[CS304] Introduction to Cryptography and Network Security

Course Instructor: Dr. Dibyendu Roy
Scribed by: Archit Verma (202151192)

Winter 2023-2024
Lecture (Week 6)

1 One-Time Pad (OTP)

In cryptography, the one-time pad emerges as a method wherein a uniquely generated private key is employed once to encrypt a message, with the recipient decrypting it using a corresponding one-time pad and key.

Employing randomly generated keys for message encryption confers a theoretical advantage due to the inherent unpredictability of each encryption instance. This randomness ensures that analyzing a series of messages does not compromise the security of the encryption, as each encryption is distinct and bears no relation to subsequent ones. However, the necessity for the recipient to possess the exact key used for encryption introduces concerns regarding securely transmitting or safeguarding both keys.

The One-Time Pad (OTP) encryption method hinges on the XOR operation, which combines each character of the message M with a corresponding character from the secret key K to produce the ciphertext C. Symbolically, this operation is expressed as:

$$C = M \oplus K$$

Consider two characters of the message, M_1 and M_2, each XORed with the key K to yield their respective ciphertexts C_1 and C_2:

$$C_1 = M_1 \oplus K$$

$$C_2 = M_2 \oplus K$$

For this encryption scheme to maintain its security, the length of the key K must equal or exceed the length of the message M.

The function $F(K, IV)$ employs a pseudo-random function to generate a bit sequence N_i, where K represents the secret key and IV denotes the initialization vector (public key). Subsequently, these bits undergo an XOR operation with the message to produce the ciphertext. This process can be elucidated as follows:

$$\begin{array}{rc} & N_0, N_1, \ldots, N_{n-1} \\ \oplus & M_0, M_1, \ldots, M_{n-1} \\ \hline & M_0 \oplus N_0, M_1 \oplus N_0, \ldots, M_{n-1} \oplus N_{n-1} \\ \hline \end{array}$$

Some notable aspects and implications of this encryption approach include:

- With knowledge of the bit sequence $N_0, N_1, \ldots, N_{n-1}$ and the key K, one can generate N_i using the function $F(K, IV)$ and subsequently obtain the ciphertext C_i by XORing N_i with the corresponding message character M_i. - Randomly chosen and securely kept secret, the key K

ensures that the output bits $N_0, N_1, \ldots, N_{n-1}$ are indistinguishable from those generated by a true random bit generator. - The length of the output bits N_i significantly exceeds that of the key K, bolstering security, particularly when K remains confidential while IV is public. - Even a minor alteration to a single bit of either the initialization vector IV or the key K results in a wholly unpredictable change in the output bits N_i. - The independence of generated bits persists even when utilizing distinct initialization vectors, as evidenced by the uncorrelated sequences $N_i^{(1)}$ and $N_i^{(2)}$ corresponding to different IVs.

2 Stream Ciphers

Stream ciphers are a type of symmetric encryption algorithm that encrypts plaintext data one bit or byte at a time, typically in a continuous stream. They are often used in situations where the plaintext data is transmitted continuously, such as in real-time communication systems like voice or video calls, or in applications where low latency is critical.

Overview

Let the message be a sequence of bits:

$$01010100\,01111000$$

Let

$$10101000\,01111000$$

be a sequence of random bits.

The ciphertext is obtained by performing bitwise XOR operation between the message and the random sequence:

$$01010100 \oplus 10101000 = 11111100$$

$$01111000 \oplus 01111000 = 00000000$$

So, the ciphertext is:

$$11111100\,00000000$$

where

$$10101000\,01111000$$

is the random sequence.

Synchronous Stream Ciphers

In a synchronous stream cipher, the generation of the key stream occurs independently of both the plaintext and ciphertext bits. This independence ensures the security and efficiency of the encryption process.
The state update function is represented as:

$$S_{i+1} = f(S_i, c)$$

Meanwhile, the key stream is generated using a dedicated function, where S_i represents the current state and K denotes the secret key:

$$Z_i = g(S_i, K)$$

The ciphertext is then generated by combining the key stream with the plaintext character m_i through a designated function:

$$C_i = h(Z_i, m_i)$$

Here, S_0 denotes the initial state, which is typically derived from the secret key K and the Initialization Vector (IV).

In a synchronous stream cipher, a stream of pseudorandom digits, usually binary, is generated independently of the plaintext and ciphertext messages. This stream, known as the key stream, is combined with the plaintext using the exclusive OR (XOR) operation to encrypt the message. Similarly, for decryption, the key stream is XORed with the ciphertext to recover the plaintext.

One critical aspect of synchronous stream ciphers is the need for synchronization between the sender and receiver. If synchronization is lost due to additions or deletions in the message during transmission, decryption may fail. Strategies to restore synchronization include systematically trying different offsets or using markers in the ciphertext to aid in synchronization.

However, if a digit in the ciphertext is corrupted during transmission, only the corresponding digit in the plaintext is affected, and the error does not propagate. While this property can be advantageous in environments with high error rates, it also makes synchronous stream ciphers vulnerable to active attacks. An attacker could potentially manipulate the ciphertext to cause predictable changes in the plaintext, such as flipping specific bits.

Self-Synchronizing Stream Cipher

A self-synchronizing stream cipher, also known as an asynchronous stream cipher or ciphertext autokey (CTAK), operates by generating keystream bits based on both previous ciphertext digits and a secret key. This method facilitates automatic synchronization between the sender and receiver, bolstering the encryption's robustness.
The cipher's state at any given moment, denoted σ_i, relies on a subset of preceding ciphertext bits:

$$\sigma_i = (C_{i-t}, C_{i-t+1}, C_{i-1})$$

The keystream generator function, $g(\sigma_i, K)$, employs the current state σ_i and the secret key K to produce keystream bits Z_i, defined as $Z_i = g(\sigma_i, K)$.
Subsequently, the ciphertext C_i forms by combining the keystream Z_i with the corresponding plaintext bit M_i using the function $h(Z_i, M_i)$, yielding $C_i = h(Z_i, M_i)$.

This approach enhances security and ensures that even if digits are added, lost, or corrupted during transmission, the receiver can automatically synchronize with the keystream generator, making recovery more feasible. An example of a self-synchronizing stream cipher is a block cipher in cipher feedback (CFB) mode.

3 Linear Feedback Shift Register (LFSR)

A Linear Feedback Shift Register (LFSR) is a type of shift register where the next state is determined by applying a linear function to the current state. This function typically involves bitwise XOR operations, known for their simplicity and effectiveness in generating pseudo-random sequences. The bits influencing the state transition in subsequent bits are referred to as "taps," strategically chosen to ensure desirable properties in the generated sequence.

At $t = 0$, the initial state S_0 is represented as $S_0 : S_{n-1}S_{n-2}...S_1S_0$. Here, t represents the clocking number.

At $t = 1$, the state S_1 becomes $S_1 : S_{n-1}S_{n-2}...S_2S_1$, and the output is S_0.

At $t = 2$, the state S_2 becomes $S_2 : S_{n-1}S_{n-2}...S_2S_1$, and the output is S_1.

The function $L(S_0, ..., S_{n-1}) = S_n \in \{0, 1\}$ determines the state transition, where $L : \{0, 1\}^n \to \{0, 1\}$. It is defined as:

$$L_a = a_0S_0 \oplus a_1S_1 \oplus ... \oplus a_{n-1}S_{n-1} \oplus a_n$$

Here, $a_i \in \{0, 1\}$.

For a linear function $L(x, y) = x \oplus y$, the property holds that $L(x) \oplus L(y) \oplus L(x \oplus y) = 0$.

For an affine function $L(x, y) = 1 \oplus x \oplus y$, the property holds that $L(x) \oplus L(y) \oplus L(x \oplus y) = 1$.

An LFSR cycles through states until it returns to its initial configuration, marking the completion of one full sequence. This return to the starting state signifies the period of the LFSR. If the LFSR returns to the same state after 'm' steps, it is termed a 'm'-period LFSR.

If we start with a non-zero initial state and return to that same state after 'm' steps, we refer to this duration as the 'm' period. An n-bit LFSR is termed fully periodic when its cycle length, or period, equals $2^n - 1$.

As an example, consider an LFSR with the following steps:

1. $t = 0$: 111 (Initial state = 0)
2. $t = 1$: $111 \to 1$
3. $t = 2$: $010 \to 1$
4. $t = 3$: $001 \to 0$
5. $t = 4$: $100 \to 1$
6. $t = 5$: $110 \to 0$
7. $t = 6$: $111 \to 0$
8. $t = 7$: $011 \to 1$
9. $t = 8$: $101 \to 1$

This example illustrates the cyclical behavior of an LFSR. If we start with an initial state of 111, the LFSR cycles through a sequence of states until it returns to 111, completing one full cycle. The period of this LFSR is 7.

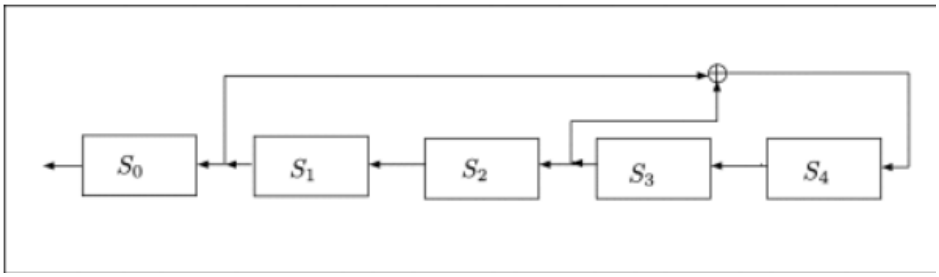

Figure 1: Enter Caption

1. If the polynomial is irreducible then period of LFSR will divide $2^n - 1$.
2. If it is reducible then different state will have different cycle length (different period).

3.1 n-bit LFSR

An n-bit Linear Feedback Shift Register (LFSR) consists of n stages, each capable of storing one bit. The register is represented by a sequence of n bits denoted as $K_{n-1}K_{n-2}...k_1K_0$, where the output bits X_i correspond to the key stream bits Z_i.

In an LFSR: - Each ciphertext bit c_i is computed as the XOR of the corresponding plaintext bit m_i and the key stream bit z_i, where z_i belongs to the set $\{0, 1\}$. - The LFSR undergoes state updates through two operations: linear feedback and shifting. - The state update function of the LFSR is denoted by α, and it updates the state and key stream according to:

$$S_{t+1} = \alpha(S_t)$$

$$Z_{t+1} = f(S_{t+1})$$

1. Known plaintext attack: If we possess the key stream bits ($Z_0 = k_0, Z_1 = k_1, ..., Z_{n-1} = k_{n-1}$), we can formulate a set of linear equations to deduce the key.

2. LFSR with non-linear filter function: For an n-bit LFSR ($n \geq 1$), where the function f maps elements from the set $\{0, 1\}$ to $\{0, 1\}$.

These properties and operations characterize the behavior of an n-bit LFSR in cryptographic systems.

3.2 Non-linear Feedback Shift Register (NFSR)

A Non-linear Feedback Shift Register (NFSR) is characterized by a feedback function that is non-linear. This non-linearity introduces complexity and enhances the cryptographic properties of the shift register.

In an NFSR, a function $h : A \to B$ is applied, where $h(x) = y$, and $f : \{0, 1\}^l \to \{0, 1\}$. The function f is defined as $f = f(x) + f(y) + f(x + y)$, adding another layer of non-linearity to the register.

The output bits Z_i are combined with plaintext bits m_i through the operation $m_i \oplus Z_i = C_i$, resulting in the ciphertext C_i.

3.3 State Update Function of an LFSR

The state update function of an LFSR (Linear Feedback Shift Register) is typically represented mathematically as follows:

$$S_{t+1} = \alpha(S_t)$$

In this equation:

- S_t represents the state of the LFSR at time t.
- S_{t+1} represents the updated state of the LFSR at time $t + 1$.
- α denotes the state update function, which determines how the current state S_t transitions to the next state S_{t+1}.

The specific form of α depends on the design and configuration of the LFSR, including the feedback taps and any additional logic applied to the state transition.

3.4 LFSR with Combiner Function

In some cryptographic schemes, multiple LFSRs (Linear Feedback Shift Registers) are used in parallel, and their outputs are combined using a combiner function. This approach helps introduce non-linearity into the system, which can enhance the security of the encryption process.

Let's denote $LFSR_1, LFSR_2, ..., LFSR_n$ as n parallel LFSRs, each with its own state and feedback configuration. The output of each LFSR, denoted $S_{1,t}, S_{2,t}, ..., S_{n,t}$ at time t, is fed into a combiner function F to generate the keystream bit Z_t.

Mathematically, the keystream bit Z_t is computed as:

$$Z_t = F(S_{1,t}, S_{2,t}, ..., S_{n,t})$$

where F is a binary Boolean function that takes the outputs of the parallel LFSRs as inputs and produces the keystream bit Z_t.

The choice of the combiner function F is crucial for the security of the encryption scheme. It should possess certain cryptographic properties to resist attacks and ensure the unpredictability of the keystream. Commonly used combiner functions include logical AND, OR, XOR, and more complex non-linear functions.

By combining the outputs of multiple LFSRs using an appropriate combiner function, cryptographic systems can achieve higher levels of security and resistance against cryptanalysis.

3.5 Hash Functions

A hash function $h : A \to B$, where $h(X) = Y$, exhibits several key properties:

1. Modifying X to X' results in a completely different value for $h(X')$.
2. Given Y, it is impossible to determine X such that $h(X) = Y$.
3. For a given X and $Y = h(X)$, there exists no X' such that $h(X) = h(X')$.

A hash function is defined by a four-tuple (P, S, K, H), where:

1. P represents the set of all possible messages.
2. S represents the set of all possible message digests or authentication tags.
3. K denotes the key space.
4. For each $K_1 \in K$, there exists a hash function $h_{K_1} \in H$ such that $h_{K_1} : P \to S$, where $|P| \geq |S|$. Moreover, $|P| \geq 2 \times |S|$.

If the hash function incorporates a key in its computation, it is referred to as a keyed hash function; otherwise, it is termed as an unkeyed hash function.

Problem 1: Pre-image Finding Problem

Given $h : P \to S$, the pre-image finding problem involves finding $x \in P$ such that $h(x) = y$. If x cannot be found in a reasonable time, then h is considered a pre-image resistant hash function; otherwise, it is pre-image friendly.

Problem 2: Second Pre-image Finding Problem

Given $h : P \to S$, and $x \in P$ with $h(x)$, the second pre-image finding problem involves finding $x' \neq x$ such that $h(x') = h(x)$. If finding x' is hard, then h is considered a second pre-image resistant hash function.

Problem 3: Collision Finding Problem

Given $h : P \to S$, the collision finding problem involves finding $x, x' \in P$ such that $x \neq x'$ and $h(x) = h(x')$. If finding such x and x' is hard, then h is called a collision resistant hash function.

4 Ideal Hash Function

An ideal hash function $h : P \to S$ ensures that obtaining $h(x)$ for a value x in P requires either computing $h(x)$ directly or consulting the corresponding entry in the hash table associated with h.

It is a theoretical construct that exhibits desirable properties, including:

1. **Uniformity**: The hash function distributes the output values uniformly across the output space, minimizing collisions.
2. **Pre-image Resistance**: It is computationally infeasible to find a pre-image for any given hash value, ensuring the security of hashed data.
3. **Second Pre-image Resistance**: Given an input, finding another input that produces the same hash value is computationally difficult, enhancing the security of hash-based schemes.
4. **Collision Resistance**: The probability of finding two distinct inputs that produce the same hash value is negligible, preventing attackers from forging hash collisions.

While ideal hash functions serve as a theoretical benchmark for cryptographic primitives, real-world hash functions aim to approximate these properties as closely as possible to ensure robust security in practical applications.

5 Pre-image Finding Algorithm

The pre-image finding algorithm aims to find $x \in X$ such that $h(x) = y$, given $y \in Y$ and $h : X \to Y$. The algorithm iterates over a subset $X_0 \subset X$ with $|X_0| = Q$, attempting to find x such that $h(x) = y$. The probability of success is proportional to $\frac{1}{\text{complexity}}$, denoted by A. The probability of successfully finding x is calculated as $\frac{A}{M}$, where M represents the complexity.

This algorithm illustrates the approach to solving the pre-image finding problem efficiently in hash functions.