Stream Ciphers

Çetin Kaya Koç
koc@cs.ucsb.edu

Gilbert Vernam

Gilbert Sandford Vernam was an AT&T Bell Labs engineer who, in 1917, invented an additive polyalphabetic stream cipher and later co-invented an automated one-time pad cipher. Wikipedia.

Born: April 3, 1890
Died: February 7, 1960
Block Ciphers

- Plaintext: $M_i$ with $|M_i| = n$, where $n$ is the block length (in bits)
- Ciphertext: $C_i$ with $|C_i| = m$, where $m \geq n$, however, generally output size is equal to input size: $m = n$
- If $m < n$, there will be more than one ciphertext for a given plaintext — ambiguity in decryption
- If $m > n$, some ciphertexts will never appear
- Encryption and decryption functions:
  \[
  E_k(M_i) = C_i \quad ; \quad D_k(C_i) = M_i
  \]
- Key size: $|K|$, the length of the key in bits
Stream Ciphers

- Plaintext: $m_i$ with $|m_i| = k$, where $k$ is the plaintext length (in bits), which is generally a small number: 1, 2, 4, 8, etc
- Ciphertext: $c_i$ with $|c_i| = k$, in other words, $|m_i| = |c_i|$
- Running key: $r_i$ with $|c_i| = k$, a sequence of symbols length $k$
- Plaintext, ciphertext, and running keys are from the same alphabet; for example, for $k = 4$ this would be $\{0000, 0001, \ldots, 1111\}$
- Encryption and decryption functions:

  $$E(m_i) = c_i = m_i \oplus r_i ; \quad D(c_i) = m_i = c_i \oplus^{-1} r_i$$

  where $\oplus$ is the (appropriate) addition function
A Stream Cipher — à la Vigenère

- Plaintext, Ciphertext, Running Key Alphabet: \( \{a, b, c, \ldots, z\} \)
  encoded as elements of \( \mathbb{Z}_{26} \)
- Given a plaintext message: \( m_i \in \mathbb{Z}_{26} \) for \( i = 1, 2, 3, \ldots \)
- Given a sequence of running keys: \( r_i \in \mathbb{Z}_{26} \) for \( i = 1, 2, 3, \ldots \)
- The ciphertext sequence is computed using the encryption function
  \[
  c_i = m_i + r_i \pmod{26}
  \]
- Similarly, the plaintext is computed using the decryption function
  \[
  m_i = c_i - r_i \pmod{26}
  \]
Stream Ciphers

Definitions and Concepts

A Stream Cipher — à la Vigenère

- The encryption and decryption function are

\[ c_i = m_i \oplus r_i \equiv m_i + r_i \pmod{26} \]
\[ m_i = c_i \oplus^{-1} r_i \equiv c_i - r_i \pmod{26} \]

- The sequence of running keys \( r_i \) needs to have certain properties in order for a stream cipher to be cryptographically strong

- For the classic Vigenère:
  - The running key sequence is repeating:
    herbalistherbalistherbalistherbalistherbalistherbalistherbali⋯
  - The period is equal to the length of the key word, which is generally a small integer
In order to understand what properties the running key sequence needs to have we need to see if the stream cipher can be cryptanalyzed under the usual attack scenarios: CO, KP, CP, CT.

Under the CO scenario, given the ciphertext sequence \( c_i \), the purpose of the adversary is to guess or to compute:

- A portion or all of the running key sequence \( r_i \)
- A portion or all of the plaintext sequence \( m_i \)

These actions produce equivalent results in the sense that:

- If a portion of \( r_i \) is obtained, we compute \( m_i \) using \( m_i = c_i \oplus^{-1} r_i \)
- If a portion of \( m_i \) is obtained, we compute \( r_i \) using \( r_i = c_i \oplus^{-1} m_i \)
Cryptanalyzing Stream Ciphers

- On the other hand, under the known or chosen text attack scenarios, the adversary obtains (or chooses) a portion of the plaintext sequence \( m_i \).
- This immediately implies that the adversary can compute a portion of the running key sequence \( r_i \) (which is of the same length as \( m_i \)) using

\[
r_i = c_i \oplus^{-1} m_i
\]

- In order to obtain longer portions of the plaintext, we cannot assume that the adversary will receive further known (or chosen) text.
- At this stage, the adversary can try guess what the other (past or future) portions of the running key would be, given a portion of the running key.
Properties of Running Key Sequences

- As we have said: the sequence of running keys $r_i$ needs to have certain properties in order for a stream cipher to be cryptographically strong.

- Considering the CO attack scenario: The running key sequence needs to have **uniformly distributed** or **statistically random** finite segments so that all segments appear with equal probability, and any segment of the sequence cannot be guessed with better probability than the probability of that segment appearing in the sequence — **Requirement R1**

- Considering the CT attack scenario: *Given any finite segment(s) of the running key sequence*, any past or future segments need to be **unpredictable** which means they cannot be computed or guessed with better probability than the probability of that segment appearing in the sequence — **Requirement R2**
For the rest of our discussions, we will consider the binary stream cipher in which the plaintext $m_i$, ciphertext $c_i$, and the running key $r_i$ words are binary bits, $m_i, c_i, r_i \in \{0, 1\}$ — The plaintext, ciphertext, and running key sequences are binary bit streams.

The encryption and decryption functions are the same:

$$c_i = m_i \oplus r_i = m_i + r_i \pmod{2}$$
$$m_i = c_i \oplus r_i = c_i + r_i \pmod{2}$$

The operation $\oplus$ is the mod 2 addition, which is its own inverse.

\[
\begin{align*}
\text{mi} & \quad 0101 \ 0010 \ 1101 \ 1001 \ 0011 \\
\text{ri} & \quad 0110 \ 0101 \ 0110 \ 0110 \ 0101 \\
\text{ci} & \quad 0011 \ 0111 \ 1011 \ 1111 \ 0110
\end{align*}
\]
A running key sequence generator needs to work in both sides of the channel, at the side of the sender and the receiver, and produce exactly the same sequence $r_i$ in order for the stream cipher to function properly.

Sender: $r_i$ is produced; $c_i = m_i \oplus r_i$ is computed; $c_i$ is sent.

Receiver: $c_i$ is received; the same $r_i$ is produced; $m_i = c_i \oplus r_i$ is computed.

Therefore, we need to have a deterministic state machine producing the running key sequence.

Furthermore, in order for it to be computable, the state machine needs to be finite, i.e., it needs to have a finite number of states (memory).

Therefore: A stream cipher running key generator is a deterministic finite state machine whose sequences $r_i$ satisfy Requirements R1 and R2.
A random number generator (RNG) produces a sequence of random (or random-looking) numbers in a predetermined range, such as \( r_i \in \{0, 1\} \) or \( r_i \in [0, 1] \).

Random (or random-looking) numbers have many applications: statistical physics, simulation, industrial testing and labeling, games, gambling, Monte Carlo methods, and cryptography.

True random numbers cannot be computed on deterministic computers.

True random numbers are best produced using physical random number generators which operate by measuring a well controlled and specially prepared random physical process.

An information-theoretic provable RNG seems to be possible only by exploiting randomness inherent to certain quantum systems.
Random Number Generators (RNGs)

There are two basic categories of RNGs: True RNGs (TRNGs) and Deterministic RNGs (DRNGs).

TRNGs are produced using physical or quantum processes; physical processes include free running oscillators, electrical noise from a resistor or semiconductor, and decay times from a radio-active material.

We cannot use TRNGs as stream ciphers, except for the special case of the Vernam cipher, called the one-time pad.

In order to understand the properties of the one-time pad, we need to define perfect secrecy, a concept introduced by Claude Elwood Shannon, an American mathematician, electronic engineer, and cryptographer known as “The father of Information Theory” — however, we will study perfect secrecy after we study block ciphers.
Stream Ciphers and DRNGs

- Our purpose is to build and understand the properties of stream ciphers.
- DRNGs are finite state machines that have a fixed but large number of starting conditions and states, and thus, very long periods.
- Having long periods is an essential quality for stream ciphers; repeated sequences of running keys will yield information about the plaintext.
- In addition to long period, we also would like to have DRNGs that satisfy Requirement R1 (uniform distribution or statistical randomness) and Requirement R2 (unpredictability).
- In this course, we will limit our attention to DRNGs, and study linear congruential generators, linear feedback shift registers, and cellular automata.
A linear congruential generator produces a sequence of integers $x_i$ for $i = 1, 2, \ldots$ starting with the given initial (seed) value $x_0$ as

$$x_{i+1} = a \cdot x_i + b \pmod{n}$$

where the multiplication and addition operation is performed modulo $n$, and therefore, $x_i \in \mathbb{Z}_n$.

This is a deterministic algorithm; the same $x_i$ value will always produce the same $x_{i+1}$ value, and the same seed $x_0$ will produce the same sequence $x_1, x_2, \ldots$.

There are only finitely many $x_i \in \mathbb{Z}_n$, and the sequence will repeat.

The period of the sequence is $w$ such that $x_{i+w} = x_i$. 


Linear Congruential Generators

- For \((a, b, n) = (3, 4, 15)\), and \(x_0 = 1\), we obtain the following sequence: \(1, 7, 10, 4, 1, 7, 10, 4 \ldots\); the period is \(w = 4\)
- For \((a, b, n) = (3, 4, 15)\), and \(x_0 = 2\), we obtain the following sequence: \(2, 10, 4, 1, 7, 10, 4, 1, 7, \ldots\); the period is \(w = 4\)
- For \((a, b, n) = (3, 4, 17)\), and \(x_0 = 1\), we obtain the following sequence: \(1, 7, 8, 11, 3, 13, 9, 14, 12, 6, 5, 2, 10, 0, 4, 16, 1, 7, 8 \ldots\); the period is \(w = 16\)
- For \((a, b, n) = (3, 4, 17)\), and \(x_0 = 15\), we obtain the following sequence: \(15, 15, 15, \ldots\); the period is just \(w = 1\)
- For \((a, b, n) = (2, 4, 17)\), and \(x_0 = 2\), we obtain the following sequence: \(1, 6, 16, 2, 8, 3, 10, 7, 1, 6, 16, 2, \ldots\); the period is \(w = 8\)
Period of LCGs

- **Theorem:** Given a LCG with parameters \((a, b, p)\) such that \(p\) is prime, the period \(w\) is equal to the order of the element \(a\) in the multiplicative group \(\mathbb{Z}_p^*\) for all \(x_0\) seed values except \(x_0 = -(a - 1)^{-1} \cdot b \mod p\).

- Since the group order is \(p - 1\), the period \(w\) is always a divisor of \(p - 1\). The maximum period occurs when \(a\) is a primitive element, whose order is \(p - 1\).

- For \((a, b, n) = (3, 4, 17)\), the order of the group is equal 16, while the order of the element \(a = 3 \mod 17\) is found as 16 since

\[
\{3^1, 3^2, 3^3, \ldots, 3^{16}\} = \{3, 9, 10, 13, 5, 15, 11, 16, 14, 8, 7, 4, 12, 2, 6, 1\}
\]

On the other hand, the “bad seed” value is

\[
x_0 = -(a - 1)^{-1} \cdot b \mod 11
\]

\[
x_0 = -(3 - 1)^{-1} \cdot 4 = -2^{-1} \cdot 4 = -2 = 15 \pmod{17}
\]
A Practical LCG

- Since our processors have fixed data length, it is a good idea to select a prime as large as the word size, since we will perform mod $p$ arithmetic.

- It turns out that $2^{31} - 1 = 2,147,483,647$ is a prime number; furthermore, the smallest primitive element in $\mathbb{Z}_p$ for $p = 2^{31} - 1$ is found as $a = 7^5 = 16,807$.

- Also, $a$ is fairly close to the square root of $p$, therefore, we have a good, practical, general-purpose LCG, given as

$$x_{i+1} = a \cdot x_i \pmod{p}$$

$$p = 2^{31} - 1 = 2,147,483,647$$

$$a = 7^5 = 16,807$$

Since $a$ is a primitive element, the period of LCG is $w = 2^{31} - 2$. 

\[\text{(http://cs.ucsb.edu/~koc)}\]
Cryptographyc Strength of LCGs

- Does the LCG satisfy requirements R1 and R2?

- Analysis and experiments show that LCGs with large $p$ (such as the previous practical LCG) are (almost) acceptable as statistically random, but there are some deficiencies.

- Unfortunately, the LCGs do not satisfy R2 since they are highly predictable: Assuming $a$ and $p$ are known, given a single element $x_i$, any future element of the sequence can be computed as $x_{i+k} = a^k x_i \mod n$.

- Similarly, given $x_i$, any past element of the sequence can be computed as $x_{i-k} = a^{-k} x_i = (a^{-1})^k \mod n$.

- Inversion: the seed $x_0$ can be computed if any element $x_i$ of the sequence is known, by working back from $i$ down to 0.
Cryptographic Strength of LCGs

- In general, we need to assume that \( a \) and \( p \) are fixed parameters of the RNG and therefore they are not changeable, i.e., they are not part of the key \((x_0, \text{the seed})\) — they can be discovered by reverse engineering.

- If we can bundle \( a \) and \( p \) with the seed \( x_0 \), then we can claim more security — it would be much harder to discover the key \((a, p, \text{and } x_0)\) given a limited number of elements \( x_i \) from the sequence \( x_1, x_2, \ldots \).

- Note that \( x_{i+1} = a \cdot x_i \mod p \) implies \( x_{i+1} = a \cdot x_i + N \cdot p \) for some integer \( N \); however, \( N \) is different for every pair \((x_{i+1}, x_i)\), we have

\[
x_{i+1} = a \cdot x_i + N_i \cdot p
\]

and therefore, if we have \( k \) pairs of the known elements \((x_j, x_k)\) then we will also have \( k + 2 \) unknowns, i.e., \( a \), \( p \), and \( N_i \) for \( i = 1, 2, \ldots, k \).
Cryptographic Strength of LCGs

Still, equations of the form $x_{i+1} = a \cdot x_i + N_i \cdot p$ can be solved using lattice reduction techniques, and therefore, we do not have strong assumptions of cryptographic strength.

There is also practical constraint in a LCG with all three parameters $(a, p, x_0)$ are considered as the key.

We know that $p$ has to be a prime and $a$ has to be a primitive element of the group, that means a key generation algorithm has needs to incorporate these properties and generate such keys.

On the other hand, in a LCG with fixed parameters $(a, p)$ we need not worry about key with special properties — the only key, the seed $x_0$, is just a random integer: any integer would be fine; also, since $b = 0$, the only “bad seed” is 0, and easy to avoid.
GLIBC random()

- The GNU C library’s `random()` function is a LCG with three steps.
- The first step is based on the prime modulus $p = 2^{31} - 1$ and the primitive element $a = 16,807$.
- Given the seed value $s$, the first step computes 33 elements $x_1, x_2, \ldots, x_{33}$:

$$
\begin{align*}
    x_0 &= s \\
    x_i &= a \cdot x_{i-1} \pmod{p} \text{ for } i = 1, 2, \ldots, 30 \\
    x_{31} &= x_0 \\
    x_{32} &= x_1 \\
    x_{33} &= x_2
\end{align*}
$$
The second step is based on the addition operation mod \( q = 2^{32} \)

In the second step, new \( x_i \) values are computed for \( i = 34, 35, \ldots, 343 \)

\[
x_i = x_{i-3} + x_{i-31} \pmod{q} \quad \text{for} \quad i = 34, 35, \ldots, 343
\]

In the final step, the output values are generated using the previous mod \( q \) addition operation and the logical right shift operation \((\cdot)_{rs}\) as follows

\[
x_i = x_{i-3} + x_{i-31} \pmod{q} \quad \text{for} \quad i \geq 344
\]

\[
r_j = (x_{j+344})_{rs} \quad \text{for} \quad i \geq 0
\]

Inversion: Two consecutive different moduli and the right shift make the inversion more difficult, however, since there are \( 2^{32} \) different seed values, exhaustive search is possible
Using Stream Cipher Modes of Block Ciphers

- An efficient way to generate a stream of deterministic random numbers is to use block ciphers, turning a block cipher box into a stream cipher.
- There are 3 basic methods: OFB (output feedback), CFB (cipher feedback), and CTR (counter).
- In block cipher context, these methods are called “modes of operation”.
- There are other modes of operation for block ciphers, each one of which is serving a different purpose.
For all three modes, we assume the following:

A block cipher encryption function is available, which produces an \( m \)-bit ciphertext \( C \) from an \( m \)-bit plaintext \( M \) using a \( n \)-bit key \( K \):

\[
C = E_k(M) \quad \text{such that} \quad |C| = |M| = m \quad \text{and} \quad |K| = n
\]

Also assume an initial value is available, which is generally called \textit{initializing variable} and written as IV.
The Output Feedback Mode

- The OFB produces a key stream $r_i$ of $s$ bits at each step, for $s = 0, 1, 2, \ldots$, and computes the ciphertext $c_i = r_i \oplus m_i$
- We have $s \leq m$, and generally $s$ is a small number, such as 1, 2, or 8
- The algorithm performs for $i = 0, 1, 2, \ldots$, starting with $S_0 = IV$

\[ T_i = E_K(S_i) \]
\[ r_i = TR_s(T_i) \]
\[ c_i = r_i \oplus m_i \]
\[ S_{i+1} = r_i || RS_s(S_i) \]

- $TR_s(T_i)$ is the function that truncates the $m$-bit number $T_i$ to $s$ bits, either by taking the leftmost $s$ bits
- Then, $S_i$ is shifted $s$ bits to right using $RS_s(S_i)$ function, and the $s$-bit $r_i$ is left-appended to get the new $m$-bit $S_{i+1}$
The Cipher Feedback Mode

- The CFB produces a key stream $r_i$ of $s$ bits at each step, for $s = 0, 1, 2, \ldots$, and computes the ciphertext $c_i = r_i \oplus m_i$
- We have $s \leq m$, and generally $s$ is a small number, such as 1, 2, or 8
- The algorithm performs for $i = 0, 1, 2, \ldots$, starting with $S_0 = IV$

\[
T_i = E_K(S_i) \\
r_i = \text{TR}_s(T_i) \\
c_i = r_i \oplus m_i \\
S_{i+1} = c_i \| \text{RS}_s(S_i)
\]

- $\text{TR}_s(T_i)$ is the function that truncates the $m$-bit number $T_i$ to $s$ bits, either by taking the leftmost $s$ bits
- Then, $S_i$ is shifted $s$ bits to right using $\text{RS}_s(S_i)$ function, and the $s$-bit $c_i$ is left-appended to get the new $m$-bit $S_{i+1}$
The Counter Mode

- In the counter mode the $m$-bit initial state $S_0$ consists of two parts: The $u$-bit count value $I$ on the right and a $(m - u)$-bit nonce value $N$ on the left: $S_0 = N || I$
- The initial value of $I = 1$, and $u$ is selected appropriately
- The algorithm performs for $i = 0, 1, 2, \ldots$, starting with $S_0 = N || 1$

$$T_i = E_K(S_i)$$
$$r_i = TR_s(T_i)$$
$$c_i = r_i \oplus m_i$$
$$I = I + 1$$
$$S_{i+1} = N || I$$

- The new value $S_{i+1}$ is obtained by incrementing the counter value $I$ and keeping the nonce $N$ as the same