Eötvös Loránd Tudományegyetem
Természettudományi Kar

Gyarmati Máté
Matematika MSc

Az internetes kommunikáció kriptográfiai biztonságának néhány matematikai kérdése

Szakdolgozat

Témavezető: Szabó István, egyetemi docens
Valószínűségszámítási tanszék

Budapest, 2017

Köszönetnyilvánítás

Szeretném megköszönni témavezetőmnek Szabó Istvánnak a témafelvetést, a rengeteg ajánlott szakirodalmat, és az építő jellegű megjegyzéseit.

Tartalomjegyzék

Tartalomjegyzék 4

1. Bevezetés 5

2. Kriptográfiai algoritmusok 7
- 2.1. Titkos (Szimmetrikus) kulcsú titkosítás . . . 7
- 2.2. Nyílt (asszimetrikus) nyílt kulcsú titkosítás . . . 7
- 2.3. RSA . . . 7
- 2.4. Diffie-Hellman probléma . . . 8
- 2.5. ElGamal . . . 9
- 2.6. Digital Signature Sheme (DSA) . . . 9
- 2.7. ECC (Ellyptic Curve Cryptography) . . . 10
- 2.8. Hibrid titkosítás . . . 11

3. Hibás véletlenek választása 12
- 3.1. Hibás véletlen Padding esetén . . . 12
- 3.2. Titkos kulcs felfedése DSA esetén . . . 13
- 3.3. RSA támadása hibás véletlen esetén . . . 14
- 3.4. Hibás véletlenszám generátor . . . 14

4. Véletlenszám generátorok 15
- 4.1. TRNG (True Random Number Generator) . . . 15
- 4.2. PRNG (Pseudo random number generator) . . . 15
- 4.3. CSPRNG (Cryptograhically Secure PRNG) . . . 16
- 4.4. LRNG (Linux véletlenszám generátora) . . . 16
  - 4.4.1. Mixing függvény . . . 17
  - 4.4.2. Output függvény . . . 19
  - 4.4.3. Entrópia becslés LRNG-ben . . . 20
- 4.5. Fortuna . . . 21
  - 4.5.1. Generátor . . . 21
  - 4.5.2. Accumulator . . . 22
  - 4.5.3. Entrópia források . . . 22
  - 4.5.4. Pool . . . 22
  - 4.5.5. Seed fájl kezelés . . . 24

5. Véletlenek tesztelése **25**
5.1. Statisztikai tesztek 25
5.2. Entrópia becslés 26
5.3. Plug-in vagy Maximum likelihood becslés 27
5.4. LZ becslés 27
5.5. PV becslő 33
5.6. Min-entrópia becslés 35
5.7. Bayes becslés entrópiára 37

6. Összegzés **38**

1. Bevezetés

Az internetes kommunikáció napjainkban már az életünk része, a fejlett országokban szinte mindenki használja az internetes levelezést, de rengeteg a netes vásásrlás, és az egyéb pénzügyi tranzakció is, sőt az interneten keresztül szerződás és hitelesíthető a digitális aláírások segítségével. Természetes elvárás, hogy a levelezésünket ne tudják illetéktelen személyek elolvasni, vagy hogy egy tranzakció során a bankszámlánk adatai titokban maradjanak a támadó előtt, aki esetleg lehallgatja a kommunikációs csatornát. Nem szeretnénk továbbá, hogy mások nevünkben üzeneteket, vagy digitális aláírásokat hamisítsanak, vagy hogy a szerződő fél letagadhassa a szerződés kötést. Ezek mellett természetes elvárás, hogy a kommunkiáció minél gyorsabb legyen.

A problémák megoldására már rengeteg algoritmust fejlesztettek ki, mint a számításilag hatákony, de közös titkos kulcsot igénylő titkos kulcsú titkosítók (pl. AES,DES), vagy a lassabb, de előzetes kommunikációt nem igénylő nyílt kulcsú titkosítók (pl. RSA,ElGamal). Digitális aláírásra is több lehetőség közül lehet válsztani (pl. RSA, DSA). A titkosító algoritmusnak közös vonása, hogy szükséges hozzájuk egy kulcs, ami a támadó előtt titokban van. Fontos, hogy ez a kulcs titokban maradjon a támadó előtt, mert a kommunikáció csak addig lehet biztonságos, amíg a támadó nem ismeri a kulcsot. Több algoritmus a titkos kulcs előállításán kívül is használ véletleneket (DSA), és mint később látni fogjuk, az itteni véletlenek megismerése is törhetővé teszi az algoritmust. Tehát az eddigieket összegezve a titkosító algoritmusok biztonságához nélkülözhetetlen, hogy valódi, a támadó számára megtippelhetetlen véletleneket használjunk fel. Hogy ezt a kijelentést megerősítsem, bemutatok majd néhány példát a korábbi évekből, amikor a rosszul megválasztott véletlenek az algoritmus töréséhez vezettek.

A véletlenek előállítását véletlenszám generátorok végzik. Habár rendelkezésünkre állnak egyenletes eloszlás szerinti véletleneket előállító, különböző fizikai jelenségeken alapuló valódi véletlenszám generátorok (TRNG), ezek költsége, és sebessége miatt inkább pszeudo-véletlenszám generátorokat (PRNG) alkalmaznak, amik determinisztikus algoritmus segítségével kis mennyiségű valódi véletlenből állítanak elő nagy mennyiségű pszeudo-véletlent. A pzseudo-véletlenek nem valódi véletlenek, de a jelenlegi módszerekkel statisztikailag megkülönbözhetetlenek a valódi véletlenektől. Két széles körben használt PRNG-t be is mutatok. Ezek a Linux beépített véletlen szám generátora (LRNG), és a Fortuna. Ezek nem valódi PRNG-k, mert nem determinisztikusak, ugyanis folyamatosan felhasználnak a számítógépből nyert véletlen információkat, mint például a merev lemez adatai, egérmozgatás, billentyűleütés.

A pszeudo-véletlenszám generátor csak akkor lehet biztonságos, ha valódi véletleneket tud felhasználni. A valószínűségi változók információtartalmának mérésére vezette be Shannon az entrópiát, amit a $H(X) = \sum_x -\Pr(X = x) \log \Pr(X = x)$ képlettel adott meg (a log a 2-es alapú logaritmust jelenti az egész dolgozatban). Az entrópia az adott valószínűségi változó bizonytalanságát adja meg. Tehát 1 egyenletes eloszlású véletlen bit entrópiája 1, 128 egyenletes eloszlású véletlen bitté 128. De ha mondjuk 4 véletlen bitről tudom, hogy pontosan 1 db 1-es van közöttük, akkor

már csak 2 az entrópia. A felhasznált véletlenektől általában 128 entrópiát várnak el, ugyananis 128 entrópiájú véletlen biztos kitalálásához 2^{128} tipp szükséges, amit jelenlegi eszközökkel nem lehet végigvinni. Az entrópia könnyen számolható ha ismert valószínűségi változók eloszlása, de nehéz feladat, ha az eloszlás ismeretlen. Rengeteg módszer van az entrópia becslésére, de hogy ezek tényleg pontosak legyen, különböző feltételek szükségesek a forrásról. Például a tömörítő algritmusok, mint amilyenek a Lempel-Ziv algortimusok is, használhatók az entrópia becslésére, de a pontossághoz szükséges, hogy az adatokat előállító forrás ergodikus, és amellett stacionárius vagy markov legyen. Ezek a feltételek statisztikai módszerekkel nem ellenőrizhetők, de elég természetesek, hogy elvárhatók legyenek a forrástól.

A következőkben bevezetem a szakdolgozat során felhasznált alapfogalmakat. Legyen X valószínűségi változó értékkészlete A. Ekkor X Shannon entrópiája, röviden entrópiája

$$H(X) = \sum_{x \in A} - \Pr(X = x) \log \Pr(X = x)$$

A min-entrópia a valószínűségi változó lehető legkisebb információtartalma, azaz

$$H_{\textit{min-entropy}}(X) = - \log \max_{x \in A}(Pr(X = x)).$$

A definiciókból nyilvánvaló, hogy $H_{\textit{min-entropy}}(X) \leq H(X)$. Az $\mathbb{X}$ forrás betűnkénti entrópiája

$$H(\mathbb{X}) = \lim_{n \to \infty} \frac{1}{n} H(X_1, ..., X_n)$$

ha ez a határérték létezik. Ha $\mathbb{X}$ stacionárius, akkor a határérték létezik, és

$$H(\mathbb{X}) = \lim_{n \to \infty} H(X_n | X_1, ...X_{n-1}).$$

Ennél több is mondható, ha $\mathbb{X}$ ergodikus és véges értékkészletű. A Shannon-McMillan-Breiman tétel alapján

$$-\frac{1}{n} \log \Pr(X_1^n) \to H(\mathbb{X}) \text{ 1 valószínűséggel}$$

ahol $X_1^n = X_1, ..., X_n$.

A második fejezetben bemutatok néhány széles körben használt kriptográfiai algoritmust. A harmadikban példákat mutatok támadásokra, amik a hibás véletleneket használják ki. A negyedik fejezetben két véletlenszám generátort mutatok be, a Linux RNG-t és a Fortunát. Az ötödik fejezetben az entrópia becslésére mutatok módszereket. Ezek között bemutatok majd az LZ77 algoritmus működésén alapuló entrópia becslőket, és belátom, hogy stacionárius, ergodikus forrás esetén az előállított sorozat becsült értéke tart a forrás betűnkénti entrópiájához.

2. Kriptográfiai algoritmusok

Napjainkban számtalan titkosító eljárást használunk. A titkosítási algoritmusok általában három fázisból áll: Kulcsgenerálás, Titkosítás, Dekódolás. Alapvetően két felé oszthatók a protokollok attól függően, hogy a titkosításban részt vevő szereplők ugyanazzal a kulccsal rendelkeznek (szimmetrikus), vagy saját titkos kulcsuk van (asszimetrikus).

2.1. Titkos (Szimmetrikus) kulcsú titkosítás

Az alapja, hogy a két kommunikáló fél rendelkezik egy közös kulccsal, amit csak ők ismernek. A titkosítás és a dekódolás is ezen kulcs felhasználásával történik. Ide tartoznak a stream titkosítók, mint a One-Time Pad (OTP), és a blokk titkosítók, mint a DES, és az AES. A DES a kis kulcsmérete (56-bit) miatt nem biztonságos, helyette használható a Triple DES, ami nagyobb, 112 vagy 168 bites kulcsmérettel rendelkezik.

A szimmetrikus titkosítók előnye, hogy bitműveleteket használnak, amiket a számítógépek hatékonyan tudnak számolni. Ezért a titkos kulcsú titkosítók számításilag hatékonyak. A hátrányuk, hogy a kommunikáló feleknek előzetesen meg kell egyezniük egy titkos kulcsban, ami problémás lehet, ha a felek között nincs biztonságos csatorna.

2.2. Nyílt (asszimetrikus) nyílt kulcsú titkosítás

Minden résztvevő rendelkezik saját (p_k, s_k) kulcspárral. Alice p_k kulcsa publikus, mindenki számára ismert. Ha Bob üzenetet akar küldni Alice-nek, akkor ehhez Alice p_k kulcsát használja fel. Az s_k kulcs titkos, csak a kulcs tulajdonosa, jelen esetben Alice ismeri. Alice a kapott üzenetet s_k segítségével állítja vissza. A titkosítás alapja egy nehéz matematikai probléma, amit a jelenlegi számítástechnikai eszközökkel nem tudunk megoldani. Ilyen problémák a faktorizálás és a diszkrét logaritmus. A nyílt kulcsú titkosítás előnye, hogy a résztvevőknek nem kell előre találkozniuk. Hátránya, hogy lassú, így nem alkalmas nagy mennyiségű üzenet hatékony titkosítására.

2.3. RSA

A legismertebb asszimetrikus séma az RSA. A titkosítás a faktorizálás nehézségén múlik. Bob szeretne $m < n$ hosszú üzenetet küldeni Alicenak.

Kulcsgenerálás

1. Alice generál két különböző prímet, p-t és q-t, és legyen $N = pq$.
2. Választ $e \in \{1, ..., \varphi(N)\}$-t, amire $(e, \varphi(N)) = 1$.
3. Kiszámolja $d = e^{-1} \mod \varphi(N)$-t.
4. Nyilvános kulcs (N, e), titkos kulcs (d).

Titkosítás

1. Bob kiszámolja $c = m^e \mod N$-et-t, és elküldi Alicenak.

Dekódolás

1. Alice kiszámolja $m = e^d \mod N$-et

Helyesség
$m^{de} = m^1 = m \mod N$

A faktorizálás biztonságához a NIST szabvány szerint N mérete legalább 2048-bit (3072, 4096 szokott még előfordulni). Az RSA ebben a formában nem biztonságos, például ha az exponens kicsi, és m üzenetre $m < N^{1/e}$ teljesül, akkor m meghatározható sima e-dik gyökvonással, ráadásul az algoritmus determinisztikus. Ezenkívül további támadások is ismertek [1].

A támadások kivédésére az RSA használata előtt az üzenetbe véletlen biteket ágyaznak be (padding). Az RSA-hoz leggyakrabban az OAEP [2] padding sémát alkalmazzák, ami egyfajta Feistel hálózat.

Legyen n az RSA modulus bithossza, k_0 és k_1 előre rögzített konstansok. $m \in \in \{0,1\}^{n-k_0-k_1}$ az üzenet, $G : \{0,1\}^{k_0} \rightarrow \{0,1\}^{n-k_0}$ generátor és $H : \{0,1\}^{n-k_0} \rightarrow \{0,1\}^{k_0}$ kriptográfiai Hash-függvény. A kódolás a következő módon működik:

1. m-et kiegészíti k_1 0-val, $m' = m \| 0^{k_1}$
2. $r \in \{0,1\}^{k_0}$ véletlen
3. $X = m' \oplus G(r)$
4. $Y = H(X) \oplus r$
5. output: $(X \| Y)$

A dekódolás is egyszerű:

1. $r = H(X) \oplus Y$
2. $m' = X \oplus G(r)$
3. Levágja m' végéről a k_1 0-t, így kapjuk m-et

2.4. Diffie-Hellman probléma

A Diffie-Hellmann kulcscsere során Alice és Bob megegyeznek egy közös kulcson. Közösen választanak egy q rendű ciklikus csoportot, és annak egy g generátorát. Alice választ egy $a \in \{1, ..., q-1\}$ számot, Bob pedig egy $b \in \{1, ..., q-1\}$. Alice átküldi g^a-t Bobnak, Bob pedig g^b-t Alicenak. Így mindketten ki tudják számolni

g^{ab}-t, ez lesz a közös kulcs. A protokoll alapja a diszkrét logaritmus probléma, itt egész pontosan az, hogy g, g^a, g^b ismeretében nem lehet hatékonyan számolni g^{ab}-t. Ezt a feltételezést hívják Computional Diffie-Hellmann (CDH) feltevésnek.

A Diffie-Hellman kulcscsere nem terjedt el ebben a formában, elsősorban azért, mert az RSA-val szemben, ezt nem lehet használni hitelesítések aláíráásra, de a Computional Diffie-Hellmann feltevés az alapja az ElGamal, a DSA és még néhány további protokolloknak is.

2.5. ElGamal

Most egy másik igen gyakran használt asszimterikus sémát mutatok be, ami RSA-val ellentétben a diszkrét logaritmus problémát használja ki. Bob szeretné az m üzenet elküldeni Alice-nak.

Kulcsgenerálás

1. Alice generál egy q rendű G ciklikus csoportot, annak g generátorával.
2. Választ egy $x \in \{1, ..., q-1\}$ véletlent, és kiszámolja $h := g^x$.
3. Publikálja a (G, q, g, h)-t, ez lesz Alice publikus kulcsa, (x) pedig a titkos kulcsa.

Titkosítás

1. Bob választ egy $y \in \{1, ..., q-1\}$ véletlent, és kiszámolja $c_1 = g^y$-t.
2. Kiszámolja $s := h^y = g^{xy}$-t.
3. Átalakítja az m üzenetet m' G-beli csoportelemmé, majd kiszámolja $c_2 = m' \cdot s$-t.
4. (c_1, c_2)-t visszaküldi Alicenak.

Dekódolás

1. Alice kiszámolja $s := c_1^x = g^{xy}$-t.
2. Majd $m' = c_2 \cdot s^{-1}$-et, és ezt visszaalakítja az m üzenetté.

A protokoll helyességét mutatja, hogy $c_2 \cdot s^{-1} = m' \cdot h^y \cdot (g^{xy})^{-1} = m' \cdot g^{xy} \cdot g{-}xy = m'$ A szabvány szerint q legalább 2048-bites.

2.6. Digital Signature Sheme (DSA)

Bár az RSA és az ElGamal is módosítható, hogy használható legyen digitális aláíráshoz, bemutatok még egy algoritmust, amit kifejezetten digitális aláíráshoz készült. Ez is egy széles körben elterjedt algoritmus.

Kulcsgenrálás

1. Legyen H Hash-függvény (pl SHA-2), (L, N) pedig kulcshosszak (2048,256) vagy (3072,256) bitmérettel.
2. q legyen N-bites prím, p L-bites prím, amire $q|p-1$.
3. Válasszák meg g-t úgy, hogy g rendje q a p multiplikatív csoportjában.
4. Megosztják (p, q, g)
5. Alice választ egy $x \in \{0, ..., q\}$ véletlent, ez lesz a titkos kulcsa, $y = g^x$ pedig a publikus kulcs.

Aláírás

1. Alice választ egy $k \in \{0, ..., q\}$ véletlent
2. Kiszámolja $r = (g^k \mod p) \mod q$ -t, és ha $r = 0$ akkor újra kezdi egy másik k értékkel
3. Kiszámolja $s = k^{-1}(H(m) + xr) \mod q$ értékéet, és ha $s = 0$, akkor újra kezdi másik k-val.
4. (r, s) pár lesz az aláírás

Megerősítés

1. Elutasít, ha $0 < r < q$ és $0 < s < q$ nem teljesül.
2. $w = s^{-1} \mod q$
3. $u_1 = H(m) \cdot w \mod q$ és $u_2 = r \cdot w \mod q$.
4. $v = g^{u_1} y^{u_2} \mod p \mod q$
5. Aláírás érvényes, ha $v = r$

Helyesség

Mivel g rendje q a p multiplikatív csoportjában, így a g kitevőjében a műveleteket $\mod q$ végezhetjük el. Ez alapján $v = g^{u_1} y^{u_2} = g^{u_1} g^{u_2 x} = g^{H(m)w + rwx} = g^{(H(m)+rx)w} = g^{sks^{-1}} = g^k$, és ezt kellett belátni a helyességhez.

2.7. ECC (Ellyptic Curve Cryptography)

Jelentősége miatt szeretném még megemlíteni az ECC-t, azaz elliptikus görbe alapú titkosítást, ami szintén a publikus kulcsú titkosítások közé tartozik. Az alapja, hogy véletlen elliptikus görbén nehéz megoldani az elliptikus görbékre vonatkozó diszkrét logaritmus problémát. Kulcscserére (ECDH) és digitális aláírásra (ECDSA) használják, egy már létező, diszkrét logaritmus alapú algoritmust (pl DSA) aalkalmaznak elliptikus görbére (ECDSA). Ezek mellett különböző PRNG-k alapja is elliptikus görbe ($Dual_E C_P RNG$). Az előnye, hogy szignifikánsan rövidebb kulcsmérettel tudja elérni ugyanazt a biztonságot, mint az RSA tipusú algoritmusok. Például a 256-bites elliptikus görbe publikus kulcsa hasonló biztonságot eredményez, mint 3072-bites RSA publikus kulcs.

2.8. Hibrid titkosítás

A nyílt kulcsú titkosítás előnye, hogy az üzenet váltáshoz a résztvevőknek nem kell előtte találkozni, hogy megállapodjanak egy közös kulcsban, mint a szimmetrikus esetén. Hátránya, hogy a műveletek elvégzése sokkal lassabb. Ezért a gyakorlatban az internetes kommunikáció során a kétfajta titkosítási séma, a szimmetrikus és az asszimetrikus titkosítás egyfajta hibrid keverékét használják. Ez azt jelenti, hogy nyílt kulcsú titkosítást csak kapcsolatfelvételre és kulcscserére használják, és miután megállapodtak a felek a közös kulcsban, utána a sokkal gyorsabb blokk titkosítókat használják (pl TSL,SSL).

3. Hibás véletlenek választása

Sok titkosítási algoritmus használ véletleneket. Egyrészt az algoritmus inicializálásánál, például az RSA esetén a p, q prímek véletlenül vannak választva. Másrészt a titkosító algoritmus is használhat véletlent, például az El Gamal titkosításnál. De a szimmetrikus kulcsú titkosítok kulcsát is valamilyen véletlent generáló algoritmus segítségével határozzák meg. Az algoritmusok biztonsága szempontjából létfontosságú, hogy a véletlen titokban maradjon a támadó előtt, mert különben az üzenet általában egyszerűen visszafejthető a véletlen ismeretében. A következőkben bemutatok néhány olyan támadást, ami nem megfelelően választott véletlent használja ki.

Ha kis halmazból választ véletlent, vagy kis entrópiájú véletlent használ az algoritmus, akkor az összes lehetséges érték végig próbálható a támadó számára. Tehát ebben az esetben tekinthetjük úgy is, hogy a támadó ismeri a véletlent. Ez gyakran teljes feltöréshez vezet. Például ha a támadó ismeri a szimmetrikus kulcsú titkosítók titkos kulcsát, akkor a titkosított üzenetet azonnal fel tudja törni.

A Netscape SSL korai változata egy PRNG-t használt, ahol a seed-et a dátum, a folyamat azonosítója, és a folyamat szülőjének azonosítója alkotta, ez pedig kevés entrópiát tartalma miatt könnyen megtippelhető [3] a támadó számára. A hibát a ésőbbi változatban javították.

Az El Gamal esetén a (G, q, g, h) publikus kulcs mellett az r ismeretében a (c_1, c_2) üzenetből a $c_1 = g^r$ és $s = h^r$ is ismert a támadó számára, innen pedig egyszerűen megkapható $m = c_2 * s^{-1}$, ami az üzenet.

3.1. Hibás véletlen Padding esetén

Tekintsük az előző fejezetben bemutatott RSA-OAEP algoritmust. Ha m az üzenet, r a véletlen, akkor a padding ebből az

$$m' = \left(m \| 0^{k_1} \oplus G(r)\right) \| \left(r \oplus H(m \| 0^{k_1} \oplus G(r))\right)$$

bitsorozatot állítja elő. Tegyük fel, hogy az m üzenet rövid, azaz az első néhány jegye 0. Legyen m^* az üzenet érdemi része, azaz elhagyjuk m-ből az első néhány 0-t. Jelölje x az

$$m \| 0^{k_1} \oplus G(r)$$

m^*-tól függő részét. Az utolsó k_1-bit nem függ m^*-től, és ha m^* rövid, akkor az első néhány sem, tehát x lesz a köztes részt. Legyen továbbá

$$y := r \oplus H(m \| 0^{k_1} \oplus G(r)).$$

Ekkor

$$m' = x \cdot 2^{k_0 + k_1} + y + K$$

alakban írható, ahol K csak $G(r)$-től függ, tehát K ismert. $c = (x \cdot 2^{k_0+k_1} + y)^e \mod N$ lesz a titkosított szöveg, ez speciális esete a $p(x, y) = c \mod N$ egyenletnek, ahol $p \in \mathbb{Z}/n$. Tekintsük az alábbi tételt [4].

3.1. Tétel (Coppersmith). *Legyen adott N egész, és a $p \in \mathbb{Z}[x]$ 1 főegyütthatós d fokú polinom. Ekkor* $(\log N, 2^d)$ *függvényében polinom időben megtalálható minden olyan x_0 egész, amire $p(x_0) = 0 \mod N$, és $|x_0| < N^{1/d}$.*

Habár Coppersmith algoritmusa csak egy változós esetre szól, de heurisztikusan kiterjeszthető kétváltozós esetre is. Világos, hogy x és y ismeretében visszafejthető m is. Ha m^* nem túl hosszú, azaz durva becsléssel x és y együttes hossza legfeljebb $N^{1/e}$, akkor a Coppersmith algoritmus megtalálja őket.

A [5] cikk szerint 2048-bites kulcs esetén, ha legfeljebb 440-bit hosszú, vagy 3072-bites kulcs esetén, ha legfeljebb 750-bit hosszú a tényleges üzenet, akkor sikeresen megtalálható.

Tekintsük a $PKCS\#1v1.5$ padding sémát. Legyen k az RSA modulus bájtjainak száma. Az m üzenet bájtjainak száma legfeljebb $k-11$. A padding tartalmaz $k-3-|m| \geq 8$ bájtnyi véletlen bájtot. A paddig továbbá tartalmaz az elején két konstans bájtot, és a véletlen bájtok és az üzenet között további egy konstans bájtot. Tehát a paddig így áll össze

$$m' = 0002\|r\|00\|m.$$

Ez felírható a következő formában is $m' = 2^f \cdot a + m$ alakban is. Mivel $f < 1024$, és az r kis entrópiájú véletlen, így az (f, a) pár lehetséges értékei végigpróbálhatók a támadó számára. Tehát feltehetjük, hogy a támadó ismeri az (f, a) párt. Most tegyük fel, hogy a támadó valamilyen módon hozzájut ugyanannak az üzenetnek két különböző titkosításához, például Bob elküldi az üzenetet Alicenak, de a támadó megakadályozza, hogy a levél eljusson Alicehoz. Mivel Bob nem kap választ, újraküldi az üzenetet Alicenak, de most más véletlent használ az algoritmus. A támadó lehallgatja mindkét üzenetet, így rendelkezik a $c_1 = (2^f \cdot a_1 + m)^e$ és $c_2 = (2^f \cdot a_2 + m)^e$ értékekkel. Legyen $g_1(x) = x^e - c_1$ és $g_2(x) = (x + a_2 - a_1)^e + c_2 - c_1$. Ekkor $2^f \cdot a_2 + m$ gyöke g_1-nek és g_2-nek, így a legnagyobb közös osztónak. Euklédeszi algoritmus segítségével meghatározható a legnagyobb közös osztó. Ha $e = 3$, akkor innen már $2^f \cdot a_2 + m$ is, azaz m is egyszerűen megkapható.

3.2. Titkos kulcs felfedése DSA esetén

Vegyük az előző fejezetben szereplő DSA aláíró algoritmus. Tegyük fel, hogy az aláíráskor generált k véletlen fix. Megmutatjuk, hogy ebben ez az esetben a titkos kulcs meghatározható. Ha az m_1, m_2 üzenetekre $z_1 := H(m_1)$ és $z_2 := H(m_2)$, akkor $r = g^k$ szintén fix.

$$s_1 = k^{-1}(z_1 + xr) \tag{1}$$
$$s_2 = k^{-1}(z_2 + xr) \tag{2}$$

Tekintve (1)-(2)-t, kapjuk, hogy $s_1 - s_2 = k^{-1}(z_1 - z_2)$. $s_1 - s_2$ és $z_1 - z_2$ ismert, innen $k = (z_1 - z_2)(s_1 - s_2)^{-1}$ könnyen számolható. Most már 1-ből x kivetelével minden ismert, így a titkos kulcs, x is kiszámolható, $x = (s_1 k - z_1)r^{-1}$.

A Sony ECDSA, azaz elliptikus görbe alapú DSA algoritmust használt a Playstation 3 aláírására, de az implementálás hibás volt, ugyanis a k véletlen értéke rögzített

volt. A fent bemutatott támadással a titkos kulcs egyszerűen meghatározható. De nem ez volt az egyetlen támadás az ECDSA ellen [6]. 2013-ban felfedték, hogy az Androidon a Bitcoin implementációjában az ECDSA-hoz használt *SecureRandom* véletlen generátorában hiba van, ugyanis időnként összeütközéseket generál [7]. Mint láttuk, összeütközés esetén kiszámolható a titkos kulcs, ami Bitcoinok ellopásához vezethetett.

3.3. RSA támadása hibás véletlen esetén

Az RSA biztonságában létfontosságú, hogy két nagy prím szorzatát ne tudjuk hatékonyan faktorizálni. 2012-ben Lenstra, Hughes, Augier, Bos, Kleinjung, és Wachter több millió RSA publikus kulcsot gyűjtött össze, és ezek 0,2%-át képes volt faktorizálni, mindössze az euklideszi algoritmus felhasználásával [8]. Ennek oka, hogy ugyanazt a p prímet, szándékosan vagy véletlenül felhasználták több $n_1 = pq_1$, és $n_2 = pq_2$ modulus képzésére is, ahol q_1, q_2 különböző prímek. n_1, n_2 ismeretében az euklideszi algoritmus segítségével könnyen meghatározható a legnagyobb közös osztó, azaz p, innen pedig már könnyen megy a faktorizáció is.

3.4. Hibás véletlenszám generátor

Egy másik fajta probléma adódott az NSA által tervezett elliptikus görbe alapú CSPRNG-vel, a *Dual_EC_DRBG*-vel. Itt a probléma, hogy a *Dual_EC_DRBG* által használt konstansok megválaszthatók úgy is, hogy azok túl sok bit generálásakor egy hátsó ajtót biztosítsanak. Ez azt jelenti, hogy ha konstansok között egy bizonyos kapcsolat fennáll, amit a konstansok megválasztója tud, akkor ő nem túl sok generált bit segítségével vissza tudja fejteni a seed-et [9]. A *Dual_EC_DRBG* a NIST által ajánlott 4 CSPRNG közé is bekerült az NSA által ajánlott konstansok felhasználásával [10]. Mivel a konstansokat megválasztását csak az NSA ismeri, fennáll az esélye, hogy azok úgy lettek megtervezve, hogy hátsó ajtót biztosítson számukra. Ezen okokból *Dual_EC_DRBG*-t később visszavonták az ajánlott CSPRNG-k közül.

4. Véletlenszám generátorok

Véletlenek felhasználása igen széleskörű, például szerencsejáték, minta vétel, számítógépes szimulációk, és kriptográfia. Rövid idő alatt nagy mennyiségű véletlenre van szükség, erre a célra számos véletlen szám generátort terveztek. Azonban ezek nagyrészénél a generált érték könnyen megtippelhető. Az előző fejezetben láttuk, hogy a titkosító algoritmusok törhetőek rossz véletlenek választása esetén. Éppen ezért nagyon körültekintően kell megválasztani a titkosításhoz szükséges véletleneket előállító véletlenszám generátort. A fejezetben bemutatok két, széles körben használt véletlenszám generátort.

4.1. TRNG (True Random Number Generator)

TRNG, ahogy a neve is jelzi valódi véletlenszámot állít elő, amik valamilyen fizikai jelenségen alapulnak, amit entrópia forrásnak nevezünk. Ilyenek például az érme feldobás és a kocka dobás is. Azonban kriptográfiai felhasználásban inkább a mikroszkópikus folyamatokat szokás figyelembe venni. Ezek közül a két legfontosabb forrás kvantum folyamatok atomi és szubatomi szinten és termikus zajok. Például az atomok bomlása megtippelhetetlen, így kíváló forrást nyújtanak véletlen számok gyűjtéséhez. Azonban az ezek méréséhez szükséges eszközök gyakran nem állnak rendelkezésre, vagy nem generálnak megfelelő mennyiségű véletlent.

A szükséges nagy mennyiségű entrópiát a számítógépek az egér mozgatásából, a billentyűk leütéséből, a merevlemez adataiból, egyes folyamatok megszakításából, és a minta vétel időpontjából gyűjtik. Habár egy-egy minta pl egy billentyű leütése nem tökéletes véltetlen, hiszen egyes billenytyűket sokkal gyakrabban nyomunk le, ráadásul a billentyűket is csak megfelelő időközönként tudjuk leütni, a folyamat mégis tartalmaz kis mennyiségű entrópiát. Viszont a kvantum jelenségekkel ellentétben gyorsan, nagy mennyiségű adatot tudunk gyűjteni. Sok, kis entrópiájú véletlenből pedig hash-függvényekkel előállítható kis mennyiségű, de tökéletes véletlen. A linux beépített generátora is ezen az elven gyűjt entrópiát, ennek működését később szeretném részletesen is bemutatni.

4.2. PRNG (Pseudo random number generator)

Habár egér mozgatással, billentyű leütéssel, merevlemezről kiolvasott adatokkal már gyorsabban lehet valódi véletleneket előállítani, de ez még mindig költséges, és időigényes. Éppen ezért a kriptográfiai algoritmusokban általában nem valódi véletlent, hanem pszeudo-véletlen számokat használnak. A pszeudo véletelenszám generátor f egy determinisztikus algoritmus, ami egy véletlen s input-ból (seed) olyan $f(s)$ számsorozatot generál, ami nagyon hasonlít egyenletes eloszlású véletlen számok egy sorozatához. A PRNG által generált sorozat általában jóval hosszabb, mint maga a seed, azaz $|s| < f|(s)|$. Mivel determinisztikus algoritmust használunk, a generált $f(s)$ sorozat entrópiája nem lehet több, mint s-é, de $f(s)$ statisztikailag megkülönböztethetetlen az azonos hosszúságú r egyenletes eloszlású véletlentől. Ez talán

látszólag ellentmondásnak tünhet, hiszen $f(s)$ nyílván kevesebb entrópiát tartalmaz, mint maga r, de a véges számolási kapacításunk miatt ez nem derül ki.

Úgy is tekinthetünk a PRNG-re, mint egy véges determinisztikus gép. Egy véletlen kezdő állapotból indul, $s \in \{0,1\}^k$, és egy lépésben átmegy egy másik $t \in \{0,1\}^k$ állapotba, közben az állapotától függően egy pszeudo-véletlen bitet generál. Mivel determinisztikus, ha a támadó valahogy megtudja az aktuális állapotot, akkor onnantól tudni fogja az összes további generált értéket, sőt akár a korábbiakat is. Ráadásul az aktuális állapot legfeljebb 2^k-féle lehet, így a generált bitek periódikusak, és a periódus maximális hossza 2^k. Ez javítható, ha az aktuális állapotba belehashelünk némi entrópiát is.

4.3. CSPRNG (Cryptograhically Secure PRNG)

Kriptográfiailag biztonságos PRNG. A PRNG-hez szükséges statisztikai feltételek mellett kriptográfiai feltételeket is kielégít. A titkosítási algoritmusokban a biztonság érdekében csak CSPRNG használható.

Kriptográfiai pszeudo-véletlenszám generátor is megtippelhető, ha valahogy megtudják a kezdő állapotot. A különbség, hogy feltéve, hogy a generátorba megfelelő mennyiségű entrópiát oltunk, és feltéve, hogy a kriptográfiai algoritmusok annyira biztonságos, mint azt elvárjuk tőlük, kriptográfiai generátorok nagyon lassan fednek fel információt a kezdő állapotról. Az ilyen generátorok nagy mennyiségű véletlent képesek előállítani, mielőtt el kéne kezdenünk aggódni a biztonságot illetően.

4.4. LRNG (Linux véletlenszám generátora)

A Linuxnak és a Windowsnak is van beépített véletlenszám generátora, mind a biztonságos kommunikációhoz, mind a felhasználók javára. Mivel a linuxé nyílt forráskódú, lehetőség volt a generátor alaposabb elemzésére is, én is ezeket a cikkeket használtam fel [11] [12] [13] [14]. Több cikk is született már, ahol a generátor működését vizsgálták, illetve kielemezték az aktuális verzió gyengeségeit [13] [14]. Ezek minden alkalommal javításra kerültek, úgyhogy feltételezhetjük, hogy az aktuális verzió biztonságos.

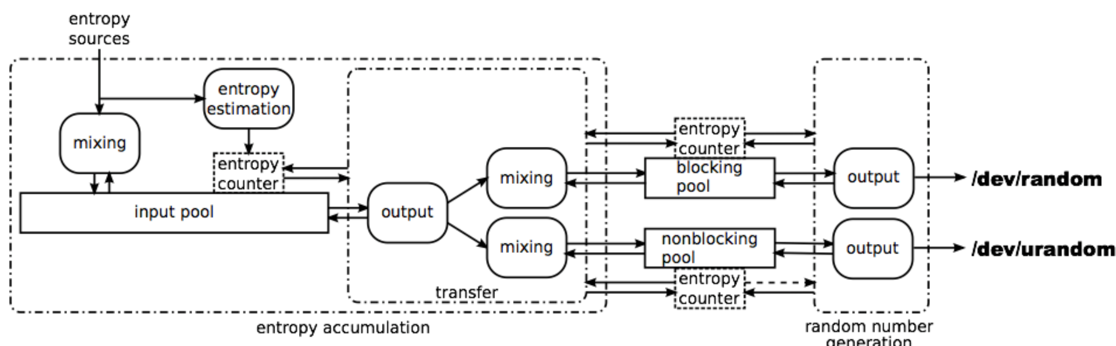
A LRNG három lehetőséget nyújt véletlenszámok generálására:

– */dev/random*

– */dev/urandom*

– *get_random_bytes*()

A *get_random_bytes*() függvény csak a számítógép kernel számára elérhető. A másik kettő elérhető a felhasználók számára is. A */dev/random* és a *dev/urandom* közötti alapvető különbség, hogy amíg a */dev/random* blokkolhatja a véletlenszám kérést, ha nem áll rendelkezésünkre elég entrópia, addig a */dev/urandom* mindig biztosít számunkra véletlent. Persze itt fennállhat a veszély, hogy a kapott véletlen nem tartalmaz elég entrópiát.



1. ábra. LRNG

A belső tér három különálló pool-ból áll, ezek az entropy pool-ok. Az input pool, a blocking pool, és a non-blocking pool. Az input pool 4096 bites, míg a blocking és non-blocking pool 1024 bites. Az utóbbi kettőt hívhatjuk output pool-nak is, hiszen ezekből állítjuk elő a kért véletlen számokat. Mindegyik pool-hoz tartozik egy entrópia számláló, ami egy becslést ad a pool bitjeinek aktuális entrópiájára.

Ha n random bitet kérünk $/dev/random$-tól, akkor az elolvassa a blocking pool-t, és generál számunkra n bit-nyi véletlent, majd csökkenti az entrópia számlálót n-nel. Ha nem áll rendelkezésre megfelelő mennyiségű entrópia a számláló szerint, akkor a folyamat blokkol, amíg nem a lesz blocking pool-ban elég entrópia.

Ha n random bitet kérünk $/dev/urandom$-tól, akkor elolvassa a blocking pool-t, és generál számunkra n bitnyi véletlent, majd csökkenti az entrópia számlálót n-nel. Ha a számláló 0-ra csökken, nem áll le a folyamat, hanem úgy működik, mint egy PRNG.

Ha kérünk a blocking vagy a non-blocking pool-tól n bitnyi véletlent, de számlálók szerint nem áll rendelkezésre elég, akkor az input pool-ból szállítunk át entrópiát a megfelelő pool-ba. Ekkor az input pool számlálója csökken, míg a megfelelő output pool-é nő.

A generátor 2 függvényt használ a bit-ek előállítására, mixing függvényt és output függvényt.

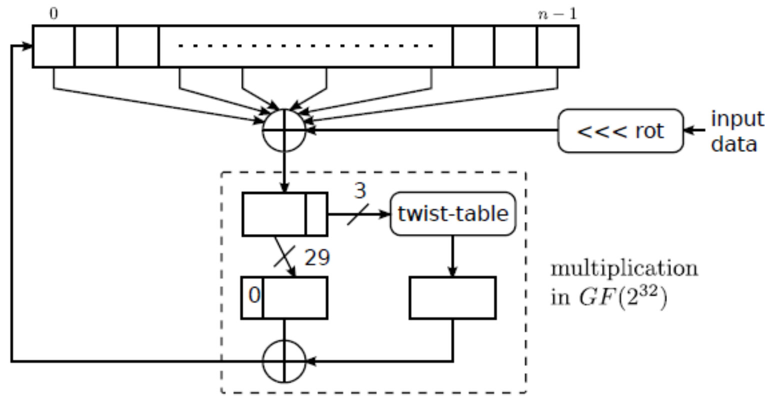
4.4.1. Mixing függvény

Információ beoltás:

4.1. Definíció. *Legyen X n-bites véletlen változó, ami a belső állapotot írja le. Legyen I m-bites véletlen válzotó, ami az entrópia forrásból származó biteket írja le. Legyen $f : \{0{,}1\}^n \times \{0{,}1\}^m \to \{0{,}1\}^n$, és H a Shannon féle entrópia függvény. f mixing függvény akkor és csak akkor, ha teljesül, hogy*

$$H(f(X,I)) \geq H(X) \wedge H((f(X,I)) \geq H(I)$$

A definició nem azt fejezi ki, hogy az új bitek beszúrásánál nem veszik el entrópia, hanem azt fejezi ki, hogy ha a támadó ismeri az entrópia forrásból származó I-biteket, akkor a belső állapotnak a beoltás után nem lesz kevesebb entrópiája, mint



2. ábra. Mixing függvény

előtte volt. Vagy megfordítva, ha a támadó ismeri a belső állapotot, akkor a beoltás után az entrópia forrásból származó bitek entrópiája átvihető a belső állapotra is

Az LRNG által használt mixing függvény először egy bájtot kiterjeszt 32-bites szóvá, majd ezt ciklikusan eltolja, végül egy LFSR-el belekeveri az input poolba. Úgy lett megtervezve, hogy entrópiát tud átadni a pool-nak, és közben nem veszik el entrópia. A későbbiekben azt is megmutatjuk, hogy kielégíti a def-et, így valóban hívhatjuk mixing függvénynek. A LRNG kétszer használja, először amikor az entrópia források outputját oltja bele az input pool-ba, majd pedig amikor az input pool-ból érkező biteket oltja bele az output pool-okba. A következőkben részletesen is bemutatom az f mixing függvény működését. Az output pool az $S_1, ...S_{32}$ 32-bites szavakból áll, most erre fogjuk megnézni (az input pool-ra kisebb változtatásokkal, ehhez nagyon hasonlóan működik f, de ott a pool 128 szóból áll).

1. A byte-ot ami az entrópiát tartalmazza először 32-bites szóvá alakítja, majd d-vel eltolja. Inicializáláskor $d = 0$, és az f mixing függvény minden egyes használatával d nő. Ha $k \equiv 0 \mod 128$, akkor $d \leftarrow d + 14 \mod 32$ és $d \leftarrow d + 7 \mod 32$.

2. A kapott w szóhoz bitenkénti XOR művelettel hozzáadjuk az S pool néhány szavát, egész pontosan S_{k+j} szavakat, ahol $j = \{0,1,7,14,19,26\}$ ($k + j$-t természetesen $\mod 32$ számoljuk).

3. A kapott W szót szétvágja, legyen w_1 az első 29 bit, w_2 az utolsó 3. Ekkor $w \leftarrow w_1 \oplus twist_table(w_2)$, ahol a $twist_table$ előre adott a forráskódban. Legyen $P(X) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$, és α a hozzá tartozó primitív elem $GF(2^{32})$-ben. Ekkor a $twisting_table$ az $\{\alpha^{32j}\}$ elemeket tartalmazza $j = \{0,1, ...7\}$-re.

4. Végül S_k elemet kicseréljük w-re, és k-t 1-gyel növeljük.

Tekintsük a következő lemmát [13].

4.2. Lemma. *Legyen* $X = (X_0, ..., X_{n-1})$*, és* Y *független val változók* $\mathcal{X}^{n-1}$ *és* $\mathcal{Y}$*-on. Továbbá legyen* $L_1 : \mathcal{Y} \to \mathcal{X}$ *injektív és* $L_2 : \mathcal{X}^n \to \mathcal{X}$ *injektív az n-dik változóban. Ekkor* $f(Y, X) = \tilde{X}_0, ..., \tilde{X}_{n-1}$ *mixing függvény, ahol* $\tilde{X}_0 = L_1(Y) \oplus L_2(X)$ *és* $\tilde{X}_i = x_{i-1}$ $i = \{1, ..., n-1\}$.

Bizonyítás. Bevezetjük a következő jelölést: $X_i, ..., X_j$ helyett egyszerűen X_i^j-t írunk.

$$H(f(Y, X)) = H(L_1(Y) \oplus L_2(X), X_0^{n-2}) =$$
$$H(L_1(Y) \oplus L_2(X) | X_0^n - 2) + H(X_0^{n-2})$$

felhasználva, hogy $H(A, B) = H(A|B) + H(B)$ tetszőleges A, B valószínűségi változóra.

4.3. Állítás. *Legyen g injektív, és Z diszkrét val változó. Ekkor* $H(g(Z)) = H(Z)$.

Rögzített $x = (X_0^{n-2}, X_{n-1})$ értékre az $L_1(.) \oplus L_2(x)$ injektív, így használva az előző állítást $H(L_1(Y) \oplus L_2(X)|X) = H(Y|X)$. Mivel X és Y függetlenek, így $H(Y|X) = H(Y)$, és ezért

$$H(L_1(Y) \oplus L_2(X)|X_0^{n-2}, X_{n-1}) = H(Y)$$

Hasonlóan, rögzített $X_0^{n-2} = x_0^{n-2}$ és $Y = y$ értékekre $L_1(y) \oplus L_2(x_0^{n-2}, .)$ injektív, így X és Y függetlensége miatt:

$$H(L_1(Y) \oplus L_2(X)|Y, X_0^{n-2}) = H(X_{n-1}|X_0^{n-2})$$

Tetszőleges Z_1 és Z_2 val változóra $H(Z_1) \geq H(Z_1|Z_2)$, így

$$H(f(Y, X)) \geq H(Y) + H(X_0^{n-2}) \geq H(Y)$$
$$H(f(Y, X)) \geq H(X_{n-1}|X_0^{n-2}) + H(X_0^{n-2}) = H(X)$$

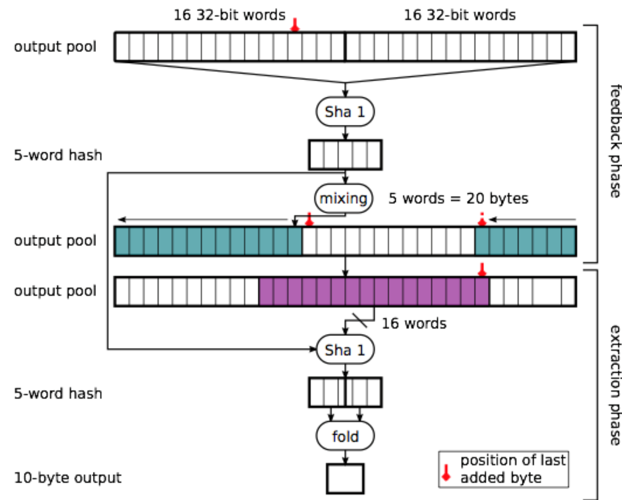
□

Legyen L_1 az algoritmus 1. lépését leíró függvény, L_2 pedig a 2. lépését. Megmutatható, hogy a 3. lépés injektív függvényt definiál, innen pedig következik, hogy a Linux mixing függvénye a definíció szerint is az.

4.4.2. Output függvény

Az output függvény az egyetlen nem-lineáris operáció a Linux PRNG-ben. Az LRNG két helyen használja: Amikor az adatok az input pool-ból az output poolba kerülnek, illetve amikor véletlent generál valamelyik output poolból. Két lépésből áll a folyamat: visszacsatolás fázis, és a kivonási fázis, és végtermékként 10 bájtot állít elő. A használt hash függvény eredetileg SHA-1 volt, később ezt lecserélték, mivel a SHA-1 már nem számít biztonságosnak [17].

Visszacsatolási fázis: Minden bájt belerak egy hash függvénybe, ami 5 szót állít elő (20 bájtot). Ezt a 20 bájtot vissza keveri a poolba a mixing függvény segítségével.



3. ábra. Output függvény

Következésképpen a pool minden egyes visszacsatolási fázisban 20 szóval eltolódik, és a mixing függvény 20 egymás utáni szót változtat meg.

Kivonási fázis: Az előző fázisban előállított 5 szót felhasználja inicializáló értéknek, amikor a pool újabb 16 szavát hash-eli. Ez a 16 bájt átfedi az visszacsatolási fázis által megváltoztatott utolsó szót. A hash outputja 5 szó, ezeket "'félbe hajtja". Ha w_0, w_1, w_2, w_3, w_4 az öt szó, akkor $w_0 \oplus w_3, w_1 \oplus w_4, w_{2,[0...15]} \oplus w_{2,[16...31]}$ 10 bájt lesz az output.

4.4.3. Entrópia becslés LRNG-ben

A becslés külön-külön megy mindegyik entrópia forrásra, ezért elég egy forrásból származó adatokra koncentrálni. A becslő inputja három 32-bites szóból áll, azaz $I = (Jiffy \| get_cycles \| num)$. A *get_cycles* a *cycle count*, a *num* az eseményhez tartozó azonosító, de a becsléshez csak a Jiffy értéket használja, ami az esemény pontos ideje. Legyen $t_0, t_1, ...$ a jiffyk egy sorozata. A becslő kiszámolja az időpillanatok differenciáját három szinten:

$$\delta_i^1 = t_i - t_{i-1}$$
$$\delta_i^2 = \delta_i^1 - \delta_{i-1}^1$$
$$\delta_i^3 = \delta_i^2 - \delta_{i-1}^2$$

Ezután a becslő kiszámolja a differenciák minimumát:

$$\Delta_i = \min(\delta_i^1, \delta_i^2, \delta_i^3)$$

Végül a következő függvényt alkalmazza:

$$H_i = \begin{cases} 0 \text{ ha } \Delta_i < 2 \\ 11 \text{ ha } \Delta_i \geq 2^12 \\ \lfloor log_2(\Delta_i) \rfloor \text{ egyébként} \end{cases}$$

4.5. Fortuna

A Fortuna Niels Ferguson és Bruce Schneier által tervezett CSPRNG, egy korábbi CSPRNG, a Yarrow továbbfejlesztése [15]. A Fortuna az LRNG-vel és a Yarrow-val ellentétben nem használ entrópia becslőket. Három része van:

– Generátor: Fix hosszú seed-ből generál tetszőlegesen hosszú adatot

– Accumulator: Összegyűjti, majd szétosztja a entrópia források output-ját, és alkalmanként frissíti a generátor seed-jét.

– Seed fájl control: Biztosítja, hogy a PRNG számítógép újraindításkor is tud véletlent előállítani.

Most ezeket szeretném részletesebben bemutatni [16] alapján.

4.5.1. Generátor

A generátor állítja elő a PRNG által kibocsátott véletleneket. Lényegében egy blokk titkosító counter módban kisebb finomításokkal. Egy AES-tipusú blokk titkosítót használ. A belső állapot (internal state) 256-bites blokk titkosító kulcsból áll, és 128-bites counterből.

Inicializálás:

1. függvény: InitializeGenerator, output:G
2. $(K,C) \leftarrow (0{,}0)$
3. $G \leftarrow (K,C)$
4. return G

A G a generátor állapotát írja le. A függvény pedig a (K,C) (kulcs,számláló) párt (0,0)-ra állítja.

Reseed:

1. függvény: *reseed*, input: s
2. Kiszámolja az új kulcsértéket: $K = SHA - 256(K\|s)$
3. Növeli a számlálót $C = C + 1$.

Blokkgenerálás:

1. függvény:GenerateBlocks, input:k, output: r
2. Ellenőrzi, hogy $C \neq 0$

3. $r \leftarrow \epsilon$, azaz r üres szó lesz.

4. for $i = 1, ..., k$ $\{r \leftarrow r \| E(K, C)$ és $C \leftarrow C + 1\}$

k az előállítandó blokkok száma, az algoritmus az r $16k$ bájtos pszeudovéletlen sorozatot állítja elő. Az $E(K, C)$ függvény pedig egy blokk titkosító K kulccsal és C szöveggel.

Véletlen adat előállítása

1. függvény: PseudoRandomData, input: n, output: r

2. Ellenőrzi $0 \leq n \leq 2^{20}$

3. $r \leftarrow first - n - bytes(GenerateBlocks(\lceil n/16 \rceil))$

4. $K \leftarrow GenerateBlocks(2)$

n bájtból álló r pszeudovéletlent állítja elő az algoritmus. Biztonsági okokból a PRNG egy lépésben legfeljebb 2^{20} bájtot állít elő, hogy elkerülje az ismétlődéseket. Ha ennél többre van szükség, akkor az algoritmus iterálásával ez is könnyen elérhető. Az utolsó lépésben pedig lecseréli a kulcsot, ezzel biztosítva, hogy a generátor minden információt elfelejt a korábban generált sorozatról, ezáltal lehetetlenné téve a a korábban generált pszeudo-véletlenek visszafejtését.

4.5.2. Accumulator

Valódi véletleneket gyűjt különböző forrásokból, és ezeket felhasználva újratölti a generátort.

4.5.3. Entrópia források

Gyakorlatilag bármi használható, ami állít elő valódi véletleneket. A források meg vannak számozva 0-tól 255-ig. Egy-egy eseményt legalább 3 bájt ír le - az első bájt a forrás számát tartalmazza, a második bájt a további bájtok számát, a következő bájtok pedig azt, amit a forrás előállított.

4.5.4. Pool

A generátor újratöltéséhez pool-okat használ, ezekenek elég nagyoknak kell lenni, hogy a támadó számára megtippelhetetlen legyen a tartalmuk. Az LRNG megpróbálja megbecsülni, hogy a pool elég entrópiát tartalmaz-e az újratöltéshez. Ezzel szemben a Fortuna egyáltalán nem használ entrópia becslőt.

Összesen 32 poolt tartalmaz: $P_0, ... P_{31}$. Elméletben mindegyik pool egy korlátlan hosszú bájt sorozatot tartalmaz. A gyakorlatban ez nem lenne túl praktikus. Ezért mindegyik pool fix hosszúságú, és ha ez megtelt, akkor egy hash-függvényt alkalmaz rá. Mindegyik forrás ciklikusan osztja el az eseményeit a pool-ok között. Ez biztosítja, hogy mindegyik pool többé-kevésbé egyenlően részesül a forrás entrópiájából.

Inicializálás.

1. függvény: InitializePRNG, output: R
2. for $i = 0, ..., 31$ $\{P_i \leftarrow \epsilon\}$
3. $ReseedCnt \leftarrow 0$
4. $G \leftarrow InitializeGenerator()$
5. $R \leftarrow (G, ReseedCnt, P_0, ..., P_31$

Kezdetben mindegyik pool üres. A ReseedCnt az újratöltéseket számolja, ezt is 0-ra állítja. A PRNG aktuális állapotát R írja le, amit a G generátor állapot, $ReseedCnt$ és a Pool-ok alkotnak.

Véletlen generálás

1. függvény: RandomData, input: R, n output: r
2. If $(length(P_0) \geq MinPoolSize) \wedge (lastreseed > 100ms$ ezelőtt$)$
   a) $ReseedCnt \leftarrow ReseedCnt + 1$
   b) $s \leftarrow \epsilon$
   c) for $i = 0, ...31$
      α) If $2^i | ReseedCnt$ $\{s \leftarrow s \| SHA - 256(P_i)$ és $P_i \leftarrow \epsilon\}$
   d) $Reseed(G, s)$
3. If ReseedCnt=0, akkor ERROR, PRNG még nincs feltöltve
4. Else return $PseudoRandomData(G, n)$

R a PRNG állapota, n a generálandó bájtok száma, r a generált bájtok. Akkor következik be a generátor újratöltése, ha P_0 elég hosszú (általában 64 bájt), és az előző újratöltés óta eltelt elég idő (100 ms). Az újratöltéshez az i-dik pool-t pontosan akkor használja fel, ha 2^i osztja a $ReseedCnt$-t. Tehát P_0-t minden újratöltésnél felhasználja, P_1-et minden másodiknál, P_2-t minden negyediknél, és így tovább. Az újratöltéshez a megfelelő pool-ok tartalmát hasheli, majd ezeket egymásután írja, és a kapott sorozatot használja fel az újratöltéshez. A pool-okat pedig, miután tartalmukat felhasználta, kiüríti.

Ha van legalább egy forrás, ami nincs a támadó hatása alatt, akkor a pool-ok entrópiája felhasználásuk pillanatában a sorszámuk exponenciális függvénye, azaz P_1 kétszer, P_2 négyszer, P_3 nyolcszor annyi entrópiát tartalmaz a felhasználásakor, mint P_0. Ezért lesz olyan pool, ami elég entrópiát gyűjtött már össze, hogy ha ezzel poollal töltjük újra a generátort, akkor az teljes biztonságot fog adni. Ezért a rendszer még akkor is vissza tudja nyerni a biztonságát, ha a támadó megtudja a belső állapotot, és a források egy része is az irányítása alatt van.

Esemény hozzáadása

1. függvény: AddRandomEvent input: R, s, i, e
2. Ellenőrzi $(1 \leq length(e) \leq 32) \wedge (0 \leq s \leq 255) \wedge (\leq i \leq 31)$
3. $P_i \leftarrow P_i \| s \| length(e) \| e$

s a forrás sorszáma, i a pool, amibe az accumuator belerakja az adatot, és e maga a forrás által előállított adat. Az Accumulator nem enged egy forrásból egyszerre túl sok adatot, mert azoknak úgy sincs elég információtartalma. Ha a forrás és a pool sorszáma is a megfelelő tartományba esik, akkor a pool tartalma mögé írja az $(s, length(e), e)$ hármast.

4.5.5. Seed fájl kezelés

A PRNG biztonságosan működik az első újratöltés után. Újraindítás után azonban várni kell, hogy elég véletlen adat összegyűljön, ami nem praktikus. Ezért a Fortuna egy seed fájlt használ, ami nem érhető el senki más számára sem. Újraindítás után a PRNG ezt a seed fájlt használja fel, hogy ismeretlen állapotba kerüljön.

Seed fájlba írás

1. függvény: WriteSeedFile, input: R, f
2. $write(f, RandomData(R,64)$

R PRNG állapota, f fájl amibe ír. Generál 64 bájtot, és beleírja az f fájlba.

Seed fájl frissítés

1. függvény: UpdateSeedFile, input: R, f
2. $s \leftarrow read(f)$
3. ellenőrzi, hogy $length(s) = 64$
4. $Reseed(G, s)$
5. $write(f, RandomData(R,64))$

R, f, mint előbb. Beolvas az f fájlból, ellenőrzi a hosszát, majd újratölti a generátort. Végül pedig átírja a seed fájlt új véletlen adattal. Fontos, hogy az újratöltés és a fájlba írás azonnal kövesse egymást, mert ha, és közben a számítógép kikapcsol, akkor a következő újraindításkor ugyanazt a seed fájlt fogja felhasználni, mint előbb. Továbbá amikor a számítógép a bekapcsolás után elég entrópiát gyűjtött, érdemes bizonyos időközönként újra írni a seed fájlt.

5. Véletlenek tesztelése

Az előző fejezetekben már láttuk a fontosságát, hogy valódi véletleneket előállító véletlen szám generátort alkalmazzunk, legyen itt szó TRNG-ről, vagy PRNG-ről. Kérdés azonban, hogyan tudjuk eldönteni egy forrásról (ez lehet TRNG, PRNG, vagy bármi más), hogy megfelelő véletleneket állít elő számunkra. Az első módszer, hogy statisztikai teszteket alkalmazunk a forrás által előállított sorozatokra. Ezek a tesztek megvizsgálják, hogy a generált sorozat eléggé véletlen, azaz hogy rendelkezik-e az egyelentes eloszlás szerinti véletlen bit-sorozat tulajdonságaival. A másik módszer, hogy megmutatjuk, hogy a forrása rendelkezik megfelelő mennyiségű entrópiával. Általában a forrás eloszlása nem ismert, így az előállított bitsorozat alapján próbálhatunk becslést adni a forrás entrópiájára.

5.1. Statisztikai tesztek

Azt mérik, hogy a vizsgált sorozat mennyire teljesíti azokat a kritériumokat, amiket egy egyenletes eloszlású véletlen sorozattól elvárunk. A gyakorlatban rengeteg ilyen teszt létezik, a sorozat végtelen sok tulajdonságát mérhetnénk. A hatékonyabb tesztelés miatt a tesztsorozatokat szoktak alkalmazni, minr például NIST Statistical Test Suite [18], Dieharder [19], TestU01 [20], de a BSI is rendelkezik saját statisztikai teszttel [21]. A NIST STS talán a legfontosabb, ugyanis ez NIST szabvány és használja az AES és különböző hitelesítések is. A NIST STS 15 különböző tesztet tartalmaz, ezek a következők (a tesztek mellé röviden odaírtam, hogy az adott teszt milyen elvárt tulajdonságot ellenőriz):

1. Frequency (Monobit) Test: A 0-k és 1-esek száma a teljes sorozatban nagyjából ugyanannyi

2. Frequency Test within a Block: 0-k és 1-k száma M nagyságú blokkokon belül is kb ugyanannyi

3. The Run Test: Milyen hosszú egymást követő csupa 0, csupa 1 részek vannak, azaz milyen gyorsan váltakoznak a 0-k és 1-k.

4. Test for the Longest-Run-of-Ones in a Block: A leghosszabb egymást követő csupa 1-esekből álló rész nem lehet sem túl hosszú, sem túl rövid

5. Binary Matrix Rank Test: Lineáris összefüggéseket keres fix hosszú részek között

6. Discrete Fourirer Transform Test: Periódikusságot vizsgálja

7. Non-Overlapping Template Matching Test: Egy m hosszú, nem periódikus M minta nem fordulhat elő túl sokszor sorozatban. Egy ablak halad végig a sorozaton, és keresi az M mintát. Itt a minták nem átfedőek, azaz ha talált egyet, akkor az ablak a talált minta végétől kezdi újra a keresést.

8. Overlapping Template Matching Test: Ugyanaz, mint előbb, csak átfedéses. Azaz találat esetén csak 1-gyel jobbra csúszik az ablak

9. Maurer's Universal Test: Két azonos minta közti távolságot méri, azaz lehet-e szignifikánsan tömöríteni a sorozatot

10. Linear Complexity Test: Egy LFSR hosszára koncentrál. Ha a sorozat elég összetett (ahogyan egy ténylegesen véletlen sorozattól elvárnánk), akkor hosszú LFSR-rel írhat le.

11. Serial Test: Az m hosszú minták átfedéses előfordulására koncentrál. Ha tényleg véletlen a sorozat, akkor 2^m féle sorozat gyakorisága nagyjából ugyanannyi

12. Approximate Entropy Test: Ez is m hosszú minták átfedéses előfordulását. Itt szomszédos hosszúságú (m, $m+1$) minták gyakoriságát hasonlítja össze

13. Cumulative Sums Test: Vegyük azt a sétát, amit úgy kapunk, hogy a 0-ból indulva a sorozatbeli 0-k és 1-esek hatására -1-et és 1-et lépünk. Valódi véletlen sorozat esetén nem távolodhatunk el túlságosan a 0-tól.

14. Random Excursion Test: Véletlen séta, mint előbb. A K hosszú körök száma, ahol a kör olyan séta, ami 0-ból indul és 0-ba érkezik.

15. Random Excursion Variant Test: Véletlen séta, mint előbb. Egy helyet hányszor látogatunk meg a séta folyamán.

Gyakran tapasztalható, hogy egy tökéletes véletlent generáló forrás által előállított sorozat is rendelkezik kisebb-nagyobb szabályosságokkal. Habár a teszteket több különböző, a forrás által előállított sorozatokra végzik el, mégis előfordulhat, hogy egy túl szigorú teszt kizár egy valódi véletlent generáló forrást. [22] szerint egy valódi véletlen sorozat is nagy eséllyel(80%) megbukik a NIST STS valamelyik tesztjén a standard által ajánlott paraméterekkel.

5.2. Entrópia becslés

Az entrópia becslésével igen széles szakirodalom foglalkozik. Egyrészt fontos szerepe van véletlenek tesztelésében. Egyes tesztek, mint például a BSI által ajánlott véletlen generátor teszt [21], alkalmaznak entrópia becslést a véletlen generátor által előállított sorozatra, ezzel vizsgálva a véletlen generátort. De a PRNG töltésére szolgáló entrópia forrásokra is érdemes entrópia becslést alkalmazni [35].

Most egy X forrás entrópiáját szeretnénk meghatározni. Ha X eloszlása ismert, akkor $H(X)$ a definíció alapján egyszerűen számolható. Az általunk használt entrópia források eloszlása azonban ismeretlen, és nehezen meghatározható, ezért csak a forrásból származó sorozat alapján próbálunk becslést adni a forrás entrópiájára. Sok különböző módszer van, ezek közül szeretnék néhányat bemutatni.

5.3. Plug-in vagy Maximum likelihood becslés

Ez a legkézenfekvőbb az összes becslés közül. Adott az x_1^n n hosszú sorozat, és legyen $y_1^m \in A^m$ tetszőleges $m < n$ hosszú minta. Jelölje $\tilde{p}_m(y_1^m)$ az y_1^m tapasztalati valószínűségét x_1^n-ben, azaz $\tilde{p}_m(y_1^m) = s(y_1^m)/n$, ahol $s(y_1^m)$ az y_1^m előfordulásának száma x_1^n-ben. A tapasztalati valószínűségek által meghatározott eloszlás $\tilde{p}_m$. A plug-in becslést a következő képpen definiáljuk:

$$\tilde{H}_{n,m,plug-in} = \frac{1}{m}H(\tilde{p}_m) = -\frac{1}{m}\sum_{y_1^m \in A^m} \tilde{p}_m log \tilde{p}_m(y_1^m)$$

Ha a folyamat, ami x_1^n-et előállítja stacionárius és ergodikus, akkor a nagy számok törvénye miatt $\tilde{p}_m$ közel lesz p_m-hez, ahol p_m az m hosszú részsorozatok valódi eloszlása. Ha m-et elég nagyra választjuk, hogy $\frac{1}{m}H(X_1^m)$ közel legyen H-hoz, feltéve, hogy n elég nagy ahhoz, hogy az m rendű tapasztalati eloszlás közel van a valódi eloszláshoz, akkor a $\tilde{H}_{n,m,plug-in}$ plug-in becslés pontos értéket fog adni a bitenként entrópiára. Pontosabban igaz az alábbi tétel [23]:

5.1. Tétel. *A plug-in becslés erősen univerzálisan konzisztens, azaz $n \to \infty$ esetén*

$$\tilde{H}_{n,m,plug-in} \to H \text{ 1 valószínűséggel}$$

Egy becsléstől azt is elvárnánk, hogy ne becsülje túl az entrópiát. A tétel mellett $E(\tilde{H}_{n,m,plug-in}) \geq H$ is belátták, úgyhogy ez is teljesül.

A gyakorlati alkalmazás esetén az egyik jelentős probléma, hogy ha m nagy, azaz pl $m = 30$, akkor már bináris esetben is 2^{30} féle szó létezik, így lehetetlen megadni a $\tilde{p}_m$ tapasztalati eloszlást. Egy másik probléma, hogy lehetetlen számszerűsíteni a torzítást.

5.4. LZ becslés

Intuitívan egy sorozat információ tartalma ugyanannyi, mint a legrövidebb sorozaté, amivé össze lehet tömöríteni. Innen adódik, hogy a veszteség mentes tömörítések jó becslést adhatnak az entrópiára. Ráadásul a Lempel-Ziv algoritmusra teljesül a következő tétel [24]:

5.2. Tétel. *A Lempel-Ziv tömörítő algoritmust Markov forrásból származó adatokra használva az egy szimbólum kódolásához szükséges bitszám tart az entrópiához.*

Mivel az entrópia becslés egyszerűbb, mint maga a tömörítés, gyakorlatban az eredeti tömörítő algoritmusok módosított változatait használják. Ezek mind ismétlődő minták hosszúságának számolásán alapulnak.

Tekintsük az $x = (..., x_{-1}, x_0, x_1, ...)$ mindkét irányba végtelen sorozatot. x_i^j jelöli az $x_i, ..., x_j$ részsorozatot. Minden i pozicióra és $n \geq 1$ ablak hosszra keressük meg azt leghosszabb mintát, (a hosszát jelölje λ), ami i-ben kezdődik, és előfordul x_{i-n}^{i+n-1}-ben is. Ekkor $L_i^n = L_i^n(x) = \lambda + 1$. Formálisan:

5.3. Definíció.

$$L_i^n = 1 + \max\{0 \leq \lambda \leq n : x_i^{i+\lambda-1} = x_j^{j+\lambda-1} \textit{ valamilyen } i-n \leq j \leq i-1\} \quad (3)$$

Bevezetünk egy további fogalmat is, ez azt fogja jelenteni, hogy meddig kell visszamenni a múltba, hogy egy adott sorozatot megtaláljunk.

5.4. Definíció.

$$R_n = R_n(x_1^n) = \min\{0 < r : x_{1-r}^{n-r} = x_1^n\}$$

A következő tétel összegfüggést mutat R_n, L_i^n aszimptotikus viselkedése és az entrópia között [25].

5.5. Tétel. *Legyen X stacionárius és ergodikus folyamat. Ekkor*

$$\frac{\log R_n}{n} \to H \textit{ ha } n \to \infty \textit{ 1 valószínűséggel} \quad (4)$$

$$\frac{L_i^n}{\log n} \to \frac{1}{H} \textit{ ha } n \to \infty \textit{ 1 valószínűséggel,} \quad (5)$$

ahol H a bitenkénti entrópiát jelöli.

Bizonyítás. A 4 bizonyításához [26] gondolatmenetét használtam.

5.6. Lemma. *Legyen X véges értékű, stacionárius, ergodikus folyamat, és $c(n)$ tetszőleges nem-negatív számokból álló sorozat, amelyre $\sum_{n=1}^{\infty} n2^{c(n)} < \infty$. Ekkor*

$$(i)\ \log(R_n \cdot \Pr(X_1^n)) \leq c(n) \textit{ 1 valószínűséggel}$$

$$(ii)\ \log(R_n \cdot \Pr(X_1^n)|X_{-\infty}^0) \leq c(n) \textit{ 1 valószínűséggel}$$

5.7. Lemma. *X véges értékű, stacionárius, ergodikus folyamat. Ekkor*

$$\log(R_n \cdot \Pr(X_1^n)) = o(n) \textit{ 1 valószínűséggel}$$

Mármost a 5.7 szerint

$$\frac{\log(R_n)}{n} - \frac{\log \Pr(X_1^n)}{n} \to 0 \text{ 1 valószínűséggel}$$

Mivel stacionárius, ergodikus folyamatról van szó, a Shannon-McMillan tétel miatt

$$\frac{\log \Pr(X_1^n)}{n} \to H \text{ 1 valószínűséggel,}$$

úgyhogy

$$\frac{\log(R_n)}{n} \to H \text{ 1 valószínűséggel.}$$

Most belátjuk 4-ből 5-t. Vegyünk tetszőleges x realizációt, és $n \geq 1$-re legyen $m = L_1^n$. Ekkor R_m definicója miatt $R_m > n$ és így

$$\frac{\log R_m}{m} > \frac{\log n}{L_1^n}$$

Ha n-nel tartunk végtelenhez, kapjuk, hogy

$$\limsup_{n\to\infty} \frac{\log n}{L_1^n} < \lim_{m\to\infty} \frac{\log R_m}{m} = H$$

A megfordításhoz legyen n tetszőleges, és $m = L_1^n - 1$. Ekkor $R_m \leq n$ teljesül, innen

$$\frac{\log R_m}{m} \leq \frac{\log n}{L_1^n - 1}$$

Ha ismét $n \to \infty$, akkor

$$\liminf_{n\to\infty} \frac{\log n}{L_1^n} = \liminf_{n\to\infty} \frac{\log n}{L_1^n - 1} \geq \limsup_{m\to\infty} \frac{\log R_m}{m} = H$$

, innen pedig adódik 5 $\square$

A 5.5 tételből következik, hogy

$$\frac{\log n}{L_i^n}$$

felhasználható az entrópia becslésére, és hogy csökkentsük a szórást, érdemes L_i^n értéket kiátlagolni különböző i pozíciókra. Ez alapján [27] a következő két becslést javasolta, amikről belátta, hogy egy bizonyos feltételek mellett konzisztensek.

$$\hat{H}_{n,k} = \left[\frac{1}{k}\sum_{i=1}^{k} \frac{L_i^n}{\log n}\right]^{-1} \tag{6}$$

$$\hat{H}_n = \left[\frac{1}{n}\sum_{i=2}^{n} \frac{L_i^i}{\log i}\right]^{-1} \tag{7}$$

Később [28] további két, ugyanezen az elven működő becslést javasolt, de itt a konzisztenciához kevesebb feltétel is elég.

$$\tilde{H}_{n,k} = \frac{1}{k}\sum_{i=1}^{k} \frac{\log n}{L_i^n} \tag{8}$$

$$\tilde{H}_n = \frac{1}{n}\sum_{i=2}^{k} \frac{\log i}{L_i^i} \tag{9}$$

Az 6 és 8 sliding-window LZ estimator, a 7 és 9 increasing-window LZ estimator. A különbség a kettő között, ahogy a nevük is elárulja, hogy $\hat{H}_{n,k}$, és $\tilde{H}_{n,k}$ esetén fix hosszú ablak mozog jobbra, míg a $\hat{H}_n$ és $\tilde{H}_n$ a növekedő ablak méretével felhasználja a teljes előzményt.

A következő tétel megmutatja, hogy valóban jól használhatók az entrópia becslésére. A bizonyítás [28] gondolatmenetét használja.

5.8. Tétel. *(i) Tetszőleges adatsorozatra alkalmazva a fenti becslőket, minden k, n-re*

$$\hat{H}_{n,k} \leq \tilde{H}_{n,k} \text{ és } \hat{H}_n \leq \tilde{H} \leq \hat{H}_n$$

(ii) A $\hat{H}_{n,k}$ és $\hat{H}_n$ becslők konzisztensek, ha véges értékű stacionárius, ergodikus folyamat által generált adatokra alkalmazzuk, amikre teljesül még a Doeblin kritérium (DC). Tehát 1 valószínűséggel:

$$\hat{H}_{n,k} \to H, \hat{H}_n \to \ H, \text{ ha } k, n \to \infty$$

(iii) A $\tilde{H}_{n,k}$ és $\tilde{H}_n$ becslők konzisztensek, ha véges értékű stacionárius, ergodikus folyamat által generált adatokra alkalmazzuk. Tehát 1 valószínűséggel:

$$\tilde{H}_{n,k} \to H, \tilde{H}_n \to \ H, \text{ ha } k, n \to \infty$$

5.9. Definíció. *Doeblin Feltétel (DC): Létezik olyan $r \geq 1$ egész és $\beta \in (0,1)$ valós szám, hogy minden $x_0 \in A$-ra*

$$Pr(X_0 = x_0 | X_{-\infty}^r) > \beta \text{ 1 valószínűséggel}$$

A DC teljesül memória nélküli forrásokra, tetszőleges rendű ergodikus Markov láncra, és más nem Markov folyamatokra is, tehát feltételezhető, hogy a vizsgált folyamatra is teljesül.

Bizonyítás. (i) Alkalmazzuk a Jensen-egyenlőtlenséget az $x \mapsto 1/x$ konvex függvényre.

$$\tilde{H}_{n,k} = \sum_{i=1}^{k} \frac{1}{k} \frac{\log n}{L_i^n} = \sum_{i=1}^{k} \left[\frac{L_i^n}{\log n} \right]^{-1} \geq \left[\sum \frac{1}{k} \frac{L_i^n}{\log n} \right]^{-1} = \hat{H}_{n,k}$$

. A másik állítás hasonlóan adódik.

(ii) bizonyítása [27] alapján megy, először belátom az alábbi lemmát.

5.10. Lemma. *Legyen $\{X_i\}$ stacionárius folyamat. Ha DC teljesül, akkor*

$$(i) P(L_1^n > k) \leq n\beta^{\lfloor k/r \rfloor} \; k \geq 1$$
$$(ii) E\left(\frac{L_1^n}{\log n} < \infty \right) \text{ azaz } \frac{L_1^n}{\log n} \; L^1\text{-dominált}$$

Bizonyítás(lemma). Ha $L_1^n > k$, akkor L_1^n definiciója miatt X_0^{k-1} megjelenik X_{-n}^1-ben, tehát valamely $k \leq j \leq n$ -re $X_0^{k-1} = X_{-j}^{-j+k+1}$ teljeseül. Így

$$\Pr(L_1^n > k) \leq \sum_{k \leq j \leq n} \Pr(X_0^{k-1} = X_{-j}^{-j+k+1}) \leq (n-k+1) \max_{k \leq j \leq n} \Pr(X_0^{k-1} = X_{-j}^{-j+k+1})$$

A teljes valószínűség tétel szerint

$$\Pr(X_0^{k-1} = X_{-j}^{-j+k+1}) = \\ = \sum_{x^k \in A^k} \Pr(X_0^{k-1} = x_0^{k-1} | X_{-j}^{-j+k+1} = x_0^{k-1}) \Pr(X_0^{k-1} = x_0^{k-1} | X_{-j}^{-j+k+1} = x_0^{k-1}) \tag{10}$$

A jobb oldalon lévő feltételes valószínűséget tovább alakítva kapjuk

$$\begin{aligned}
\Pr(X_0^{k-1} =&x_0^{k-1}|X_{-j}^{-j+k+1} = x_0^{k-1}) \\
&\le \Pr\left(X_0^{rt-1} = x_0^{rt-1}, 0 \le t \le \left\lfloor \frac{k}{r} \right\rfloor |X_{-j}^{-j+k+1} = x_0^{k-1}\right) \\
&\le \prod_{t=0}^{\lfloor \frac{k}{r} \rfloor} \Pr\left(X_0^{rt-1} = x_0^{rt-1}|X_{-j}^{-j+k+1} = x_0^{k-1}, X_0^{rs-1} = x_0^{rs-1}, 0 \le s < t\right)
\end{aligned}$$

A DC feltétel miatt a szorzat minden tényezője felülbecsülhető β-val, így

$$\Pr(X_0^{k-1} = x_0^{k-1}|X_{-j}^{-j+k+1} = x_0^{k-1}) \le \beta^{\lfloor \frac{k}{r} \rfloor}.$$

Ezt visszahelyettesítve 10-be kapjuk, hogy $\Pr(X_0^{k-1} = X_{-j}^{-j+k+1}) \le \beta^{\lfloor \frac{k}{r} \rfloor}$, innen pedig már világos, hogy $P(L_1^n > k) \le n\beta^{\lfloor \frac{k}{r} \rfloor}$.

Most megmutatjuk, hogy L_1^n/n L^1-dominált. Legyen $\gamma = (-\log \beta)/2r$. Vegyük észre, hogy $k \ge 4r$ esetén

$$\left\lfloor \frac{\lfloor k \log n \rfloor}{r} \right\rfloor \ge \frac{k \log n}{r} - 1 - \frac{1}{r} \ge \frac{k \log n}{r} - \frac{k \log n}{2r} \ge \frac{k \log n}{2r} \tag{11}$$

$$E\left(\sup_{n \ge 2} \frac{L_1^n}{\log n}\right) = \int_0^\infty \Pr\left(\sup_{n \ge 2} \frac{L_1^n}{\log n} > k\right) dk \le K + \int_K^\infty \sum_{n \ge 2} \Pr(L_1^n > k \log n) dk,$$

ahol K tetszőkleges konstans. Felhasználva a lemma első részét, majd 11-t, kapjuk

$$\begin{aligned}
E\left(\sup_{n \ge 2} \frac{L_1^n}{\log n}\right) &\le K + \sum_{n \ge 2} \int_K^\infty n\beta^{\lfloor \frac{\lfloor k \log n \rfloor}{r} \rfloor} dk \\
&\le K + \sum_{n \ge 2} \int_K^\infty n\beta^{\frac{k \log n}{2r}} dk \\
&= K + \sum_{n \ge 2} n^{1-\gamma k} dk \\
&= K + \sum_{n \ge 2} \frac{n^{1-\gamma K}}{\gamma \ln n}
\end{aligned}$$

Ha K-t úgy választjuk meg, hogy $K > 2/\gamma$, akkor a végtelen összeg konvergens. □

(ii) bizonyításához szükség lesz még Maker általánosított ergodikus tételére [29], [27]

5.11. Tétel. *(Maker) Legyen T mértéktartó transzformáció az (X, B, P) valószínűségi mezőn és jelölje I az invariáns események σ-algebráját. Legyen $g_{n,i}$ két dimenziós függvénye valós értékű valószínűségi változóknak. Ekkor*

1. *Ha $E(\inf_{n,i} g_{n,i}) > -\infty$ és $g = \liminf_{n,i} \to g_{n,i}$, akkor*

$$\liminf_{n,i \to \infty} \frac{1}{n} \sum_{1 \le i \le n} g_{n,i}(T^i x) \ge E(g|I) \text{ 1 valószínűséggel}$$

2. *Ha* $\sup_{n,i} |g_{n,i}|$ *integrálható, és* $g_{n,i} \to g$ *1 valószínűséggel, ahogy* $n, i \to \infty$*, akkor*

$$\frac{1}{n} \sum_{1 \le i \le n} g_{n,i}(T^i x) \to E(g|I) \text{ 1 valószínűséggel és } L^1\text{-ben}$$

Az $X = A^{\mathbb{Z}}$ halmaz Borel σ-algebráján tekinthetjük a folyamat eloszlása által definiált valószínűségi mértéket. Ha stacionárius a folyamat, akkor a $(A^{\mathbb{Z}}, B, \Pr)$ valószínűségi mezőn a $Tx_i = x_{i+1}$ $\forall i \in \mathbb{Z}$ eltolás mértéktartó transzformáció.

Alkalmazzuk Maker tételét $g_{n,i} = \frac{L_1^n}{\log n}$ és $g_{n,i} = \frac{L_1^i}{\log i}$ értékekre. A lemma miatt $\sup_{n,i} |g_{n,i}| < \infty$ teljesül és a 5.5 tétel miatt $g_{n,i} \to 1/H$ majdnem biztosan mindkét esetben, ezzel beláttuk (ii)-t.

A (iii) bizonyítása hasonlóan végig vihető, mint (ii), a különbség, hogy az L^1 domináltsághoz nem szükséges DC, hanem a (iii) feltételeiből azonnal belátható. Tehát

5.12. Lemma. *Stacionárius, ergodikus folyamatra a fent definiált* L_1^n*,* R_m *függvényekre teljesül*

$$E\left\{\sup_{n \ge 1} \frac{\log n}{L_i^n}\right\} < \infty \tag{12}$$

$$E\left\{\sup_{m} \frac{\log R_m}{m}\right\} < \infty \tag{13}$$

Bizonyítás (lemma). Tetszőleges x realizációra és $n \ge 1$-re, ha $m = L_i^n$, akkor R_m és L_1^n definiciója szerint $R_m > n$, amiből azonnal adódik

$$\frac{\log R_m}{m} > \frac{\log n}{L_1^n},$$

Mindkét oldalt a szuprémumot véve fennáll

$$\sup_m \frac{\log R_m}{m} \ge \sup_{n \ge 1} \frac{\log n}{L_i^n}.$$

Tehát elég belátni 13-t, mert innen azonnal adódik 12 is. A diszkrét várható érték kiszámítása miatt.

$$E\left\{\sup_{n \ge 1} \frac{\log R_m}{m}\right\} = \sum_{k \ge 0} \Pr\left\{\sup_{n \ge 1} \frac{\log R_m}{m} \ge k\right\}$$

Ha $k < K$, akkor a valószínűség felülbecsülhető 1-gyel, K értékét majd később választjuk meg.

$$\sum_{k \ge 0} \Pr\left\{\sup_{n \ge 1} \frac{\log R_m}{m} \ge k\right\} \le K + \sum_{k \ge K} \Pr\left\{\sup_{n \ge 1} \frac{\log R_m}{m}\right\}$$

A $\Pr\{\sup_m \frac{\log R_m}{m} \ge k\}$ értéket nyilvánvalóan felülbecsüli $\sum_{m \ge 1} \Pr\{\frac{\log R_m}{m} \ge k\}$, mivel az előbbi tag az összegben, a többi tag pedig nem negatív, így adódik

$$K + \sum_{k \geq K} \Pr\left\{\sup_{n \geq 1} \frac{\log R_m}{m}\right\} \leq K + \sum_{k \geq K} \sum_{m \geq 1} \Pr\left\{\frac{\log R_m}{m}\right\}$$

Markov egyenlőtlenséget használva $\Pr\left\{\frac{\log R_m}{m}\right\} \leq E(R_m) \cdot 2^{-mk}$. Ha az X folyamat lehetséges értékeinek száma $\alpha = |A|$, akkor összesen α^m m hosszú x_1^m szó van. Kac tétele szerint $E(R_m | X_1^m = x_1^m) = 1/\Pr(X_1^m = x_1^m)$ [30] [25]. Innen, és a teljes várható érték tételéből azonnal adódik

$$E(R_m) = \sum_{x_1^m} E(R_m | X_1^m = x_1^m) \cdot \Pr(X_1^m = x_1^m) = \alpha^m$$

Az előzőeket összegezve, majd a geometria sor összegképletét alkalmazva

$$\begin{aligned} E\left\{\sup_{n \geq 1} \frac{\log R_m}{m}\right\} &\leq K + \sum_{k \geq K} \sum_{m \geq 1} 2^{-m(k - \log \alpha)} \\ &= K + \sum_{k \geq K} \frac{2^{-(k - \log \alpha)}}{1 - 2^{-(k - \log \alpha)}} \\ &= K + \sum_{k \geq K} \frac{1}{2^k/\alpha - 1} < \infty \end{aligned}$$

adódik, ahol $K > \log \alpha$ a választás.

□

5.12 lemma és Maker tételének segítségével 5.11 (iii) ugyanúgy befejezhető, mint (ii). Ezzel beláttuk a tételt.

□

5.5. PV becslő

Egy igen egyszerű becslés bináris sorozat esetén a következő, ami [31] cikkben jelenik meg, ami a bit-váltásokat számolja össze. Most tegyük fel, hogy a bitek egymástól függetlenül, azonos eloszlás szerint állnak elő. Ekkor egy bit p valószínűséggel lesz 0, $1 - p$ valószínűséggel 1. A bit-váltás valószínűsége új bit érkezésekor $p(1-p) + (1-p)p = 2p(1-p)$. Tehát n hosszú sorozat esetén a bit váltások várható értéke $2(n-1)p(1-p)$, és a következő becslés alkalmazható.

$$\begin{aligned} 2(n-1)p(1-p) &\leq -np(\ln 2 \cdot \log p) + n(1-p)(-\ln 2 \cdot \log p) < \\ & n(-p \log p - (1-p)\log(1-p)) = nH(p) \end{aligned}$$

A becslés során felhasználtuk, hogy $-\log x \geq \frac{1}{\ln 2}(1-x)$, és hogy $\ln 2 < 1$.

Az előbbi becslésnek egy módosított változata, ha nem 1 hosszú, hanem B hosszú egymást nem fedő blokkokat veszünk, és megkeressük azt a blokkot, ami vele azonos, és legközelebb van hozzá a nála korábbiaknál. Pontosabban legyen adva az $\mathbb{X}$ bináris forrás, ami az $X_1, ...X_n$ sorozatot állítja elő. Erre úgy is tekinthetünk, mint $Y_1, ...Y_{n'}$ sorozatra, ahol Y_i B hosszú bináris blokk minden i-re.

5.13. Definíció.

$$\hat{\lambda}_n = \begin{cases} n \ ha \ y_{n-j} \neq y_n \ 1 \leq j \leq n \\ \min\{0 \leq j \leq n-1 : y_{i-1-j} = x_i\} \ egyébként \end{cases} \tag{14}$$

Mauer a véletlen szám generátor tesztjében [32] alkalmazta a

$$H_m = \frac{1}{K} \sum_{n=Q+1}^{Q+K} \log(\hat{\lambda}_n)$$

becslést, ahol Q és K előre meghatározott konstansok. Megmutatható [32], hogy

$$\lim_{B \to \infty} \frac{E(H_m)}{B} = H$$

ahol H a forrás bitenkénti entrópiáját. Ennek javítását javasolja Conor [33]. A log függvény helyett tekintsünk most g függvényt, amit később fogunk meghatározni.

$$H_c = \frac{1}{K} \sum_{n=Q+1}^{Q+K} g(\hat{\lambda}_n)$$

Olyan g függvényt szeretnénk, amire ha a blokkok függetlenek, akkor $E(H_c) = H$.

$$E(H_c) = \Pr[\hat{\lambda}_n(X) = i]g(i)$$

$$\Pr[\hat{\lambda}_n(X) = i] = \sum_{y \in Y} \Pr(Y_n = y, Y_{n-1} \neq y, ..., Y_{n-i-1} \neq y, Y_{n-i} = y)$$

Ha a blokkok, azaz y_i-k függetlenek, akkor

$$\Pr[\hat{\lambda}_n(X) = i] = \sum_{y \in Y} \Pr(y)^2 (1 - Pr(y)^{i-1})$$

. Ez alapján

$$E(H_c) = \sum_{y \in Y} \Pr(y)^2 (1 - Pr(y)^{i-1} g(i) = \sum_{y \in Y} \Pr(y) \gamma(y)$$

ahol $\gamma(y) = \sum_{i=1}^{\infty} \Pr(y)(1-y)^{i-1} g(i)$ Ha γ oly módon van megválasztva, hogy $\gamma(y) = -\log(y)$, akkor $E(H_c) = \sum_{y \in Y} - \Pr(y) \log(y)$, ami éppen az entrópia definiciója. Tehát a

$$\sum_{i=1}^{\infty} \Pr(y)(1 - \Pr(y))^{i-1} g(i) = -\log(y)$$

egyenletet szeretnénk megoldani k-ra. Legyen $t = 1 - \Pr(y)$, és felhasználjuk, hogy $-\ln(1-t) = \frac{t^i}{i}$, így

$$\frac{1}{\ln 2} \sum_{i=1}^{\infty} \frac{t^i}{i} = (1-t) \sum_{i=1}^{\infty} t^i g(i) = g(0) + \sum_{i=1}^{\infty} t^i ((g(i+1) - g(i))$$

Innen $g(0) = 0$ és $g(i+1) - g(i) = 1/(i \ln 2)$. Tehát a

$$g(k) = \frac{1}{\ln 2} \sum_{i=1}^{k-1} \frac{1}{i}$$

választással a

$$H_c = \frac{1}{\ln 2} \sum_{i=1} \hat{\lambda}_n - 1 \frac{1}{i}$$

entrópia becslő várható értéke tart az egy bitre jutó entrópiához. A H_c becslőt alkalmazza a [21] statisztikai teszt is. A becslés hátránya, hogy általában a forrás nem független elemeket bocsát ki, ezáltal a becslő felülbecsülheti a tényleges entrópiát. De ha a blokkok hosszát elég nagyra választjuk, akkor feltételezhetjük, hogy a blokkok függetlenek.

Az előző eredményt bináris blokkokból álló sorozat helyett tetszőleges véges forrás által generált sorozatra lehet általánosítani, ugyanis a bizonyítás során nem használtuk fel, hogy Y bináris blokk. Tekintsünk tetszőleges véges forrás által előállított $Y_1, ..., Y_n$ sorozatot, ahol az Y_i-k függetlenek. Definiáljuk a következőt:

5.14. Definíció. *Legyen $i \geq r+1$ tetszőleges. Ekkor:*

$$\lambda_i^r = \begin{cases} r \quad ha\ x_{i-j} \neq x_i\ 1 \leq j \leq r \\ \min\{0 \leq j \leq r-1 : x_{i-1-j} = x_i\}\ egyébként \end{cases}$$

A λ_i^r nem nézi meg a teljes memóriát, csak legfeljebb r-et. Ez az algoritmus sebességét javíthatja. Az entrópia becslés a következő módon van definiálva:

5.15. Definíció.

$$\hat{H}_{pv}^r(x_{i-r}^i) = \frac{1}{\ln 2} \sum_{j=1}^{\lambda_i^r} \frac{1}{j}$$

Az előző bizonyítás gondolatmenetét alkalmazva kimondhatjuk az alábbi tételt:

5.16. Tétel. *Legyenek az X véges, stacionrius, ergodikus forrás által előállított $X_1, ..., X_n$ függetlenek. Ha $r, n \to \infty$, akkor az $x_1, ...x_n$ realizációra*

$$\frac{1}{n-r} \sum_{i=r-1}^{n} \hat{H}_{PV}^r(x_{i-r}^i) \to H(p) \quad \textit{1 valószínűséggel}$$

Mint azt korábban kifejtettek, általában nem igaz, hogy a forrás által előállított adat elemei függetlenek, de az [34] cikkben alkalmazott tesztekben $\hat{H}_{pv}^r < \tilde{H}_{n,k}$ teljesült, ez alapján feltételezhető, hogy $\hat{H}_{pv}^r$ tényleg alúlról becsli az entrópiát.

5.6. Min-entrópia becslés

Amikor azt próbáljuk megbecsülni, hogy az entrópia forrás mennyi entrópiával rendelkezik, akkor a Shanon entrópia helyett érdemes lehet a min entrópiát becsülni, azaz azt mérni, hogy legrosszabb esetben mennyi bizonytalansággal rendelkezik a

forrás által előállított sorozat. A NIST által kiadott [35] is bemutat erre különböző statisztikai módszereket. Én ezek közül most csak egyet szeretnék bemutatni, mely az LZ78 működésén alapszik, ez pedig az LZ78Y predictor. A mintán végig haladva szótárt épít, számolja az egyes stringek előfordulását, és ez alapján becsüli meg a következő értéket.

Legyen $X = (x_1, ..., x_n)$ a vizsgált minta, ahol $x_i \in A = a_1, ...a_\alpha$ $i = 1, ...N$-re.

1. Legyen $k = 16$ és $m = n-k-1$. B egy m méretű bool vektor, és $max_dict_size = 65536$ lesz a szótár maximális mérete.

2. D szótár kezdetben üres, $dictq_size = 0$.

3. for $i = k + 2, ..., m$

   a) for $j = k, ...,1$

      – Ha $(x_{i-j-1}, ..., x_{i-2})$ nincs benne D-ben, és $dict_size < max_dict_size$, akkor $(x_{i-j-1}, ..., x_{i-2})$-t berakjuk D-be, $D[(x_{i-j-1}, ..., x_{i-2})][x_{i-1}] = 0$, és $dict_size$ 1-gyel nő.

      – Ha $(x_{i-j-1}, ..., x_{i-2})$ benne van D-ben, akkor $D[(x_{i-j-1}, ..., x_{i-2})]$ 1-gyel nő.

   b) Legyen $maxcount = 0$, $tipp = NULL$. for $j = k, ...,1$

      – Legyen $prev = (x_{i-j}, ..., x_{i-1})$

      – Ha $prev$ benne van D-ben, akkor keressük meg $a \in A$-t, amire $D[prev][a]$ maximális.

      – Ha $D[prev][a] > maxcount$, akkor $tipp = a$, és $maxcount = D[prev][a]$.

   c) Ha $tipp = x_i$, azaz ha helyes a tipp, akkor $B[i - k - 1] = 1$.

   Az a) részben építem a súlyozott szótárat, a b) részben tippelek x_i-re a szótár és az x_i-t megeleőző elemek alapján, végül c)-ben ellenörzöm, hogy helyes volt-e a tipp.

4. Legyen c a helyes tippek, azaz az 1-esek száma B-ben. Ekkor $p'_{global} = c/n$.

$$p_{global} = p'_{global} + 2{,}576\sqrt{\frac{p'_{global}(1 - p'_{global})}{m - 1}}$$

5. Legyen r B-ben a leghosszabb csupa 1-esekből álló sorozat hossza. Ekkor p_{local} a következő egyenlet megoldása, amit bináris kereséssel kapható meg:

$$0{,}99 = \frac{1 - p_{local}z}{(r + 1 - rz)q} \cdot \frac{1}{z^{n+1}},$$

   ahol $q = 1 - p_{local}$ és $z = z_{10}$-et a következő rekurzió határozza meg: $z_j = 1 + qp_{local}^r z_{j-1}^{r+1}$, ahol $z_0 = 1$.

6. Végül a becslés értékét a min-entrópiára a következő adja:

$$min\text{-}entropy = -\log\max(p_{global}, p_{local})$$

5.7. Bayes becslés entrópiára

Az entrópia becslése nem csak véletlen szám generátorok tesztelésére jó, hanem igen széles körben alkalmazzák matematikán kívül eső területeken, mint például neurális folyamatok információtartalmának becslésére. Azonos eloszlású, független valószínűségi változók entrópiáját akarják megbecsülni mintavétel alapján. A feladatra nagyon sok féle entrópia becslő módszert találtak ki, ezek általában véges diszkrét eloszlású val változók becslésére alkalmasak. A plug-in becslő nem túl hatékony, ha kicsi a minta mérete, de nagy a val változó érték készlete. A becslők általában ezt próbálják javítani. Most csak kettő jelentősebbet szeretnék röviden bemutatni.

A bayes becslés egy előre meghatározott priori eloszlásból indul ki, hogy megbecselülje az entrópiát a poszteriori eloszlás felhasználásával. Az első bayes becslés az entrópiára David Wolpert és David Wolf nevéhez fűződik 1995-ből [36]. Ők a dirichlet multinomiális modelt használták. Legyen K a valószínűségi változó értékkészletének elemszáma és $a_1, ..., a_K$ az értékkészlet. A priori eloszlás π K dimenziós dirichlet eloszlású $(\alpha_1, ..., \alpha_K)$ paraméterekkel és jelöle n_i, a mintavétel során aza_i előfordulását. Ekkor a poszteriori eloszlás is dirichlet eloszlású, ráadásul a $\alpha_1 + n_1, ..., \alpha_K + n_K$ paraméterekkel. A becslést az entrópiára a következő adja

$$H_{bayes} = E(H|n_1, ...n_K) = \psi(n + \alpha + 1) - \sum_{i=1}^{K} \frac{n_i + \alpha_i}{n + K} \psi(n_i + \alpha_i + 1)$$

ahol $n = \sum_{i=1}^{K} n_i$ és $\alpha = \sum_{i=1}^{K} \alpha_i$ és ψ a digamma függvény, azaz $\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$. A kérdés, hogyan válasszuk meg az α paraméter vektort. A korábbi javaslatok α értékére 1,1/K,1/2,0.

2002-ben Nemenman, Shafee, and Bialek megmutatta, hogy rögzített *alpha* esetén az entrópia becsült értékét a priori eloszlás dominálja a minta vétel ellenére. Ezért a priori eloszlás meghatározására kevert dirichlet eloszlást használtak [37]. Az NSB nagyon hatékony, de nagy a műveletigénye, és lassú a gyakorlati alkalmazáshoz [38].

6. Összegzés

A szakdolgozatomban bemutattam a leginkább használt titkosítási eljárásokat, és megmutattam, hogy nem biztonságosak, ha az algoritmushoz felhasznált véletleneket nem kellő körültekintéssel választjuk. Bemutattam még két széles körben használt pszeudo-véletlenszám generátort, amik kis mennyiségű valódi véletlenből állítanak elő nagy mennyiségű pszeudo-véletlent. Fontos, hogy pszeudo-véletlenek elég entrópiájú véletlennel legyenek feltöltve, ezért bemutattam módszereket stacionárius, ergodikus források betűnkénti entrópiájának becslésére.

Hivatkozások

[1] J. Hastad *On using RSA with Low Exponent in a Public Key Network.* Advances in Cryptology — CRYPTO '85 Proceedings. Lecture Notes in Computer Science. 218. pp. 403–40. 1986

[2] M. Bellare, P. Rogaway: *Optimal Assymetric Encryption - How to Encrypt RSA.* Advances in Cryptology Eurocrypt 94 Proceedings, Lecture Notes in Computer Science Vol. 950, A. De Santised. ,Springer-Verlag, 1994

[3] I. Goldberg, D. Wagner: *Randomness and Netscape Browser.* Dr. Dobb's Journal. 1996

[4] D. Coppersmith: *Finding a small root of a univariate modular equation.* Advances in Cryptology — EUROCRYPT '96 (Ueli Maurer, ed.), LNCS, no. 1070, IACR, Springer, pp. 155–165. 1996

[5] Daniel R. L. Brown: *A Weak-Randomizer Attack on RSA-OAEP with e=3.* IACR ePrint Archive, 2005

[6] M. Schmid: *ECDSA -Application and Implementation Failures* UC SANTA BARBARA, CS 290G, FALL 2

[7] R. Chirgwin: *Android bug batters Bitcoin wallets.* The Register. 2013

[8] A. Lenstra,; J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, C. Wachter: *Ron was wrong, Whit is right* Santa Barbara: IACR: 17. 2012

[9] D. Shumow, N. Ferguson: *On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng" (PDF).* http://cr.yp.to/ 2007

[10] E. Barker, J. Kelsey: *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* NIST. 2012

[11] F. Goichon, C. Lauradoux, G. Salagnac, T. *Entropy transfers in the Linux Random Number Generator.* [Research Report] RR-8060, INRIA. pp.26. <hal-00738638> 2012

[12] K. Alzhrani, A. Aljaedi: *Windows and Linux Random Number Generation Process: A Comparative Analysis.* International Journal of Computer Applications (0975 – 8887) Volume 113 – No. 8, March 2015

[13] P. Lacharme, A. Rock, V. Strubel, M. Videau. *The linux pseudorandom number generator revisited.* Cryptology ePrint Archive, Report 2012/251, 2012

[14] Z. Gutterman, B. Pinkas, T. Reinman. *Analysis of the Linux random number generator.* In Proceedings of the 2006 IEEE Symp. on Sec. and Privacy, pages 371–385, 2006

[15] J. Kelsey, B. Schneier, N. Ferguson: *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Generator* SAC 1999: Selected Areas in Cryptography pp 13-33

[16] N. Ferguson, B. Schneier, T. Kohno: *Cryptography Engineering*, Wiley Publishing, Inc., pages 138-161, 2010

[17] B. Schneier: *Schneier on Security: Cryptanalysis of SHA-1* 2005

[18] A. Rukhin, J. Soto J et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications,* NIST, US. http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pd, 2010

[19] R. G. Brown: *Dieharder: A Random Number Test Suite, Version 3.31.1*, 2004

[20] L'ecuyer P., Simard *R.,TestU01: A C library for empirical testing of random number generators,* ACM Trans. Math. Softw., vol.33, 2007

[21] W. Killmann, W. Schindler: *A proposal for: Functionality classes for random number generators Version 2.0*

[22] Marek Sys, Zdenek Riha, Vashek Matyás, Kinga Márton, Alin Suciu: *On the Interpretation of Results from the NIST Statistical Test Suite* Romanian Journal of Information Science and Technology Volume 18, Number 1, 18-32. 2015

[23] A. Antos, I. Kontoyiannis: *onvergence Properties of Functional Estimates for Discrete Distributions* Random Structures and Algorithms 19(3-4):163-193. 2001

[24] A. Wyner and J. Ziv *The sliding window Lempel-Zi algorithm is asymptotically optimal,* Proc. IEEE, pp. 872-877, 1994.

[25] A. D. Wyner, J. Ziv: *Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression.* Information Theory, IEEE Transactions on 35(6) 1250-1258. 1989

[26] I. Kontoyiannis. *Asymptotic recurrence and waiting times for stationary processes.* J. Theoret. Probab., 11:795811, 1998

[27] I. Kontoyiannis: Y. Suhov, A. Wyner: *Nonparametric entropy estimation for statinary processes and random fields, with applications to English text* IEEE Trans. Inform. Theory 44, 1319-1327. 1998

[28] Y. Gao, I. Kontoyiannis, E. Bienenstock: *Estimating the entropy of binary time series: Methodology, some theory and a simulation study.* Entropy 10(2) 71-99. 2008

[29] P. T. Maker: *The ergodic theorems for a sequence of functions* Duke Math. J., vol. 6, pp. 27-30. 1940

[30] M. Kac: *On the notion of recurrence in discrete stohastic processes* Bull. Amer. Math. Soc. vol. 53, pp. 1002-1010.

[31] Bucci, M., Luzzi, R.: *Design of testable random bit generators. In: Cryptographic Hardware and Embedded Systems* - CHES 2005. Volume 3659 of Lecture Notes in Computer Science., Edinburgh, UK, Springer 147-156. 2005

[32] U. Maurer, *A universal statistical test for random bit generators,* Journal of cryptology, vol. 5, no. 2, pp. 89105, 1992

[33] J.S. Coron: *On the Security of Random Sources,* Gemplus Corporate Product R&D Division, Technical Report IT02-1998; also in: H. Imai and Y. Zheng (eds.): Public Key Cryptography, Second International Workshop on Practice and Theory in Public Key Cryptography, PKC'99, Springer, Lecture Notes in Computer Science,Vol. 1560, Berlin, pp. 29-42. 1999

[34] C. Lauradoux, J. Ponge, A. Roeck. *Online Entropy Estimation for Non-Binary Sources and Applications on iPhone.* [Research Report] RR-7663, INRIA. pp.19. <inria-00604857v2> 2011

[35] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish, M. Boyle: *Recommendation for the Entropy Sources Used for Random Bit Generation* (Second DRAFT) NIST Special Publication 800-90B 2016

[36] D. Wolpert, D. Wolf *Estimating functions of probability distributions from a finite set of samples,* Physical Review E, Vol. 52, No. 6, 1995

[37] I. Nemenman, F. Shafee, W. Bialek *Entropy and inference, revisited* Advances in Neural Information Processing Systems 14, MIT Press 2002

[38] J. Hausser, K. Strimmer: *Entropy Inference and the James-Stein Estimator, with Application to Nonlinear Gene Association Networks* Journal of Machine Learning Research 10, 1469-1484. 2009