CS387 - Applied Cryptography

Ángel Sola Orbaiceta

December 2021

1 Concepts

Given a message $m \in M$, where M is the set of all possible messages, and a key $k \in K$, where K is the set of all possible keys, an encryption function E can be defined as:

$$E : M \times K \to C$$

where $c \in C$ is the *ciperthext* (being C the set of all possible ciphertexts). Conversely, a decryption function D can be defined as:

$$D : C \times K \to M$$

The **correctness property** states that, for all messages and keys, decrypting the result of encrypting a message must result in the message itself. Mathematically:

$$\forall m, k : D_k(E_k(m)) = m$$

The **security property** states that the ciphertext reveals nothing about the key or original message.

1.1 One-Time Pad

The one-time pad is based in the XOR ($\oplus$) function. The XOR function satisfies the property that any value XOR-ed with itself equals zero: $x \oplus x = 0$. The one-time pad uses this property so that, by using a key that's the same size as the ciphertext, we can do:

$$c = m \oplus k$$
$$m = c \oplus k$$

The one-time pad encryption and decryption functions are implemented in the *source/one_time_pad.py* file.

1.2 Perfect Cipher

Recall that given two events A and B in the same probability space, the **conditional probability** of B given that A occured is:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

Now, given two messages, m and m^* drawn from the set of messages M are encrypted using a key $k \in K$ to produce a cypertext $c \in C$, a **perfect cipher** must hold that:

$$P\left(m = m^* | E_k\left(m\right) = c\right) = P\left(m = m^*\right)$$

We can now prove that the one-time pad is a perfect cipher as follows. For the one-time pad:

$$P\left(E_k\left(m\right) = c\right) = \sum_{m_i \in M} \sum_{k_i \in K} \frac{P(E_{k_i}(m_i) = c)}{|M| \times |K|} = \frac{|M|}{|M| \times |K|} = \frac{1}{|K|}$$

and, assuming a uniform distribution for the key space:

$$P\left(m = m^* \cap E_k(m) = c\right) = P(m = m^*) \times P(k = k^*) = \frac{P(m = m^*)}{|K|}$$

Therefore:

$$P\left(m = m^* | E_k(m) = c\right) = \frac{\frac{P(m=m^*)}{|K|}}{\frac{1}{|K|}} = P(m = m^*)$$

which means that the ciphertext reveals nothing about the key or original message.

Shanon's theorem If a cipher is perfect, it must be impractical ($|K| \geq |M|$).

2 Application of Symmetric Ciphers

Kolmogorov complexity The complexity K of a sequence s ($K(s)$) is the length of the shortest possible description of s. s is random if $K(s) = |s| + C$. The Kolmogorov complexity is uncomputable.

A Pseudo-Random Number Generator (PRNG) produces a long sequence of seemingly random bytes given an initial seed value. In Linux, *dev/random* can be used as a randomness pool.

2.1 Modes Of Operation

We assume the message we want to encrypt or decrypt can be broken into n blocks of size b:

$$m = m_0, m_1, m_2, \ldots, m_{n-1}$$

Electronic Codebook Mode (ECB) each block is encrypted independently from each other:

$$c_i = E_k(m_i)$$

and decryption:

$$m_i = D_k(c_i)$$

The problem with ECB mode is that it doesn't hide repetition: equal blocks encrypt to equal ciperthexts. To run a cipher using the ECB mode of operation:

```
$ py symmetric/block_cli.py -e -m ecb -f <file> -k <key>
```

Cipher Block Chaining Mode (CBC) the output of each block is XOR-ed with the input to the next block. Encryption:

$$c_i = E_k(m_i \oplus c_{i-1})$$

and if an **initialization vector** is used, the first block:

$$c_0 = E_k(m_0 \oplus IV)$$

and decryption:

$$m_i = D_k(c_i) \oplus c_{i-1}$$
$$m_0 = D_k(c_0) \oplus IV$$

To run a cipher using the CBC mode of operation:

```
$ py symmetric/block_cli.py -e -m cbc -f <file> -k <key>
```

Counter Mode (CTR) uses a counter that increments for each block, which concatenated with a **nonce** is passed to the encryption function. The result is then XOR-ed with the message blocks. The benefit of CTR is that blocks can be encrypted and decrypted in parallel.

$$c_i = E_k(\texttt{nonce}||i) \oplus m_i$$

and decryption:

$$m_i = c_i \oplus D_k(\texttt{nonce}||i)$$

If the block size is 32 bytes (256 bits), the nonce and the counter can be 16 bytes (128 bits) each. When concatenated, both make a 32 bytes sequence.

To run a cipher using the CTR mode of operation:

```
$ py symmetric/block_cli.py -e -m ctr \
  -f <file> \
  -k <key> \
  -n <nonce>
```

2.2 Protocols

A **protocol** is a precisely defined sequence of steps agreed upon two parties. A **cryptographic protocol** involves shared secrets between the two parties, and, for it to be **secure**, it must provide some guarantees even if some participants cheat.

2.3 Cryptographic Hash Functions

A **hash function** $h = H(x)$ is a function defined over a large input domain which output is of a fixed –usually small– length. A regular hash function has the following properties:

- compression: a large input domain is mapped to a small fixed output
- well distributed: $P(H(x) = i) \sim \frac{1}{N}$, where N is the size of the output

Additionally, **cryptographic hash functions** must have the following properties:

- pre-image resistance ("one-way ness"): given $h = H(x)$, it's hard to find x
- weak collision resistance: given $h = H(x)$, it's hard to find any x' such that $H(x') = h$
- strong collision resistance: it's hard to find any pair x and y such that $H(x) = H(y)$

Given an ideal hash function with N possible outputs, an attacker is expected to need:

- weak: $\sim N$ guesses to find x' where $H(x') = h$
- strong: $\sim \sqrt{N}$ guesses to find x, y where $H(x) = H(y)$.

Salted password scheme To securely store passwords and avoid dictionary attacks –where the attacker precomputes a list of password hashes–, a good scheme is to hash the result of concatenating a **salt** (sequence or random bits) with the password: $H(\texttt{salt}||\texttt{password})$. The salt can be stored together with the password; it's just used to avoid the dictionary attack.

3 Key Distribution

With symmertic ciphers, the parties need to share a secret before they can communicate securely: **the key**. But distributing keys among all possible pairs of parties who want to communicate isn't practical.

Pairwise distribution If n people want to communicate with each other, that'd mean that $\frac{n\times(n-1)}{2}$ different keys need to be shared among them. For 100 people, this would be $\frac{100\times 99}{2} = 4950$ keys. This is obviously impractical.

Trusted third party A third party might be trusted to maintain a secret with each person who wants to communicate. Then, whenever two persons –say Alice and Bob– want to communicate, they must first ask the third party to create a new secret key for them.

3.1 Diffie-Hellman Key Exchange

Published in 1976 by Whitfield Diffie and Martin Hellman in their famous paper “New Directions in Cryptography”.

Diffie-Hellman Key Exchange This protocol –assuming Alice and Bob want to share a private key– works as follows:

1. Alice and Bob agree on two shared numbers: a large prime number q and g, a primitive root[1] of q
2. Alice chooses a random secret value $x_A = \{0,1\}^n$ and computes $y_A = g^{x_A} \,\texttt{mod}\, q$
3. Bob chooses a random secret value $x_B = \{0,1\}^n$ and computes $y_B = g^{x_B} \,\texttt{mod}\, q$
4. Alice and Bob exchange the y_A and y_B values
5. Alice can compute the shared secret key: $k_{AB} = y_B^{x_A} \,\texttt{mod}\, q$
6. Bob can compute the shared secret key: $k_{BA} = y_A^{x_B} \,\texttt{mod}\, q$

We can prove that the keys that both Alice and Bob compute are the same ($k_{AB} = k_{BA}$) since, for Alice:

$$k_{AB} = y_B^{x_A} \,\texttt{mod}\, q = g^{x_B \times x_A} \,\texttt{mod}\, q$$

and for the case of Bob:

$$k_{BA} = y_A^{x_B} \,\texttt{mod}\, q = g^{x_A \times x_B} \,\texttt{mod}\, q$$

A **passive eavesdropper** would know about q, g, y_A and y_B, so, how can be sure they wouldn't be able to compute the private key k_{AB}? An adversary would need to be able to solve the discrete logarithm problem in order to compute the shared secret key, and we know the discrete logarithm problem is very hard to solve.

[1] g is a primitive root of q if $\forall x \in \{1, 2, \ldots, q-1\} \exists k$ such that $g^k = x$.

The Discrete Logarithm Problem With continuous logarithms, if we're given the equation $a^x = b$, we know of efficient ways to compute x if we rearrange the equation as $x = \log_a b$. In the discrete logarithm case: computing x in $a^x = b \bmod n$, there isn't a known way to solve it. One thing to note about $a^x = b \mathtt{mod} n$ is that it doesn't always have a solution (as opposed to the continuous case), but if n is a prime number and a is a generator[2] number, then there must be a solution for x.

Computing a^n can be done using $O(\log n)$ multiplications. Modular exponentiation is linear in the size (bits to represent) of the power.

Rabin-Miller and AKS Primality Tests To test if a large number is prime in a reasonable amount of time.

4 Asymmetric Cryptosystems

[2]A number g is a generator if its powers form a permutation of the numbers in Z_n: $\{1, 2, \ldots, n-1\}$.