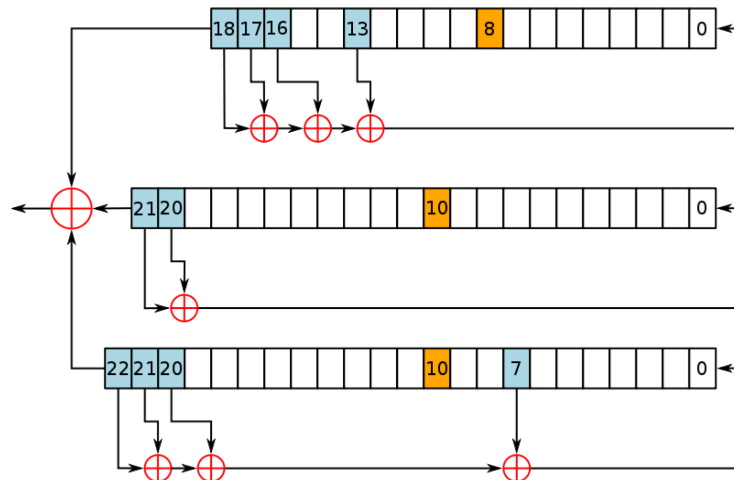
Az alábbiakban áttekintjük néhány széles körben használt folyamtitkosító jellemzőit.

- Az **RC4** (1987) 1 byte-os blokkokkal dolgozik, tízszer gyorsabb a DES-nél, az SSL egyik alap-algoritmusa. A titkos kulcs a 0..255 sorozat egy permutációja, tehát |K| = 256!, ami egy 1680 bites titkos kulcsnak felel meg, a periódus nagyon hosszú. Minden adatbyte-nál változik a 0..255 számokat tartalmazó kulcs-tömb elemeinek a sorrendje, majd kiválasztják belőle az aktuális kulcsbyte-ot. A kulcsbyte-ot XOR-olják a nyíltszöveg byte-tal. Nem találtak érdemi törést hozzá. Nem publikus, de az RC4-gyel gyakorlatilag ekvivalens algoritmus hozzáférhető.
- A GSM rendszerek a beszélgetések titkosítására visszacsatolt léptetőregiszterekre (LFSR) alapozott folyamtitkosítókat alkalmaztak **A5/1**, később a túl könnyű (gyakorlatilag real time egy PC-n) törhetőség miatt továbbfejlesztve **A5/2** néven[15]. A 128 bites titkos kulcs a SIM kártyán van tárolva, az ebből challenge-response alapon generált 64 bites viszonykulcsot az LFSR-ek inicializálására használják[16]. Alább az A5/1 blokkvázlata a 19, 22 és 23 bites léptető regiszterekkel. Ezekbe töltik indításkor a 64 bites titkos kulcsot. A kimeneti álvéletlen bitfolyam a 3 regiszterből kilépő 3 bit XOR-függvénye. A regiszterek középső bitjeinek (sárgán árnyékolva) az órajelek kapuzásában van szerepe.

---

[15] Ettől függetlenül még ma (2012-ben) is nagyon sok telefon alkalmazza a rendkívül gyenge A5/1-et. Az erősebb algoritmusok bevezetését elsősorban politikai okok miatt gátolják, elsősorban az USA-ban.

[16] Ez a fajta megoldás a “key encrypting key” (KEK) koncepciója.



- A jövő ígéretes folyamtitkosító algoritmusai az EU ECRYPT által ajánlott Rabbit és Grain. Rendkívül alacsony hardver költség és energiafelhasználás, magas működési sebesség, könnyű szoftver implementálhatóság és paraméterezhetőség, ugyanakkor megfelelő biztonság jellemzi őket. Ezek implementációs részleteit lásd az V. Mellékletben.

5. Nyilvános kulcsú kriptorendszerek és alkalmazásaik

Az internet térhódításával szükségessé vált olyan módszerek kifejlesztése, melyek nem igénylik titkos kulcsok védett leosztását, mivel a kommunikációban résztvevők köre előre nem ismert (például internetes boltok, szolgáltatások), vagy más ok miatt egyetlen egyszer sem lehet védett csatornát létesíteni köztük. A nyilvános kulcsú rendszerek olyan módon kerülik meg a kulcskezelési problémát, hogy a kulcs egy részét olyan formában (úgynevezett **egyirányú művelet,** trap function alkalmazásával) hozzák nyilvánosságra, hogy abból a kulcsra a jelenleg ismert módszerekkel csak nagyon nagy számítási teljesítmény felhasználásával lehet visszakövetkeztetni. Megmutatható, hogy bármilyen nyilvános kulcsú kriptorendszerhez szükség van ilyen egyirányú művelet alkalmazására. Többféle egyirányú műveletet használnak, melyek hatásukban mind olyanok, mint Hamupipőke próbatétele az összekevert hamuval és liszttel: összekeverni nagyon könnyű, szétválasztani szinte lehetetlen. Néhány nevezetes egyirányú művelet:

- A diszkrét logaritmus (DLP). Ha a, x, és p ismert, akkor $y = a^x$ mod p nagyon gyorsan számítható, de a, y, és p ismeretében x számítása nagy számok esetén nagyon komplex, idő- és energiaigényes feladat. A Diffie-Hellman kulcsmegosztási algoritmus és az Elgamal algoritmus alapja.
- Nagy prímszámok szorzatának faktorizálása (PFP). Ha p és q nagy prímszámok, akkor $n = pq$ számítása könnyen elvégezhető, de n-ből p és q meghatározása nagyon komplex, idő- és energiaigényes feladat. Az RSA algoritmus alapja.
- Véges testek felett definiált elliptikus görbék pontjainak szorzása természetes számmal (ECDLP), bővebben lásd A kriptográfia jövője alfejezetben.

Fontos megjegyezni, hogy az egyirányú műveletek inverzének számítása a matematika aktív kutatási területe, és elvileg bármikor születhet olyan módszer, amely gyors invertálást tesz lehetővé, miáltal az

erre épülő kriptográfiai algoritmusok értelmetlenné vagy legalábbis sebezhetővé válnak. A titkos kulcsú kriptorendszerek esetén sokkal kevésbé fenyeget ilyen veszély. A nyilvános kulcsú kriptorendszerek másik hátránya a relatív lassúságuk és nagy erőforrás-igényük. Ezen két probléma miatt a nyilvános kulcsú rendszereket általában nem önmagukban, hanem titkos kulcsú rendszerekkel és üzenetpecsét-algoritmusokkal kombinálva, különféle protokollokba ágyazottan alkalmazzák. Az ilyen kriptorendszert **hibrid kriptorendsze**rnek nevezik. A hibrid rendszerben a nyilvános kulcsú kriptorendszer tipikus feladata az autentikáció és a véletlen viszonykulcs megosztása a résztvevők között. A sikeres kulcsmegosztás után a felek a kommunikáció érdemi részét már egy titkos kulcsú kriptorendszer használatával folytatják.

Hibrid kriptorendszer a HTTPS mögött álló SSL (Secure Socket Layer) protokoll, melyben a felek a session elején állapodnak meg abban, hogy melyik nyilvános és titkos kulcsú kriptorendszereket, milyen paraméterekkel fogják használni a továbbiakban. Szintén TDES-RSA alapú hibrid rendszer a mobilbank-szolgáltatások alapja.

A Diffie-Hellman kulcsmegosztási protokoll

Az első nyilvános csatornán működő nyilvános kulcsú kulcsmegosztási protokoll, a Diffie-Hellman protokoll 1976-ból származik. A két kommunikáló fél célja a későbbi kommunikáció során használandó titkos viszonykulcs biztonságos megosztása. A protokoll lépései:

1. Alice és Bob nyilvánosan megállapodnak egy nagy (több ezer bites) *p* prímszámban és egy kis *g* számban, amely a *p* alapú véges test elemeiből álló ciklikus csoport generátora, tehát

$$\forall x \in GF[p] - re \; \exists \, k : g^k = x \, mod \, p$$

2. Alice választ egy véletlen *a* < *p* részkulcsot, és elküldi Bobnak a $k_a = g^a mod\, p$ számot. Hasonlóképpen Bob is választ egy véletlen *b* < *p* részkulcsot, és elküldi Alice-nak a $k_b = g^b mod\, p$ számot a nyilvános csatornán. A támadónak az *a* és *b* meghatározásához meg kell oldania a diszkrét logaritmus problémát.
3. Mindkét fél kiszámítja a $K = {k_b}^a \, mod\, p = {k_a}^b \, mod\, p = g^{ab} \, mod\, p$ számot. Ezt a továbbiakban egy titkos kulcsú rendszer kulcsaként használják.

A Diffie-Helmann protokoll kis módosítással alkalmas üzenet-titkosításra is. Ez a megoldás a feltalálója után **ElGamal kriptorendszer**[17] néven vált ismertté (1984).

Az RSA kriptorendszer

Az RSA a legelterjedtebben használt nyilvános kulcsú, **aszimmetrikus** kriptorendszer. Az aszimmetrikus jelleg azt jelenti, hogy Alice és Bob tevékenysége és az általuk használt kulcs is különbözik egy üzenet rejtésekor, illetve fejtésekor. A támadó számára rendelkezésre álló kulcs-részből, az ún. **nyilvános kulcs**ból a rejtett szöveg megfejtéséhez szükséges kulcs-rész, az ún. **privát kulcs** csak a PFP probléma megoldásával lehetséges. A nyilvános kulcs birtokában üzenetet rejteni bárki tud, fejteni azonban csak a publikus kulcshoz tartozó privát kulccsal lehet, melyet a címzett soha nem tesz közzé. Az RSA részletes

[17] Az ElGamal pedig a DSA amerikai digitális aláírás-szabvány alapja.

leírása megtalálható a IV. Mellékletben, egy demo alkalmazás pedig a http://vassanyi.ginf.hu/info/rsa/ lapon.

Az RSA már 1977 óta használatban van, ezért számos támadást dolgoztak ki ellene. Biztonságosan alkalmazni csak az ajánlások szigorú betartásával, és kellőképpen nagy kulcsmérettel lehet. Az ismert támadások:

- alapvető aritmetikai alkalmazási hiba kihasználása (n vagy x túl kicsi)
- egyéb rosszul választott paraméterekből adódó algoritmikus gyengeségek kihasználása
- választott rejtett szöveg típusú támadás (1998): megfigyelt Y alapján konstruált Y' és az ehhez tartozó X' alapján X. A címzettet (illetve annak számítógépétí) egy RSA-t használó protokoll hibáját kihasználva vették rá arra, hogy a konstruált Y'-t dekódolja és az eredményt visszaküldje. A kriptográfiában az ilyen rejtett, előre tervezett, kikényszerített műveletet **oracle service**-nek hívják.
- időzítéses támadás: a fejtési művelet időszükséglete összefügg a privát kulccsal, és a fejtés időszükségletét egy protokoll gyengeségét kihasználva meg lehetett határozni. Ezáltal a támadó (az üzenet küldője) jelentősen le tudta szűkíteni a lehetséges privát kulcsok körét.

Beékelődéses támadás nyilvános kulcsú kriptorendszerek ellen

Nemcsak az RSA, hanem minden nyilvános kulcsú kriptorendszer alapproblémája, hogy éppen mivel a felek sohasem találkoznak, ezért nehéz megbizonyosodniuk egymás kilétéről. A beékelődéses (**man in the middle**) támadás jellemző lépései az RSA esetén, ha B kíván titkos üzenetet küldeni A-nak, és a támadónak (E-nek) módjában áll manipulálni az üzeneteket:

1. B elkéri A-tól A nyilvános kulcsát
2. E elfogja ezt a kérést, elkéri A nyilvános kulcsát, de nem ezt, hanem a saját nyilvános kulcsát küldi vissza B-nek
3. Ezután E minden B által írt üzenetet megfejt a saját titkos kulcsával, majd újra elrejti A nyilvános kulcsával és elküldi A-nak.

E-nek azon túl, hogy minden üzenetet el tud olvasni, módja van az üzenetek átírására, illetve B nevében írt hamis üzenet konstruálására is. A és B mindebből nem vesz észre semmit, sőt éppen a sikeresen megfejtett üzenetek győzik meg őket arról, hogy a kommunikáció titokban folyt le. Ez a támadás csak akkor védhető ki, ha A és B valamilyen külső segítséggel meg tud győződni egymás kilétéről, úgy ahogy a személyi igazolvány is igazolja az ismeretlen tulajdonos bizonyos adatait. Ilyen igazolást a kriptográfiában a külső hatóság vagy cég által digitálisan aláírt digitális tanúsítványok (certificate, lásd alább) tudnak nyújtani.

A digitális aláírás az RSA segítségével

Az RSA, és más nyilvános kulcsú kriptorendszerek is, használhatók az üzenet-titkosításos algoritmus megfordításával is. Ekkor A, a privát kulcs birtokosa a privát kulcs használatával először rejti az üzenetet, melyet aztán közzétesz, és amelyet a nyilvános kulcs használatával bárki meg tud fejteni. Ennek gyakorlati értelme akkor van, ha nem az üzenet titkosságát, hanem annak **letagadhatatlanságát** és **sérthetetlenségét** (Non-repudiation and integrity) szeretnénk a nyilvános kulcsú rendszerrel biztosítani:

- ha a rejtett üzenetet valaki megváltoztatja, utána már nem lesz A nyilvános kulcsával (értelmesen) megfejthető , tehát ha a fejtés sikeres, az üzenet nem sérült
- mivel a nyilvános és a privát kulcs egyedi párt alkot, ezért ha a fejtés sikeres, akkor azt biztosan A rejtette el vagy legalábbis az ő privát kulcsát használták a rejtéshez

Hatékonysági megfontolásokból nem az egész üzenetet, hanem csak egy abból készített kis méretű ún. **üzenetpecsétet** (hash) rejtenek el egy dokumentum digitális aláírása során. A hash algoritmus biztosítja, hogy ha a dokumentum változik, akkor a belőle készített hash érték (a pecsét) is változzon. A digitális aláírás gyakorlati alkalmazásának lépései:

1. A elkészíti (vagy másoktól elfogadja) az aláírandó dokumentumot, választ egy hash algoritmust, és ezzel kiszámítja a dokumentum hash értékét
2. a hash értéket és az algoritmus nevét, paramétereit a privát kulcsával rejti egy állományba, amit **digitális aláírás**nak hívnak
3. a dokumentumot és az aláírást publikálja a saját nyilvános kulcsával együtt
4. ha valaki meg akar győződni a dokumentum hiteléről, akkor a nyilvános kulccsal először is megfejti a dokumentumhoz tartozó aláírást, majd
5. az aláírásban található algoritmussal és paraméterekkel kiszámítja a dokumentum hash értékét
6. ha a kiszámított érték egyezik az aláírásban lévő hash értékkel, akkor a dokumentum nem sérült (eredeti) és azt A privát kulcsával írták alá.

Magyarországon 2001. óta a digitális aláírás egyenértékű a papír-alapú aláírással (2001. évi XXXV. törvény).

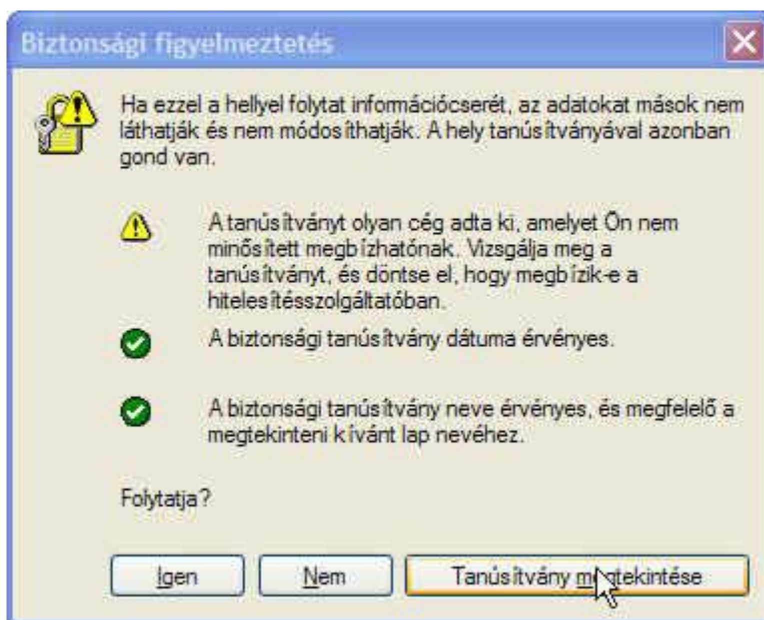
Digitális tanúsítványok és tanúsítási rendszerek

A digitális aláírás technológiáját fel lehet használni a beékelődéses támadás kivédésére. Ehhez egy olyan dokumentumot az ún. **digitális tanúsítványt** szerkesztenek, mely A személyazonosságát és A nyilvános kulcsát együtt tartalmazza, majd ezt egy olyan szervezet, az ún. **certificate authority** (CA) látja el digitális aláírással a saját privát kulcsa használatával. A CA nyilvános kulcsa mindenki számára elérhető kell, hogy legyen. Ezután, ha bárki meg akar győződni arról, hogy A-nak tényleg az-e a nyilvános kulcsa, ami a tanúsítványon áll, akkor a CA nyilvános kulcsával megfejti az aláírást, és ellenőrzi a dokumentum sértetlenségét. A digitális tanúsítvány kötelező részei az X509 szabvány szerint:

- a tanúsítvány sorszáma és verziója
- a kibocsátó CA neve és egyéb adatai
- a tanúsítvány időbeli érvényessége (a lejárt tanúsítvány érvénytelen)
- a tulajdonos neve és egyéb adatai
- a tulajdonos nyilvános kulcs algoritmusa és a nyilvános kulcs értéke
- az aláíráshoz használt algoritmus
- a fentiekhez, mint dokumentumhoz a CA által készített digitális aláírás

A tanúsítvány használatával elkerülhető a beékelődéses támadás, mert a támadó hiába írja át a tanúsítvány dokumentum részében a nyilvános kulcsot, a CA privát kulcsa nélkül az aláírást nem tudja átírni, ezért B észreveszi a módosítást.

Természetesen felmerül a kérdés, hogy egy interneten kapott tanúsítvány ellenőrzésekor hogyan szerezzük be a CA nyilvános kulcsát. A beékelődéses támadás elkerülésére ezt a CA *saját* tanúsítványából kell kiolvasnunk, melyet egy CA feletti hatóság vagy szervezet írt alá, és így tovább. Az egymásra hivatkozó tanúsítványok sorozatát **hierarchikus tanúsítási láncnak** (chain of trust) nevezzük. A lánc végén egy olyan szervezetnek kell állnia, melynek nyilvános kulcsa minden résztvevő (web-böngésző program, operációs rendszer) számára eleve ismert, beégetett. Az ilyen nyilvános kulcs neve **bizalmi horgony** (trust anchor). A tanúsítványokkal biztosított kommunikáció, például egy https protokollal elérhető weblap böngészése előtt a magát igazolni kívánó fél (jelen esetben a webszerver) a teljes, horgonyig vezető tanúsítási lánc összes tanúsítványát elküldheti a másik félnek. Ha a láncot nem sikerül egy horgonyig követni, akkor a felhasználónak kell megítélnie, hogy elfogadja-e a tanúsítványt, tehát belép-e a weblapra, vagy telepíti-e a bizonytalan eredetű szoftvert. Alább egy böngésző által adott ilyen témájú figyelmeztetés.



A tanúsítványokkal is vissza lehet élni. Ahogy a személyi igazolványt, vagy a bankkártyát, a privát kulcsot is el lehet lopni, illetve a hatóság is bevonhatja például az eljárás alá vont cég tanúsítványát. Ezért a bevont vagy a tulajdonos kérésére érvénytelenített tanúsítványokról erre szakosodott szolgáltatók bevonási listákat (**certificate revocation list**, CRL) vezetnek, melyeket a tanúsítvány végleges elfogadása előtt célszerű ellenőrizni[18].

[18] Például a Mozilla Firefox böngészőben a CRL-szerverek listája telepítéskor üres, a felhasználónak kell konfigurálni.

A fenti tanúsítási rendszerhez szükség van egy CA szervezetre, és különösen a magasabb biztonsági fokozatú tanúsítványok esetén egyszeri és éves költségek merülnek fel. Ez nem okoz problémát az üzleti és hivatalos világban, azonban a civil világnak nincs erre szüksége. A hierarchikus tanúsítási rendszer helyett elterjedt a hálózatos tanúsítás vagy bizalmi háló (**web of trust**), amely a személyes ismeretségre és bizalomra épül. Ilyen rendszerben működik a PGP (Pretty Good Privacy). Ki-ki elkészíti a saját privát-nyilvános kulcspárját, aztán az ismerőseivel aláírják egymás tanúsítványait. Egy tanúsítványt többen is aláírhatnak, és minél többen írják alá, annál nagyobb a hitele egy új ismerős számára. A tanúsítványokat nyilvános szerverekre töltik fel, mint amilyen a pgp.mit.edu.

A tanúsítványok tárolásához, kezeléséhez, terjesztéséhez, ellenőrzéséhez használt hardver lés szoftver elemeket, beállításokat, házirendeket és eljárásokat összefoglaló néven nyilvános kulcsú infrastruktúrának (**public key infrastructure**, PKI) nevezzük. A jelenleg biztonságosnak tekinthető PKI megoldás a kulcstároló/titkosító célhardver alkalmazása, melyből a privát kulcs sohasem lép ki. Már egy olcsó, USB-key formájú eszköz funkcionalitása tartalmazza a kulcsgenerálást (fizikai véletlenszám-generálás) és kulcstárolást, X509 tanúsítványok tárolását, és az elterjedt titkosítási és aláírási algoritmusok végrehajtását.

6. Üzenetpecsétek

Üzenetpecsét (hash) algoritmusokon alapul a kriptográfiai adatintegritás-védelem. A hash függvény bemenete egy gyakorlatilag tetszőlegesen nagy méretű adatblokk, a kimenete pedig egy kis méretű, tipikusan néhány száz bites hash érték. A számítás gyorsasága nagyon fontos (lásd az alábbi táblázatot).

Módszer	Relatív sebesség	Értékelés
Hash függvény (üzenetpecsét)	3	Leggyorsabb
Folyamtitkosító	2	Gyorsabb
Blokk-titkosító	1	Gyors
Nyílt kulcsú titkosító	0.02	Nagyon lassú

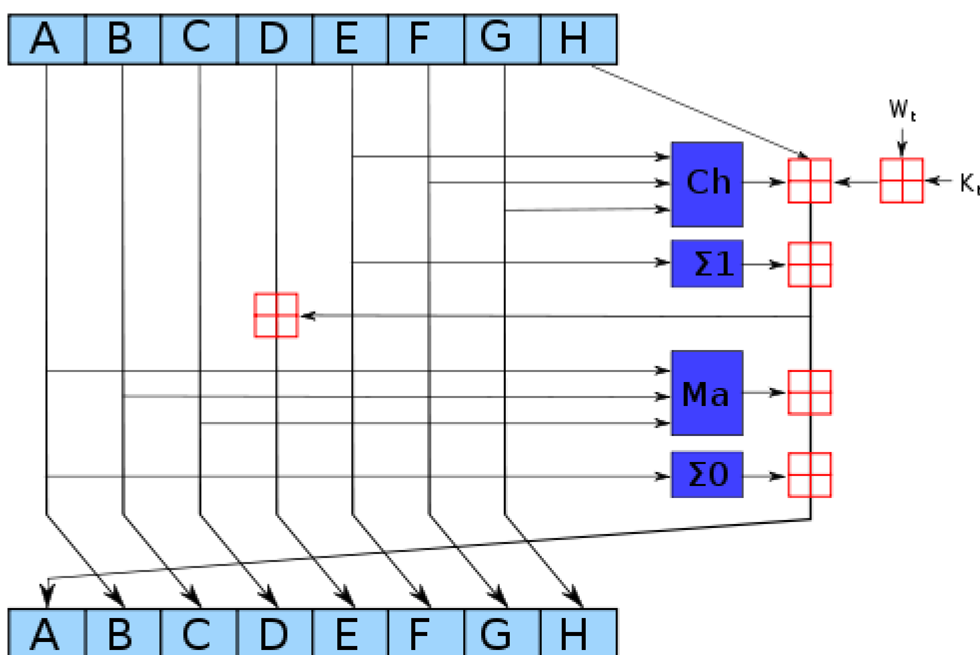
A hash függvényektől elvárt tulajdonságok:

- Bármekkora x bemenetre alkalmazható legyen a függvény.
- A pecsét kicsi, fix méretű legyen.
- A pecsét számolása hatékonyan és könnyen megvalósítható legyen.
- Lavinahatás: a bemenet egy bitjének megváltoztatása 50% valószínűséggel változtasson meg minden bitet a pecsétben.
- Álljon ellen a támadásoknak:
    - egy adott pecsétből egy ilyen pecsétet adó üzenetet nehéz meghatározni (az erre irányuló kísérlet a **preimage** támadás)
    - egy adott üzenethez nehéz egy ugyanolyan pecsétet adó másik üzenetet találni (**second preimage** támadás)

- nehéz két, ugyanolyan pecsétet adó üzenetet találni (ütközés, **collision** támadás)

A jelenleg széles körben használt algoritmusok az MD5, az SHA-1 és az SHA-2. Az MD5 gyakorlatilag törtnek tekinthető, alkalmazása nem javasolt. Az SHA-1 algoritmus részletes leírását lásd a III. Mellékletben. Mivel az SHA-1-ben 2005-ben sebezhetőségeket találtak, ezért SHA-2 néven továbbfejlesztették. Ennek jellemzői:

- 224, 256, 384 vagy 512 bites hash méret
- a 256 bites változatban 64 iteráció a bemenet 512 bites blokkjain, melyeket 8 db. 32 bites state regiszterben tárolnak
- eddig nem találtak rá érdemi törést.



A fenti ábrán az SHA-2 szerkezete látható. A kék hátterű dobozok AND, XOR, OR, rotálás és shiftelés műveleteket tartalmaznak, W_t a bemenetből az adott iterációra kiterjesztett darab, K_t az iterációhoz tartozó konstans.

A születésnap-támadás

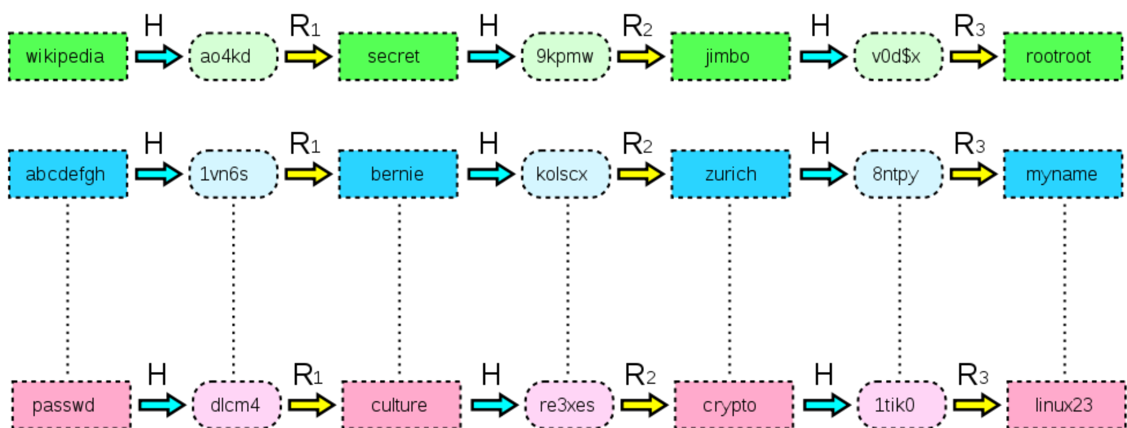
Minden üzenetpecsétre alapuló módszer támadható az ún. születésnap-támadással, ami lényegében egy ütközéses támadás. A digitális aláírás elleni támadás alapgondolata az, hogy bár egy *konkrét* üzenetpecsétet adó dokumentumot nagyon nehéz találni, ennél a születésnap-paradoxonnak köszönhetően lényegesen könnyebb *két azonos* üzenetpecsétet adó dokumentumot találni. A születésnap-paradoxon matematikai hátterét lásd a IV. Mellékletben. Tehát a támadó az aláírandó dokumentumnak, például egy szerződésnek eleve két példányát készíti el: az egyik tartalma a számára kedvezőbb. Ezután mindkét dokumentum-változaton lényegtelen változtatásokat hajt végre, mint

például a szóközök beszúrása üres sorokba stb., és közben mindig kiszámolja a két változat hash értékét. Ha a két hash egyezik, aláíratja a dokumentumot, majd lecseréli a számára kedvezőbb változatra. A hash egyezése (az ütközés) biztosítja az aláírások egyezését a két változatra. A születésnap-támadás kivédhető az elegendően nagy üzenetpecsét-mérettel (ami 2012-ben 256 bit vagy annál több). De a fenti példában van egy egyszerűbb megoldás is: nyomjunk még egy szóközt aláírás előtt a dokumentum egy üres sorába…

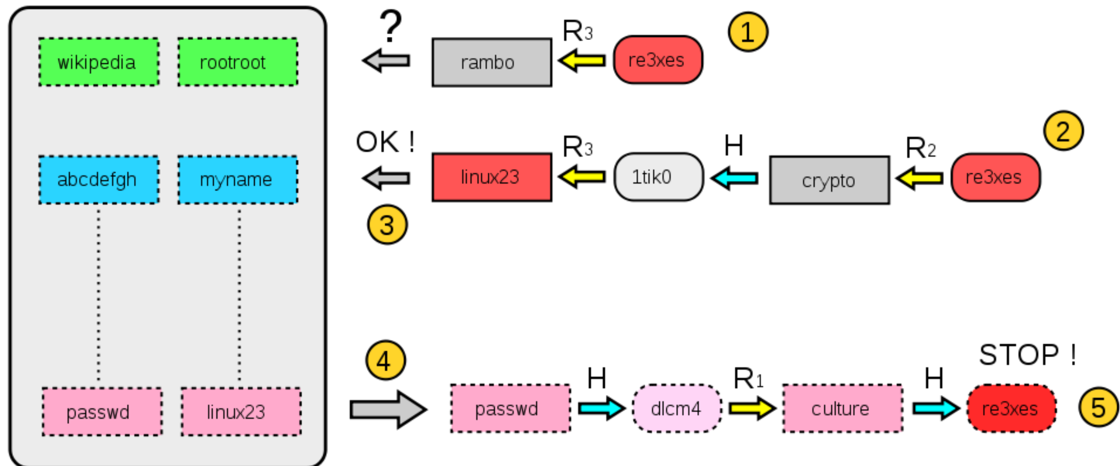
A szivárványtáblás támadás

A szivárványtáblás támadás az általános idő-tárhely ekvivalencia támadás hash függvényekre, különösen jelszó-hash visszafejtésre kifejlesztett változata. A támadás alapfeltevése, hogy a hash érték a jelszó-string nem invertálható random függvénye. A támadás lépései:

1. előfeldolgozás, a szivárványtáblák feltöltése. A támadó feltételez egy jelszóhosszat és egy karakterkészletet, melyet a feltörni kívánt hash-hoz tartozó jelszó használt. Ezután készít egy olyan ún. redukciós függvényt, mely egy hash értékből a megfelelő hosszúságú és karakterkészletű stringet állít elő. Ezután egy véletlen jelszóból indulva kiszámítja az ehhez tartozó hash értéket, majd a hash-re alkalmazza a redukciós függvényt, ennek eredményére ismét a hash értéket számít és így tovább. A lánc hossza a tábla paramétere. A szivárványtábla egy rekordjába csak a lánc első és utolsó elemét menti el.



2. a szivárványtáblák keresése. A támadó most betölti a táblát a memóriába, és a feltörni kívánt hash értékre alkalmazza a redukciós függvényt, majd megnézi, szerepel-e az így kapott string a tábla valamelyik rekordjában, mint utolsó érték. Ha nem, alkalmazza a stringre a hash függvényt, majd a redukciós függvényt, újra keres a táblában és így tovább. Ha megtalálja stringet, akkor az illető rekord feljegyzett első elemétől indulva megtalálható az a jelszó, amely a kérdéses hash értéket adja.



Mivel a sikeres támadáshoz szükséges táblák mérete a jelszó hosszával és a karakterkészlet számosságával gyors ütemben nő, ezért a szivárványtáblás támadás ellen hatékonyan lehet védekezni a jelszó méretének megnövelésével. A jelszóhoz a hash számítása előtt hozzáfűznek egy felhasználó-specifikus stringet, miáltal az előre elkészített táblák használhatatlanná válnak. A műveletet **sózás**nak (salting) nevezik.

7. A kriptográfia jövője

A tömeges alkalmazású kriptográfia gerincét jelenleg (2012-ben) az alábbi elemek alkotják:

- titkos kulcsú kommunikáció: AES (128 bites)
- kulcsmegosztás és autentikáció: RSA
- dokumentum-integritás: SHA-2

Ezen algoritmusok, módszerek részletes leírása megtalálható a Mellékletekben. A kriptográfia jelentősége azonban az exponenciális ütemben gyűlő információtömeggel egyre nő, és a jövőben új megoldások elterjedése várható. Ebben a fejezetben néhány ilyen ígéretes módszert mutatunk be.

Az elliptikus görbékre alapuló kriptográfia (ECC)

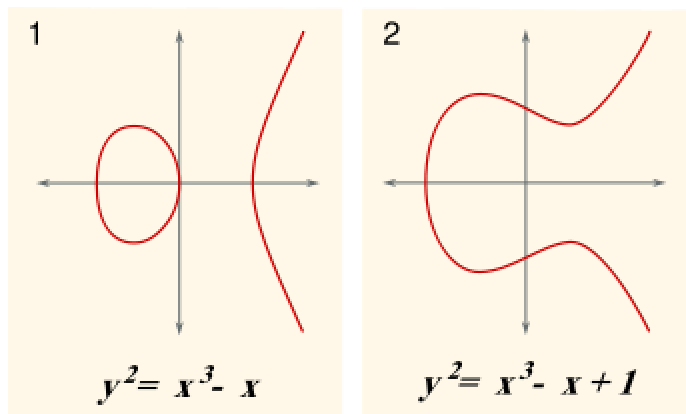
Mint említettük, a nyílt kulcsú kriptorendszerek biztonsága erősen függ egyrészt az egyirányú függvény invertálására irányuló gyors matematikai módszerek fejlődésétől, másrészt a támadó számára rendelkezésre álló, egyre nagyobb számítástechnikai kapacitástól. Ennek gyakorlati következménye, hogy a például a leginkább elterjedt kulcsmegosztási és autentikációs protokoll, az RSA évről évre egyre nagyobb kulcsmérettel üzemeltethető csak, ha egy adott biztonsági szintet meg kívánunk tartani. Az alábbi táblázat az ECRYPT 2010-es kulcsméret-ajánlásait tartalmazza.

Blokkos kriptorendszer kulcsmérete	*Biztonsági szint*	*RSA kulcsméret*	*ECC kulcsméret*
72	Rövid idő alatt, egyszerű módszerekkel törhető	1008	144
80	“elméletileg” megfelelő	1248	160

96	Az ajánlható “abszolút minimum”	1776	192
112	Az ajánlható “megfelelő minimum”	2432	224
128	Megfelelő, kivéve a szigorúan titkos dokumentumokat	3248	256
256	A szigorúan titkos dokumentumok számára is megfelelő	15424	512

Látható, hogy az RSA számára ajánlott „megfelelő minimum” a több ezer bites kulcsméret. Ennek támogatása természetesen nem jelent gondot egy asztali számítógépnek, azonban egyre inkább terjednek azok a smart card (chipkártya) alapú alkalmazások, melyek rendkívül korlátozott energia-felhasználással és memóriával kénytelenek működni. Ezeken a növekvő kulcsméret támogatása egyre nagyobb nehézségekbe ütközik. Az RSA algoritmikus sebezhetőségei mellett ezért is lényeges a kisebb kulccsal dolgozó, gyengébb hardvert igénylő alternatívák kutatása. Mint a fenti táblázatból látszik, az ECC ugyanazt a biztonsági szintet lényegesen kisebb kulcsmérettel tudja garantálni, ezért a jövőben valószínűleg le fogja váltani az RSA-t.

Az **elliptikus görbe** azon pontok halmaza, melyek kielégítik az $y^2 = x^3 + ax + b$ egyenletet, ahol a többszörös gyökök kizárása érdekében feltesszük, hogy $4a^3 + 27b \neq 0$. Az alábbi ábra két elliptikus görbét mutat.



Az ellitikus görbe két pontjának „összegét” úgy definiáljuk, hogy megkeressük azt a pontot a görbén, ahol a két pontot összekötő egyenes metszi a görbét, majd ezt tükrözzük az x tengelyre. A tükörkép is a görbe pontja (1), hiszen a görbe szimmetrikus az x tengelyre. Természetesen igaz, hogy P + Q = Q + P, tehát a művelet kommutatív (2). Bebizonyítható, hogy az összeadási művelet asszociatív, tehát P + (Q + R) = (P + Q) + R fennáll (3). Jelöljük 0-val a görbe azon pontját, melyre $y = \infty$. Bármely ponthoz ezt a 0-t adva, az összeadási szabály szerint P-be jutunk vissza, tehát P +0 = 0 + P fennáll (4). Továbbá, a P + -P = 0 feltételt kielégítő –P pont létezik: ez a P x tengelyre vett tükörképe, P additív inverze (5), mivel a két pontot összekötő egyenes párhuzamos az y tengellyel. A fenti (1)-(5) öt tulajdonság megléte biztosítja, hogy a görbe pontjai az összeadás művelettel Abel-csoportot alkotnak. Ha pedig a P pontot saját magával adjuk össze, akkor az egyenes a P pontban húzott érintő. Egy pont többszöröseit tehát könnyen kiszámíthatjuk ismételt összeadásokkal, így értelmezhetjük *k*P-t, ahol k természetes szám.

A kriptográfiai alkalmazáshoz az ellitpikus görbét diszkretizáljuk olyan módon, hogy a pontok koordinátái a valós számok helyett egy *p* prímszám alapú F_p véges test elemei (a *p*-nél kisebb természetes számok) legyenek, az aritmetikai számítások megfelelő (modulo *p*) módosításával. A véges test feletti E görbét $E(F_p)$-vel jelöljük. $E(F_p)$ pontjainak számára ismeretes Hasse eredménye (1933):

$$p + 1 - 2\sqrt{q} \leq card\left(E\left(F_p\right)\right) \leq p + 1 + 2\sqrt{q}$$

Az $E(F_p)$ pontjai által alkotottt csoport ciklikus. Egy P pontot generátornak nevezünk, ha annak többszörösei (P, 2P, 3P, …) a görbe összes pontját előállítják.

Az elliptikus görbe feletti diszkrét logaritmus problémát ekkor (ECDLP) úgy definiálhatjuk, mint a *k* szorzótényező meghatározását a P és a Q = *k*P pontokból, egy adott diszkretizált $E(F_p)$ görbére. Ha p nagy, legalább 200 bites prímszám, akkor a feladat a diszkrét logaritmushoz hasonlóan nagyon komplex, idő- és energiaigényes. A Diffie-Hellman kulcsmegosztási algoritmust könnyű úgy módosítani, hogy az ECDLP-t használja:

1. A felek nyilvánosan megegyeznek egy $E(F_p)$ diszkretizált görbében, és a görbe egy P pontjában (ezekre például a NIST ajánlásokat tesz közzé).
2. Alice választ egy véletlen k_a részkulcsot ($1 < k_a < p-1$), és elküldi Bobnak a $k_a \cdot P$ pont koordinátáit. Hasonlóképpen Bob is választ egy véletlen k_b részkulcsot ($1 < k_b < p-1$), és elküldi Alice-nak a $k_b \cdot P$ pont koordinátáit.
3. Mindketten kiszámítják a $k_b \cdot k_a \cdot P = k_a \cdot k_b \cdot P$ pont koordinátáit. A titkos visznykulcs ezen pont x koordinátája lesz.

Például a $p = 61$ feletti $y^2 = x^3 + 2x + 4$ görbe kiszemelt pontja legyen a P = (7, 19). Ha $k_a = 14$, akkor az Alice által küldött pont a (47, 22), a Bob által küldött pont pedig $k_b = 50$ esetén a (8, 31). A közösen kiszámított $k_b \cdot k_a \cdot P = k_a \cdot k_b \cdot P$ pont a (23, 54), tehát a 23 lesz a titkos viszonykulcs.

Kvantum-kriptográfia

A „kvantum-kriptográfia" kifejezést általában olyan módszerre értik, mellyel a csatornát a kulcsmegosztás idejére kvantumfizikai törvényeket kihasználva biztosítják, ezáltal elérik, hogy ne kelljen nyilvános kulcsú kulcsmegosztást alkalmazni (Quantum Key Distribution Networks, QKDN).

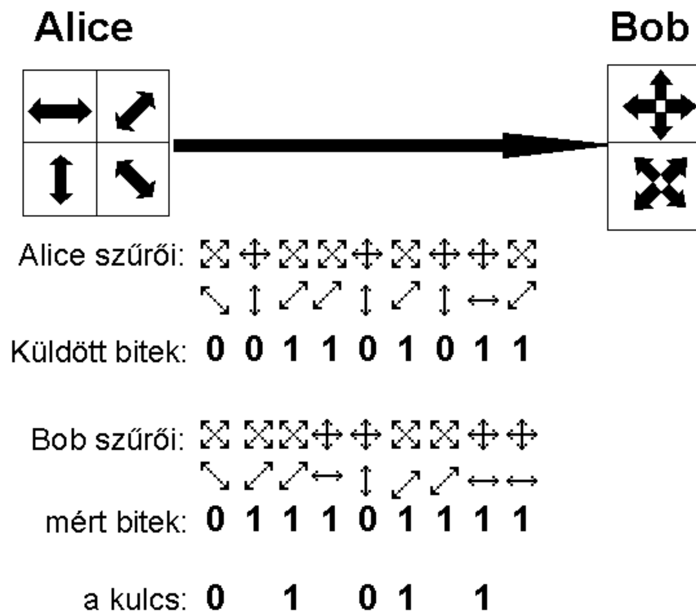
A makroszkopikus világban, ha egy folyamatot elindítunk - például leejtünk egy követ -, akkor a klasszikus fizika törvényei segítségével kiszámíthatjuk, hogy melyik időpillanatban mi történik (milyen gyorsan zuhan a kő, hol tart éppen), és ami nagyon fontos: a zuhanó kő helyét anélkül megállapíthatjuk, hogy megzavarnánk az esésben.

Mi történik az atomok szintjén? Képzeljünk el egy részecskét, amiről tudjuk, hogy észak-dél irányban rezeg. Ha meg akarjuk mérni, hogy a részecske kelet-nyugati vagy észak-dél irányban rezeg-e, akkor a mérés eredménye az, hogy észak-dél irányban. Mi van azonban, ha azt mérjük meg, hogy északkelet-délnyugat, vagy északnyugat-délkelet irányban rezeg-e? Azt várnánk, hogy a mérés eredménye az, hogy egyik irányban sem. Valójában, ha egymás utáni, kezdetben észak-dél irányban rezgő részecskéken

végeznénk el ezt a mérést, akkor az esetek felében azt kapjuk, hogy északkelet-délnyugat, máskor meg, hogy a másik irányban, méghozzá teljesen véletlenszerűen. Sőt, a mérés után a részecske valóban a mért irányban fog tovább rezegni! A kvantummechanika kimondja, hogy nem lehet megmérni egy fizikai mennyiséget anélkül, hogy hatással ne lennénk a mérendő mennyiségre. A makroszkopikus mérésnél is tapasztalunk hasonló jelenséget (például áramerősség mérésekor a mérőműszer beleavatkozik az áram folyásába), ám ez azért van, mert a műszereink nem ideálisak. A kvantummechanika szerint viszont, még akkor is beleavatkoznánk a fenti részecske rezgésébe, ha ideális műszerünk lenne: a részecske „észreveszi", hogy őt most mérik, ezért gyorsan beállítja a rezgését egyik, vagy a másik irányba. A kvantumkriptográfia ezt a furcsa viselkedést felhasználva elméletileg is feltörhetetlen titkosítási módszert ad a kezünkbe.

1984-ben Charles Bennett és Gilles Brassard kidolgozták a BB84 protokollt, amely a kvantummechanikán alapulva megoldást nyújt a fenti problémára, vagyis hogy hogyan cseréljen egymás között két személy, Alice és Bob titkos kulcsot anélkül, hogy az illetéktelen – Eve – kezébe jutna. A módszer mikroszkopikus részecskéket használ, a gyakorlatban a fotonok a legkönnyebben alkalmazhatóak. Képzeljünk el egy 3 dimenziós, derékszögű koordináta rendszert. A fotonok a Z tengely mentén haladnak, ám van egy, a haladás irányára merőleges rezgésük az XY síkban, amit polarizáltságnak hívnak. A hagyományos izzó mindenféle polarizáltságú fotont bocsát ki magából. Polárszűrő segítségével azonban ki tudjuk választani, például az Y irányban polarizált fotonokat, hogy csak azokat küldjük tovább.

Alice és Bob két fajta szűrőkészlettel rendelkezik: Az egyik a *rektilineáris*, a másik a *diagonális*. A rektilineáris készlet egy X és Y irányú szűrőből áll: ↔ és ↕. A diagonális készlet szűrői ehhez képest 45 fokkal el vannak forgatva: ⤢és ⤡. A rektilineáris szűrőkészlet jele +, míg a diagonálisé ╳. A bináris 1-es a következőképpen polarizált fotonoknak felel meg: ↔ vagy ⤢. A 0 pedig: ↕ vagy ⤡. Alice véletlenszerűen küldi az 1-eseket és 0-kat jelképező fotonokat úgy, hogy a szűrőkészleteit is véletlenszerűen váltogatja. A másik oldalon Bob fogadja Alice fotonjait, és megpróbálja detektálni azok polarizáltságát: ő is véletlenszerűen használja a szűrőkészletet. Akkor állapítja meg helyesen az érkező fotonok polarizáltságát, ha az adott fotont ugyanazzal a szűrőkészlettel méri meg, mint amit Alice használt arra a fotonra. Ha rosszat választ, akkor ½ a valószínűsége annak, hogy ugyanazt a bitet kapja, mint amit Alice küldött. Bob gondosan feljegyzi, hogy mikor milyen szűrőt alkalmazott, és milyen biteket kapott. Miután Alice elküldte a biteket, egy nyilvános csatornán (amit bárki lehallgathat) közli Bobbal, hogy mikor milyen szűrőt használt. Bob megnézi a feljegyzéseit, és ekkor már tudja, hogy azokat a biteket detektálta helyesen, ahol ő is azt a szűrőt használta, amit Alice. A többi bitet eldobja. Ekkor Bob közli Alice-al szintén egy nyilvános csatornán, hogy mely sorszámú biteket találta el. Ennél a pontnál Alice tudja, hogy Bob mely biteket tartotta meg, és ezt a bitsorozatot használják a titkos kulcsként (lásd az alábbi ábrát).



Nézzük meg, mi történik, ha Eve beiktat egy saját szűrőkészlet Alice és Bob közé. Ő is ugyanúgy tudja csak detektálni Alice fotonjait, ahogy Bob. Az esetek egy részében ő is rossz szűrőt használ. Viszont, hogy ne keltsen gyanút, a fotonokat továbbküldi Bob felé. Azonban, ha nem megfelelő szűrőt alkalmaz, például Alice ↕-t küld, ami 0, és Eve a × szűrőt használja, akkor véletlenszerűen ⤢-t vagy ⤡-t mér, azaz 1-est vagy 0-t. Sajnos nem tudja az eredeti, detektálás előtti fotont továbbküldeni, hanem a tévesen detektáltat továbbítja Bobnak[19]. Tehát a támadó az esetek ¼-ében elrontja az elküldött bitet. Ekkor, ha Bob ugyanazt a szűrőt használja, mint amit Alice, akkor nem jól állapítja meg a foton polarizáltságát, vagyis a küldött bit értékét, hiszen Eve csavart egyet a fotonon. Miután az adás végén Alice és Bob a nyilvános csatornán egyeztetik a használt szűrőket, és a Bob által elvileg jól detektált bitek sorszámát, a kapott kulcs egy részét elküldik egymásnak szintén a nyílt csatornán. Ha azt látják, hogy ez a kis részlet nem egyezik, akkor tudják, hogy Eve hallgatózott (és ezzel elrontotta a kulcsot). Ekkor átkapcsolnak egy másik csatornára. Ha az összehasonlítás után azt látják hogy nem volt hallgatózás, akkor a kulcs maradék részét fogják alkalmazni. Ebből adódik a módszer hátránya is: Ha nagy távolságra akarunk kulcsot küldeni, akkor nem alkalmazhatunk jelerősítőket, hiszen az olyan, mint mikor Eve hallgatózik. Az egyes eseteket az alábbi táblázat foglalja össze.

Alice véletlen bitsorozata	0	1	1	0	1	0	0	1
Alice véletlen szűrői	+	+	×	+	×	×	×	+
Az Alice által küldött polarizációk	↑	→	↘	↑	↘	↗	↗	→
Eve véletlen szűrői	+	×	+	+	×	+	×	+

[19]tehát a fotont nem lehet másolni ("no cloning theorem")

Eve által mért és továbbküldott polarizációk	↑	↗	→	↑	↘	→	↗	→
Bob véletlen szűrői	+	×	×	×	+	×	+	+
A Bob által mért polarizációk	↑	↗	↗	↘	→	↗	↑	→
A szűrők nyilvános egyeztetése								
A megosztott titkos kulcs	0		0			0		1
Hibák a kulcsban	✓		✗			✓		✓

Ha a felek n bitet egyeztetnek, akkor a hallgatózás felfedezésének a valószínűsége $P = 1 - \left(\frac{3}{4}\right)^n$, tetszőlegesen közelíthető 1-hez. Arra az esetre, ha a felek a felfedezett hallgatózás ellenére ezt a csatornát használják tovább, kidolgoztak olyan protokollokat, mellyel bit-blokkonként paritást számolva nyilvános csatornán tetszőlegesen kis mértékig csökkenthetik a két oldal titkos kulcsa közti különbséget (information reconciliation).

A QKDN elleni lehetséges támadások:

- Hallgatózás (észrevehető, lásd fent)
- Beékelődés. A megbízható, tanúsítvány alapú autentikációt nem pótolja a kvantumkriptográfia sem.
- Fotonhasítás. Ha a fotonforrás nem ideális, csak *átlagosan* 1 fotont ad ki, időnként kettőt is, akkor a dupla foton egyik felét a támadó kvantum-memóriában tárolhatja.
- Hacking a véletlenszám-generátor és a protokoll hibái ellen
- DoS: elvágják a biztonságos optikai kábelt

Számos helyen, több éve stabilan működnek QKDN rendszerek. 2008-tól Ausztriában Bécsben és Sankt Pöltenben már sikerült létrehozni egy hat helyszínt összekötő hálózatot, 200 kilométernyi optikai kábellel és 69 km fizikai átmérővel, ami ezt a protokollt használja. A szükséges hardver azonban jelenleg (2012-ben) még rendkívül drága[20]. Amennyiben ez a módszer bárki számára elérhető lesz, úgy biztosak lehetünk abban, hogy a két végpont között senki nem tudja majd lehallgatni a kommunikációt.

[20] kvantumkritográfiai eszközökkel foglalkozik például a MagiQ és id Quantique: www.magiqtech.com, www.idquantique.com.

Irodalom

[1] Simon Singh: Kódkönyv. Park, 2001.
[2] Virrasztó Tamás: Titkosítás és adatrejtés. Netacademia, 2004.
[3] Richard B. Wells: Applied Coding and Information Theory for Engineers, Prentice Hall, 1999
[4] R.E. Smith: Internet security. Addison-Wesley, 1997.
[5] Andrew S. Tannenbaum: Számítógép-hálózatok, Panem, 2004

I.MELLÉKLET:
AZ AES KRIPTOGRÁFIAI ALGORITMUS

Bevezetés

Korábban a DES (Data Encryption Standard) egy szabványos, 56 bites blokk kódoló algoritmus volt, amelyet az IBM-nél fejlesztettek ki. Az algoritmust és változatait évtizedeken át alkalmazták, ám számos kritika érte a kis kulcsméret miatt, valamint sokan bizalmatlanok is voltak vele szemben. Az amerikai kormányzati szervek nem szerették volna, ha olyan titkosítási módszer jut a lakosság – és így a bűnözők – kezébe, amelyet a hatóságok nem képesek megfejteni. Ám nyilván a civil szférának szüksége volt egy biztonságos kódoló eljárásra, elsősorban az üzleti életben. A kormányzat úgy döntött, hogy a DES 56 bites kulcsokat használhat, ugyanis az akkori vélekedés szerint a civilek, bűnözők nem rendelkeztek olyan számítási kapacitással, amellyel egy ilyen kulccsal kódolt üzenetet meg lehetett fejteni, viszont a hatóságok igen. Ez természetesen az összeesküvés-elmélet gyártók számára remek táptalajt adott. Továbbá a DES több s-dobozt is alkalmazott, amelyeket az IBM szerint véletlenszerűen generáltak. Sokan feltételezték, hogy valójában valamilyen hátsó ajtó van elrejtve ezekben az s-dobozokban, így aki megfelelően használja őket, az könnyen feltörhet bármilyen üzenetet. Eddig azonban még nem sikerült igazolni ezt a sejtést.

A 90-es évek végére nyilvánvalóvá vált, hogy a DES elavult, így szükség van egy új titkosítási szabványra. Szerették volna azt is elkerülni, hogy a közvélemény bizalmatlan legyen a szabvánnyal. 1997-ben az NIST (National Institute of Standards and Technology) kiírt egy kriptográfiai versenyt, amelyben öt szempontnak megfelelő algoritmusokat vártak. A szempontok az alábbiak voltak:

- Szimmetrikus blokk-kódoló legyen
- Az algoritmus minden részlete nyilvános
- Támogassa a 128, 192 és 256 bites kulcsokat
- Hardveresen és szoftveresen nagy hatékonysággal implementálható legyen
- Szabadon felhasználható legyen

A versenyre számos pályázat érkezett, a nyertes a belga szerzők által megalkotott Rijndael algoritmus lett. A módszer neve a szerzők nevéből származik: Joan Daemen és Vincent Rijmen. Az algoritmus mindegyik követelménynek kiválóan eleget tesz, a harmadikon még túl is tett: Amellett, hogy alkalmazható 128, 160, 192, 224 és 256 bites kulcsokkal, a kódolható blokk mérete szintén 128, 160, 192, 225 és 256 bit lehetett. A hivatalos szabványba viszont csak a 128 bites blokkméret, és a verseny követelményeinek megfelelő 128, 192 és 256 bites kulcsok kerültek be. Az új szabvány neve AES lett, azaz Advanced Encryption Standard. Ettől fogva bizonyos források a DES rövidítést már Data Encryption System-nek tüntetik fel.

Az alábbiakban először ismertetjük az algoritmushoz szükséges matematikai alapműveleteket, majd bemutatjuk a kódolás, illetve dekódolás részleteit.

Matematikai alapok

Mint a legtöbb modern kriptográfiai algoritmus, az AES is nagyban támaszkodik a bitenkénti kizáró vagy, azaz XOR műveletre. A műveletnek eredménye akkor 1, ha a két operandus értéke eltér egymástól, egyébként az eredmény 0, ahogy az 1. táblázat is mutatja.

1. táblázat Az XOR művelet igazságtáblája

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

A működéséből adódik egy fontos tulajdonsága, amely miatt a kriptográfiában előszeretettel alkalmazzák:

(A XOR B) XOR B = A

Ennek igazolása látható a 2. táblázatban. Ezt a tulajdonságot arra használják, hogy egy kulcs bitsorozat és egy kódolandó üzenetet-et az XOR művelettel adjanak össze. A dekódoláskor pedig elég annyit tenni, hogy a korábban kapott kódolt üzenethez újra hozzáadjuk a kulcsot az XOR segítségével, így előáll az eredeti üzenet. Ennek alkalmazására a legegyszerűbb példa a korábban már bemutatott OTP algoritmus, de az AES egyes iterációiban is elvégezzük ezt a műveletet.

A	B	A XOR B = C	C XOR B = A
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Az XOR művelet hardveresen rendkívül egyszerűen megvalósítható, akár két tranzisztor is elég hozzá. Továbbá szinte minden mai általános célú processzor támogatja.

Az AES főbb lépései polinomokkal való összeadásra és szorzásra épülnek. Először ismertetjük az összeadást, majd arra alapozva bemutatjuk a szorzást.

Összeadás

Két polinomot a GF(2^8) algebra szerint adunk össze, valamint az összeadást az $\oplus$-al jelöljük. A polinomok együtthatói a $\{0,1\}$ halmazból kerülnek ki. Amennyiben a két polinom i. tagjának együtthatója 1, az eredmény együtthatója 0 lesz, és ez nincsen hatással az $i+1$. együtthatóra, ahogy ezt az alábbi példában láthatjuk:

$$(x^6 + x^4 + x + 1) \oplus (x^7 + x^5 + x^3 + x + 1) = x^7 + x^6 + x^5 + x^4 + x^3$$

Mivel a polinomok együtthatói a $\{0,1\}$ halmazba tartoznak, ezért a polinomokat egyszerűen ábrázolhatjuk bitsorozatokkal. Az x^6+x^4+x+1 polinomot a `01010011` bitsorozat reprezentálja: Az i. bit megegyezik az i. együttható értékével. Ez alapján látható, hogy a $\oplus$ művelet könnyen megvalósítható az XOR segítségével. A fenti példa tehát a következőképpen néz ki bitsorozatokkal:

01010011	x^6+x^4+x+1
10101011	$x^7+x^5+x^3+x+1$

11111000	$x^7+x^6+x^5+x^4+x^3$

Ennek köszönhetően az összeadást egyetlen processzorutasítással is elvégezhetjük. Megjegyezzük, hogy az AES esetében a kivonás művelet azonos az összeadással, erre a szorzásnál lesz szükségünk.

Szorzás

A szorzást szintén olyan polinomok között értelmezzük, mint az összeadásnál, a művelet jele pedig: $\circ$. Az eredmény polinomban továbbra is a 0, illetve 1 együtthatók szerepelhetnek. Fontos, hogy a műveletet modulo aritmetikával végezzük el, amelyhez a következő polinomot használjuk:

$$x^8+x^4+x^3+x+1$$

Ennek következtében az eredménye az, hogy az eredmény polinom legfeljebb 7-ed fokú lehet, így az elfér egy byte-ban. A szorzás két fő lépésből áll: Először összeszorozzuk a két polinomot, majd elvégezzük a maradékos osztást. A szorzás hasonlóan működik, mint ahogy azt korábban megtanultuk: Az első polinom minden tagját szorozzuk a második minden tagjával, így rész polinomokat kapunk. Ezeket a rész polinomokat a $\oplus$ művelettel összeadjuk, ahogy az alábbi példában is láthatjuk:

$$(x^6+x^4+x+1)\circ(x^7+x^5+x^3+x+1)=[(x^{13}+x^{11}+x^8+x^7)\oplus(x^{11}+x^9+x^6+x^5)]$$
$$\oplus[(x^9+x^7+x^4+x^3)\oplus(x^7+x^5+x^2+x)]\oplus(x^6+x^4+x+1)=$$

$$(x^{13}+\text{x^9}+x^8+x^7+x^6+x^5)\oplus(x^9+x^5+x^4+x^3+x^2+x)\oplus(x^6+x^4+x+1)=$$

$$(x^{13}+x^8+x^7+x^6+x^4+x^3+x^2+x)\oplus(x^6+x^4+x+1)=$$

$$x^{13}+x^8+x^7+x^3+x^2+1$$

A kapott polinomot maradékosan osztani kell a $x^8+x^4+x^3+x+1$ polinommal, vagyis az osztás az

$$(x^6+x^4+x+1)\circ(x^7+x^5+x^3+x+1)\bmod(x^8+x^4+x^3+x+1)=$$

$$x^{13}+x^8+x^7+x^3+x^2+1\bmod(x^8+x^4+x^3+x+1)$$

művelet elvégzését jelenti. A maradékos osztás ugyanazon az elven működik, mint számok esetében, visszavezethető a kivonásra: Addig vonjuk ki az osztandóból az osztót, amíg ez utóbbi nem kisebb, mint az osztandó. Ehhez az osztó polinomot meg kell szorozni x olyan hatványával, hogy a szorzat

legmagasabb fokú tagja megegyezzen az osztandó legmagasabb fokú tagjával. A kapott szorzatot és az osztandót pedig az $\oplus$ művelettel összeadjuk. Ezt addig ismételjük, amíg van az eredmény polinomban 7-nél magasabb fokú tag. Az algoritmust a fenti példán a következő módon hajtjuk végre:

Az osztandó polinom: $x^{13}+x^{8}+x^{7}+x^{3}+x^{2}+1$

Az osztó polinom: $x^{8}+x^{4}+x^{3}+x+1$

Az osztandó polinom legmagasabb fokú tagjának foka 13, ezért x^{5}-el beszorozzuk az osztót, majd hozzáadjuk az osztandóhoz:

$$(x^{13}+x^{8}+x^{7}+x^{3}+x^{2}+1)\oplus(x^{13}+x^{9}+x^{8}+x^{6}+x^{5})=x^{9}+x^{7}+x^{6}+x^{5}+x^{3}+x^{2}+1$$

Az eredmény legmagasabb fokú tagjának foka 9, tehát még nem végeztünk, legyen ez az osztandó. Az osztót most x-el kell szoroznunk, majd a szorzatot hozzáadjuk az osztandóhoz:

$$(x^{9}+x^{7}+x^{6}+x^{5}+x^{3}+x^{2}+1)\oplus(x^{9}+x^{5}+x^{4}+x^{2}+x)=x^{7}+x^{6}+x^{4}+x^{3}+x+1$$

A kapott eredményben már nincs olyan tag, amely foka nagyobb 7-nél, tehát megkaptuk a végeredményt.

Ahogy az összeadást, úgy a szorzást is bitsorozatokon végzett műveletek segítségével implementálják. Vegyük észre, hogy ha egy polinomot megszorzunk x^{i}-vel, akkor az azt reprezentáló bitsorozatot balra toljuk i bittel, az új bitek pedig a 0 értéket veszik fel. Ezen felül még az XOR-ra lesz szükségünk. Jelöljük a polinomokat reprezentáló bitsorozatokat A-val és B-vel. A polinomoknál első lépésként az egyik polinom minden tagjával megszoroztuk a másik polinomot, és így rész polinomokat kaptunk. Ezt bitekkel úgy végezzük el, hogy feljegyezzük, hogy B-ben mely $0 \le i \le 7$ pozícióiban van 1. Legyen ezen 1-es bitek száma n. Előállítunk n darab új bitsorozatot úgy, hogy A-t rendre eltoljuk a feljegyzett i értékekkel. A kapott n sorozatot pedig XOR segítségével összeadjuk, ahogy az alábbi példán láthatjuk:

Számítsuk ki a 01010011 · 10101011 mod 100011011 értékét. Legyen A = 01010011, valamint B = 10101011. B-ben a következő pozíciókon van 1: 7, 5, 3, 1 és 0. Tehát ezen értékekkel kell A-t eltolnunk:

```
    0 1 0 1 0 0 1 1 0 0 0 0 0 0 0     Eltolva 7 bittel
        0 1 0 1 0 0 1 1 0 0 0 0 0     Eltolva 5 bittel
            0 1 0 1 0 0 1 1 0 0 0     Eltolva 3 bittel
                0 1 0 1 0 0 1 1 0     Eltolva 1 bittel
                  0 1 0 1 0 0 1 1     Eltolva 0 bittel
XOR -----------------------------
    0 1 0 0 0 0 1 1 0 0 0 1 1 0 1
```

A következő lépés pedig a maradékos osztás elvégzése, mivel a kapott eredményben a 13. pozíción 1-es bit található. Az osztó bitsorozatot rendre eltoljuk annyira, hogy annak legmagasabb helyi értéken lévő 1-es bitje fedje az osztandó bitsorozat legmagasabb helyi értékű 1-es bitjét. A két sorozatot XOR-al

összeadjuk. Amennyiben az összegben van olyan 1-es bit, amely 7-nél magasabb pozíción található, úgy a fenti lépést megismételjük.

```
         0 1 0 0 0 0 1 1 0 0 0 1 1 0 1      Osztandó bitsorozat
           1 0 0 0 1 1 0 1 1 0 0 0 0 0      Osztó bitsorozat eltolva 5 bittel
     XOR -----------------------------
                   1 0 1 1 1 0 1 1 0 1      Legmagasabb 1-es bit: 9, folytatjuk
                   1 0 0 0 1 1 0 1 1 0      Osztó bitsorozat eltolva 1 bittel
     XOR -----------------------------
                       1 1 0 1 1 0 1 1      Legmagasabb 1-es bit: 7, vége
```

A kapott 11011011 bitsorozat pedig az $x^7+x^6+x^4+x^3+x+1$ polinomot reprezentálja.

Multiplikatív inverz

Az AES-ben multiplikatív egység elemnek tekintjük a 00000001 bitsorozat által reprezentált polinomot. Egy A polinom multiplikatív inverze az az A^{-1} polinom, amivel megszorozva az egység elemet kapjuk: $AA^{-1} \equiv 1 \bmod M$, ahol $M = x^8+x^4+x^3+x+1$. Az S-doboz megalkotásához szükségünk lesz minden lehetséges polinom multiplikatív inverzére. Mivel összesen 256 polinomot használunk, akár brute force módszerrel is megkereshetünk minden multiplikatív inverzet, majd azokat egy táblázatban eltárolhatjuk, de használhatjuk akár a kiterjesztett euklideszi algoritmust (lásd az RSA-nál). Vegyük észre, hogy a 00000000 bitsorozatnak a fenti definíció szerint nem létezik a multiplikatív inverze, ezért ebben az esetben a 00000000-t választjuk invernek.

2. táblázat Byte-ok multiplikatív inverzei az AES-ben

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
	1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
	2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
	3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
	4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
	5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
	6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
	7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82
	8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
	9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
	a	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
	b	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
	c	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
	d	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
	e	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3
	f	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

A kulcs kiterjesztése

Az alábbiakban az AES 128 bites kulcsos változatát ismertetjük. Az algoritmus a kódolandó blokkot először egy 4*4-es méretű állapot táblába másolja, majd több iteráción keresztül transzformálja ezt, az állapotot. Egy transzformáció egyik lépése az, hogy a kezdeti kulcsból előállított byte-okat XOR művelettel hozzáadja az állapothoz. Ennek érdekében a 128 bites kulcsot ki kell terjeszteni, hogy minden transzformációs lépésben mást adhassunk hozzá az állapothoz. A kulcs kiterjesztést elég akkor elvégezni, ha a kódolónknak, illetve dekódolónknak új kulcsot adunk meg, azaz nem szükséges ezt minden blokk kódolásánál végrehajtani.

Jelöljük a kulcs i. $(0 \le i \le 15)$ byte-ját k_i-vel. Ez alapján a kulcs byte-okat egy 4*4-es táblázatba rendezzük az alábbi módon:

w[0]	w[1]	w[2]	w[3]
k_0	k_4	k_8	k_{12}
k_1	k_5	k_9	k_{13}
k_2	k_6	k_{10}	k_{14}
k_3	k_7	k_{11}	k_{15}

A táblázat oszlopait a w tömbben tároljuk, a tömb legkisebb indexe a 0. Ezt a táblázatot kell kiterjeszteni 44 oszlopra az első 4 alapján. Az új oszlopokat egymást követően számítjuk ki az alábbi algoritmussal:

Ciklus i = 4-től 43-ig

1. Legyen temp = w[i – 1]
2. Ha i osztható 4-el, akkor
    i. temp byte-jait forgatjuk: temp = [w_0, w_1, w_2, w_3] $\rightarrow$ [w_1, w_2, w_3, w_0]
    ii. temp minden byte-ját kicseréljük az S-doboz alapján
    iii. temp új első byte-jához hozzáadjuk az $x^{i/4-1} \bmod (x^8 + x^4 + x^3 + x + 1)$ polinomot reprezentáló byte-ot
3. elágazás vége
4. w[i] = w[i – 4] XOR temp

Ciklus vége

Kódolás

A bemenet 16 byte-ját első lépésként egy 4*4-es állapot táblázatba másoljuk, majd ezt az állapotot transzformáljuk 10 iteráción keresztül. Jelöljük a bemenet i. $(0 \le i \le 15)$ byte-ját a_i-vel. Ez alapján a bemenet byte-okat egy 4*4-es táblázatba rendezzük az alábbi módon:

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}

a_2	a_6	a_{10}	a_{14}
a_3	a_7	a_{11}	a_{15}

Az állapot táblázaton 4 különböző transzformációt hajthatunk végre:

1. Kulcs hozzáadás
2. Byte helyettesítés
3. Sorforgatás
4. Oszlopkeverés

Az alábbiakban sorra ismertetjük ezeket a lépéseket, majd bemutatjuk, hogy ezek segítségével hogyan történik a kódolás.

Kulcs hozzáadás

Ebben a lépésben az állapot táblázat minden byte-jához XOR művelettel hozzáadunk egy másik byte-ot a kiterjesztett kulcs táblázatból. Az első kulcs hozzáadáskor a táblázat első 4 oszlopát használjuk, a következő alkalommal pedig a következő 4 oszlopot. Az AES iterációs lépései előtt előkészítésként elvégezzük ezt a műveletet, így az i. iterációban a kulcs táblázat [4*i, (i+1) * 4 – 1] oszlopait használjuk fel, ahol $1 \le i \le 10$. A műveletet az alábbi ábrán mutatjuk be:

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

$\oplus$

$w_{0,i}$	$w_{0,i+1}$	$w_{0,i+2}$	$w_{0,i+3}$
$w_{1,i}$	$w_{1,i+1}$	$w_{1,i+2}$	$w_{1,i+3}$
$w_{2,i}$	$w_{2,i+1}$	$w_{2,i+2}$	$w_{2,i+3}$
$w_{3,i}$	$w_{3,i+1}$	$w_{3,i+2}$	$w_{3,i+3}$

=

$s_{0,0} \oplus w_{0,i}$	$s_{0,1} \oplus w_{0,i+1}$	$s_{0,2} \oplus w_{0,i+2}$	$s_{0,3} \oplus w_{0,i+3}$
$s_{1,0} \oplus w_{1,i}$	$s_{1,1} \oplus w_{1,i+1}$	$s_{1,2} \oplus w_{1,i+2}$	$s_{1,3} \oplus w_{1,i+3}$
$s_{2,0} \oplus w_{2,i}$	$s_{2,1} \oplus w_{2,i+1}$	$s_{2,2} \oplus w_{2,i+2}$	$s_{2,3} \oplus w_{2,i+3}$
$s_{3,0} \oplus w_{3,i}$	$s_{3,1} \oplus w_{3,i+1}$	$s_{3,2} \oplus w_{3,i+2}$	$s_{3,3} \oplus w_{3,i+3}$

Byte behelyettesítés

Az AES-ben kettő S-dobozt használunk, az egyiket a kódolásnál, a másikat a dekódolásnál, ez utóbbi az első inverze. A kódolásnál alkalmazottra csak, mint S-dobozra hivatkozunk, a másikat inverz S-doboznak nevezzük. Egy xy alakú, hexadecimálisan leírható byte-ot az S-doboz x. sorában lévő y. cellában található értékre cserélünk ki. Ahogy a bevezetőben említettük, az S-doboz nem egy véletlenszerűen generált táblázat. Egy tetszőleges a byte helyettesítő eleme a következő módon számolható ki:

1. Legyen b = az a byte multiplikatív inverze, amennyiben $a = 0$, úgy $b = 0$.
2. Jelölje b byte i. bitjét b_i.

Legyen $\bar{b}_i = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i$, ahol c = 01100011. Ezt a lépést mátrixos alakban is felírhatjuk:

$$\begin{bmatrix} \bar{b}_0 \\ \bar{b}_1 \\ \bar{b}_2 \\ \bar{b}_3 \\ \bar{b}_4 \\ \bar{b}_5 \\ \bar{b}_6 \\ \bar{b}_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

A 2. lépésre a szakirodalomban mint affin-transzformációra is hivatkoznak. A 3. táblázatban megjelenítettük az így generálható helyettesítő byte-okat hexadecimális formában.

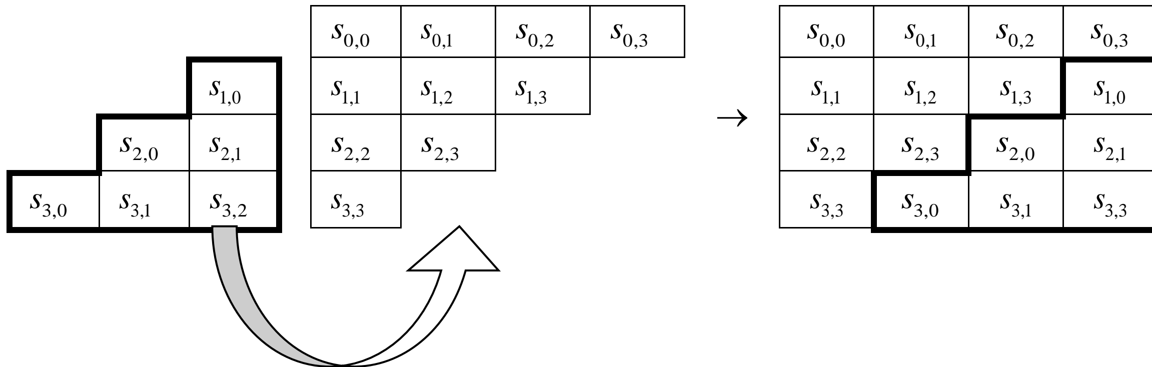
3. táblázat Az AES-ben használt S-doboz

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Sorforgatás

A sorforgatást és az azt követő oszlopkeverést együttesen diffúziós rétegnek is nevezik. Céljuk az, hogy e két lépés segítségével a bemenet minden byte-ja hatással legyen a kimenet minden byte-jára. A forgatás során az i. sor elemeit ($0 \le i \le 3$) i-vel balra forgatjuk, ahogy azt az 1. ábra Sorforgatás a kódolásnálábrán láthatjuk:

1. ábra Sorforgatás a kódolásnál

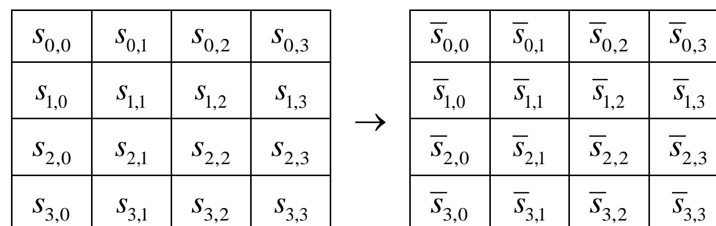


Oszlopkeverés

Ez a transzformáció az állapot tábla oszlopain belül keveri meg a byte-okat. Az oszlopok között nincsen kapcsolat, minden oszlopra ugyanazt a műveletsort végezzük el, a többitől függetlenül: Kiválasztunk egy oszlopot, ennek indexe legyen c, majd az oszlopon belüli cellákat a következő képletek alapján változtatjuk meg:

- $\overline{s}_{0,c} = (0x02 \circ s_{0,c}) \oplus (0x03 \circ s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$
- $\overline{s}_{1,c} = s_{0,c} \oplus (0x02 \circ s_{1,c}) \oplus (0x03 \circ s_{2,c}) \oplus s_{3,c}$
- $\overline{s}_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (0x02 \circ s_{2,c}) \oplus (0x03 \circ s_{3,c})$
- $\overline{s}_{3,c} = (0x03 \circ s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (0x02 \circ s_{2,c})$

A kiszámolt byte-okat pedig visszaírjuk az állapot táblázatba:



Az oszlopon belüli keverést a sorok eltolása után hajtjuk végre. A sor eltolást és oszlopkeverést háromszor egymás után elvégezve elérhetjük, hogy minden bemeneti byte hatással legyen a kimenet minden byte-jára. Az AES esetében ez okozza a lavinahatást: A bemenet egy bitjének megváltoztatása 50% valószínűséggel változtat meg minden bitet a kimenetben.

Kódoló iterációk

Feltételezzük, hogy a bemenetről beolvastuk a kódolandó üzenetet az állapot táblázatba. Innentől a kódolás menete a következő:

1. Az állapothoz hozzáadjuk a kulcs táblázat [0,3] intervallumba eső oszlopait.
2. Ciklus i := 1-től 9-ig:
   a. Byte csere
   b. Sorok forgatása
   c. Oszlopok keverése
   d. Az állapothoz hozzáadjuk a kulcs táblázat [i*4, (i+1)*4-1] oszlopait
3. Byte csere
4. Sorok forgatása
5. Az állapothoz hozzáadjuk a kulcs táblázat [40, 43] oszlopait

Az utolsó iterációban tehát elhagyjuk az oszlopok keverését. Az eljárás kimenete pedig az állapot táblázat tartalma, ahol a byte-okat oszlop folytonosan kell kiolvasni.

Dekódolás

A dekódolási folyamat a kódolás ismeretében roppant egyszerű, csupán „visszafele” kell csinálni minden lépést, amelyet a kódolás során végrehajtottunk. A dekódolandó üzenetet beolvassuk az állapot táblázatba, hasonlóan, mint a kódoláskor. Tudjuk, hogy a kódolás utolsó lépése az volt, hogy az állapothoz hozzáadtuk a kulcs tábla [40-43]-as oszlopait. Ezt a lépést semlegesíthetjük ennek a lépésnek a megismétlésével, hiszen tudjuk, hogy ha XOR-al kétszer adunk hozzá egy kulcsot valamilyen értékhez, akkor visszakapjuk az eredeti értéket. A kódolás többi lépését is visszavonhatjuk, ha az egyes műveletek inverzeit alkalmazzuk, így az eljárás végén megkaphatjuk az eredeti üzenetet. Az alábbiakban bemutatjuk az egyes inverz műveleteket.

Inverz byte helyettesítés

Ebben a lépésben az eredeti S-doboz inverzét használjuk. A 4. táblázatban megfigyelhető, hogy az inverz S-doboz x. sorában és y. oszlopában olyan ij érték szerepel, hogy az S-doboz i. sorában és y. oszlopában xy található. Az inverz S-dobozt előre eltárolhatjuk, vagy menet közben is kiszámíthatjuk az egyes byte-ok helyettesítő értékeit az alábbi módon:

1. Az S-doboz előállításánál használt affin-transzformáció inverzét alkalmazzuk. A B byte bitjeit a következő módon transzformáljuk:

$$\bar{b}_i = b_{(i+2)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+7)\bmod 8} + d_i$$

$$\begin{bmatrix} \bar{b}_0 \\ \bar{b}_1 \\ \bar{b}_2 \\ \bar{b}_3 \\ \bar{b}_4 \\ \bar{b}_5 \\ \bar{b}_6 \\ \bar{b}_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

2. A kapott $\bar{B}$ értéknek megkeressük a multiplikatív inverzét, az eredmény lesz a B helyettesítő értéke.

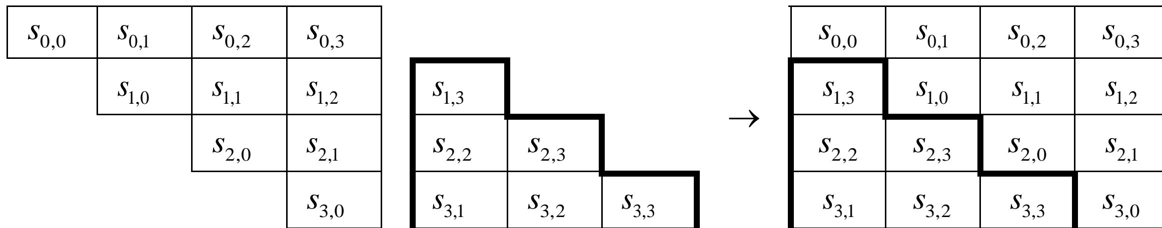
4. táblázat Az AES-ben használt inverz S-doboz

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Inverz sorforgatás

A kódolásnál balra forgattuk a sorokat, ennek az inverze pedig a jobbra forgatás, ahogy azt a 2. ábra láthatjuk. Minden sorban annyi pozícióval kell forgatni, amennyivel a kódolásnál is, csupán az irány változik.

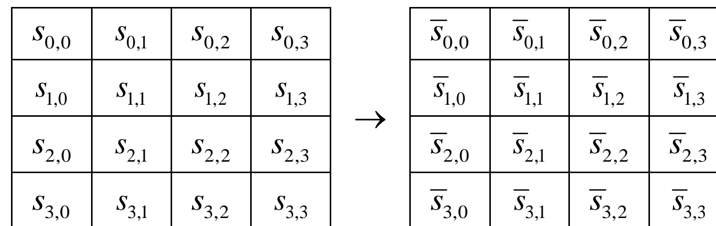
2. ábra Sorforgatás a dekódolásnál



Inverz oszlopkeverés

Ebben a lépésben minden egyes oszlopra elvégzünk egy transzformációt. Az eljárás működési elve hasonló a kódolás oszlopkeveréséhez, csupán az egyes szorzatok szorzói különböznek:

- $\overline{s}_{0,c} = (0x0E \circ s_{0,c}) \oplus (0x0B \circ s_{1,c}) \oplus (0x0D \circ s_{2,c}) \oplus (0x09 \circ s_{3,c})$
- $\overline{s}_{1,c} = (0x09 \circ s_{0,c}) \oplus (0x0E \circ s_{1,c}) \oplus (0x0B \circ s_{2,c}) \oplus (0x0D \circ s_{3,c})$
- $\overline{s}_{2,c} = (0x0D \circ s_{0,c}) \oplus (0x09 \circ s_{1,c}) \oplus (0x0E \circ s_{2,c}) \oplus (0x0B \circ s_{3,c})$
- $\overline{s}_{3,c} = (0x0B \circ s_{0,c}) \oplus (0x0D \circ s_{1,c}) \oplus (0x09 \circ s_{2,c}) \oplus (0x0E \circ s_{3,c})$



Dekódoló iterációk

Feltételezzük, hogy a bemenetről beolvastuk a dekódolandó üzenetet az állapot táblázatba. Innentől a dekódolás menete a következő:

1. Az állapothoz hozzáadjuk a kulcs táblázat [40,43] intervallumba eső oszlopait.
2. Ciklus i := 9-től 1-ig:
   a. Inverz sorforgatás
   b. Inverz byte-csere
   c. Az állapothoz hozzáadjuk a kulcs táblázat [i*4, (i+1)*4-1] oszlopait
   d. Inverz oszlopkeverés
3. Inverz sorforgatás
4. Inverz byte-csere

5. Az állapothoz hozzáadjuk a kulcs táblázat [0, 3] oszlopait

Az utolsó iterációban tehát elhagyjuk az oszlopok inverz keverését. Az eljárás kimenete pedig az állapot táblázat tartalma, ahol a byte-okat oszlop folytonosan kell kiolvasni.

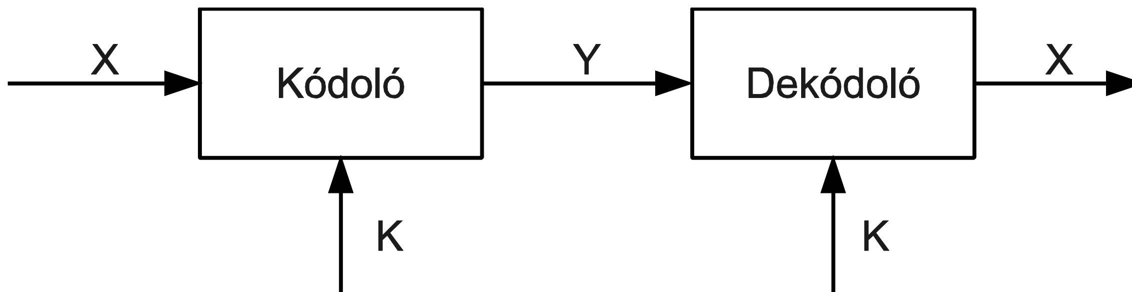
II. MELLÉKLET
BLOKK KÓDOLÓK ÜZEMMÓDJAI

Elektronikus kódkönyv (ECB)

Az elektronikus kódkönyv üzemmód (Electronic Code Book, azaz ECB) a blokk kódolók legegyszerűbb felhasználási módja. A következő jelöléseket fogjuk használni:

- X : Kódolatlan szöveg
- Y : Kódolt szöveg
- K : Kulcs
- $Y = E_K(X)$, azaz E_k függvény végzi el a kódolást a K kulcs alapján.
- $X = D_K(Y)$, azaz D_k függvény végzi el a dekódolást a K kulcs alapján.

A kódolandó bemenetet azonos bithosszúságú blokkokra bontjuk, majd azokat egyenként kódoljuk. A kódolt blokkokat elküldjük a fogadó oldalra, ahol azokat dekódolják, és a beérkezett, majd megfejtett blokkokból összerakják a teljes üzenetet. A lényeg tehát az, hogy a blokkokat egymástól függetlenül kódoljuk, illetve dekódoljuk, ahogy azt a 3. ábra is mutatja.



3. ábra Elektronikus kódkönyv üzemmód működése

Ennek az üzemmódnak több előnye is van. Egyrészt ez a legegyszerűbb. Amennyiben nagy adatmennyiséget szeretnénk kódolni, úgy egyszerűen párhuzamosíthatunk. Végül, ha az üzenet átvitele során egy blokk sérül, akkor a dekódoláskor szintén csak egy blokkot kapunk vissza hibásan, a többi helyes marad. Itt azonban ki is merülnek az előnyök, lássuk, milyen problémák adódnak.

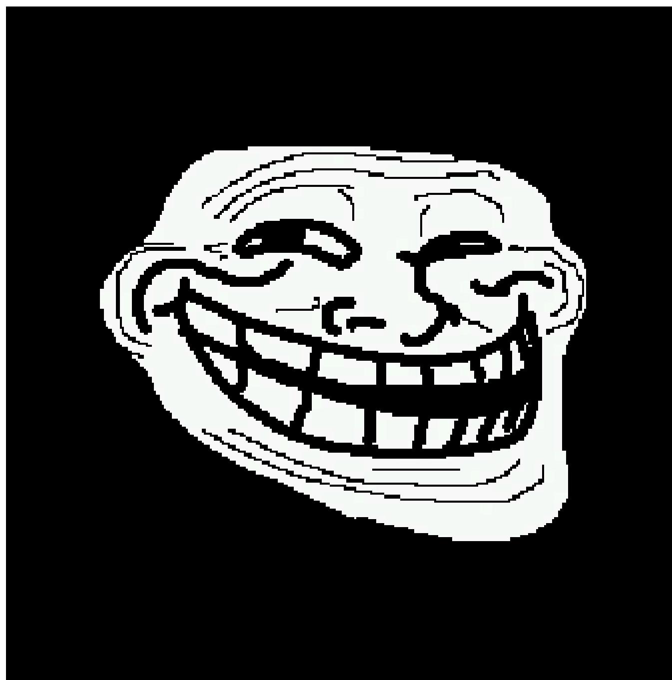
Tegyük fel, hogy Alice-nak és Bobnak van 1-1 saját bankjuk. A két bank időnként pénzt utal át egymásnak, méghozzá valamilyen számítógépes hálózaton keresztül. Egy átutalást a következő szerkezetű adatcsomag átküldésével bonyolítanak le:

1	2	3	4	5
Küldő bank azonosítója	Küldő bankszámlaszám	Fogadó bank azonosítója	Fogadó bankszámlaszám	Pénzösszeg

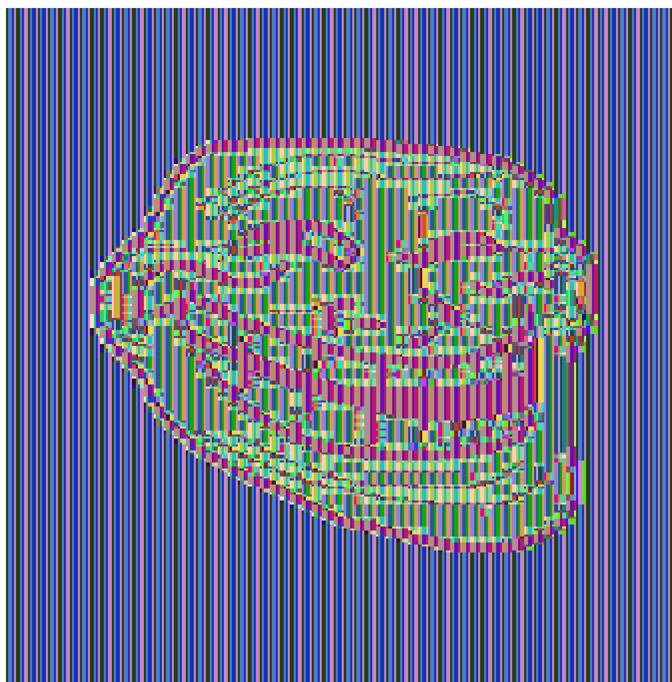
Természetesen ezt az adatcsomagot valamilyen blokk kódoló segítségével titkosítják, és úgy küldik el a hálózaton. Tegyük fel, hogy az adatcsomag egyes szeletei akkorák, amekkorák az általunk használt blokk kódoló által igényelt blokkméret. Továbbá a Alice és Bob bankjai csupán hetente változtatják meg a felhasznált kulcsot. Eve pedig ezt kihasználva úgy dönt, hogy megpróbál a rendszer gyengeségeit kihasználva gyorsan meggazdagodni. Az üzeneteket nem képes visszafejteni, viszont más módja is van célja elérésének. Eve a következőt fogja tenni: Először nyit magának 1-1 bankszámlát Alice-nál és Bobnál. A következő lépésben csekély összegeket utal magának Alice bankjából Bob bankjába. Eközben figyeli, hogy a bankok között milyen adatcsomagok mozognak. Észreveszi, hogy van 5 blokk, amelyek ismétlődnek az utalásoknál. Lementi az 1. 3. és 4. blokkot, hogy ezeket később majd felhasználja. Sejti, hogy az 1. blokk jelöli a bank azonosítóját, a 3. tartalmazza azt a bankot, ahová a pénz megy, és végül a 4. blokk tárolja azt a számlaszámot, ahol a pénzt el kell helyezni. Ezek után figyeli, hogy a gyanútlan ügyfelek utalásai hatására milyen adatcsomagok indulnak meg a hálózaton. Eve számára azok a csomagok érdekesek, amelyek arról szólnak, hogy Alice-éból Bob bankjába kell pénzt küldeni. Ezeket Eve könnyen kiszűrheti, ugyanis rendelkezik olyan kódolt blokkokkal, amely biztosan Alice, illetve Bob bankjának azonosítóját jelöli. Tehát amelyik elfogott csomag 1. és 3. blokkjában a megfelelő értéket látja, abban kicseréli a 4. blokkot arra, amit ő korábban elmentett. Ugyanis a behelyettesített blokk Eve számlaszámának azonosítóját tartalmazza. A csere után Eve tovább küldi az üzenetet, amely hatására Bob bankjában Eve számlájára íródik valamekkora pénzösszeg. Mivel a bankok között rendkívül gyakran utaznak ilyen adatcsomagok, Eve számláján hamar igen nagymennyiségű pénzösszeg halmozódik fel.

Egy másik példában fogtunk egy 32 bites színmélységű tömörítetlen bitképet, és a fejléc kivételével egy 128 bites kulcsot használó AES-el tikosítottuk a tartalmát. Az eredeti képet a 4. ábra, a kódolt változatot pedig az 5. ábra ábrázolja. Látható, hogy a titkosított képen még mindig jól felismerhető az eredeti ábra. A problémán az se segít, ha erősebb kulcsot használunk.

A bemutatott két probléma oka az, hogy az ECB üzemmód determinisztikus. Azaz ha egy adott blokkot többször egymás után kódolva ugyanazt az eredményt kapjuk. Azaz lényegében egy behelyettesítő kódolást kapunk, amelyről közismert, hogy rendkívül egyszerűen feltörhető. Az üzemmód innen is kapta a nevét: elektronikus kódkönyv. Régen gyakran használtak un. kódkönyveket, ahol minden szimbólumhoz leírtak 1-1 helyettesítő szimbólumot. A később ismertetésre kerülő üzemmódok ezt a gyengeséget védik ki más-más megközelítéssel.



4. ábra Az eredeti, kódolatlan kép

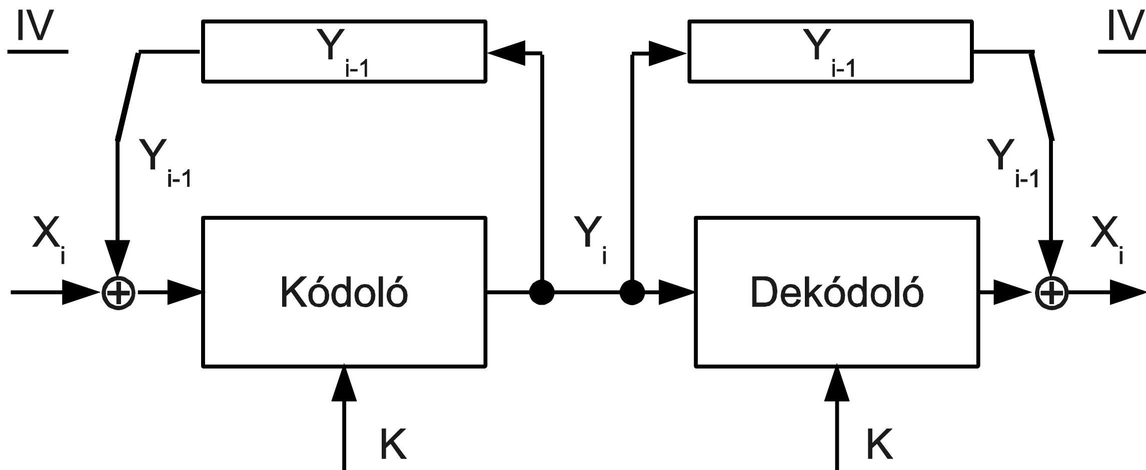


5. ábra 128 bites AES kulccsal, ECB módban titkosított kép

Titkosított blokkok láncolása (CBC)

Az ECB gyengeségének kijavítására alkalmazhatjuk azt a módszert, hogy a kódolás kimenetére nem csak a kódolandó blokk van hatással, hanem az előzőleg kódolt blokk is. Mivel az első kódolandó blokk előtt

nem volt semmi, ezért szükségünk van egy IV-vel jelölt inicializáló vektorra. Az üzemmód működését a 6. ábra láthatjuk.



6. ábra A CBC mód működése

Az első blokk kódolása előtt bitenkénti XOR művelettel hozzáadjuk az IV-t, majd az így módosított blokkot kódoljuk. Az eredményt tovább küldjük a fogadó oldalra, valamint a kódolás helyén betöltjük egy regiszterbe. A következő blokkhoz már az ebbe a regiszterbe betöltött adatsort adjuk hozzá. A fogadó oldalon az első blokk fogadásakor elvégezzük a dekódolást, majd az eredményhez újra hozzáadjuk az IV-t. Az XOR tulajdonságainak köszönhetően ekkor visszakapjuk az eredeti blokkot. A fogadó oldalon a dekódolás előtt elmentjük a beérkezett titkosított blokkot egy regiszterbe, hogy azt felhasználhassuk a következő blokk dekódolásához. Tehát az egyes blokkokat egymásra láncoljuk, innen jön az üzemmód neve is: Cipher Block Chaining, azaz CBC. Fontos, hogy az inicializáló vektor azonos legyen mindkét oldalon. Formálisan a következőképpen néz ki a folyamat:

- Első blokk kódolása: $Y_1 = E_K(IV \oplus X_1)$
- Első blokk dekódolása: $X_1 = D_K(Y_1) \oplus IV$
- Következő (i.) blokkok kódolása: $Y_i = E_K(Y_{i-1} \oplus X_i)$
- Következő (i.) blokkok dekódolása: $X_i = D_K(Y_i) \oplus Y_{i-1}$

Gondoljuk végig, hogy az ECB-nél ismertetett bankos problémát megoldja-e a CBC. Amennyiben üzenetenként más IV vektort használnak, Eve nem képes mintákat felfedezni az üzenetekben. Továbbá, ha az egymás utáni üzeneteket folyamatosan egymás után láncolják, akkor szintén megoldódik a probléma. Tegyük fel azonban, hogy üzenetenként újra inicializáljuk a fogadó és küldő oldalt, azaz minden üzenet elején újra elővesszük ugyanazt az IV vektort. Ekkor Eve újra elfogja az egyes üzeneteket, majd kicseréli a 4. blokkot és az üzenetet tovább küldi. A vevő fogadja az üzenetet és visszafejti azt. Eve pechére a fogadó bank rosszul dekódolja a 4. és 5. blokkot. Ugyanis a 4. blokkra hatással van az első három is, ahol a 2. blokk üzenetenként más és más. Mivel a fogadó oldalon tévesen dekódolják a 4. és 5.

blokkot, ezért a pénz nem fog megérkezni Eve számlájára. A beavatkozás észlelése egy másik probléma, ezzel részletesebben foglalkozunk majd a digitális aláírás témakörén belül.

Nézzük, hogy ha a korábban bemutatott képet CBC módban kódoljuk, akkor milyen ábrát kapunk. Az eredményt a 7. ábra láthatjuk. Ezen már felismerhetetlen az eredeti kép, úgy tűnik, mintha egy véletlenszerűen generált ábrát látnánk.

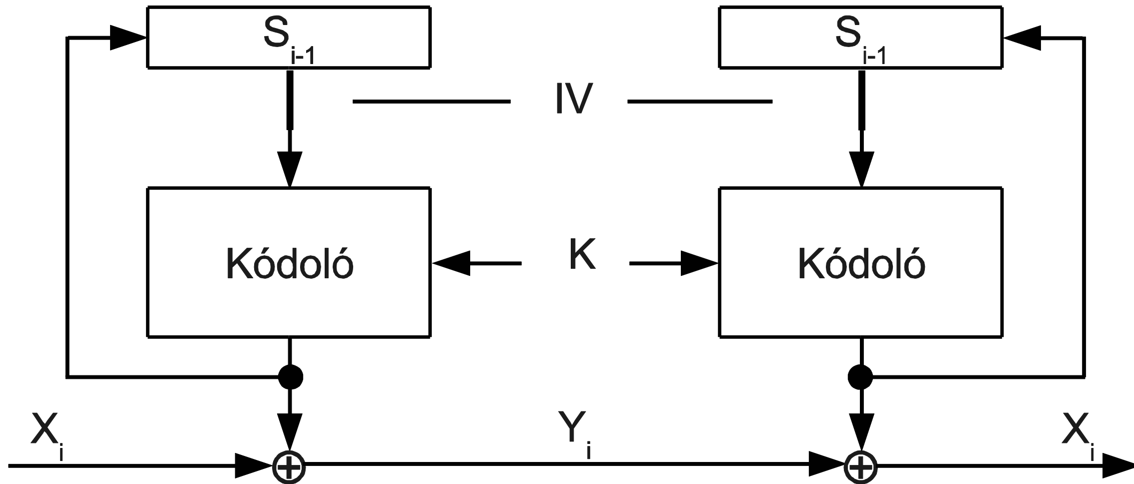
7. ábra 128 bites AES-el kódolt kép CBC üzemmódban

Látható tehát, hogy a CBC megszűnteti az ECB determinisztikusságát, így kivédi annak a gyengeségét. Az IV vektorban valamilyen módon meg kell egyezniük a kommunikáló feleknek, ezt akár szimmetrikus kódolóval is megtehetik. Az üzemmódnak van azonban egy komoly hátránya: Amennyiben nem garantálható, hogy minden blokk hibamentesen átjut a hálózaton, akkor ha csak egy blokknak egyetlen bitje hibás lesz, akkor a lavina hatás miatt az összes ezt követő blokk hibásan lesz dekódolva. A következő üzemmód ezt küszöböli ki.

Kimenet visszacsatolása (OFB)

Ennél az üzemmódnál szintén szükségünk van egy IV vektorra. Itt a kódoló nem kapja meg magát az X blokkot. Az első blokk elküldésekor a kódoló egy K kulccsal titkosítja az IV vektort. Az eredményt egyrészt elmenti egy S regiszterbe, másrészt XOR-al hozzáadja a titkosítandó X blokkhoz. Az eredményt pedig elküldi a fogadó oldalra, ahol szintén egy kódolóval és az IV vektorral előállítunk egy bitsorozatot, azt XOR-al hozzáadjuk a beérkező üzenethez, és megkapjuk az eredetit. A következő blokk elküldése előtt már nem az IV-t, hanem az S regiszter tartalmát kell kódolni. A küldő és fogadó oldalon tehát pontosan ugyanazt kell végrehajtani. Mivel itt a kódoló kimenetét csatoljuk vissza annak a

bemenetére, az üzemmódot Output Feedback-nek, azaz OFB-nek nevezzük. A módszer működésének vázlatát a 8. ábra láthatjuk.



8. ábra Az OFB mód működése

Formálisan a következő módon írhatjuk fel a módszer működését:

- Első blokk kódolása: $S_1 = E_K(IV)$, $Y_1 = X_1 \oplus S_1$
- Első blokk dekódolása: $S_1 = E_K(IV)$, $X_1 = Y_1 \oplus S_1$
- Következő (i.) blokkok kódolása: $S_i = E_K(S_{i-1})$, $Y_i = X_i \oplus S_i$
- Következő (i.) blokkok dekódolása: $S_i = E_K(S_{i-1})$, $X_i = Y_i \oplus S_i$

A kódoló kimenete tehát teljesen független a titkosítandó szövegtől. A kódoló kimenetét kulcsfolyamnak is nevezhetjük, ezt akár előre is kiszámolhatjuk. Az átviteli hibákra ez a módszer a legkevésbé érzékeny, ugyanis ha a hálózaton átvitt üzenetben 1 bit sérül meg, akkor az eredményben is csak 1 bit hiba keletkezik. Fontos, hogy a vevő oldalon is kódolót kell alklamazni a kulcsfolyam előállítására!

Azonban oda kell figyelnünk arra, hogy a kulcs folyamot ne ismételjük meg. Tegyük fel, hogy a P_1 és P_2 szöveget ugyanazzal a K kulcsfolyammal titkosítjuk. Ekkor a keletkezett kódolt üzenetek:

$$C_1 = P_1 \oplus K \text{ és } C_2 = P_2 \oplus K .$$

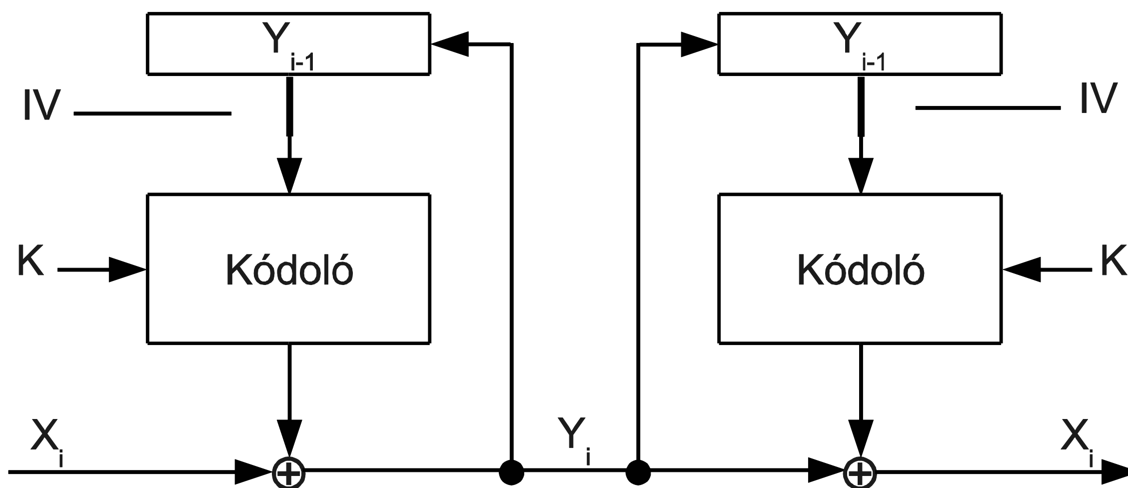
Eve elfogja C_1-et és C_2-t, majd XOR-al összeadja őket:

$$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = P_1 \oplus P2$$

Amennyiben Eve ismeri P_1-et, vagy P_2-t, úgy a másik egy XOR segítségével könnyen meghatározható. Ellenkező esetben pedig $P_1 \oplus P_2$ gyakoriságelemzéssel törhető, ugyanis az egyes szimbólumpároknak megfeleltethető egy helyettesítő szimbólum.

Kódolt üzenet visszacsatolása (CFB)

Az alábbi üzemmódban szintén kódolót használunk a küldő és fogadó oldalon. A módszer annyiban különbözik az OFB-től, hogy a kódolóra nem annak kimenetét, hanem magát a kódolt üzenetet csatoljuk vissza, ezért ennek a neve Cipher Feedback, azaz CFB.



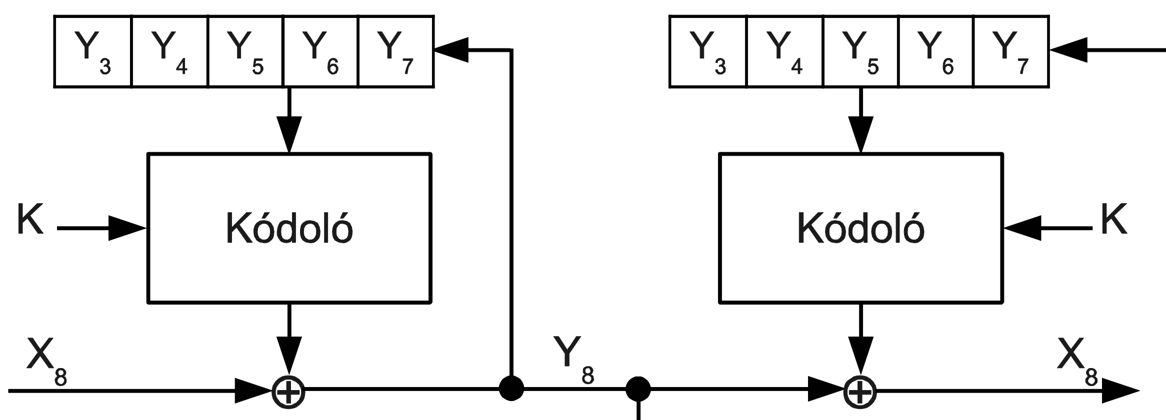
9. ábra A CFB mód működése

Formálisan a következőképpen néz ki az algoritmus:

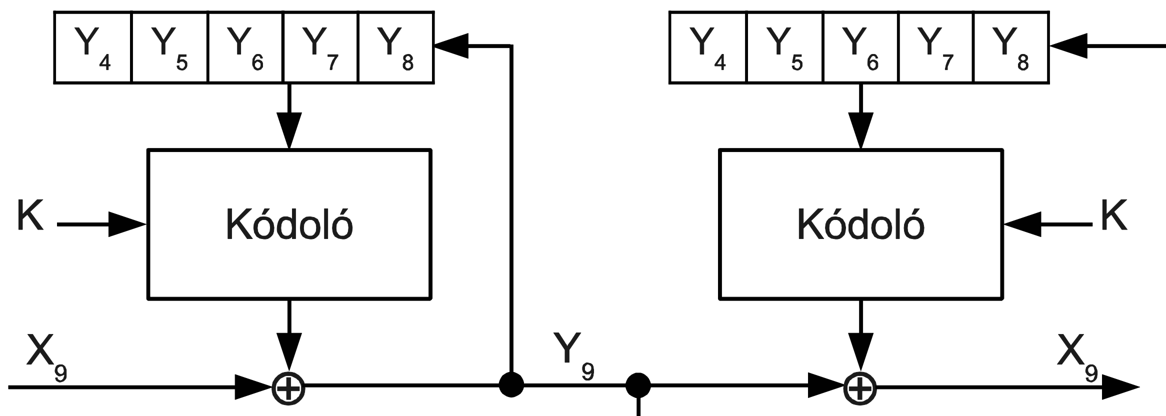
- Első blokk kódolása: $Y_1 = X_1 \oplus E_K(IV)$
- Első blokk kódolása: $X_1 = Y_1 \oplus E_K(IV)$
- Következő (i.) blokkok kódolása: $Y_i = X_i \oplus E_K(Y_{i-1})$
- Következő (i.) blokkok dekódolása: $X_i = Y_i \oplus E_K(Y_{i-1})$

Eddig látszólag ez a módszer nem nagy előrelépés az előzőekhez képest. Tegyük fel azonban, hogy olyan üzeneteket kell kódolni, amelyek jóval rövidebbek, mint a blokk kódolónk által igényelt blokk méret. Például ha a felhasználó gépel, akkor lehet, hogy a szoftvernek nincs ideje megvárni, míg összegyűlik a megfelelő mennyiségű karakter, hogy azokat majd egyben kódolva elküldje. Az egyik megoldás az lehet, hogy az egyes karaktereket kiegészítjük a megfelelő blokk méretre, majd elvégezzük a kódolást. Könnyen látható, hogy ekkor azonban pazarlóan bánunk a hálózat sávszélességével. A megoldás a CFB mód megfelelő használatában rejlik.

Tegyük fel, hogy a kódolók bemenetére egy shift-regisztert kötünk. A regiszterbe betölthetünk egy új byte-ot, ennek hatására a regiszterben lévő legrégebbi byte kiesik. A regiszter kezdetben az IV byte-jait tartalmazza. Az új üzenet küldésekor a shift-regisztert kódoljuk. Az elküldendő byte-ot XOR-al összeadjuk a kódoló kimenetének meghatározott indexű byte-jával, majd az eredmény byte-ot visszatöltjük a shift-regiszterbe, valamint elküldjük a hálózaton keresztül a fogadónak. A fogadó oldalon a küldőhöz hasonlóan előállítanak egy kulcsot, amelynek a megfelelő byte-jával XOR kapcsolatba hozzák a beérkező byte-ot, és megkapják a dekódolt byte-ot. Továbbá, a hálózaton érkező kódolt byte-ot betöltik a shift-regiszterbe. A 10. ábra és 11. ábra látható, ahogy az újabb elküldött, illetve fogadott byte-ok hatására a shift-regiszterek tartalma változik. A módszer hátránya is ebből fakad: Amennyiben az egyik byte átvitele során hiba lép fel, akkor amíg a hibás byte a fogadó oldal shift-regiszterében van, addig a beérkező byte-okat hibásan dekódolják.



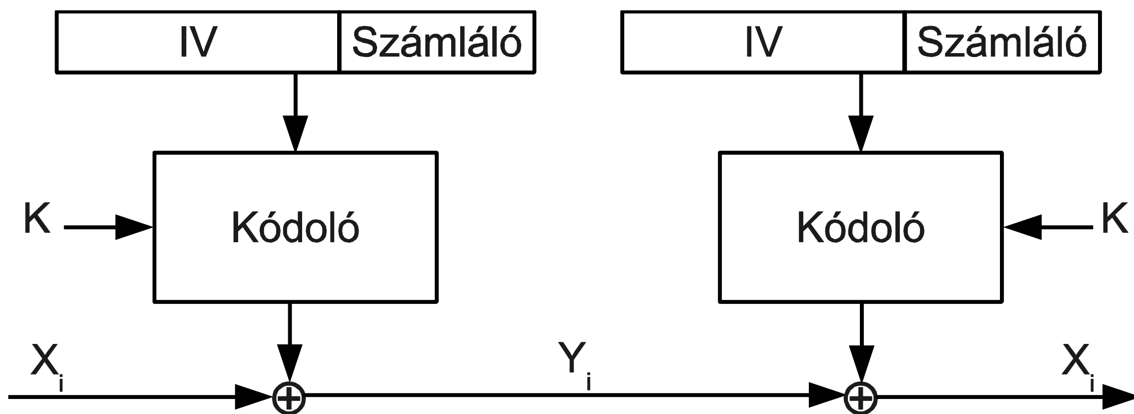
10. ábra A shift-regiszterek tartalma a 8. byte küldésekor



11. ábra A shift-regiszterek tartalma a 9. byte küldésekor

Számláló mód (CTR)

A CBC, OFB és CFB üzemmódok nagy hátránya az, hogy ha rendelkezünk egy nagy mennyiségű kódolt adathalmazzal, például egy merevlemez tartalmával, és szeretnénk ezen adathalmaz egy adott szakaszához hozzáférni, akkor az előtte lévő kódolt adatokat is fel kell dolgozni. Tehát mivel az egymás utáni blokkok szoros kapcsolatban állnak egymással, a direkt hozzáférés meglehetősen problémás lehet. Ennek a megoldására dolgozták ki a számláló módot (Counter mode, azaz CTR). Itt is kódolót alkalmaznak a küldő és fogadó oldalon. A kódolók bemenetét úgy kapjuk meg, hogy összefűzünk egy inicializáló vektort és egy számlálót. A számláló értéke kezdetben nulla. Az első blokk elküldésekor az IV-t és a számlálót összefűzve kódoljuk, majd az eredményt XOR-al hozzáadjuk a küldendő blokkhoz. A fogadó oldalon hasonlóképpen cselekszünk a kódolóval, majd annak kimenetével megfejtjük a hálózaton beérkezett titkosított üzenetet. Az üzenet elküldése illetve fogadása után mindkét oldalon 1-el növeljük a számlálót. Lényegében újfent egy kulcsfolyamot állítunk elő, amelyre nincsenek hatással sem a múltban elküldött blokkok, sem a múltban generált kulcsfolyam-szeletek.



12. ábra A CTR mód működése

Formálisan a következő módon működik a CTR mód, ha CTR_e jelöli a küldő, CTR_d a fogadó számlálóját:

- Kezdetben $CTR_e = CTR_d$.
- Az i. blokk kódolása: $Y_i = X_i \oplus E_K(IV \mid CTR_e)$, majd CTR_e-t 1-el megnöveljük.
- Az i. blokk dekódolása: $X_i = Y_i \oplus E_K(IV \mid CTR_d)$, majd CTR_d-t 1-el megnöveljük.

Ez az üzemmód kifejezetten hasznos többek között merevlemezek titkosításakor. Ekkor a számláló jelölheti az aktuális merevlemez szektor sorszámát.

III. MELLÉKLET
AZ RSA ALGORITMUS

Az RSA szó a fejlesztőinek neveiből származik: Ron Rivest, Adi Shamir és Len Adleman. Az algoritmust 1976-ban publikálták. 1997-ben nyilvánosságra hozták, hogy Clifford Cocks 1973-ban már kidolgozta az eljárást, a brit GCHQ (Government Communications Headquarters) nevű szervezetnél, ám több mint két évtizedig nem engedélyezték, hogy ezt publikálja.

Matematikai alapok

Az RSA számelméleti eredményeken alapul, ezért először ezeket ismertetjük. A bemutatásra kerülő számelméleti tételek helyességének bizonyításától eltekintünk.

Legnagyobb közös osztó

Definíció: Egy A és B egész szám legnagyobb közös osztójának nevezzük azt a legnagyobb egész számot, amely A-nak és B-nek is osztója.

A jegyzetben A és B legnagyobb közös osztóját lnko(A, B)-vel jelöljük. Kiszámítására több módszer is van, próbáljuk meghatározni lnko(84, 30)-at. Bontsuk a két számot prímtényezőkre:

- 84 = 2 * 2 * 3 * 7 és
- 30 = 2 * 3 * 5

A két szám osztói közül a 2 és 3 a közös, tehát lnko(84, 30) = 2 * 3 = 6. Kis számoknak nem okoz nehézséget az előbbi módszer, ám nagy számoknál ez az algoritmus használhatatlanná válik. Szerencsére létezik ennél jóval gyorsabb módszer is, mégpedig az euklideszi algoritmus. A bemenet az A és B egész számok, tegyük fel, hogy A > B.

1. R := A mod B
2. Ha R = 0 akkor végeztünk, a legnagyobb közös osztó := B
3. A := B
4. B := R
5. Vissza az 1. pontra

A fenti algoritmusban a mod a maradékos osztást jelöli. Az lnko(84, 30) a következő módon számolható ezzel az algoritmussal:

Lépés	A	B	R
1	84	30	24
2	30	24	6
3	24	6	0

Multiplikatív inverz

Az RSA-ban moduláris aritmetikát használunk, ezért definiáljuk a multiplikatív inverzet:

Definíció: Egy A egész szám multiplikatív inverze az a B szám modulo M-ben, amire igaz, hogy:

$$AB \equiv 1 (\bmod M)$$

Például legyen A = 23, M = 120. Ekkor, ha B-t 47-nek választjuk, akkor

$$AB = 1081$$

$$1081 \bmod 120 = 1$$

Tehát 23 multiplikatív inverze modulo 120 esetén 47. Mivel az RSA kulcsválasztási lépésében multiplikatív inverzet is kell számolni, ezért szükségünk van egy hatékony algoritmusra ennek a meghatározásához. Erre alkalmas a kiterjesztett euklideszi algoritmus.

A kiterjesztett euklideszi algoritmus a következő egyenlet megoldását teszi lehetővé:

$$ax + by = lnko(a,b),$$

ahol a és b egészek adottak, x és y egészeket kell meghatározni. Tegyük fel, hogy x az a multiplikatív inverze modulo m-ben:

$$ax \equiv 1 (\bmod m)$$

Tehát m osztója ax – 1-nek, az osztás eredményét jelöljük q-val:

$$ax - 1 = qm$$

Ezt átrendezve azt kapjuk, hogy

$$ax - qm = 1$$

Amennyiben lnko(a, m) = 1, és a kiterjesztett euklideszi algoritmussal meghatározzuk x-et és q-t, akkor x lesz a keresett multiplikatív inverz. Az lnko(a, m) = 1 feltétel az RSA-hoz elegendő, hiszen ott erre az esetre lesz csak szükségünk.

Ahogy a neve is utal rá, az algoritmus hasonlóan működik, mint az euklideszi algoritmus, csupán azt kell kiegészíteni néhány utasítással. A könnyebb tárgyalhatóság kedvéért határozzuk meg az lnko(a = 120, b = 23)-at:

Osztandó	**Osztó**	**Hányados (q)**	**Maradék (r)**
120	23	5	5
23	5	4	3
5	3	1	2
3	2	1	1
2	1	2	0

A maradékokat jelöljük r_i-vel. Az algoritmust az ax + by = lnko(a, b) egyenlet megoldására használjuk, tehát az ismeretleneket x-el és y-al jelöljük. Fejezzük ki az r_i-t a és b alapján:

$$r_i = ax_i + by_i$$

A táblázat alapján könnyen látszik, hogy a táblázat egy sorában lévő maradék értéke 1 sorral lejjebb szintén megjelenik, mint osztó, 2 sorral lejjebb, pedig mint osztandó. Tehát ha i > 2 akkor:

$$r_i = r_{i-2} - q_i r_{i-1}$$

Az előző kettő összefüggésből a következőt kapjuk:

$$r_i = (ax_{i-2} + by_{i-2}) - q_i(ax_{i-1} + by_{i-1})$$

Emeljük ki a -t és b -t:

$$r_i = a(x_{i-2} - q_i x_{i-1}) + b(y_{i-2} - q_i y_{i-1})$$

Az algoritmus során az ismeretlen x és y értékét fokozatosan megközelítjük. Tegyük fel, hogy

- $x_1 = 1$, $y_1 = 0$ és
- $x_2 = 0$, $y_2 = 1$

Valamint, ha i > 2, akkor

- $x_i = x_{i-2} - q_i x_{i-1}$
- $y_i = y_{i-2} - q_i y_{i-1}$

Definiáljuk továbbá r_1 -et és r_2 -t:

- $r_1 = ax_1 + by_1 = a$
- $r_2 = ax_2 + by_2 = b$

A fentiek alapján az algoritmus a következő:

1. $r_1 = a$, $r_2 = b$
2. $x_1 = 1$, $y_1 = 0$, $x_2 = 0$, $y_2 = 1$
3. $i := 3$
4. $q_i := r_{i-2} / r_{i-1}$
5. $r_i := r_{i-2} \bmod r_{i-1}$
6. Ha $r_i = 1$ akkor vége
7. $x_i := x_{i-2} - q_i x_{i-1}$
8. $y_i := y_{i-2} - q_i y_{i-1}$

9. $i := i + 1$
10. Ugrás a 4. pontra

A fenti algoritmusban az euklideszi algoritmussal ellentétben akkor állunk meg, ha a maradék 1. Mivel nekünk az RSA miatt van szükségünk erre az algoritmusra, a fenti megközelítés elegendő lesz.

Az alábbi példában megkeressük x és y értékét, ha a = 120 és b = 23. Korábban meghatároztuk, hogy lnko(120, 23) = 1.

i	q_i	r_i	x_i	y_i	$r_i = a\underline{x_i} + b\underline{y_i}$
1		120	1	0	$120 = 120 * \underline{1} + 23 * \underline{0}$
2		23	0	1	$23 = 120 * \underline{0} + 23 * \underline{1}$
3	5	5	1	-5	$5 = 120 * \underline{1} + 23 * \underline{(-5)}$
4	4	3	-4	21	$3 = 120 * \underline{(-4)} + 23 * \underline{21}$
5	1	2	5	-26	$2 = 120 * \underline{5} + 23 * \underline{(-26)}$
6	1	1	-9	47	$1 = 120 * \underline{(-9)} + 23 * \underline{47}$

Tehát 1 = 120 * (-9) + 23 * 47, azaz x = -9 és y = 47. A korábbi megállapítások alapján az alábbi egyenleteket írhatjuk fel:

$$47 * 23 \equiv 1 \pmod{120}$$

$$120 * (-9) \equiv 26 * 38 \equiv 1 \pmod{47}$$

$$120 * (-9) \equiv 14 * 5 \equiv 1 \pmod{23}$$

Azaz 47 és 23 egymás multiplikatív inverzei modulo 120, vagy 26 és 38 egymás modulo inverzei modulo 47, továbbá ugyanez igaz 14-re és 5-re modulo 23-ban.

Euler-függvény

Az Euler-függvény definíciójához először szükségünk van a relatív prímek fogalmára:

Definíció: A és B egész számok relatív prímek, ha lnko(A, B) = 1.

Ez alapján az Euler-függvény definíciója:

$$\varphi(m) = |\{a : lnko(m, a) = 1 \text{ és } a < m\}|$$

Azaz az Euler-függvény egy adott m pozitív egész szám esetén megadja, hogy hány darab m-nél kisebb és m-el relatív prím létezik.

Számoljuk ki $\varphi(6)$ értékét!

- lnko(6, 1) = 1 ✓

- lnko(6, 2) = 2
- lnko(6, 3) = 3
- lnko(6, 4) = 2
- lnko(6, 5) = 1 ✓

Tehát $\varphi(6) = 2$. Bár egy számpárra az euklideszi algoritmussal hatékonyan tudjuk ellenőrizni a relatív prím definíciójának való megfelelést, egy nagy m számra $m-1$ ellenőrzésre van szükség, ami elfogadhatatlan mennyiségű számítást tesz szükségessé.

Amennyiben ismerjük m prímtényezős felbontását, úgy $\varphi(m)$ értékét hatékonyan meghatározhatjuk. Legyen

$$m = p_1^{e1} p_2^{e2} \dots p_n^{en},$$

ahol p_i prímszámok. Ez alapján az Euler-függvény értékét így határozhatjuk meg:

$$\varphi(m) = \prod_{i-1}^{n} (p_i^{ei} - p_i^{ei-1})$$

Határozzuk meg $\varphi(240)$ értékét a fenti képlet alapján!

$$m = 16 * 15 = 2^4 * 3^1 * 5^1 = p_1^{e1} p_2^{e2} p_3^{e3},\ n = 3$$

$$\varphi(240) = (2^4 - 2^3)(3^1 - 3^0)(5^1 - 5^0) = 8 * 2 * 4 = 64$$

Ennek a módszernek előnye a hatékonyság, hátránya viszont az, hogy ismerni kell hozzá m prímtényezős felbontását. Nagy számoknál ehhez rengeteg számításra van szükség, viszont az RSA-nál ezt a tulajdonságot használjuk ki.

Vegyük észre, hogy ha m két prímszám, nevezetesen p és q szorzata, akkor

$$\varphi(m) = (p-1)(q-1)$$

A kis Fermat-tétel

A következő összefüggés Pierre de Fermat-tól származik, bár ő ezt csupán sejtésként fogalmazta meg. A kis jelzőt azért kapta, hogy megkülönböztessük a nagy Fermat-tételtől, amely hosszú évszázadokig, egészen 1994-ig csupán sejtés volt. A kis Fermat-sejtést Fermat 1636-ban fogalmazta meg, és Leibniz viszonylag hamar, 1683-ban közölte, hogy ő már ismert egy bizonyítást rá.

A tétel a következőt mondja ki: Legyen p prímszám, továbbá a bármely egész szám, ekkor igaz az, hogy

$$a^p \equiv a \pmod{p}$$

Az RSA megértéséhez a fenti összefüggésnek egy másik változatára lesz szükségünk. Legyen p továbbra is prímszám, ám a olyan egész, amely p-hez relatív prím. Ekkor igaz a következő:

$$a^{p-1} \equiv 1 \ (\text{mod } p)$$

Legyen a = 34, p = 37. Ekkor

$$34^{37} = 4620829355366080837126178409035022147187035888137210429 44$$

$$4620829355366080\text{-}2186728080903502214718 70\text{-}706153575042944 \text{ mod } 37 = 34$$

A tétel igaz akkor is, ha p < a, legyen a = 34 és p = 7.

$$34^{7} = 52523350144$$

$$52523350144 \equiv 6 \equiv 34 \ (\text{mod } 7)$$

Az RSA algoritmusa

Az RSA azt használja ki, hogy két nagy prímszámot könnyen összeszorozhatunk, viszont pusztán a szorzat ismeretében jelenlegi tudásunk szerint nem tudjuk elfogadható időn belül visszafejteni a prímszámokat.

Az algoritmus előkészítő lépésében mindenki generál magának privát és publikus kulcsokat, majd a publikus kulcsokat nyilvánosságra hozzák. A kulcsgenerálás lépései a következők:

1. Legyen p és q elegendően nagy prímszámok, és $p \neq q$
2. $N = pq$
3. $\varphi(N) = (p-1)(q-1)$
4. Legyen e olyan szám, amelyre igaz, hogy $lnko(e, \varphi(N)) = 1$ és $e \geq 3$
5. Legye d olyan szám, amelyre igaz, hogy $ed \equiv 1 \ (\text{mod } \varphi(N))$
6. A nyilvános kulcs legyen: (N, e), a privát kulcs pedig d

A nyilvános kulcsból e-t az euklideszi algoritmussal keressük meg. Ez a szám lehet kicsi is, így az is megfelelő, ha 3-tól kezdve egyesével teszteljük az egész számokat, hogy megfelelőek-e. Rendszerint 11-nél tovább ritkán kell elmenni. A d meghatározásához pedig a kiterjesztett euklideszi algoritmusra van szükség. Jelöljük x-el a kódolandó üzenetet, y-al pedig a kódolt változatát. Ekkor a kódolás a következő módon zajlik:

$$y = x^{e} \ (\text{mod } N)$$

A dekódolás menete pedig a következő:

$$x = y^{d} \ (\text{mod } N)$$

A támadónak szüksége van d értékére ahhoz, hogy az üzenetet meg tudja fejteni. Ehhez a nyilvános e mellett szüksége van $\varphi(N)$ értékére is, amit elvileg a szintén nyilvános N-től meg lehet határozni. Korábban viszont láttuk, hogy egy megfelelően nagy szám esetén annak prímtényezős felbontása túl számításigényes, gyakorlatilag kivitelezhetetlen, így a támadó N alapján nem képes $\varphi(N)$-t, és így d-t meghatározni.

Az alábbiakban bemutatjuk az RSA működését egy egyszerű példán.

1. Legyen $p = 73$ és $q = 151$
2. $N = pq = 11023$
3. $\varphi(N) = (p-1)(q-1) = 10800$
4. e legyen 11, mert $lnko(10800,11) = 1$
5. d értéke 5891, mert $ed \equiv 1 \pmod{10800}$
6. A nyilvános kulcs tehát: (11023, 11), míg a privát kulcs 5891
7. Titkosítsuk az $x = 17$ üzenetet!
8. $17^{11} \pmod{11023} = 1782$
9. Dekódolás: $1782^{5891} \pmod{11023} = 17$

Az ajánlásokban általában minimum 1024 bites kulcsméret szerepel, de katonai alkalmazásoknál 2048, vagy ennél nagyobb kulcsméretek is előfordulnak. A tapasztalatok azt mutatják, hogy az RSA, hasonlóan a többi nyilvános kulcsú kódoló eljáráshoz, nagyságrendekkel lassabb, mint a blokk-kódolók. Emiatt az RSA-t az üzenetváltásnak csupán bizonyos lépéseinél, például a kulcs megosztásnál alkalmazzák.

Az alábbiakban megmutatjuk, hogy az RSA helyesen működik, azaz bizonyítsuk az alábbi állítást:

$$(x^e)^d \equiv x \pmod{N}$$

Tudjuk, hogy d az e-nek $\bmod\ \varphi(N)$-ben multiplikatív inverze, azaz

$$ed \equiv 1 \pmod{\varphi(N)}$$

Tehát ez azt jelenti, hogy *ed* egy olyan egész szám, amit $\varphi(N)$-el elosztva 1-et kapunk maradékul, az osztás egész részét pedig jelöljük v-vel:

$$ed = v\varphi(N) + 1$$

Az eredeti állítást a következő módon írhatjuk át:

$$x^{v\varphi(N)+1} \equiv x \pmod{N}$$

Az állítás igazolásához először egy segédállítást kell belátnunk. Bizonyítsuk be, hogy tetszőleges x és s értékek, és tetszőleges u prímszám esetén igaz a következő:

$$x^{s(u-1)+1} \equiv x \pmod{u}$$

Két esetet kell megvizsgálnunk, az első az, mikor x-et nem osztja u, ekkor a kis Fermat-tétel segítségével az állítás belátható:

$$x^{u-1} \equiv 1 \pmod{u} \rightarrow (x^{u-1})^s \equiv 1 \pmod{u} \rightarrow x(x^{u-1})^s \equiv x \pmod{u}$$

A másik eset viszont az, mikor x-et osztja u. Ekkor u és x nyilván nem relatív prímek, tehát ekkor a kis Fermat-tételt nem alkalmazhatjuk, tehát az előző eset bizonyítása ide nem jó. Mivel x-et osztja u, ezért az osztás maradéka 0, azaz:

$$x \equiv 0 \equiv x^{s(u-1)+1} \equiv x \pmod{u}$$

Tehát a segédállításunk igaz. Ezek után már minden adott az eredeti állítás bizonyításához. Ezt szeretnénk belátni:

$$x^{v\varphi(N)+1} \equiv x \pmod{N}$$

Tudjuk, hogy

$$x^{s(u-1)+1} \equiv x \pmod{u},$$

ha u prím. Valamint

$$ed = v\varphi(N) + 1 = v(p-1)(q-1) + 1.$$

Legyen $s_1 = v(p-1)$, és $s_2 = v(q-1)$. Ezek alapján igazak a következők:

$$x^{s_1(q-1)+1} \equiv x \pmod{q}$$

$$x^{s_2(p-1)+1} \equiv x \pmod{p}$$

Vagyis q osztja $x^{s_1(q-1)+1} - x$-et és p osztja $x^{s_2(p-1)+1} - x$-et, tehát $pq = N$ osztja $x^{v\varphi(N)+1} - x$-et. Ebből következik, hogy $x^{v\varphi(N)+1} \equiv x \pmod{N}$, ami pedig az eredeti állítással ekvivalens. ☐

Ez alapján igaz az is, hogy $(x^d)^e \equiv x \pmod{N}$, erre majd az üzenethitelesítésnél lesz szükségünk.

Az RSA alkalmazása digitális aláírásra

Tegyük fel, hogy Bob az x üzenetet szeretné elküldeni Alice-nak. Bob privát kulcsa legyen $k_{pr} = d$, nyilvános kulcsa pedig $k_{pub} = (n, e)$. Bob eljuttatja (n, e)-t Alice-nak. Az üzenet elküldése előtt Bob kiszámolja x-re a digitális aláírást: $s \equiv x^d \mod n$. Bob elküldi Alice-nak (x, s)-t. Alice fogadja az

üzenetet, és ellenőrzi a kapott aláírást, mégpedig úgy, hogy dekódolja s-t: $\overline{x} \equiv s^e \mod n$. Amennyiben Alice azt tapasztalja, hogy $x = \overline{x}$, akkor biztos lehet vele, hogy az aláírást Bob privát kulcsával készítették, tehát az aláírás helyes. Mivel csak Bob ismeri d értékét, ezért más nem képes a megfelelő s-t előállítani, tehát biztos, hogy Bob küldte az üzenetet. Abban az esetben, ha Eve elfogja az (x, s) párost, megváltoztatja x-et, a megfelelő s-t is ki kellene számolnia, de erre nem képes, hiszen nem rendelkezik Bob privát kulcsával. Lássunk erre egy konkrét szám példát!

Legyen az elküldendő üzenet $x = 4$. Generáljunk megfelelő RSA privát és publikus kulcsot, ahol a lépések a következők:

1. Legyen $p = 3$ és $q = 11$
2. $n = pg = 33$
3. $\varphi(n) = (3-1)(11-1) = 20$
4. Legyen $e = 3$
5. $d \equiv e^{-1} \equiv 7 \mod 20$

Bob elküldi Alice-nak a saját nyilvános kulcsát, mégpedig a $(33,3)$-at. Az üzenet elküldésekkor pedig kiszámolja az $x = 4$ üzenet digitális aláírását:

$$s = x^d \equiv 4^7 \equiv 16 \mod 33$$

Bob ezek után elküldi $(x, s) = (4,16)$-ot Alice-nak. Alice ellenőrzi az üzenet hitelességét, kiszámolja $\overline{x}$-et:

$$\overline{x} = s^e \equiv 16^3 \equiv 4 \mod 33.$$

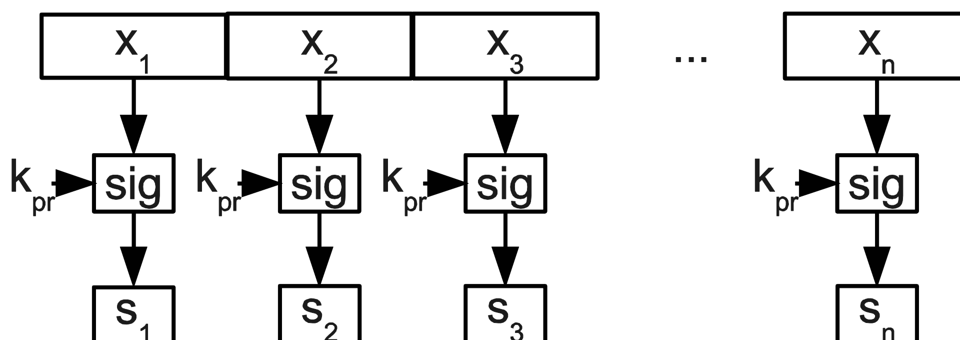
Mivel $x = \overline{x}$, ezért biztos, hogy az aláírás helyes.

Ez a módszer egyszerű, ám van egy hátránya: RSA esetében könnyen látható, hogy az x nem lehet nagyobb, mint n. Sajnos hasonló korlátozás érvényes más nyilvános kulcsú kódolóra is, ezért bonyolultabb aláírási módszert kell alkalmazni.

Bontsuk az x üzenetet n darab szeletre, ahol minden x_i $(1 \leq I \leq n)$ szeletre igaz, hogy $x_i < n$. A fenti módszerrel állítsuk elő a privát kulcs segítségével minden x_i szelet aláírását:

$$s_i = x_i^d \mod n.$$

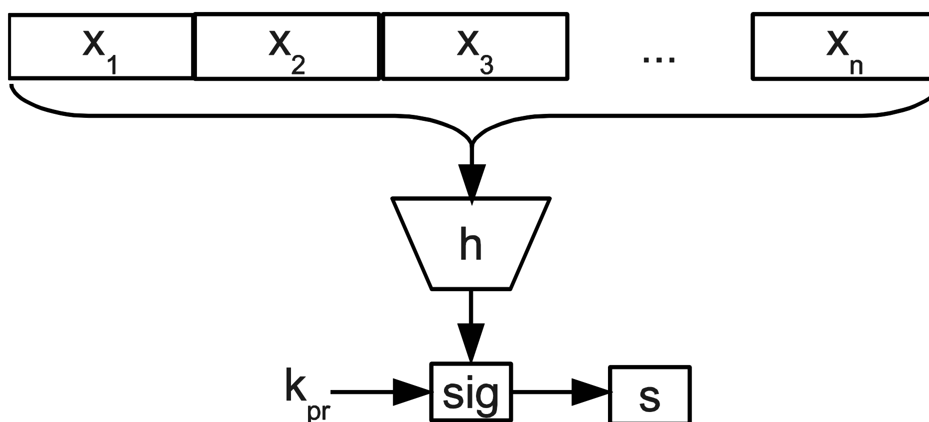
A kapott s_i-et pedig összefűzve megkapjuk a teljes x üzenet digitális aláírását.



13. ábra Naiv módszer a digitális aláírás előállítására

Az aláírást továbbá sem képes más reprodukálni. A módszer azonban egyrészt nem hatékony, másrészt nem is biztonságos. A nyilvános kulcsú kódolók közismerten lassúak, tehát nagy üzenet esetén rendkívül sokáig tartana előállítani az egyes üzenetpecsét szeleteket. A másik hátrány, hogy a kapott aláírás mérete nagyságrendileg akkora, mint az eredeti üzenet, tehát az elküldendő adatmennyiség közel megkétszereződik, ez pedig elfogadhatatlan tárolás, illetve hálózati átvitel szempontjából. A legsúlyosabb ellenérv pedig a módszer támadhatósága: Eve a nélkül átrendezheti a továbbítandó (x_i, s_i) párokat, hogy arról Alice vagy Bob tudomást szerezne.

A gyakorlatban ténylegesen alkalmazott megoldás az, hogy először számoljunk ki egy rövid hash-t a teljes üzenetre, ezt később *h*-val jelöljük, majd csupán ez utóbbit kódoljuk nyilvános kulcsú kódolóval.



14. ábra A helyes aláírás generálás

Ez a módszer kijavítja az előző hibáit. A hash függvényt úgy kell megadni, hogy annak kimenetére alkalmazható legyen az nyilvános kulcsú kódoló. Mivel csak egy rövid bitsorozatra alkalmazzuk a kódolást, ezért a kapott aláírás mérete kicsi lesz, valamint az gyorsan is számolható. Továbbá a támadó nem tud mit átrendezni az elküldött (x, s)-ben anélkül, hogy ez a fogadónak fel ne tűnne. Lássuk

formálisan, hogyan néz ki a fenti módszer. Az elküldendő üzenet továbbra is x. Bob nyílt kulcsa: k_{pub}, a privát pedig k_{pr}.

1. Bob először kiszámolja az x -re a hash-t: $z = h(x)$
2. Bob z-ből kiszámolja a digitális aláírást: $s = \mathrm{sig}_{k_{pr}}(z)$
3. Bob elküldi (x, s) -t Alice-nak, aki elvégzi az ellenőrzést. Ő is kiszámolja a hash függvény értékét: $\overline{z} = h(x)$. Végül ellenőrzi, hogy ha dekódolja s -t, akkor ugyanazt a hash értéket kapja-e, mint amit ő kiszámolt. Tehát, ha $\overline{z} = k_{pub}(s)$, akkor a digitális aláírás helyes, egyébként pedig helytelen. Most lássuk, hogy Eve hogyan támadhatja a digitális aláírást. Nyilván Eve, ahogy bárki más, Bobnak csak a nyilvános kulcsát ismeri, a privátat nem. Eve elfogja Bob üzenetét, és megváltoztatja az x üzenetet $\overline{x}$ -re. Ekkor nyilván új digitális aláírást kell mellékelni a megváltoztatott üzenethez, ha ez sikerülne, akkor Alice nem venné észre, hogy átverés áldozata lett. Eve $\overline{x}$ hash értékét nyilván ki tudja számolni: $\overline{z} = h(\overline{x})$. A következő lépés pedig az, hogy Bob privát kulcsával kódolja $\overline{z}$ -t, ám erre nem képes, hiszen nem rendelkezik a megfelelő kulccsal.

IV. MELLÉKLET
A SZÜLETÉSNAP-TÁMADÁS MATEMATIKAI HÁTTERE

Az alábbi feladat a születésnap paradoxon nevet viseli. A kérdés az, hogy hány embernek kell lennie egy szobában ahhoz, hogy legalább 0.5 legyen annak a valószínűsége, hogy van két ember, akiknek azonos napon van a születésnapjuk? Tegyük fel, hogy nincsenek a szobában ikrek és szökőév sincsen. Jellemzően az első válasz, ami az esetek többségében elhangzik, az, hogy 365 / 2 emberre van szükség, holott valójában elég csupán 23. A feladat maga tehát nem paradoxon, csupán azért alkalmazzák rá ezt a megjelölést, mert első hallásra talán furcsának hangzik a helyes megfejtés.

Ahhoz, hogy megértsük, miért elég 23 ember, vegyük észre, hogy a feladat nem azt kérdezi, hogy egy kiválasztott személyhez találunk-e olyan párt, akinek ugyanazon a napon van a születésnapja. Itt tetszőleges párokat keresünk. 23 személy esetén pedig $\binom{23}{2} = \frac{23*22}{2} = 253$ pár alkotható. Annak a valószínűsége, hogy mindenki másik napon született:

$$P(\text{mindenki másik napon született}) = \frac{364}{365}\frac{363}{365}\dots\frac{365-n+1}{365}$$

Ennek ellenkezőjének valószínűsége pedig:

$$P(\text{legalább két embernek azonos napon van a születésnapja}) = 1 - \frac{364}{365}\frac{363}{365}\dots\frac{365-n+1}{365}$$

Ez utóbbi valószínűség $n = 23$ esetben pedig 0.507. Ezt a gondolatmenetet alkalmazhatjuk az üzenetpecsétekre is. Határozzuk meg, hogy ha $\lambda = 0.5$ valószínűséggel lehet találni két olyan hash függvény bemenetet, amelyekre a kimenet azonos, akkor egy n bites kimenettel rendelkező hash függvénynél mennyi próbálkozásra van szükség?

A fenti valószínűség képletet az alábbi módon írhatjuk át:

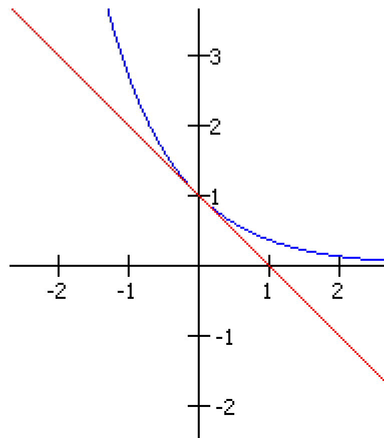
$$P(\text{nincs ütközés}) = \left(1-\frac{1}{2^n}\right)\left(1-\frac{2}{2^n}\right)\dots\left(1-\frac{t-1}{2^n}\right) = \prod_{i=1}^{t-1}\left(1-\frac{i}{2^n}\right)$$

Itt a t jelöli azt, hogy mennyiszer érdemes próbálkozni, míg megfelelő bemeneteket találunk. A folytatáshoz szükségünk lesz egye közelítő képletre. Az 15. ábra láthatjuk pirossal az $1-x$, kékkel pedig az e^{-x} függvényt. Látható, hogy amennyiben x nagyon közel van az 1-hez, akkor

$$e^{-x} \approx 1-x\,.$$

Mivel i/e^n rendkívül közel van 0-hoz, ezért a fenti közelítést alkalmazhatjuk a valószínűség kiszámítására:

$$P(\text{nincs ütközés}) \approx \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}} \approx e^{-\frac{1+2+3+\ldots+t-1}{2^n}}$$



15. ábra Az e^{-x} és az $1-x$ függvények

A képlet jobb oldalán látható számlálót helyettesíthetjük az alábbi módon:

$$1+2+3+\ldots+t-1=\frac{t(t-1)}{2}$$

Ezt felhasználva a valószínűség képletét a következő módon közelíthetjük:

$$P(\text{nincs ütközés}) \approx e^{-\frac{t(t-1)}{2*2^n}} \approx e^{-\frac{t(t-1)}{2^{n+1}}}$$

Ahogy korábban is írtuk, λ jelöli annak a valószínűségét, hogy legalább egy ütköző bemenetpárt találunk:

$$\lambda \approx 1-e^{-\frac{t(t-1)}{2^{n+1}}}$$

Mivel arra vagyunk kíváncsiak, hogy mennyi t próbálkozásra van szükség ahhoz, hogy λ valószínűséggel ütközést találjunk, rendezzük a fenti egyenletet t-re:

$$\ln(1-\lambda) \approx -\frac{t(t-1)}{2^{n+1}}$$

$$t(t-1) \approx 2^{n+1} \ln\left(\frac{1}{1-\lambda}\right)$$

Mivel *t* egy elegendően nagy szám, ezért a fenti egyenlet bal oldalát így közelíthetjük:

$$t(t-1) \approx t^2$$

Ezek után az egyenlet a következőképpen rendezhető:

$$t \approx \sqrt{2^{n+1}\ln\left(\frac{1}{1-\lambda}\right)} \approx 2^{(n+1)/2}\sqrt{\ln\left(\frac{1}{1-\lambda}\right)}$$

A fenti képlet ismeretében határozzuk meg, hogy *n* = 80 bites hash függvény kimenetnél mennyire érdemes *t*-t, választani, azaz mennyi próbálkozásra van szükség ahhoz, hogy 0.5 valószínűséggel találjunk ütközést:

$$t = 2^{81/2}\sqrt{\ln\left(\frac{1}{1-0.5}\right)} \approx 2^{40.2}$$

A fenti eredmény ismerete nélkül azt gondolnánk, hogy ha egy hash algoritmus kimenete 80 bites, akkor a támadónak 2^{79} üzenetet kell átnéznie, mire megfelelőt talál. Ám valójában elég 2^{40} darabot átnéznie, ez pedig a ma használatos számítógépek teljesítményét figyelembe véve rendkívül kicsi mennyiség, akár egy egyszerű laptop segítségével is találhatunk egyező kimenettel rendelkező bemeneteket.

V.MELLÉKLET
AZ SHA-1 ÜZENETPECSÉT

Alapműveletek

Mielőtt belemerülnénk az SHA-1 algoritmus részleteibe, bemutatjuk a felhasznált alapműveleteket.

Bitenkénti XOR

A művelet megegyezik az AES kódolónál ismertetettel, azaz a művelet két operandusa 1-1 bitsorozat, az eredmény bitsorozatban azon a helyiértéken van 1, ahol a két operandusban a bitek különböztek egymástól, formálisan, ha A és B bitsorozatok:

$$A \oplus B = C \text{, ahol } C_i = 1 \text{ akkor és csak akkor, ha } A_i \neq B_i \text{.}$$

Példa: 01101 $\oplus$ 01010 = 00111.

Bitenkénti AND

A bitenkénti AND, vagy ÉS műveletben a kimenet i. bitje akkor és csak akkor 1, ha a bemenetek i. bitjei szintén 1-ek:

$$A \wedge B = C \text{, ahol } C_i = 1 \text{ akkor és csak akkor, ha } A_i = B_i = 1 \text{.}$$

Példa: 01101 $\wedge$ 01010 = 01000.

Bitenkénti OR

A bitenkénti AND, vagy ÉS műveletben a kimenet i. bitje akkor és csak akkor 1, ha a bemenetek i. bitjei közül legalább az egyik 1.

$$A \vee B = C \text{, ahol } C_i = 1 \text{ akkor és csak akkor, ha } A_i = 1 \text{ vagy } B_i = 1 \text{.}$$

Példa: 01101 $\vee$ 01010 = 01111.

Balra_forgatás

Adott egy x bitsorozat, ezt balra forgatjuk n pozícióval, és feltételezzük, hogy n kisebb, mint x hossza. A forgatás során n-el balra toljuk a biteket, és az eredeti sorozat n darab legmagasabb helyiértéken lévő bitet a sorozat végéhez illesztjük.

Példa: balra_forgatás(110010, 2) = 001011.

Összeadás mod 2^{32}-ben

Adottak A és B 32 bites előjel nélküli egész változók. Az összeadást az alábbi módon végezzük el:

$$A + B \equiv C \mod 2^{32}$$

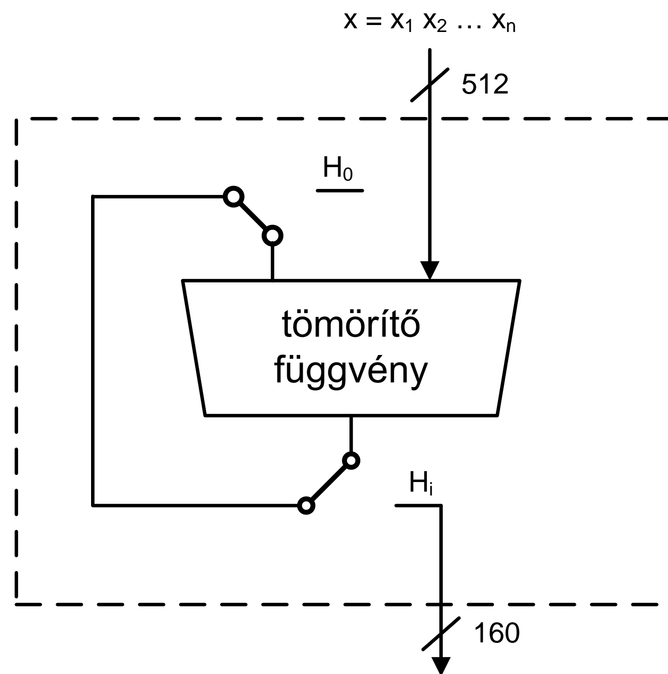
Példa: adjunk össze két számot, amelyeket hexadecimális formában írunk fel:

$$0xCA62C1D6 + 0x8F1BBCDC \equiv 0x597EEB2 \bmod 0x100000000$$

Megjegyezzük, hogy ezt a műveletet C/C++-ban rendkívül egyszerűen implementálhatjuk, ugyanis ezen nyelvekben a 32 bites előjel nélküli típusok összeadás művelete pontosan így működik.

Az SHA-1 működése

A legtöbb hash függvény úgy működik, hogy valamilyen előkészítő lépés után azonos hosszúságú blokkokra bontja a bemenetet. Ezen blokkokon egyenként végrehajt valamilyen tömörítő eljárást, azaz az l bit hosszú blokkot leképezi egy rövidebb, m bites blokkra. A tömörítőnek kettő bemenete van, az egyik az előzőleg leképezett m bites blokk, a másik pedig a következő blokk. Miután elfogytak az l bites blokkok, a tömörítő legutolsó kimenete lesz a hash függvény kimenete is, ezt a működési folyamatot reprezentálja a 16. ábra. Az SHA-1 függvény kimenete 160 bites, és a bemenetet 512 bit hosszúságú blokkokra bontja.



16. ábra Az SHA-1 működése

A fent vázolt algoritmus előtt azonban ki kell egészíteni a bemenetet, ugyanis nem biztos, hogy a bemenet hossza osztható 512-vel. Első lépésként a bemenetet ki kell egészíteni egy 64 bites információval, amely leírja, hogy az eredeti bemenet hány bit hosszú. Amennyiben szükséges, kiegészítő 0 biteket fűzünk a 64 bites információ elé, hogy kijöjjenek az 512 bit hosszú blokkok. A lépések a következők:

1. Az x bitsorozat után egy darab 1-es bitet fűzünk.
2. Kiszámoljuk, hogy hány darab 0-val kell kibővíteni a sorozatot:

$$k \equiv 512 - 64 - 1 - l = 448 - (l + 1) \bmod 512$$

3. Az 1-es bit után fűzünk k darab 0-t az x bitsorozatban.
4. Végül az x-hez hozzáfűzzük a 64 bites előjel nélküli egész értéket, mégpedig úgy, hogy ez l-t, azaz x eredeti bithosszát tárolja. Fontos, hogy ez a 64 bites érték big endian kódolásban kerüljön a sorozat végére!

Ebből látszik, hogy az SHA-1 algoritmus legfeljebb 2^{64}-1 bit hosszúságú adathalmazt képes feldolgozni. Lássunk egy példát a bemenet kiegészítésére: Legyen a bemenet a következő string: x = „abc”.

$$\underbrace{01100001}_{a}\ \underbrace{01100010}_{b}\ \underbrace{01100011}_{c}$$

Az üzenet után egy 1-es bitet fűzünk, ekkor a következő sorozatot kapjuk:

01100001 01100010 01100011 $\underline{1}$

Tudjuk, hogy l = 24, ezért k = 448-(24+1) mod 512 = 423 darab 0-t is a sorozat végére kell fűznünk:

$$01100001\ 01100010\ 01100011\ \underline{1}\ \underbrace{0000\ldots000}_{423\text{ db }0}$$

Végül az l = 24-et egy 64 bites számként a sorozat végéhez illesztjük:

$$01100001\ 01100010\ 01100011\ \underline{1}\ \underbrace{0000\ldots000}_{423\text{ db }0}\ \underbrace{0000\ldots11000}_{64\text{ bites egész}}$$

A blokkok feldolgozása

Az alábbiakban bemutatjuk, hogy milyen transzformációkat végez a tömörítő függvény egy blokkon. Az egyszerűség kedvéért ezek után jelöljük x-el magát a blokkot. Az SHA-1 rendelkezik 5 darab 32 bites belső változóval, amelyek végül a kimenetet tárolják majd, ám ezeknek a legelső blokk feldolgozása előtt beállítunk bizonyos kezdőértékeket (a többi blokk előtt már nem):

- $H_0 = 0\text{x}67452301$
- $H_1 = 0\text{xEFCDAB}89$
- $H_2 = 0\text{x}98\text{BADCFE}$
- $H_3 = 0\text{x}10325476$
- $H_4 = 0\text{xC}3\text{D}2\text{E}1\text{F}0$

A blokk feldolgozása két főbb lépésből áll. Először fel kell töltenünk egy 80 elemű, w nevű tömböt, amely 32 bites egészeket tárol. A tömböt 0-tól kezdjük indexelni. Az x-et, vagyis a blokkot pedig 16 darab 32 bites szóra bontjuk: x = $x_0, x_1, \ldots, x_{15}$. A w tömböt a következő módon töltjük ki:

- Ha $0 \le i \le 15$, akkor w[i] = x_i
- Egyébként w[i] = balra_forgatás(w[i-3] $\oplus$ w[i-8] $\oplus$ w[i-14] $\oplus$ w[i-16], 1).

A blokk feldolgozásához szükségünk lesz 5 darab 32 bites ideiglenes változóra is, ezeket jelöljük A, B, C, D, E-vel. A blokk feldolgozása előtt inicializáljuk ezeket a változókat:

1. A = H_0
2. B = H_1
3. C = H_2
4. D = H_3
5. E = H_4

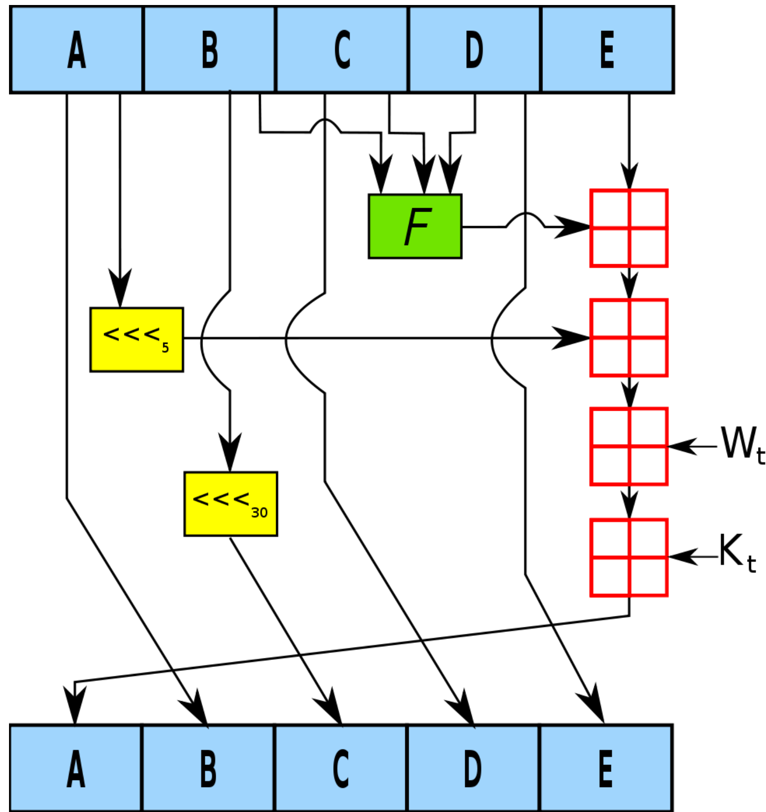
A tömörítő eljárás 4*20 iterációt hajt végre ezeken az ideiglenes változókon, majd ezen iterációk után elvégezzük az alábbi műveleteket:

1. $H_0 = \mathrm{A} + H_0 \pmod{2^{32}}$
2. $H_1 = \mathrm{B} + H_1 \pmod{2^{32}}$
3. $H_2 = \mathrm{C} + H_2 \pmod{2^{32}}$
4. $H_3 = \mathrm{D} + H_3 \pmod{2^{32}}$
5. $H_4 = \mathrm{E} + H_4 \pmod{2^{32}}$

A következő blokk feldolgozásakor pedig ezen H_i értékekből indulunk ki. Végül lássuk, miből áll a 80 iteráció, amit az A, B, C, D és E változókon végre kell hajtani:

1. Ciklus i := 0-tól 79-ig
    a. TEMP = balra_forgatás(A, 5) + f(B, C, D) + E + K + w[i]
    b. E = D
    c. D = C
    d. C = balra_forgatás(B, 30)
    e. B = A
    f. A = TEMP
2. Ciklus vége

Az alábbi ábra a fenti blokk-feldolgozás lépéseit mutatja, az F-doboz a fenti f műveletet jelöli.



A fenti algoritmusban szereplő K konstans és f függvény az i ciklusváltozó függvénye, ezt tartalmazza az 5 táblázat.

5. táblázat Az SHA-1 tömörítő algoritmusa által használt konstansok és függvények

i	f	K
$0 \le i \le 19$	$f = D \oplus (B \wedge (C \oplus D))$	0x5A827999
$20 \le i \le 39$	$f = B \oplus C \oplus D$	0x6ED9EBA1
$40 \le i \le 59$	$f = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$	0x8F1BBCDC
$60 \le i \le 79$	$f = B \oplus C \oplus D$	0xCA62C1D6

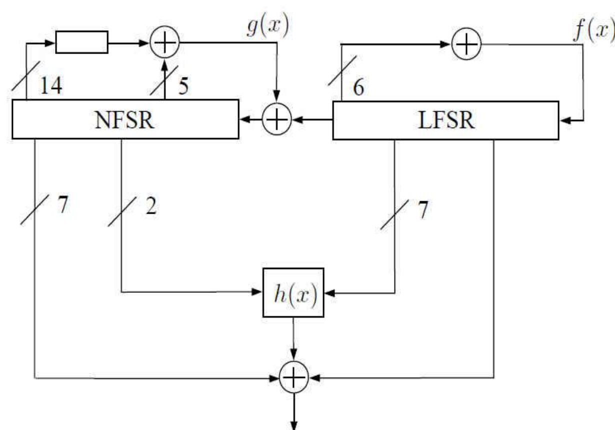
Az SHA-1 tehát képes 512 bitnél rövidebb bemenetre is kimenetet számolni. Jól ismert tesztmondat a következő: „The quick brown fox jumps over the lazy dog”. Az SHA-1 a következő hexadecimális kimenetet adja erre a mondatra: `2FD4E1C6 7A2D28FC ED849EE1 BB76E739 1B93EB12`. Hasonlóan értelmezhető az üres string is mint bemenet, ennek a kimenete: `DA39A3EE 5E6B4B0D 3255BFEF 95601890 AFD80709`.

VI. MELLÉKLET
A GRAIN-128 FOLYAMTITKOSÍTÓ

Jellemzők:

- Svéd-svájci tervezés
- 128 bites titkos kulcs + 96 bites IV
- LFSR alapú
- Nagyon alacsony hardver igény
- Kevés plusz hardverrel gyorsítható, akár 32x
- „eddig" ellenáll a támadásnak
- http://www.ecrypt.eu.org/stream/grainp3.html

Áttekintés: egy lineárisan és egy nemlineárisan visszacsatolt léptető regiszter (LFSR és NFSR) kombinációja.



A visszacsatolások egyenletei:

$$s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}.$$

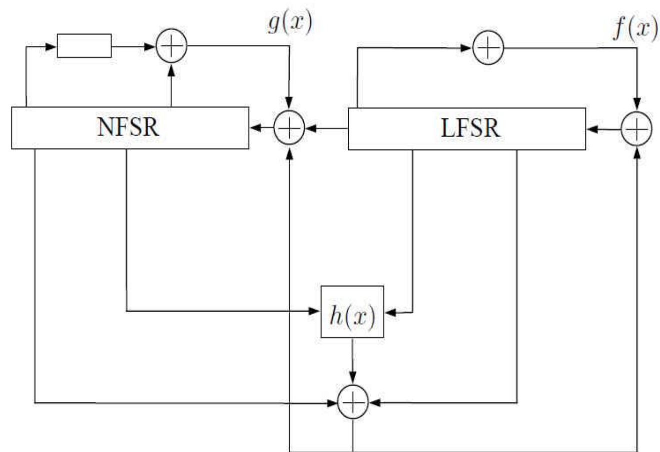
$$\begin{aligned} b_{i+128} \quad = \quad & s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} + \\ & + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} + \\ & + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + \\ & + b_{i+68}b_{i+84}. \end{aligned}$$

$$h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8$$

$$z_i = \sum_{j \in \mathcal{A}} b_{i+j} + h(x) + s_{i+93}$$

ahol $\mathcal{A}$ = {2, 15, 36, 45, 64, 73, 89}

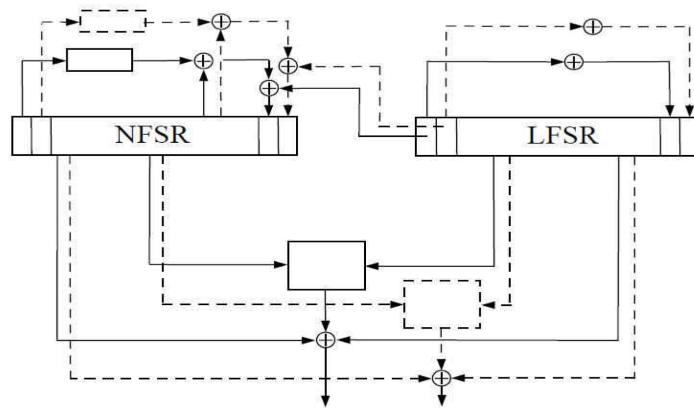
Inicializálás:



Az inicializálás lépései:

- Titkos kulcs (128 bit) >> NFSR
- IV (96 bit) >> LFSR alja (teteje 1-esekkel)
- 256 órajelig járatva

Gyorsítás a visszacsatolások többszörözésével lehetséges. A lényeg, hogy a legnagyobb területet elfoglaló léptetőregisztereket nem kell többszörözni:



A Grain hardver igénye (gate count) különböző gyorsítások esetén, ha a NAND2=1 kapu, NAND3=1.5 kapu, XOR2=2.5 kapu, a D-FF pedig 8 kapu:

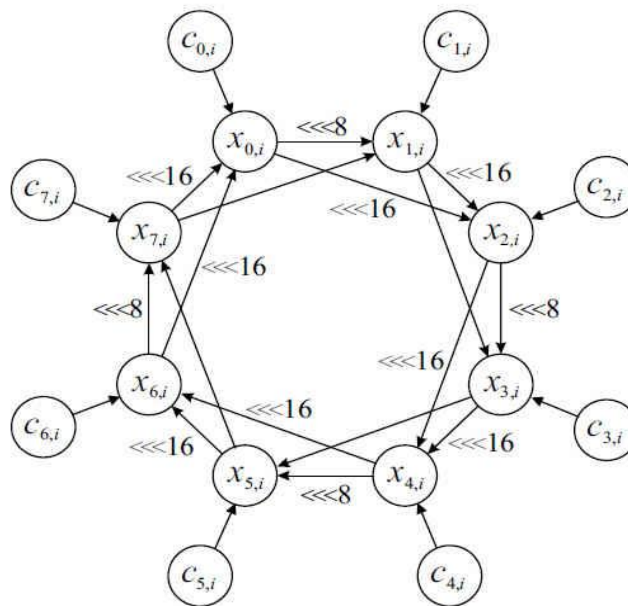
Gate Count	*Speed Increase*					
Building Block	1x	2x	4x	8x	16x	32x
LFSR	1024	1024	1024	1024	1024	1024
NFSR	1024	1024	1024	1024	1024	1024
$f(\cdot)$	12.5	25	50	100	200	400
$g(\cdot)$	37	74	148	296	592	1184
Output func	35.5	71	142	284	568	1136
Total	2133	2218	2388	2728	3408	4768

VII. MELLÉKLET
A RABBIT FOLYAMTITKOSÍTÓ

Jellemzők:

- Dán tervezés
- 128 bites titkos kulcs, 64 bites IV
- 8 db. 32 bites állapot regiszter, 8 db. 32 bites számláló és egy carry bit: összesen 513 bites state
- Az állapotváltozók nemlineárisan függnek egymástól (nincs S-doboz és LFSR!!)
- Gyors szoftver megvalósítás: byte-orientált, és az egész state elfér a regiszterekben
- „eddig” ellenáll a támadásnak
- http://www.ecrypt.eu.org/stream/p3ciphers/rabbit/rabbit_p3.pdf

Áttekintés:



Az $x_{0,i}$ a 0-dik állapotregisztert, a $c_{0,i}$ a 0-dik számlálót jelöli az i-edik ütemben. A <<< művelet a megadott számú rotálást jelenti.

Az állapotregiszterek számításának módja:

$$\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i} \lll 16) + (g_{6,i} \lll 16) \\
x_{1,i+1} &= g_{1,i} + (g_{0,i} \lll 8) + g_{7,i} \\
x_{2,i+1} &= g_{2,i} + (g_{1,i} \lll 16) + (g_{0,i} \lll 16) \\
x_{3,i+1} &= g_{3,i} + (g_{2,i} \lll 8) + g_{1,i} \\
x_{4,i+1} &= g_{4,i} + (g_{3,i} \lll 16) + (g_{2,i} \lll 16) \\
x_{5,i+1} &= g_{5,i} + (g_{4,i} \lll 8) + g_{3,i} \\
x_{6,i+1} &= g_{6,i} + (g_{5,i} \lll 16) + (g_{4,i} \lll 16) \\
x_{7,i+1} &= g_{7,i} + (g_{6,i} \lll 8) + g_{5,i}
\end{aligned}$$

$$g_{j,i} = \left((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 \gg 32)\right) \bmod 2^{32}$$

A >> művelet a jobbra shiftelést jelenti a megadott számú pozícióval. Látható, hogy az alkalmazott nemlineáris művelet a négyzetre emelés. A hatékonyságot növeli, hogy a byte-os eltolások megvalósíthatók egyszerű byte-cserével a regiszteren belül.

A számlálók léptetésének egyenletei:

$$\begin{aligned}
c_{0,i+1} &= c_{0,i} + a_0 + \phi_{7,i} \mod 2^{32} \\
c_{1,i+1} &= c_{1,i} + a_1 + \phi_{0,i+1} \mod 2^{32} \\
c_{2,i+1} &= c_{2,i} + a_2 + \phi_{1,i+1} \mod 2^{32} \\
c_{3,i+1} &= c_{3,i} + a_3 + \phi_{2,i+1} \mod 2^{32} \\
c_{4,i+1} &= c_{4,i} + a_4 + \phi_{3,i+1} \mod 2^{32} \\
c_{5,i+1} &= c_{5,i} + a_5 + \phi_{4,i+1} \mod 2^{32} \\
c_{6,i+1} &= c_{6,i} + a_6 + \phi_{5,i+1} \mod 2^{32} \\
c_{7,i+1} &= c_{7,i} + a_7 + \phi_{6,i+1} \mod 2^{32}
\end{aligned}$$

, az a_i konstansok:

$$\begin{aligned}
a_0 &= \text{0x4D34D34D} & a_1 &= \text{0xD34D34D3} \\
a_2 &= \text{0x34D34D34} & a_3 &= \text{0x4D34D34D} \\
a_4 &= \text{0xD34D34D3} & a_5 &= \text{0x34D34D34} \\
a_6 &= \text{0x4D34D34D} & a_7 &= \text{0xD34D34D3.}
\end{aligned}$$

A carry bit:

$$\phi_{j,i+1} = \begin{cases} 1 & \text{if } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \wedge j = 0 \\ 1 & \text{if } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \wedge j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Az álvéletlen bitfolyamot 128 bites darabokban állítják elő az állapotregiszterekből, az alábbi egyenletekkel:

$$
\begin{aligned}
s_i^{[15..0]} &= x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]} & s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]} \\
s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]} & s_i^{[63..48]} &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]} \\
s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]} & s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]} \\
s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]} & s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}
\end{aligned}
$$

A state inicializálásához az alábbi kezdő állapotot állítják be (K a 128 bites titkos kulcs, $k_0 = K^{[15..0]}$, $k_1 = K^{[31..16]}$, stb):

$$
x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} \diamond k_j & \text{for } j \text{ even} \\ k_{(j+5 \bmod 8)} \diamond k_{(j+4 \bmod 8)} & \text{for } j \text{ odd} \end{cases}
$$

$$
c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} \diamond k_{(j+5 \bmod 8)} & \text{for } j \text{ even} \\ k_j \diamond k_{(j+1 \bmod 8)} & \text{for } j \text{ odd.} \end{cases}
$$

Ezzez a kezdő állapottal 4 ciklusig járatják a titkosítót, majd a 64 bites IV-t is belekombinálják (XOR művelettel) a számlálók állapotába.

A Rabbit teljesítménye szoftver megvalósítás esetén, 8 KB adat titkosításakor az alábbi táblázatban látható, különféle (gyenge) processzorokra. Az első oszlop az órajelciklusok száma, a második a kód mérete byte-okban, a harmadik pedig a felhasznált memória, szintén byte-okban. A mezőkben lévő 3 érték a rejtési iteráció / kulcsinicializálás / IV inicializálás adatát mutatja. Az 513 bites state egy 68 byte-os struktúrában van tárolva.

Processor	Performance	Code size	Memory
Pentium III	3.7/278/253	440/617/720	40/36/44
Pentium 4	5.1/486/648	698/516/762	16/36/28
PowerPC	3.8/405/298	440/512/444	72/72/72
ARM7	9.6/610/624	368/436/408	48/80/80
MIPS 4Kc	10.9/749/749	892/856/816	40/32/32

Célhardveres megvalósítás esetén 0.18 um-es CMOS technológiával 4100 kapuval valósítható meg a Rabbit, 0.048 mm^2 felületen, és ezzel 500 MBit/s működési sebesség érhető el. Ez a sebesség több hardver felhasználásával növelhető, a 32 bites négyzetre emelés pipeline-os gyorsításával (3 db 16*16 bites szorzás és egy összeadás). Az elérhető sebességek:

Pipelines	Gate count	Die area	Performance
1	28K	0.32 mm^2	3.7 GBit/s
2	35K	0.40 mm^2	6.2 GBit/s
4	57K	0.66 mm^2	9.3 GBit/s
8	100K	1.16 mm^2	12.4 GBit/s