CO487 Lecture Notes

Billy Lee

Winter 2024

Contents

1 Week 1 (Jan 8-12)

1.1 Course Preview

1.1.1 What is Cryptography?

Cryptography is a1ut securing communications in the presence of malicious adversaries.
Adversarial capabilities include

- Read data
- Modify data
- Inject data
- Delete data
- and much more

1.1.2 Fundamental Goals of Cryptography

- Confidentiality: Keeping data secret from all but those authorized to see it.
- Data integrity: Ensuring data has not been altered by unauthorized means (Authentication).
- Data Origin Authentication: Corroborating the source of data (Authentication).
- Non-repudiation: Preventing an entity from denying previous commitments or actions (Real life examples: contracts, signatures, etc.).

1.1.3 Secure Web Transactions

1.1.3.1 Transport Layer Security (TLS)

The cryptography protocol used by web browsers to securely communicate with web sites such as gmail, facebook, amazon, etc.

TLS is used to assure an individual user (client) of the authenticity of the web site (server) they are visiting, and to establish a secure communications channel for the remainder of the session.

1.1.3.2 Symmetric-key Cryptography

The client and server share some secret information k called a key. They can subsequently engage in secure communication by encrypting their messages with **AES** and authenticating the resulting ciphertexts with **HMAC**.

Question: How do the client and server establish the shared secret key k? If transmitting over the internet is unsafe, how can they exchange the key securely?

1.1.3.3 Public-key Cryptography

The client and server share some **authenticated** (but non-secret) information.

To establish a secret key, the client selects the secret **session key** k, and encrypts it with the server's **RSA public key**. Then only the server can decrypt the resulting ciphertext with its **RSA private key** to recover k.

How does the client obtain an authentic copy of the server's RSA public key?

1.1.3.4 Signature Scheme

The server's RSA public key is signed by a **Certification Authority** (CA) using its secret signing key with the **RSA signature scheme**.

The client can verify the signature using the CA's **RSA public verification key**. In this way, the client obtains an authentic copy of the server's RSA public key.

Question: How does the client obtain an authentic copy of the CA's RSA public key? Answer: The CA's RSA public key is embedded in the client's web browser.

1.1.4 The TLS Protocol

1. When a client first visits a secured web page, the server transmits its **certificate** to the client.
   - The certificate contains the server's identifying information (e.g. the website name and URL) and RSA public key, and the RSA signature of a certification authority.
   - The certification authority (e.g. DigiCert) is trusted to carefully verify the server's identity before issuing the certificate.
2. Upon receipt of the certificate, the client verifies the signature using the certification authority's public key, which is embedded in the browser. A successful verification confirms the authenticity of the server and of its RSA public key.
3. The client selects a random session key k, encrypts it with the server's RSA public key, and transmits the resulting ciphertext to the server.
4. The server decrypts the ciphertext to obtain the session key k, which is then used with symmetric-key encryption schemes to encrypt (e.g. with AES) and authenticate (e.g. with HMAC) all sensitive data exchanges for the remainder of the session.

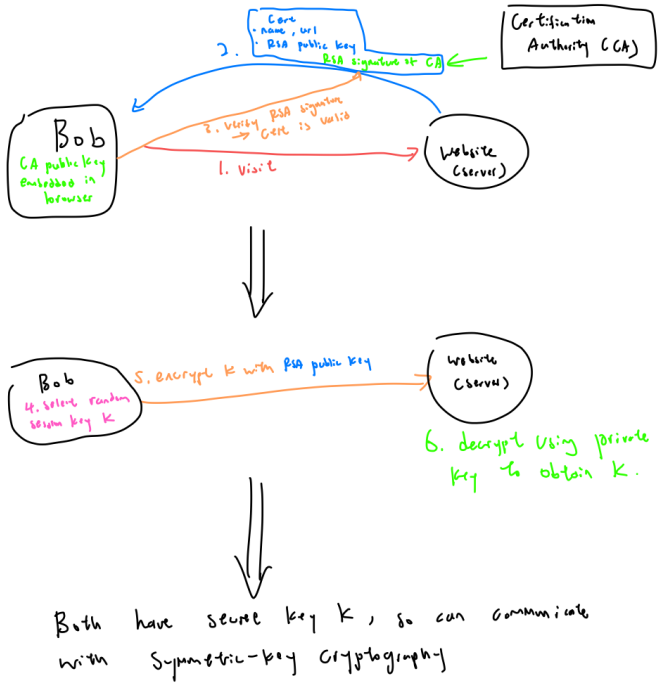


Figure 1: TLS Protocol

1.1.5 TLS Potential Vulnerabilities

There are many potential security vulnerabilities:

1. The crypto is weak (e.g. AES, HMAC, RSA).
2. The crypto can be broken using a quantum computer.
3. Weak random number generation (for session key).
4. Issuance (noun of issue) of fraudulent certificates.
5. Software bugs (both inadvertent and malicious).
6. Phishing attacks
7. TLS only protects data during transit. It does not protect data stored at the server.

1.1.6 Cybersecurity

Cybersecurity is comprised of the concepts, technical measures, and administrative measures used to protect networks, computers, programs, and data from deliberate or inadvertent unauthorized access, disclosure, manipulation, or use. Also known as information security. There are 3 main components:

Computer Security	Network Security	Software Security
Security models and policies	Internet protocols	Buffer overflows
Secure operating systems	Viruses and worms	Programming languages and compilers
Virus protection	Denial-of-service (DoS)	Digital rights management
Auditing mechanisms	Firewalls	Software tamper resistance
Risk analysis	Intrusion detection	Trusted computing
Risk management	Wireless communications	

Note that cryptography is not equal to cybersecurity. Cryptography provides some mathematical tools that can assist with the provision of cybersecurity services. It is a small, albeit indispensable, part of a complete security solution.

1.2 Symmetric-Key Encryption

Definition 1.1 – Symmetric-key Encryption Scheme
A symmetric-key encryption scheme (SKES) consists of

- M: the plaintext space
- C: the ciphertext space
- K: the key space
- a family of encryption functions $E_k : M \to C$ for all $k \in K$
- a family of decryption functions $D_k : C \to M$ for all $k \in K$

Such that $D_k(E_k(m)) = m$ for all $m \in M$ and $k \in K$.

1.2.1 Using SKES to Achieve Confidentiality

Suppose we have two parties, Alice and Bob, who wish to communicate securely.

1. Alice and Bob agrees on a secret key $k \in K$ by communicating over the secured channel.
2. Alice computes $c = E_k(m)$ and sends the ciphertext c to Bob over the unsecured channel.

3. Bob retrieves the plaintext by computing $m = D_k(c)$.

Now, a question arises: if Alice and Bob has a secured channel, why don't they just communicate through that? The answer is that the secured channel might not often be an actual physical channel for communication. For example, when we register for a mobile plan with a provider, the secret key is the SIM card which we insert into our phone. The secured channel to deliver that secret key is then the physical delivery of the SIM card to us.

Example 1.1 – Simple Substitution Cipher
Suppose we have the following permutation as the secret key k:

$$k = \begin{array}{cccccccccccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h} & \text{i} & \text{j} & \text{k} & \text{l} & \text{m} \\ \text{D} & \text{N} & \text{X} & \text{E} & \text{S} & \text{K} & \text{O} & \text{J} & \text{T} & \text{A} & \text{F} & \text{P} & \text{Y} \\ \text{n} & \text{o} & \text{p} & \text{q} & \text{r} & \text{s} & \text{t} & \text{u} & \text{v} & \text{w} & \text{x} & \text{y} & \text{z} \\ \text{I} & \text{Q} & \text{U} & \text{B} & \text{R} & \text{Z} & \text{G} & \text{V} & \text{C} & \text{H} & \text{M} & \text{W} & \text{L} \end{array}$$

An example encryption would be

$$m = \text{the big dog} \leftrightarrows c = E_k(m) = \text{GJS NTO EQO}$$

We will see below that this simple substitution cipher is not secure.

1.2.2 What Does it Mean for a SKES to be Secure?

We need a security definition, but first, we have to define some assumptions.

1. What are the computational powers of the adversary?
2. How does the adversary interact with the two communicating parties?
3. What is the adversary's goal?

Security Model: Defines the computational abilities of the adversary, and how she interacts with the communicating parties.
Convention: We will always strive to model maximal adversary capabilities and minimal adversary goals.
Basic Assumption: The adversary knows everything about the SKES, except for the secret key k chosen by Alice and Bob.

1.2.2.1 Computational Power of the Adversary

There are 3 types of assumptions for the computational power of the adversary:

1. Information-theoretic security: Eve has infinite computational resources.
2. Complexity-theoretic security: Eve is a polynomial-time Turing machine.
3. Computational Security: Eve has 36768 Intel E5-2683 V4 cores running at 2.1 GHz at her disposal. We say that Eve is **computationally bounded**.

In this course, we will only be concerned with **computational security**, so we will assume that Eve is computationally bounded.

1.2.2.2 Adversary's Interactions

There are 2 types of attacks that the adversary can perform:

1. Passive Attacks:
   - Ciphertext-only attack: The adversary knows some ciphertext (that was generated by Alice or Bob).
   - Known-plaintext attack: The adversary also knows **some** plaintext and the corresponding ciphertext.
2. Active Attacks:

- Chosen-plaintext attack: The adversary can also **choose** some plaintext and obtains the corresponding ciphertext (from Alice or Bob).

How does the chosen-plaintext attack work? Suppose we have two devices, a desktop and a phone. So Alice is the desktop and Bob is the phone in this case. People can send an email to our desktop, and our desktop will encrypt our email, then send to the phone. An adversary can then send an email to our desktop, then intercept the ciphertext sent to the phone. Thus, the adversary knows the plaintext and the corresponding ciphertext.

1.2.2.3 Adversary's Goal

These are the goals of the adversary (from strongest to weakest):

1. Recover the secret key k.
2. Systematically recover plaintext from ciphertext, without necessarily learning k.
3. Learn some partial information about the plaintext from the ciphertext, other than its length.

Definition 1.2 – Totally Insecure
If the adversary can achieve 1 or 2, the SKES is said to be **totally insecure** (or **totally broken**).

Definition 1.3 – Semanically Secure
If the adversary cannot learn any partial information about the plaintext from the ciphertext (except possibly its length), the SKES is said to be **semantically secure**.

1.2.2.4 Definition of a Secure SKES

Thus, we now have our definition of a secure SKES:

Definition 1.4 – Secure SKES
A symmetric-key encryption scheme (SKES) is said to be **secure** if it is semantically secure against chosen-plaintext attack by a computationally bounded adversary (since we are assuming computational security and minimal adversary goals).

Thus, the adversary breaks a SKES if she accomplishes the following:

1. The adversary is given a challenge ciphertext c.
2. During its computation, the adversary can select plaintext and obtain (from either Alice or Bob) the corresponding ciphertexts.
3. After a feasible amount of computation, the adversary obtains some information about the plaintext m corresponding to c (other than the length of m).

1.2.3 Desirable Properties of a SKES

1. Efficient algorithms should be known for computing $E_k(m)$ and $D_k(c)$ (the encryption and decryption algorithms should be efficient).
2. The secret key k should be small, but large enough to render exhaustive key search infeasible (there should be enough keys to make trying every key infeasible).
3. The scheme should be secure (as defined above).
4. The scheme should be secure even against the designer of the system.

1.2.4 Security of the simple substitution cipher

Recall the simple substitution cipher

Example 1.2 – Simple Substitution Cipher
Suppose we have the following permutation as the secret key k:

$$k = \begin{array}{cccccccccccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h} & \text{i} & \text{j} & \text{k} & \text{l} & \text{m} \\ \text{D} & \text{N} & \text{X} & \text{E} & \text{S} & \text{K} & \text{O} & \text{J} & \text{T} & \text{A} & \text{F} & \text{P} & \text{Y} \\ \text{n} & \text{o} & \text{p} & \text{q} & \text{r} & \text{s} & \text{t} & \text{u} & \text{v} & \text{w} & \text{x} & \text{y} & \text{z} \\ \text{I} & \text{Q} & \text{U} & \text{B} & \text{R} & \text{Z} & \text{G} & \text{V} & \text{C} & \text{H} & \text{M} & \text{W} & \text{L} \end{array}$$

An example encryption would be

$$m = \text{the big dog} \leftrightarrows c = E_k(m) = \text{GJS NTO EQO}$$

We will see below that this simple substitution cipher is not secure.

It is totally insecure against a chosen-plaintext attack. The adversary can recover the key by simply getting the ciphertext for the string "abcdefghijklmnopqrstuvwxyz" (the adversary is allowed to get ciphertext for any chosen plaintext).

How about a ciphertext-only attack (where the adversary is only given the ciphertext)? Is exhaustive key search possible? Here could be the strategy:

- Given sufficient amounts of ciphertext c, decrypt c using each possible key until c decrypts to a plaintext message that makes sense.
- In principle, 30 characters of ciphertext are sufficient on average to yield a unique plaintext that is sensible English message. In practice, a few hundred characters are needed.

For the exhaustive key search (brute force search), the number of keys to try is $26! \approx 2^{88}$. This is infeasible since if the adversary uses 1,000,000 computers, each capable of trying 10^9 keys per second, then exhaustive key search takes about 10^4 years.

Definition 1.5 – Work Factor
In this course, we will use the following work factor scale:

- 2^{40} operations is considered very easy.
- 2^{56} operations is considered easy.
- 2^{64} operations is considered feasible.
- 2^{80} operations is considered barely feasible.
- 2^{128} operations is considered infeasible.

Definition 1.6 – Security Level
A cryptographic scheme is said to have a security level of ℓ bits if the fastest known attack on the scheme takes approximately 2^{ℓ} operations.

Of course, apart from the brute force method, a simple frequency analysis of ciphertext letters can be used to recover the secret key (compare the frequency of letters in the ciphertext to the frequency of letters in the English language). Hence, the simple substitution cipher is totally insecure against a ciphertext-only attack.

1.2.5 Polyalphabetic Ciphers

The basic idea is that we use several permutations, so a plaintext letter is encrypted to one of several possible ciphertext letters.

m	=	t	h	i	s	i	s	a	m	e	s	s	a	g	e
+k	=	C	R	Y	P	T	O	C	R	Y	P	T	O	C	R
c	=	V	Y	G	H	B	G	C	D	C	H	L	O	I	V

1.2.5.1 Vegenere Cipher

The secret key is an English word having no repeated letters, e.g. $k =$ CRYPTO. An example of an encryption is Here, $A = 0, B = 1, \ldots, Z = 25$, and addition of letters is in modulo 26. Thus, the decryption is subtraction modulo 26. So,

$$m = c - k$$

The frequency distribution of ciphertext letters is flatter than for a simple substitution cipher.

However, the Vigenere cipher is totally insecure against a chosen-plaintext attack. Since an attacker can just pick the plaintext to be 30 "a"s to get the key back. Against ciphertext-only attacks, the Vigenere cipher is also totally insecure.

1.2.5.2 One-time Pad

The One-time Pad is a variation of the Vegenere cipher that is secure. It is invented by Vernam in 1917 for the telegraph system. The secret key is a random string of letters that is the exact same length as the plaintext m. An example encryption would be

m	=	t	h	i	s	i	s	a	m	e	s	s	a	g	e
+k	=	Z	F	K	W	O	G	P	S	M	F	J	D	L	G
c	=	S	M	S	P	W	Y	P	F	Q	X	C	D	R	K

However, the key should not be reused since if we have two ciphertext-plaintext pairs,

$$c_1 = m_1 + k, \quad c_2 = m_2 + k$$

then

$$c_1 - c_2 = (m_1 + k) - (m_2 + k) = m_1 - m_2$$

Thus, $c_1 - c_2$ depends only on the plaintext (and not on the key k), and hence can leak information about the plaintext m_1 and m_2.

1.2.5.3 Binary Messages

From now on, unless otherwise stated, messages and keys will be assumed to be bit (binary) strings (this makes sense because all digital communications are done in binary). We use the notation $\oplus$ to denote bitwise exclusive-or (XOR), i.e. bitwise addition mod 2. For example,

$$1011001011 \oplus 1001001001 = 0010000010$$

Thus, for the one-time pad, encryption is $c = m \oplus k$ and decryption is $m = c \oplus k$.

1.2.5.4 Security of the One-time Pad

Definition 1.7 – Perfect Secrecy
An SKES is said to have **perfect secrecy** if it is semantically secure against ciphertext-only attacks by an adversary with infinite computational resources.

The one-time pad is semantically secure.

Shannon (1949) proved that if plaintexts are ℓ-bit strings, then any symmetric-key encryption scheme with perfect secrecy must have $|K| \geq 2^{\ell}$. In other words, there are at least 2^{ℓ} keys; since keys are binary strings, this means that the length of the key must be at least as long as the plaintext. Thus, the perfect secrecy of the one-time pad is fairly useless in practice.

1.2.5.5 Stream Ciphers

Instead of using a random key in the one-time pad, we use a "pseudorandom" key.

> **Definition 1.8 – Pseudorandom Bit Generator (PRBG)**
> A pseudorandom bit generator (PRBG) is a deterministic algorithm (i.e. input can only have 1 output) that takes an input a (random) **seed**, and outputs a longer pseudorandom sequence called the **keystream**.

Thus, a stream cipher uses a PRBG for encryption. The **seed** is the secret key shared by Alice and Bob. Note that we no longer have perfect secrey since the keystream is not purely random but only pseudorandom. Thus security depends on the quality of the PRBG.

1.2.5.6 Security Requirements for the PRBG

- Indistinguishability requirement: the keystream should be **indistinguishable** from a random sequence.
- Unpredictability requirement: given portions of the keystream, it should be computationally infeasible to learn any information about the remainder of the keystream. That is, an attacker cannot deduce the rest of the keystream from the portion of the keystream that she has.

1.2.6 ChaCha20 Stream Cipher (Show and Tell)

The ChaCha20 stream cipher is conceptually very simple, word oriented (a word is 32 bits), and uses only simple arithmetic operations - integers modulo 2^{32}, XOR, and left rotations. To date, no security weaknesses have be found.

To define ChaCha20's initial state, we need the following notation:

- 256-bit **key** $k = (k_1, k_2, \ldots, k_8)$.
- 96-bit **nonce** $n = (n_1, n_2, n_3)$, a non-repeating quantity.
- 128-bit **constant** $f = (f_1, f_2, f_3, f_4) = (\texttt{0x61707865}, \texttt{0x3320646e}, \texttt{0x79622d32}, \texttt{0x6b206574})$.
- 32-bit **counter** c.

The **initial state** is

$$\begin{array}{|c|c|c|c|}\hline f_1 & f_2 & f_3 & f_4 \\ \hline k_1 & k_2 & k_3 & k_4 \\ \hline k_5 & k_6 & k_7 & k_8 \\ \hline c & n_1 & n_2 & n_3 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline S_1 & S_2 & S_3 & S_4 \\ \hline S_5 & S_6 & S_7 & S_8 \\ \hline S_9 & S_{10} & S_{11} & S_{12} \\ \hline S_{13} & S_{14} & S_{15} & S_{16} \\ \hline \end{array}$$

Then, we define the **quarter round** function. It's purpose is to mix up the bits of the 4 32-bit word inputs a, b, c, d in a complicated nonlinear fashion. We use the following notation:

- $\oplus$: XOR
- $\boxplus$: addition modulo 2^{32}
- $\lll t$: left rotation by t positions

Algorithm 1 Quarter Round Function (QR)

Input: Four 32-bit words a, b, c, d

$a \leftarrow a \boxplus b$
$d \leftarrow d \oplus a$
$d \leftarrow d \lll 16$
$c \leftarrow c \boxplus d$
$b \leftarrow b \oplus c$
$b \leftarrow b \lll 12$
$a \leftarrow a \boxplus b$
$d \leftarrow d \oplus a$
$d \leftarrow d \lll 8$
$c \leftarrow c \boxplus d$
$b \leftarrow b \oplus c$
$b \leftarrow b \lll 7$

Output: Four 32-bit words a, b, c, d.

Here is the ChaCha20 keystream generator:

Algorithm 2 ChaCha20 Keystream Generator

Select a nonce n and initialize the counter c.
while keystream bytes are still needed **do**
    Create the initial state S.
    Make a copy S' of S.
    for $i = 1$ to 10 **do**
        QR (S_1, S_5, S_9, S_{13})
        QR $(S_2, S_6, S_{10}, S_{14})$
        QR $(S_3, S_7, S_{11}, S_{15})$
        QR $(S_4, S_8, S_{12}, S_{16})$
        QR $(S_1, S_6, S_{11}, S_{16})$
        QR $(S_2, S_7, S_{12}, S_{13})$
        QR (S_3, S_8, S_9, S_{14})
        QR $(S_4, S_5, S_{10}, S_{15})$
    end for
    Output $S \oplus S'$
    Increment the counter.
end while

Encryption: The keystream bytes are XORed with the plaintext bytes to produce ciphertext bytes. The nonce is appended to the ciphertext.

2 Week 2 (Jan 15-19)

2.1 Symmetric-Key Encryption (Continued)

2.1.1 Block Ciphers

Definition 2.1 – Block Cipher
A block cipher is a symmetric-key encryption scheme that breaks up the plaintext into blocks of a fixed length (e.g. 128 bits), and encrypts the blocks one at a time.

In contrast, a stream cipher encrypts the plaintext one character (usually a bit) at a time.

2.1.1.1 Desirable Properties of Block Ciphers

Security:

- Diffusion: each ciphertext bit should depend on **all** plaintext bits. That means that changing one bit of the plaintext should change **all** bits of the ciphertext.
- Confusion: the relationship between key and ciphertext bits should be complicated.
- Key length: should be small, but large enough to preclude exhaustive search.

Efficiency:

- Simplicity: easy to implement and analyze.
- Speed: high encryption and decryption rate.
- Platform: suitable for hardware and software.

2.1.1.2 Data Encryption Standard (DES)

An example of a block cipher is the Data Encryption Standard (DES).

- Key length: 56 bits.
- Size of key space: 2^{56}.
- Block length: 64 bits.

DES has several problems.

1. Small key size: exhaustive search on the key space takes only 2^{56} operations and can easily be parallelized.
2. Small block size: if plaintext blocks are distributed "uniformly at random", then the expected number of ciphertext blocks observed before a collision occurs is approximately 2^{32}. This means that it is possible that after encrypting 2^{32} blocks, it is possible that two plaintext blocks maps to the same ciphertext block.

Definition 2.2 – Birthday Paradox
Suppose that an urn contains n numbered balls. Suppose that balls are drawn from the urn, one at a time, with replacement. The expected number of draws before a ball is selected for the second time (called a collision) is approximately $\sqrt{\pi n/2} \approx \sqrt{n}$.

One potential solution might be multiple encryption: re-encrypt the ciphertext one or more times using independent keys. However, this does not result in increased security. For example, if E_{π_1} denotes the encryption function for the simple substitution cipher with key π_1, and E_{π_2} denotes the encryption function for the simple substitution cipher with key π_2, then $E_{\pi_1} \circ E_{\pi_2}$ is equivalent to the simple substitution cipher with key $\pi_1 \circ \pi_2$. So the combination is just one permutation, which has the same key space.

2.1.1.3 Double DES

A Double-DES secret key is $k = (k_1, k_2)$, where $k_1, k_2 \in_R \{0,1\}^{56}$. $k \in_R K$ means that k is chosen uniformly and independently at random from the set K.

Encryption is

$$c = E_{k_2}(E_{k_1}(p))$$

Decryption is

$$m = E_{k_1}^{-1}(E_{k_2}^{-1}(c))$$

The Double-DES key length is $\ell = 112$ bits, so exhaustive key search takes 2^{112} operations. However, the block length is still unchanged at 64 bits.

The Double-DES encryption scheme is susceptible to the Meet-in-the-middle attack. The main idea is that

$$c = E_{k_2}(E_{k_1}(p)) \implies E_{k_2}^{-1}(c) = E_{k_1}(p)$$

Algorithm 3 Meet-in-the-middle attack on Double-DES

Input: 3 plaintext/ciphertext pairs $(m_1, c_1), (m_2, c_2), (m_3, c_3)$.

for each $h_2 \in \{0,1\}^{56}$ **do** ▷ h_2 is a guess for k_2
  Compute $E_{h_2}^{-1}(c_1)$ and store $[E_{h_2}^{-1}(c_1), h_2]$ in a table sorted by first component. ▷ Decode c_1 using all possible h_2
end for
for each $h_1 \in \{0,1\}^{56}$ **do** ▷ h_1 is a guess for k_1
  Compute $E_{h_1}(m_1)$. ▷ Encrypt m_1 using all possible h_1
  Search for the entry $[E_{h_1}(m_1), h_2]$ in the table. ▷ Main idea of the attack
  for each match $[E_{h_2}^{-1}(c_1), h_2]$ in the table **do** ▷ Checking if h_1, h_2 are correct by checking with c_2 and c_3
    if $E_{h_2}(E_{h_1}(m_2)) = c_2$ **then**
      if $E_{h_2}(E_{h_1}(m_3)) = c_3$ **then**
        return (h_1, h_2) and STOP
      end if
    end if
  end for
end for

Output: The secret key $k = (k_1, k_2)$.

Why do we need 3 plaintext/ciphertext pairs? Because there could be false keys that match c_1 and m_1, but do not match c_2 and m_2. How many plaintext/ciphertext pairs are needed for unique key determination?

Let E be a block cipher with key space $K = \{0,1\}^\ell$ ($\ell = 112$ in Double DES), and plaintext and ciphertext space $\{0,1\}^L$ ($L = 64$ in Double DES).

Let $k' \in K$ be the secret key chosen by Alice and Bob, and let $(m_i, c_i), 1 \le i \le t$, be the known plaintext/ciphertext pairs where the plaintexts are distinct. So we know that $c_i = E_{k'}(m_i)$ for $1 \le i \le t$. Then, how large should t be to ensure (with probability very close to 1) that there is only one key $k \in K$ for which $E_k(m_i) = c_i$ for all $1 \le i \le t$? We would have to select t such that the expected number of false keys, $FK \approx 0$.

For each $k \in K$, the encryption function $E_k : \{0,1\}^L \to \{0,1\}^L$ is a permutation (bijection). We make the heuristic assumption that for each $k \in K$, E_k is a random function. This assumption is false since the encryption function is deterministic, but the assumption is good for our analysis.

Now we fix $k \in K$ and assume that $k \neq k'$. The probability that $E_k(m_i) = c_i$ for all $1 \le i \le t$ is

$$\underbrace{\frac{1}{2^L} \cdot \frac{1}{2^L} \cdots \frac{1}{2^L}}_{t \text{ times}} = \frac{1}{2^{Lt}}$$

Each $\frac{1}{2^{Lt}}$ is the probability that m_i maps to c_i under the function E_k (since we are assuming that E_k is random).

So, the expected number of false keys $k \in K$ (not including k') for which $E_k(m_i) = c_i$ for all $1 \leq i \leq t$ is

$$FK = \frac{2^\ell - 1}{2^{Lt}}$$

This is the number of keys $k \neq k'$ multiplied by the probability that the key k is a key that matches all the plaintext/ciphertext pairs.

Now, let E be the DES encryption function, so Double-DES encryption is $c = E_{k_2}(E_{k_1}(m))$. We have $\ell = 112, L = 64$, let's try out different t values (the number of plaintext/ciphertext pairs) to see how many keys $k \in K$ match all the plaintext/ciphertext pairs.

1. If $t = 1$, then $FK \approx 2^{48}$.
2. If $t = 2$, then $FK \approx \frac{1}{2^{16}}$.
3. If $t = 3$, then $FK \approx \frac{1}{2^{80}} \approx 0$.

For the time complexity of the attack, the number of DES operations is

$$\approx \underbrace{2^{56}}_{\text{line 1}} + \underbrace{2^{56}}_{\text{line 4}} + \underbrace{2 \cdot 2^{48}}_{\text{line 8}} \approx 2^{57}$$

Note that we are not counting the time to do the sorting and searching.

The space requirements is

$$2^{56}(64 + 56) \text{ bits} \approx 1,083,863 \text{ TB}$$

In conclusion, the security level of Double-DES is 57 bits, so Double-DES is not much more secure than DES.

2.1.1.4 Triple DES

A Triple-DES secret key is $k = (k_1, k_2, k_3)$, where $k_1, k_2, k_3 \in_R \{0,1\}^{56}$.

Encryption is

$$c = E_{k_3}(E_{k_2}(E_{k_1}(p)))$$

Decryption is

$$m = E_{k_1}^{-1}(E_{k_2}^{-1}(E_{k_3}^{-1}(c)))$$

The Triple-DES key length is $\ell = 168$ bits, so exhaustive key search takes 2^{168} operations which is infeasible.

The Meet-in-the-middle attack takes approximately 2^{112} operations. So the security level of Triple-Des is 112 bits. However, there is no proof that Triple-DES is more secure than DES. This is because the block length is still unchanged.

2.1.1.5 Substitution-Permutation Networks

A substitution-permutation network (SPN) is an **iterated block cipher** where a **round** consists of a substitution operation followed by a permutation operation. The components of an SPN cipher are:

- n: the block length (in bits).
- ℓ: the key length (in bits).
- h: the number of rounds.
- A fixed invertible function $S : \{0,1\}^b \to \{0,1\}^b$ called a **substitution**, where b is a divisor of n.
- A fixed **permutation** $P \in S_n$ on $\{1, 2, \ldots, n\}$.

- A **key scheduling algorithm** that determines round keys $k_1, k_2, \ldots, k_h, k_{h+1}$ from a key k.

Note that all of the above, n, ℓ, h, S, P, and the key scheduling algorithm are public. The only secret in AES is the key k that is selected.

Here is a description of encryption:

Algorithm 4 Encryption in an SPN cipher

Input: Round keys $k_1, k_2, \ldots, k_h, k_{h+1}$, number of rounds h, plaintext.

1: $A \leftarrow$ plaintext
2: **for** $i = 1, 2, \ldots, h$ **do**
3: $\quad A \leftarrow A \oplus k_i$ ▷ XOR
4: $\quad A \leftarrow S(A)$ ▷ Substitution
5: $\quad A \leftarrow P(A)$ ▷ Permutation
6: **end for**
7: $A \leftarrow A \oplus k_{h+1}$
8: ciphertext $\leftarrow A$

Output: ciphertext

Decryption is the reverse of encryption.

2.1.1.6 Finite Field $GF(2^8)$

The elements of the finite field $GF(2^8)$ are the polynomials of degree at most 7 in $\mathbb{Z}_2[y]$ (the polynomials in y with coefficients in $\mathbb{Z}_2$), with addition and multiplication performed modulo the irreducible polynomial

$$f(y) = y^8 + y^4 + y^3 + y + 1$$

We can interpret an 8-bit string

$$a = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$$

as coefficients of the polynomial

$$a(y) = a_7 y^7 + a_6 y^6 + a_5 y^5 + a_4 y^4 + a_3 y^3 + a_2 y^2 + a_1 y + a_0$$

and vice versa.

For example, let $a = 11101100$ and $b = 00111011$. So

$$a = 11101100 \iff a(y) = y^7 + y^6 + y^5 + y^4 + y^2$$

$$b = 00111011 \iff b(y) = y^5 + y^4 + y^2 + y + 1$$

Addition is simply

	a	$=$	1	1	1	0	1	1	0	0
$+$	b	$=$	0	0	1	1	1	0	1	1
	$a \oplus b$	$=$	1	1	0	1	0	1	1	1

Which is equal to

$$a(y) + b(y) = y^7 + y^6 + y^4 + y^3 + y + 1$$

Multiplication is defined by

				a	$=$	1	1	1	0	1	1	0	0
			$\times$	b	$=$	0	0	1	1	1	0	1	1
						1	1	1	0	1	1	0	0
					1	1	1	0	1	1	0	0	
				0	0	0	0	0	0	0	0		
			1	1	1	0	1	1	0	0			
		1	1	1	0	1	1	0	0				
$+$	1	1	1	0	1	1	0	0					
	1	0	1	0	1	0	0	0	1	0	1	0	0

Then we divide by $f(y) = y^8 + y^4 + y^3 + y + 1 = 100011011$ to get (this is a long division)

100011011	/	1	0	1	0	1	0	0	0	1	0	1	0	0
		1	0	0	0	1	1	0	1	1				
				1	0	0	1	0	1	0	0	1	0	0
				1	0	0	0	1	1	0	1	1		
							1	1	0	0	1	0	0	0

So remainder is $11001000 = y^7 + y^6 + y^3$. Thus, $a(y) \cdot b(y) = y^7 + y^6 + y^3$.

Inversion of $a = 11101100$ is simply another element a^{-1} from $GF(2^8)$ such that $a \cdot a^{-1} = 00000001$.

2.1.1.7 Advanced Encryption Standard (AES)

AES is an SPN, where the permutation operation is comprised of two **invertible** linear transformations. All operations are **byte oriented**, e.g. $b = 8$ so the substitution operation (called the S-box), maps 8 bits to 8 bits, so $S : \{0,1\}^8 \to \{0,1\}^8$. This facilitates fast implementations on software platforms. The block length is $n = 128$. Each subkey (round key) is 128 bits (no matter the key length). AES accepts three key lengths, the number of rounds h depends on the key length.

- AES-128: $\ell = 128, h = 10$.
- AES-192: $\ell = 192, h = 12$.
- AES-256: $\ell = 256, h = 14$.

Each round updates a variable called **State** which consists of a 4×4 array of bytes (note that $4 \times 4 \times 8 = 128$, the block length). The state is first initialized with the plaintext.

$$\text{State} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \leftarrow \text{plaintext}$$

Each AES round uses four invertible operations:

1. AddRoundKey: key mixing
2. SubBytes: S-box
3. ShiftRows: permutation
4. MixColumns: linear transformation

After h rounds are completed, a final subkey (the final round key) is XORed with State, the result being the ciphertext.

AddRoundKey

Perform Bitwise-XOR of each byte of State with the corresponding byte of the round key (recall that each round key is 128 bits, so it can be divided into a 4×4 array of bytes). Suppose that we have the round key

$$k = \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

We XOR the state and the round key:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} = \begin{bmatrix} a_{0,0} \oplus k_{0,0} & a_{0,1} \oplus k_{0,1} & a_{0,2} \oplus k_{0,2} & a_{0,3} \oplus k_{0,3} \\ a_{1,0} \oplus k_{1,0} & a_{1,1} \oplus k_{1,1} & a_{1,2} \oplus k_{1,2} & a_{1,3} \oplus k_{1,3} \\ a_{2,0} \oplus k_{2,0} & a_{2,1} \oplus k_{2,1} & a_{2,2} \oplus k_{2,2} & a_{2,3} \oplus k_{2,3} \\ a_{3,0} \oplus k_{3,0} & a_{3,1} \oplus k_{3,1} & a_{3,2} \oplus k_{3,2} & a_{3,3} \oplus k_{3,3} \end{bmatrix}$$

$$= \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

SubBytes
Take each byte in State and replace with the output of the S-box (the substitution operation S). $S : \{0,1\}^8 \to \{0,1\}^8$ is a fixed, public, invertible, non-linear function.

$$\begin{bmatrix} S(a_{0,0}) & S(a_{0,1}) & S(a_{0,2}) & S(a_{0,3}) \\ S(a_{1,0}) & S(a_{1,1}) & S(a_{1,2}) & S(a_{1,3}) \\ S(a_{2,0}) & S(a_{2,1}) & S(a_{2,2}) & S(a_{2,3}) \\ S(a_{3,0}) & S(a_{3,1}) & S(a_{3,2}) & S(a_{3,3}) \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

Now we will give the definition of the S-box. Let $p \in \{0,1\}^8$, and consider p as an element of $GF(2^8)$. We first let

$$q = \begin{cases} p^{-1} & \text{if } p \neq 0 \\ p & \text{if } p = 0 \end{cases}$$

Then q is an element of $GF(2^8)$, so it can be represented as $q = (q_7 q_6 q_5 q_4 q_3 q_2 q_1 q_0)$. Then, we will compute

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \pmod 2$$

The output of the S-box is then

$$S(p) = r = (r_7 r_6 r_5 r_4 r_3 r_2 r_1 r_0)$$

More simply, the lookup table of the S-box is the following:

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

For example, $S(\texttt{0xA8}) = \texttt{0xC2}$.

ShiftRows
Permute the bytes of State by applying a cyclic shift to each row, where the first row is not shifted, the second row is shifted by 1 position to the left, the third row is shifted by 2 positions to the left, and the fourth row is shifted by 3 positions to the left. That is,

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \mapsto \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix}$$

MixColumns Read column i of State as a polynomial. For example, for the following State:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

We get

$$(a_{0,i}, a_{1,i}, a_{2,i}, a_{3,i}) = a_{0,i} + a_{1,i}y + a_{2,i}y^2 + a_{3,i}y^3$$

(read each coefficient $a_{j,i}$ as elements of $GF(2^8)$, that is $a_{j,i} = a_7a_6a_5a_4a_3a_2a_1a_0$)
Then, we multiply this polynomial with the constant polynomial

$$c(x) = 02 + 01x + 01x^2 + 03x^3 \qquad \text{Will be defined below}$$

and reduce modulo $x^4 - 1$. This gives a new polynomial

$$b_{0,i} + b_{1,i}y + b_{2,i}y^2 + b_{3,i}y^3$$

Example 2.1 – How to multiply $a(x) \otimes c(x)$
Let $a(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ and $c(x) = 02 + 01x + 01x^2 + 03x^3$, where 01 , 02 , 03 are elements in $GF(2^8)$, but written as hexadecimal.

To compute $a(x) \otimes c(x)$ do:

- Compute $d(x) = a(x) \times c(x)$, which is polynomial multiplication where coefficient arithmetic is in

$GF(2^8)$.

- Divide by $x^4 - 1$ to find the remainder polynomial $r(x)$.
- Then $a(x) \otimes c(x) = r(x)$.

As an example, let $a(x) = \texttt{d0f112bb} = \texttt{d0} + \texttt{f1}x + \texttt{12}x^2 + \texttt{bb}x^3$. Then

$$a(x) \otimes c(x) = \texttt{1a} + \texttt{a4}x + \texttt{d3}x^2 + \texttt{e5}x^3 = \texttt{1aa4d3e5}$$

Now, here is a description of encryption and decryption. From the key k derive $h + 1$ round keys $k_0, k_1, \ldots, k_h$

Algorithm 5 Encryption of AES

Input: Round keys $k_0, k_1, \ldots, k_h$, plaintext.

State $\leftarrow$ plaintext
State $\leftarrow$ State $\oplus\, k_0$
for $i = 1, 2, \ldots, h-1$ **do**
    State $\leftarrow$ SubBytes(State)
    State $\leftarrow$ ShiftRows(State)
    State $\leftarrow$ MixColumns(State)
    State $\leftarrow$ State $\oplus\, k_i$
end for
State $\leftarrow$ SubBytes(State)
State $\leftarrow$ ShiftRows(State)
State $\leftarrow$ State $\oplus\, k_h$
ciphertext $\leftarrow$ State

Output: ciphertext

Algorithm 6 Decryption of AES

Input: Round keys $k_0, k_1, \ldots, k_h$, ciphertext.

State $\leftarrow$ ciphertext
State $\leftarrow$ State $\oplus\, k_h$
State $\leftarrow$ InvShiftRows(State)
State $\leftarrow$ InvSubBytes(State)
for $i = h-1, \ldots, 2, 1$ **do**
    State $\leftarrow$ State $\oplus\, k_i$
    State $\leftarrow$ InvMixColumns(State)
    State $\leftarrow$ InvShiftRows(State)
    State $\leftarrow$ InvSubBytes(State)
end for
State $\leftarrow$ State $\oplus\, k_0$
plaintext $\leftarrow$ State

Output: plaintext

where InvMixColumns is multiplication by

$$d(x) = \texttt{0e} + \texttt{09}x + \texttt{0d}x^2 + \texttt{0b}x^3$$

modulo $x^4 - 1$.

AES Key Schedule (for 128-bit keys)
For 128-bit keys, AES has 10 rounds, so we need 11 round keys. The first round key is $k_0 = (r_0, r_1, r_2, r_3)$, which is the actual AES key. The other round keys are

$$k_1 = (r_4, r_5, r_6, r_7), \quad k_2 = (r_8, r_9, r_{10}, r_{11}), \quad \ldots, \quad k_{10} = (r_{40}, r_{41}, r_{42}, r_{43})$$

where each round key $k_i, 1 \leq i \leq 10$ are derived using previous round keys and the key schedule functions f_i.

The functions $f_i : \{0,1\}^{32} \to \{0,1\}^{32}$ are defined as follows:

1. The input is divided into four bytes: (a, b, c, d).
2. Left rotate the bytes by 1 position: (b, c, d, a).
3. Apply the AES S-box to each byte: $(S(b), S(c), S(d), S(a))$.
4. XOR the leftmost byte with the constant ℓ_i, and output the result:

$$(S(b) \oplus \ell_i, S(c), S(d), S(a))$$

The constants ℓ_i (in hexadecimal) are:

$$\ell_1 = 01, \quad \ell_2 = 02, \quad \ell_3 = 04, \quad \ell_4 = 08, \quad \ell_5 = 10, \quad \ell_6 = 20, \quad \ell_7 = 40, \quad \ell_8 = 80, \quad \ell_9 = 1b, \quad \ell_{10} = 36$$

The following is the performance of different encryption algorithms (tested on an Intel Xeon CPU)

Algorithm	Block length (bits)	Key length (bits)	Speed (Mbytes/s)
ChaCha20	N/A	256	323
Triple-DES	64	168	21
AES-128	128	128	170
AES-128-NI	128	128	2426
AES-256	128	256	129
AES-256-NI	128	256	1830

NI stands for new instruction set, these mean that there are special hardware in the CPU that can perform AES operations faster.

2.1.2 Block Cipher Modes of Operation

In practice, one usually wishes to encrypt a large quantity of data. The plaintext message is $m = m_1, m_2, \ldots, m_t$, where each m_i is an L-bit block. Now, how should we use a block cipher $E_k : \{0,1\}^L \to \{0,1\}^L$ to encrypt m? There are multiple modes of operation: ECB, CBC, CTR, GCM, CCM.

2.1.2.1 Electronic Codebook (ECB) Mode

Suppose that the plaintext message is $m = m_1, m_2, \ldots, m_t$. ECB mode encrypts the blocks **independently**, one at a time. The drawback is that identical plaintext blocks result in identical ciphertext blocks (under the same key k).

2.1.2.2 Cipher Block Chaining (CBC) Mode

For encryption, select $c_0 \in_R \{0,1\}^L$ (c_0 is a random non-secret IV), and then compute

$$c_i = E_k(m_i \oplus c_{i-1}), \quad \text{for } i = 1, 2, \ldots, t$$

The ciphertext is $(c_0, c_1, c_2, \ldots, c_t)$.

For decryption, compute

$$m_i = E_k^{-1}(c_i) \oplus c_{i-1}, \quad \text{for } i = 1, 2, \ldots, t$$

Therefore, identical plaintext with different IVs result in different ciphertexts, and for this reason CBC encryption is semantically secure against chosen-plaintext attacks (assuming that the block cipher E is secure).

3 Week 3 (Jan 22 - 26)

3.1 Hash Functions

Hash functions play a fundamental role in cryptography, they are used in a variety of cryptographic primitives and protocols. However, they are very difficult to design because of stringent security and performance requirements. The most commonly used hash functions are

- SHA-1
- SHA-2 family: SHA-224, SHA-256, SHA-384, SHA-512
- SHA-3 family

3.1.1 What is a Hash Function?

A hash function is a function that takes in an input of variable length, and outputs a message with fixed length (e.g. 256 bits). For example,

$$\text{SHA-256} : \{0,1\}^* \to \{0,1\}^{256}$$

And note that a single change in bit in the input should result in a completely different output. For example, the inputs "Hello there" and "Hello There" will result in completely different outputs.

Definition 3.1 – Hash Function
A hash function is a **mapping** H such that:

1. H maps binary messages of arbitrary lengths $\leq L$ to outputs of a fixed length n:

   $$H : \{0,1\}^{\leq L} \to \{0,1\}^n$$

   where L is usually large, e.g. $L = 2^{64}$, whereas n is small, e.g. $n = 256$.

2. $H(x)$ can be efficiently computed for all $x \in \{0,1\}^{\leq L}$

H is called a n-bit hash function. $H(x)$ is called the **hash** or **message digest** of x. Note that the description of a hash function is **public**, so there are no secret keys. For simplicity, we will usually write $\{0,1\}^*$ instead of $\{0,1\}^{\leq L}$. More generally, a hash function is an efficiently computable function from a set S to a set T. Here is an example of a toy hash function:

x	$H(x)$	x	$H(x)$	x	$H(x)$	x	$H(x)$
0	00	1	01				
00	11	01	01	10	01	11	00
000	00	001	10	010	11	011	11
100	11	101	01	110	01	011	10
0000	00	0001	11	0010	11	0011	00
0100	01	0101	10	0110	10	0111	01
1000	11	1001	01	1010	00	1011	01
1100	10	1101	00	1110	00	1111	11

For this hash function,

- since both 00 and 1000 maps to 11, $(00, 1000)$ is a **collision**.
- since 1001 maps to 01, 1001 is a **preimage** of 01.
- since 10 maps to 01 as well, 10 is a second **preimage** of 1011.

3.1.2 Hash Functions from Block Ciphers

We define the Davies-Meyer hash function. Let E_k be an m-bit block cipher with n-bit key k. Let IV be a fixed m-bit initializing value.

Algorithm 7 Davies-Meyer Hash Function

Input: E_k (block cipher), IV (initializing value), x (message)

1: Break up $x\|1$ into n-bit blocks, so $\bar{x} = x_1, x_2, \ldots, x_t$, padding the last block with 0's as necessary
2: $H_0 \leftarrow IV$
3: **for** $i = 1$ to t **do**
4: $\quad H_i \leftarrow E_{x_i}(H_{i-1}) \oplus H_{i-1}$
5: **end for**
6: $H(x) \leftarrow H_t$
7: **return** $H(x)$

Output: $H(x)$

3.1.3 Preimage Resistance (PR)

A hash function $H : \{0,1\}^* \to \{0,1\}^n$ is **preimage resistant** if given a hash value $y \in_R \{0,1\}^n$, it is **computationally infeasible** to find any $x \in \{0,1\}^*$ with $H(x) = y$.

That is, given y, it is almost impossible to find the preimage. An application of this is for password protection on a multi-user computer system.

1. The server stores $[userID, H(\text{password})]$ in a password file.
2. When a user logs in, the server checks if $H(\text{input})$ matches $H(\text{password})$.
3. If an attacker obtains a copy of the password file, she does not learn any passwords.
4. This application requires preimage resistance.

3.1.4 2nd Preimage Resistance (2PR)

A hash function $H : \{0,1\}^* \to \{0,1\}^n$ is **2nd preimage resistant** if given $x \in_R \{0,1\}^*$, it is computationally infeasible to find another $x' \in \{0,1\}^*$, where $x \neq x'$, such that $H(x) = H(x')$.

That is, it is impossible to find another string that has the same hash value. An application of this is Modification Detection Codes (MDCs):

1. Suppose there is some code on the internet.
2. A user downloads the code along with the hash value.
3. Prior to running the code, the user computes the hash value of the code and checks if it matches the hash value provided.
4. If the hash values are different, then the code has been tampered with.

3.1.5 Breaking PR and 2PR

Therefore, to show that a hash function is not PR, given $y \in_R \{0,1\}^n$, we need to find $x \in \{0,1\}^*$ such that $H(x) = y$.

To show that a hash function is not 2PR, given $x \in_R \{0,1\}^*$, we need to find $x' \in \{0,1\}^*$ such that $H(x) = H(x')$ and $x \neq x'$.

3.1.6 Collision Resistance (CR)

A hash function $H : \{0,1\}^* \to \{0,1\}^n$ is **collision resistant** if it is computationally infeasible to find $x, x' \in \{0,1\}^*$ with $x \neq x'$ such that $H(x) = H(x')$. Such a pair (x, x') is called a **collision**.

That is, it is impossible to find two different strings that have the same hash value. An application of this is digital signatures:

1. For reasons of efficiency, instead of signing a long message x, the much shorter message digest $H(x)$ is signed.
2. This application requires preimage-resistance, 2nd preimage-resistance, and collision resistance.
3. To see why collision resistance is required, suppose that the legitimate signer can find a collision (x, x') such that $H(x) = H(x')$. Alice can sign x_1, then later claim that she signed x_2 as well.

3.1.7 Summary of Cryptographic Requirements

Definition 3.2 – Preimage Resistant
A hash function $H : \{0,1\}^* \to \{0,1\}^n$ is **preimage resistant** if given $y \in_R \{0,1\}^n$, it is computationally infeasible to find any $x \in \{0,1\}^*$ with $H(x) = y$.

Definition 3.3 – 2nd Preimage Resistant
A hash function $H : \{0,1\}^* \to \{0,1\}^n$ is **2nd preimage resistant** if given $x \in_R \{0,1\}^*$, it is computationally infeasible to find another $x' \in \{0,1\}^*$, where $x \neq x'$, such that $H(x) = H(x')$.

Definition 3.4 – Collision Resistant
A hash function $H : \{0,1\}^* \to \{0,1\}^n$ is **collision resistant** if it is computationally infeasible to find $x, x' \in \{0,1\}^*$ with $x \neq x'$ such that $H(x) = H(x')$.

3.1.8 Implications

Claim 3.1
If a hash function H is collision resistant, then H is 2nd preimage resistant.

Proof. We will prove the contrapositive, so suppose that H is not 2PR, we will show that H is not CR.

We select $x \in_R \{0,1\}^*$. Since H is not 2PR, we can efficiently find $x' \in \{0,1\}^*$ such that $H(x) = H(x')$ and $x \neq x'$. Thus, (x, x') is a collision we have efficiently found, showing that H is not CR. □

Claim 3.2
CR does not imply PR

Proof. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is CR. Consider the hash function $\bar{H} : \{0,1\}^* \to \{0,1\}^{n+1}$ defined by

$$\bar{H}(x) = \begin{cases} 0 \| H(x) & \text{if } x \notin \{0,1\}^n \\ 1 \| x & \text{if } x \in \{0,1\}^n \end{cases}$$

Then $\bar{H}$ is CR. To see this, if $x \notin \{0,1\}^n$, then since H is CR, we have that $\bar{H}$ is CR. If $x \in \{0,1\}^n$, then it is impossible to find a collision with the same hash value since the hash value is just $1 \| x$.

However, $\bar{H}$ is not PR since preimages can be efficiently found for at least half of all $y \in \{0,1\}^{n+1}$, namely the hash values y that begin with 1. Since if we have a hash value y that begins with a 1, then the preimage is just the last n bits of y.
Note: The hash function $\bar{H}$ is rather contrived. For a somewhat uniform hash function, i.e. hash functions for which all hash values have roughly the same number of preimages, CR does imply PR. □

Claim 3.3
Suppose H is somewhat uniform. If H is CR, then H is PR.

Proof. We will prove the contrapositive.
Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is not PR. We will show that H is not CR.

Select $x \in_R \{0,1\}^*$ and compute $y = H(x)$. Since H is not PR, we can efficiently find $x' \in \{0,1\}^*$ such that $H(x') = y$. Since H is somewhat uniform, we can expect $x \neq x'$ with very high probability. Thus, (x, x') is a collision for H that we have efficiently found, so H is not CR. □

Claim 3.4
PR does not imply 2PR

Proof. Suppose $H : \{0,1\}^* \to \{0,1\}^n$ is PR. We define $\bar{H} : \{0,1\}^* \to \{0,1\}^n$ by

$$\bar{H}(x_1, x_2, \ldots, x_t) = H(0, x_2, x_3, \ldots, x_t)$$

for all $(x_1, x_2, \ldots, x_t) \in \{0,1\}^*$. Then, $\bar{H}$ is PR since H is PR ($\bar{H}$ just calls H). However, $\bar{H}$ is not 2PR since

$$\bar{H}(0, x_2, x_3, \ldots, x_t) = \bar{H}(1, x_2, x_3, \ldots, x_t)$$

□

Claim 3.5
Suppose H is somewhat uniform. If H is 2PR, then H is PR.

Proof. We will prove the contrapositive.
Suppose $H : \{0,1\}^* \to \{0,1\}^n$ is not PR. We will show that H is not 2PR.

Suppose we are given $x \in_R \{0,1\}^*$. We compute $y = H(x)$ and then efficiently find $x' \in \{0,1\}^*$ such that $H(x') = y$ (since H is not PR). Since H is somewhat uniform, we expect that $x \neq x'$ with very high probability. Thus, x' is a second preimage of x that we have efficiently found, so H is not 2PR. □

Claim 3.6
2PR does not imply CR

Proof. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is 2PR. We will show that H is not CR. Consider $\bar{H} : \{0,1\}^* \to \{0,1\}^n$ defined by

$$\bar{H}(x) = \begin{cases} H(x) & \text{if } x \neq 1 \\ H(0) & \text{if } x = 1 \end{cases}$$

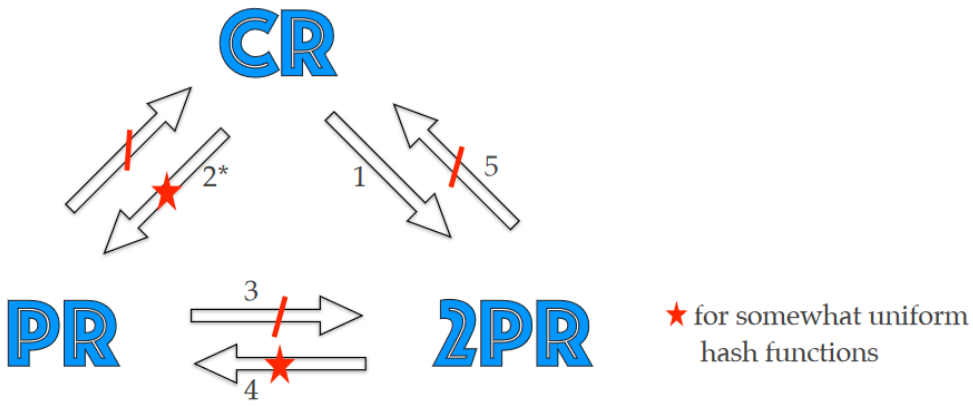
Then $\bar{H}$ is not CR, since $(0,1)$ is a collision for $\bar{H}$.

Suppose now that $\bar{H} : \{0,1\}^* \to \{0,1\}^n$ is not 2PR. We will show that H is not 2PR. Suppose we are given $x \in_R \{0,1\}^*$. Since $\bar{H}$ is not 2PR, we can efficiently find $x' \in \{0,1\}^*$ such that $\bar{H}(x) = \bar{H}(x')$ and $x \neq x'$. We can also assume that $x \neq 0, 1$ with very high probability. Hence, $\bar{H}(x) = H(x)$.

Now, if $x' \neq 1$, then $\bar{H}(x') = H(x') = \bar{H}(x) = H(x)$.
If $x' = 1$, then $\bar{H}(x') = \bar{H}(1) = H(0) = H(x)$.

In either case, we have efficiently found a second preimage x for x with respect to H. Hence, H is not 2PR, a contradiction. Thus, $\bar{H}$ is 2PR. □

Here is a summary of the implications:



Since CR implies both PR and 2PR, it is the hardest to achieve. Since PR does not imply 2PR or CR, it is the easiest to achieve.

3.1.9 Generic Attacks

> **Definition 3.5 – Generic Attack**
> A generic attack on hash functions $H : \{0,1\}^* \to \{0,1\}^n$ does not exploit any properties that the specific hash function might have.

In the analysis of a generic attack, we view H as a random function in the sense that for each $x \in \{0,1\}^*$, the hash value $y = H(x)$ was defined by selecting $y \in_R \{0,1\}^n$.

From a security point of view, a random function is an ideal hash function. However, random functions are not suitable for practical applications because they cannot be compactly described.

3.1.9.1 Generic Attacks for Finding Preimages

Given $y \in_R \{0,1\}^n$, repeatedly select arbitrary $x \in \{0,1\}^*$ until $H(x) = y$. The expected number of hash operations is 2^n.

This generic attack is infeasible if $n \geq 128$. However, it has been proven that this generic attack for finding preimages is optimal, i.e. no faster generic attack exists. Of course, for a specific hash function, there might exist a faster preimage finding algorithm.

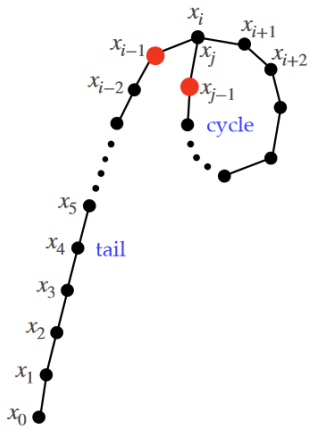
3.1.9.2 Generic Attack for Finding Collisions

Select arbitrary $x \in \{0,1\}^*$ and store $(H(x), x)$ in a table sorted by first entry. Repeat until a collision is found. By the birthday paradox, the expected number of hash operations is $\sqrt{\pi \frac{2^n}{2}} \approx \sqrt{2^n}$.

This generic attack is infeasible if $n \geq 256$. It has been proven that this generic attack for finding collisions is optimal, i.e. no faster generic attacks exists. However, the expected space required is also $\sqrt{\pi \frac{2^n}{2}} \approx \sqrt{2^n}$. Therefore, if $n = 128$, the expected running time is 2^{64} (feasible), but the expected space required is 5×10^8 Terabyes (infeasible).

3.1.10 VW Parallel Collision Search

The VW parallel collision search has expected number of hash operations $\approx \sqrt{2^n}$, and the expected space required is negligible. It is also easy to parallelize, meaning that there is a m-fold speedup with m processors. The VW collision-finding algorithm can easily be modified to find meaningful collisions. The idea is as follows:



- Problem: Find a collision for $H : \{0,1\}^* \to \{0,1\}^n$
- Assumption: H is a random function
- Notation: Let $N = 2^n$
- Idea:

1. Define a sequence $\{x_i\}_{i \geq 0}$ by $x_0 \in_R \{0,1\}^n$ and $x_i = H(x_{i-1})$ for $i \geq 1$.
2. Let j be the smallest index for which $x_j = x_i$ for some $i < j$ such a j must exist since there are only N possible values for x_i.
3. Then, $x_{j+\ell} = x_{i+\ell}$ for all $\ell \geq 1$. By the birthday paradox, the expected length of the tail and the cycle is

$$E[j] \approx \sqrt{\pi \frac{N}{2}} \approx \sqrt{N}$$

The expected length of the tail is

$$E[i] \approx \frac{1}{2}\sqrt{N}$$

The expected length of the cycle is

$$E[j-i] \approx \frac{1}{2}\sqrt{N}$$

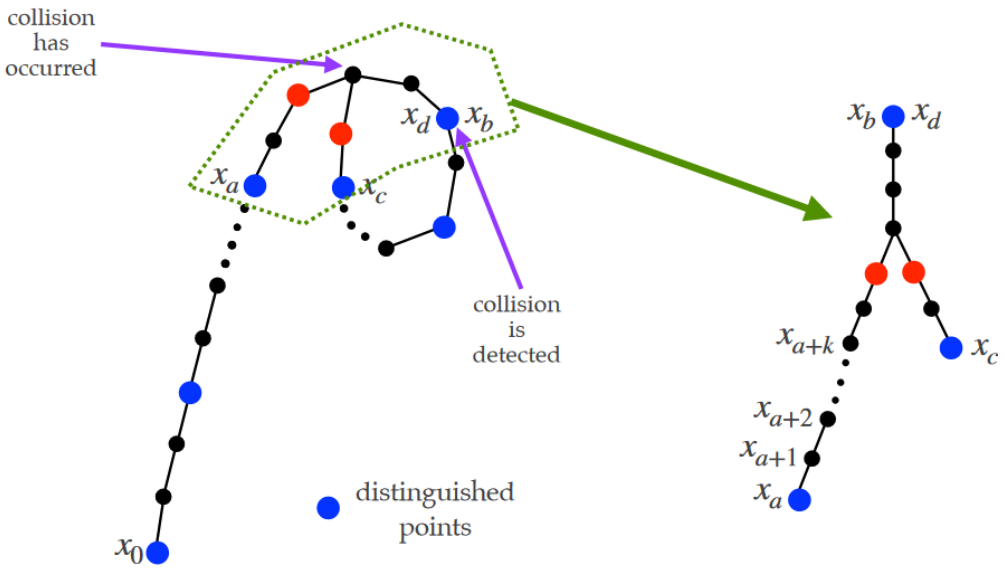
4. Now, $i \neq 0$ with overwhelming probability, in which event (x_{i-1}, x_{j-1}) is a collision for H.

Now, how to find (x_{i-1}, x_{j-1}) without using much storage? We only store **distinguished points**. We select an easily-testable distinguishing property for elements of $\{0,1\}^n$, for example, the points with leading 32 bits all 0s.

Let θ be the proportion of elements of $\{0,1\}^n$ that are distinguished. In the case where all leading 32 bits are all 0s, this is the probability

$$\theta = \frac{1}{2^{32}}$$

Then, we only store the points that are distinguished.



Algorithm 8 VW Collision Finding State 1: Detecting a collision

Select $x_0 \in_R \{0,1\}^n$
Store $(x_0, 0, -)$ in a sorted table, where the second entry is the index, and the third entry is the previous point.
$LP \leftarrow x_0$ ▷ Last point
for $d = 1, 2, \ldots$ **do**
    Compute $x_d = H(x_{d-1})$
    if x_d is distinguished **then**
        if x_d is already in the table, say $x_d = x_b$, where $b < d$ **then**
            go to Stage 2
        else
            Store (x_d, d, LP) in the table
        end if
    end if
    $LP \leftarrow x_d$
end for

Algorithm 9 VW Collision Finding State 2: Finding the collision

$\ell_1 \leftarrow b - a$ ▷ a is the previous distinguished point before x_d, following the path of the tail
$\ell_2 \leftarrow d - c$ ▷ c is the previous distinguished point before x_d, following the path of the cycle
Suppose $\ell_1 \geq \ell_2$, and set $k \leftarrow \ell_1 - \ell_2$
Compute $x_{a+1}, x_{a+2}, \ldots, x_{a+k}$
for $m = 1, 2, 3, \ldots$ **do**
    Compute (x_{a+k+m}, x_{c+m})
    if $x_{a+k+m} = x_{c+m}$ **then**
        Collision found, **return** $(x_{a+k+m-1}, x_{c+m-1})$
    end if
end for

4 Week 4 (Jan 29 - Feb 2)

4.1 Hash Functions (Continued)

4.1.1 VW Analysis

For stage 1, the expected number of H-evaluations is

$$\underbrace{\sqrt{\pi \frac{N}{2}}}_{\text{collision occured}} + \underbrace{\frac{1}{\theta}}_{\text{collision detected}} \approx \sqrt{N} + \frac{1}{\theta}$$

For state 2, the expected number of H-evaluations is

$$\leq \frac{3}{\theta}$$

Therefore, the overall expected running time is

$$\sqrt{N} + \frac{4}{\theta}$$

The expected storage is

$$\approx 3n\theta\sqrt{N} \text{ bits}$$

since each table entry is (x_d, d, LP), so this has bit length $3n$.

For example, consider $n = 128$. Take $\theta = \frac{1}{2^{32}}$. Then the expected running time of the VW collision search is 2^{64} H-evaluations (feasible), and the expected storage is 192 GB (negligible).

4.1.2 Parallelizing VW Collision Search

We can run independent copies of VW on each of m processors, then each processor reports distinguished points to a central server. The resulting running time is

$$\approx \frac{1}{m}\sqrt{N} + \frac{4}{\theta}$$

The expected storage is still the same at

$$\approx 3n\theta\sqrt{N} \text{ bits}$$

This results in a factor-m speedup. There is no communications required between processors, and only occasional communications with the central server.

4.1.3 Iterated Hash Functions (Merkle's Meta Method)

We introduce the Merkle's meta method. The components are

- Fixed initializing value $IV \in \{0,1\}^n$
- Efficiently computable compression function $f : 0,1^{n+r} \to \{0,1\}^n$

Algorithm 10 Merkle's Meta Method

Input: x where x has bitlength $b < 2^r$
Output: $H(x)$

1: Break up x into r-bit blocks: $\bar{x} \leftarrow x_1, x_2, \ldots, x_t$, padding the last block with 0 if necessary
2: $x_{t+1} \leftarrow$ right-justified binary representation of b, the length of x.
3: $H_0 \leftarrow IV$
4: **for** $i = 1$ to $t + 1$ **do**
5: $\quad H_i \leftarrow f(H_{i-1}, x_i)$ ▷ The H_i's are called chaining variables
6: **end for**
7: **return** H_{t+1}

Theorem 4.1 – Merkle's Theorem
If the compression function f is collision resistant, then the iterated hash function H is also collision resistant.

Therefore, Merkle's theorem reduces the problem of designing collision-resistant hash functions to that of designing collision-resistant compression functions, which have a fixed length as input.

Proof of Merkle's Theorem. We prove the contrapositive.
Suppose that H is not CR. We will show that f is not CR.

Since H is not CR, we can efficiently find messages $x, x' \in \{0,1\}^*$, with $x \neq x'$ and $H(x) = H(x')$. Let

$$\bar{x} = x_1, x_2, \ldots, x_t, \quad b = \text{bitlength}(x), \quad x_{t+1} = \text{length block}$$

$$\bar{x}' = x'_1, x'_2, \ldots, x'_t, \quad b' = \text{bitlength}(x'), \quad x'_{t+1} = \text{length block}$$

We can then efficiently compute

$$\begin{aligned} H_0 &= IV \\ H_1 &= f(H_0, x_1) \\ H_2 &= f(H_1, x_2) \\ &\vdots \\ H_{t-1} &= f(H_{t-2}, x_{t-1}) \\ H_t &= f(H_{t-1}, x_t) \\ H(x) = H_{t+1} &= f(H_t, x_{t+1}) \end{aligned} \qquad \begin{aligned} H'_0 &= IV \\ H'_1 &= f(H'_0, x'_1) \\ H'_2 &= f(H'_1, x'_2) \\ &\vdots \\ H'_{t'-1} &= f(H'_{t'-2}, x'_{t'-1}) \\ H'_{t'} &= f(H'_{t'-1}, x'_{t'}) \\ H(x') = H'_{t'+1} &= f(H'_{t'}, x'_{t'+1}) \end{aligned}$$

Since $H(x) = H(x')$, we have $H_{t+1} = H'_{t'+1}$. We now consider cases:

- Case 1: $b \neq b'$. Then $x_{t+1} \neq x'_{t'+1}$. Thus, $(H_t, x_{t+1}), (H'_{t'}, x'_{t'+1})$ is a collision for f that we have efficiently found.
- Case 2: $b = b'$. Then $t = t'$ and $x_{t+1} = x'_{t'+1}$.
  - Let i be the largest index, $0 \leq i \leq t$ for which $(H_i, x_{i+1}) = (H'_i, x'_{i+1})$. Such an i must exist since $x \neq x'$. In this step, we are following the above equations in reverse order.
  - Then
    $$H_{i+1} = f(H_i, x_{i+1}) = f(H'_i, x'_{i+1}) = H'_{i+1}$$
    So $(H_i, x_{i+1}), (H'_i, x'_{i+1})$ is a collision for f that we have efficiently found.

Thus, f is not collision resistant. □

4.1.4 MDx-family of Hash Functions

MDx is a family of iterated hash functions. MD4 has 128-bit outputs. However, it was discovered that collisions for MD4 can be found by hand. Moreover, an algorithm for finding MD4 preimages can be done in 2^{102} operations.

MD5 is a strengthened version of MD4, it also has 128-bit outputs. However, it was discovered that collisions for MD5 can be found in 2^{24} operations. Preimages can be found in $2^{123.4}$ operations.

4.1.5 SHA-1

SHA is a 160-bit iterated hash function, based on MD4. It was slightly modified to SHA-1. However, collisions can be found in 2^{63} operations.

4.1.6 SHA-2 Family

This is a variable output-length version of SHA-1. Output lengths are

- 224 bits (SHA-224 and SHA-512/224)
- 256 bits (SHA-256 and SHA-512/256)
- 384 bits (SHA-384)
- 512 bits (SHA-512)

The security levels of these hash functions against VW collision finding attacks are the same as the security levels of Triple-DES, AES, AES-192, and AES-256 against exhaustive key search.

$H : \{0,1\}^* \to \{0,1\}^n$	Output length n	Security Level (generic) (bits)	Security Level (nongeneric) (bits)
MD4 (1990)	128	64	4
MD5 (1991)	128	64	39
SHA (1993)	160	80	39
SHA-1 (1994)	160	80	63
SHA-224 (2001)	224	112	112
SHA-256 (2001)	256	128	128
SHA-384 (2001)	384	192	192
SHA-512 (2001)	512	256	256

4.1.7 SHA-256

Details can be found in Slides 3.45-3.49

4.2 Message Authentication Codes

A **message authentication code** (MAC) scheme is a family of functions

$$\text{MAC}_k : \{0,1\}^* \to \{0,1\}^n$$

parameterized by an ℓ-bit key k, where each function MAC_k can be efficiently computed.

$t = \text{MAC}_k(x)$ is called a MAC or tag of x with key k. MAC schemes are used for providing (symmetric-key) data integrity and data origin authentication That is, if x is received with tag t, then the receiver can verify that x has not been altered and that x was sent by the sender who knows the key k. The difference between MAC, hash functions, and encryption functions is that MACs require a key (unlike hash functions), but MACs do not have to be invertible (unlike encryption functions).

Here is an example of application of MAC schemes:

1. Alice wants to send a message x to Bob.
2. Alice and Bob establishes a secret key $k \in \{0,1\}^\ell$.
3. Alice computes the tag $t = \text{MAC}_k(x)$ and sends (x, t) to Bob.
4. Bob verifies that $t = \text{MAC}_k(x)$.

However, there is no confidentiality (since x is sent using an unsecured channel) or non-repudiation (no way to prove that Alice sent x).

4.2.1 Breaking MAC Schemes

Let k be the secret key shared by Alice and Bob. The adversary does not know k, but is allowed to obtain (from either Alice or Bob) the tags for messages of her choosing. The adversary's goal is to obtain the tag of a new message of the adversary's choosing, i.e. a message whose tag she did not already obtain from Alice or Bob.

Definition 4.1 – Security of MAC Schemes
A MAC scheme is secure if given some tags $\text{MAC}_k(x_i)$ for messages x_i's of one's own choosing, it is computationally infeasible to compute a message-pair tag $(x, \text{MAC}_k(x))$ for a new message x.

That is, the adversary cannot come up with a tag that is valid for a new message.

To break a MAC scheme, the adversary is given a MAC oracle, which is a black box that computes tags for messages of the adversary's choosing. The adversary's goal is to compute a new message-tag pair $(x, \text{MAC}_k(x))$, where x is a message that has not been in the oracle.

4.2.2 Ideal MAC Scheme

An ideal MAC scheme has the following property, for each key $k \in \{0,1\}^\ell$, the function $\text{MAC}_k : \{0,1\}^* \to \{0,1\}^n$ is a random function. However, ideal MAC schemes are useless in practice, but when we analyze generic attacks on MAC schemes, it is reasonable to assume that the MAC scheme is ideal.

4.2.3 Generic Attacks on MAC Schemes

The first generic attack is by guessing the tag of a message $x \in \{0,1\}^*$. That is, the adversary selects $y \in_R \{0,1\}^n$ and guess that $y = \text{MAC}_k(x)$. Assuming that the MAC scheme is ideal, the success probability is $\frac{1}{2^n}$. Note that guesses cannot be directly checked. MAC tag guessing is infeasible if $n \geq 128$.

The second generic attack is an exhaustive search on the key space. Given r message-tag pairs $(x_1, t_1), \ldots, (x_r, t_r)$, the adversary can check whether a guess h of the key is correct by verifying that $t_i = \text{MAC}_h(x_i)$ for $i = 1, \ldots, r$.

Assuming that the MAC scheme is ideal, the expected number of keys for which all (x_i, t_i) pairs verify is

$$1 + \underbrace{\frac{2^\ell - 1}{2^{nr}}}_{FK}$$

For example, if $\ell = 128, n = 128, r = 2$, then the expected number of false keys is

$$FK \approx \frac{1}{2^{128}}$$

Assuming that FK is negligible, the expected number of operations is $\approx 2^{\ell-1}$. Exhaustive key search is infeasible if $\ell \geq 128$.

4.2.4 MACs Based on Block Ciphers

We introduce CBC-MAC. Let E be an n-bit block cipher with key space $\{0,1\}^\ell$. We assume that plaintext messages all have lengths that are multiples of n.

Algorithm 11 CBC-MAC

Input: Plaintext x, key k **Output:** $\text{CBC-MAC}_k(x)$

1: Divide x into n-bit blocks: $x = x_1, x_2, \ldots, x_r$
2: $H_1 \leftarrow E_k(x_1)$
3: **for** $i = 2$ to r **do**
4: $\quad H_i \leftarrow E_k(H_{i-1} \oplus x_i)$
5: **end for**
6: **return** H_r

The security of CBC-MAC has been formally proven: Suppose that E is an ideal encryption scheme. Then CBC-MAC with **fixed-length** messages is a secure MAC scheme (Recall that an encryption scheme E is ideal if for each $k \in \{0,1\}^\ell$, $E_k : \{0,1\}^n \to \{0,1\}^n$ is a random permutation).

However, CBC-MAC is not secure if **variable-length** messages are allowed. Here is a chosen-message attack on CBC-MAC:

Algorithm 12 Chosen attack on CBC-MAC with variable length messages

Output: A forgery (x, t)

1: Select an arbitrary 3-block message $x = (x_1, x_2, x_3)$
2: Obtain the tag t_1 of the one-block message x_1, so $t_1 = E_k(x_1)$
3: Obtain the tag t_2 of the two-block message $(t_1 \oplus x_2, x_3)$, so $t_2 = E_k(E_k(t_1 + x_2) \oplus x_3)$
4: Output the forgery (x, t_2)

This is a valid attack because the adversary obtains a tag of a message that has not been through the oracle. We can check for correctness:

$$t_2 = E_k(E_k(t_1 + x_2) \oplus x_3) = E_k(E_k(E_k(x_1) + x_2) \oplus x_3) = \text{CBC-MAC}_k(x)$$

4.2.5 Encrypted CBC-MAC (EMAC)

One countermeasure for variable-length messages is **Encrypted CBC-MAC**, where the CBC-MAC tag is encrypted using a second key s, so

$$\text{EMAC}_{k,s}(x) = E_s(\text{CBC-MAC}_k(x))$$

The security of EMAC has been formally proven: Suppose that E is an ideal encryption scheme. Then EMAC with **variable-length** messages is a secure MAC scheme for messages of any length.

4.2.6 MACs Based on Hash Functions

Hash functions were not originally designed for message authentication. In particular, they do not require a key. Then, how to use a hash function to construct a secure MAC?

Here is one attempt. Let H be an iterated n-bit hash function. For simplicity, assume that all messages have bit lengths that are multiples of r, and suppose that the length-block is omitted. Let $n + r$ be the input block length of the compression function

$$f : \{0,1\}^{n+r} \to \{0,1\}^n$$

So for SHA-256, $n = 256$ and $r = 512$. Let $k \in_R \{0,1\}^n$. Let K denote k padded with $r - n$ 0's, so K has bit length r. Then we define

$$\text{MAC}_k(x) = H(K, x)$$

However, this is insecure. Here is a length extension attack:

Algorithm 13 Length extension attack on hash-based MAC

Input: Known plaintext tag pair (x, t), where $x = x_1$ (one block message)

1: We already know $t = f(f(IV, k), x_1)$
2: Select one-block message y
3: Compute $t' = f(t, y)$ ▷ This works because $t' = f(\underbrace{f(f(IV, k), x_1)}_{=t(\text{known})}, y)$
4: Output the forgery $(x \parallel y, t')$

Now, here is another hash based MAC, called **HMAC**. We define two r-bit strings:

- ipad = 0x36 repeated $\frac{r}{8}$ times

- opad = 0x5C repeated $\frac{r}{8}$ times

Let K denote k padded with $r - n$ 0's, so K has bit length r. Then

$$\text{HMAC}_k(x) = H((k \oplus \text{opad}), H((K \oplus \text{ipad}), x))$$

5 Week 5 (Feb 5 - 9)

5.1 Message Authentication Codes (Continued)

5.1.1 HMAC Based on Hash Functions (Continued)

The security analysis is rigorous. The informal statement of the theorem is: Suppose that the compression function f used in H is a secure MAC with fixed-length messages and a secret IV as the key. Then HMAC is a secure MAC scheme.

In practice, it is recommend that HMAC uses the hash function $H = \text{SHA-256}$.

5.1.2 Key Derivation Functions

HMAC is also commonly used as a **key derivation function** (KDF). Suppose that Alice has a secret key k, and wishes to derive several session keys sk_i, e.g. to encrypt data in different communication sessions. Alice then uses her secret key k to compute the session keys sk_i:

$$sk_1 = \text{HMAC}_k(1), \quad sk_2 = \text{HMAC}_k(2), \quad sk_3 = \text{HMAC}_k(3), \quad \ldots$$

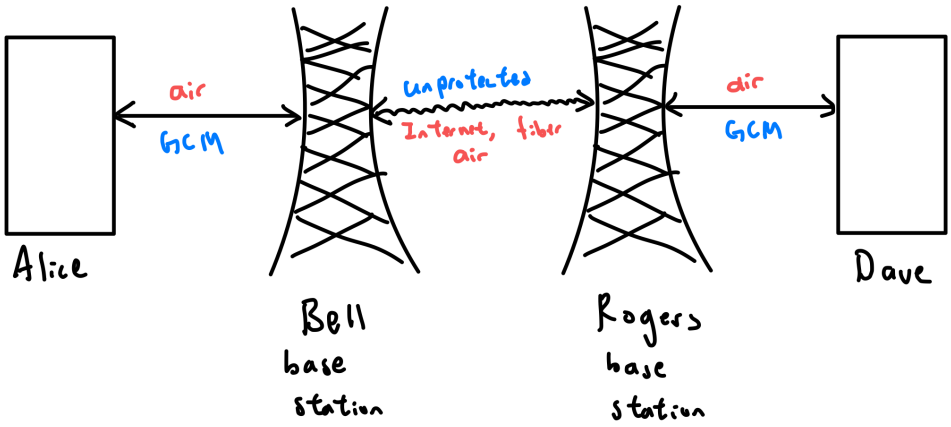
The rationale for this is that without knowledge of k, an adversary is unable to learn anything about the particular session key sk_i, even though she might have learnt some other session keys.

5.1.3 GSM

GSM is the global standard for mobile communications, it is used in 2G and 2.5G. GSM security only uses symmetric-key primitives. The objectives of GSM security are:

1. Entity authentication: The cell phone provider needs the assurance that entities accessing its service are legitimate subscribers. That is, a cell phone provider does not want to provide service to unauthorized users.
2. Confidentiality: The data exchanged between a cell phone user and their cell phone service provider should be confidential.

Note that GSM does not provide end-to-end security (confidentiality of the conversation between two cell phone users). The authentication is only one way, the phone authenticates itself to the base station, but the base station does not authenticate itself to the phone.



As we can see in this diagram, the signal between base stations are not protected. Thus, communication between two cell phones are not end-to-end encrypted.

GSM uses the following cryptographic ingredients:

- Enc: A symmetric-key encryption scheme.
- MAC: A symmetric-key MAC scheme
- KDF: A key derivation function

This is the setup:

- A SIM card manufacturer (such as Gemalto) randomly selects a secret key k, and installs it in a SIM card. A copy of k is given to the cell phone service provider (so a different key k is chosen for each user).
- When a user subscribes to a cell phone provider, she gets the SIM card which she installs in her phone.

Algorithm 14 GCM Initialization

Input: Alice, a cell phone user, and Bob, a cell phone service provider.

1: Alice sends an **authentication request** to Bob.
2: Bob selects a **challenge** $r \in_R \{0,1\}^{128}$ and sends r to Alice.
3: Alice's SIM card uses k to compute the response $t = \text{MAC}_k(r)$ and sends it to Bob.
4: Bob retrieves Alice's key k from its database, and verifies that $t = \text{MAC}_k(r)$.
5: Alice and Bob computes an **encryption key** $K_E = \text{KDF}_k(r)$, and thereafter use the encryption algorithm $\text{Enc}_{K_E}(\cdot)$ to encrypt and decrypt messages for each other for the remainder of the session.

One drawback of using only symmetric-key cryptography is that the SIM card manufacturer and the cell phone service providers have to securely maintain a large database of SIM keys k. If an adversary were to obtain a copy of this database, then the adversary can impersonate any cell phone user. This is a significant security risk.

5.1.4 Authenticated Encryption

Recall that confidentiality means that an adversary should not be able to learn anything about the plaintext from the ciphertext. Authentication includes

- data origin authentication: The receiver should be able to verify that the message was indeed sent by the sender, and
- data integrity: The receiver should be able to verify that the message has not been tampered with.

Confidentiality is achieved using a symmetric-key encryption scheme E, for example

$$E = \text{AES-CBC}$$

Authentication is achieved using a MAC scheme, for example HMAC.

Now, what if confidentiality and authentication are both required?

5.1.5 Encrypt-and-MAC

Algorithm 15 Encrypt-and-MAC

Description: Alice wants to send plaintext m to Bob.
Input: Plaintext m, keys k_1, k_2.

1: $c \leftarrow E_{k_1}(m)$
2: $t \leftarrow \text{MAC}_{k_2}(c)$
3: Alice sends (c, t) to Bob.
4: Bob decrypts c by doing $m \leftarrow E_{k_1}^{-1}(c)$.
5: Bob verifies the MAC by checking if $\text{MAC}_{k_2}(c) = t$.

This is not secure. The tag $\text{MAC}_{k_2}(c)$ might leak some information about m.

5.1.6 Encrypt-then-MAC

Algorithm 16 Encrypt-then-MAC

Description: Alice wants to send plaintext m to Bob.
Input: Plaintext m, keys k_1, k_2.

1: $c \leftarrow E_{k_1}(m)$
2: $t \leftarrow \text{MAC}_{k_2}(c)$
3: Alice sends (c, t) to Bob.
4: Bob first verifies that $\text{MAC}_{k_2}(c) = t$.
5: Bob decrypts c by doing $m \leftarrow E_{k_1}^{-1}(c)$.

This generic method has been proven to be secure, provided that the encryption scheme E and the MAC scheme employed are secure.

Definition 5.1 – AE-Secure
An authenticated encryption scheme AE is AE-secure if:

1. AE is semantically secure against chosen-plaintext attack; and
2. AE has ciphertext integrity, i.e. an adversary who is able to obtain ciphertext/tag pairs

$$(c_1, t_1), \quad (c_2, t_2), \quad \ldots, (c_\ell, t_\ell)$$

for plaintext messages $m_1, m_2, \ldots, m_\ell$ of her choosing is unable to produce a valid ciphertext/tag pair (c, t) for a new message m.

5.1.7 Special-purpose Authenticated Encryption Schemes

Many special-purpose authenticated encryption schemes have been developed, the most popular of which is Galois/Counter Mode (GCM). These modes can be faster than generic Encrypt-then-MAC, and also allows for the authentication (but not encryption) of "header" data.

In an internet packet, the payload (message itself) is encrypted and authenticated, while the header (additional information such as origin and destination IP addresses) is only authenticated. This is because the header is needed for routing, and should not be encrypted.

For example, the AES-GCM authenticated encryption scheme proposed by David McGrew and John Viega in 2004. It uses the **CTR** mode of encryption and **GMAC**, a custom-designed MAC scheme.

5.1.8 CTR: Counter mode of encryption

Let $k \in_R \{0,1\}^{128}$ be the secret key shared by Alice and Bob. Let

$$M = (M_1, M_2, \ldots, M_u)$$

be a plaintext message, where each M_i is a 128-bit block and $u \leq 2^{32} - 2$. The encryption algorithm is:

Algorithm 17 CTR Encryption

Input: Plaintext $M = (M_1, M_2, \ldots, M_u)$, secret key k

1: Select a nonce $IV \in \{0,1\}^{96}$
2: Let $J_0 = IV||0^{31}||1$
3: **for** $i = 1, 2, \ldots, u$ **do**
4: $\quad J_i \leftarrow J_{i-1} + 1$
5: $\quad C_i \leftarrow \text{AES}_k(J_i) \oplus M_i$
6: **end for**
7: Send $(IV, C_1, C_2, \ldots, C_u)$ to Bob.

Algorithm 18 CTR Decryption

Input: Secret key k, message $(IV, C_1, C_2, \ldots, C_u)$

1: $J_0 \leftarrow IV||0^{31}||1$
2: **for** $i = 1, 2, \ldots, u$ **do**
3: $\quad J_i \leftarrow J_{i-1} + 1$
4: $\quad M_i \leftarrow \text{AES}_k(J_i) \oplus C_i$
5: **end for**

CTR mode of encryption can be viewed as a steam cipher, notice its similarities of ChaCha20. As with the case with CBC encryptoin, identical plaintexts with different IVs result in different ciphertexts. It is critical that the IV should not be repeated, but this can be difficult to achieve in practice. However, unlike CBC encryption, CTR encryption is parallelizable, since each C_i does not depend on the previous C_{i-1}, it only depends on J_{i-1}. Note that the decryption function AES^{-1} is not used. The secret key can have bitlength 128, 192, or 256.

5.1.9 Multiplying Blocks

Now, before we introduce GMAC, we need to introduce the concept of multiplying blocks. Let

$$a = a_0 a_1 a_2 \ldots a_{127}$$

be a 128-bit block. We can associate the binary polynomial

$$a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{127} x^{127} \in \mathbb{Z}_2[x]$$

with a. Let

$$f(x) = 1 + x + x^2 + x^7 + x^{128}$$

If a and b are 128-bit blocks, then we can define $c = a \cdot b$ to be the block corresponding to the polynomial

$$c(x) = a(x) \cdot b(x) \pmod{f(x)}$$

That is, $c(x)$ is the remainder upon dividing $a(x) \cdot b(x)$ by $f(x)$ in $\mathbb{Z}_2[x]$. This is multiplication in the Galois field $GF(2^{128})$.

5.1.10 Galois Message Authentication Code (GMAC)

Algorithm 19 GMAC

Input: $A = (A_1, A_2, \ldots, A_v)$, where each A_i is a 128-bit block. L, the bitlength of A encoded as a 128-bit block. $k \in_R \{0,1\}^{128}$, the secret key.
Output: (IV, A, t), where IV is the 96-bit nonce, A is the authenticated data, and t is the tag.

1: $J_0 \leftarrow IV||0^{31}||1$, where IV is a 96-bit nonce.
2: $H \leftarrow \text{AES}_k(0^{128})$
3: $f_A(x) \leftarrow A_1 x^{v+1} + A_2 x^V + \cdots + A_{v-1} x^3 + A_v x^2 + Lx$
4: $t \leftarrow \text{AES}_k(J_0) \oplus f_A(H)$
5: Send (IV, A, t) to Bob.

5.1.11 GMAC Security

Recall that the tag is $t = \text{AES}_k(J_0) \oplus f_A(H)$. Consider the simplified tag $t' = f_A(H)$.

An adversary can guess the tag t' of a message A with success probability $\frac{1}{2^{128}}$. She can also guess the tag t' by making a guess H' for H and computing $f_A(H')$. Her success probability is at most $\frac{v+1}{2^{128}}$, where v is the blocklength of A. However, if the adversary sees a single valid message-tag pair (A, t'), then she can solve the polynomial equation

$$f_A(H) = t'$$

for H. Since $f_A(x)$ has degree $v + 1$, there is at most $v + 1$ solutions.

To circumvent this attack, a second secret $\text{AES}_k(J_0)$ is used to hide t':

$$t = \text{AES}_k(J_0) \oplus t'$$

The secret $\text{AES}_k(J_0)$ serves as a one-time pad for t'.

5.1.12 Authenticated Encryption: AES-GCM

Algorithm 20 AES-GCM Encryption

Input:

- Data to be encrypted and authenticated: $M = (M_1, M_2, \ldots, M_u)$, where each M_i is a 128-bit block, and $u \leq 2^{32} - 2$.
- **AAD** (Additional Authenticated Data), also called encryption context, which are data to be authenticated but not encrypted: $A = (A_1, A_2, \ldots, A_v)$, where each A_i is a 128-bit block.
- Secret key $k \in_R \{0,1\}^{128}$, shared between Alice and Bob.

Output: (IV, A, C, t), where

- IV is a 96-bit initialization vector.
- $A = (A_1, A_2, \ldots, A_v)$ is the additional authenticated data.
- $C = (C_1, C_2, \ldots, C_u)$ is the encrypted/authenticated data.
- t is a 128-bit authentication tag.

1: $L \leftarrow L_A, L_M$, where L_A is the bitlength of A and L_M is the bitlength of M expressed as 64-bit integers.
2: $IV \in_R \{0,1\}^{96}$
3: $J_0 \leftarrow IV || 0^{31} || 1$
4: **for** $i = 1, 2, \ldots, u$ **do** ▷ Encryption
5: $\quad J_i \leftarrow J_{i-1} + 1$
6: $\quad C_i \leftarrow \text{AES}_k(J_i) \oplus M_i$
7: **end for**
8: $H \leftarrow \text{AES}_k(0^{128})$ ▷ Authentication
9: $t \leftarrow \text{AES}_k(J_0) \oplus f_{A,C}(H)$ ▷ Authentication
10: **return** (IV, A, C, t)

Note that

$$\begin{aligned} A &= (A_1, A_2, \ldots, A_v) \\ C &= (C_1, C_2, \ldots, C_u) \\ f_{A,C}(x) &= A_1 x^{u+v+2} + \cdots + A_{v-1} x^{u+3} + A_v x^{u+2} + C_1 x^{u+1} + \cdots + C_{u-1} x^3 + C_u x^2 + Lx \\ t &= \text{AES}_k(J_0) \oplus f_{A,C}(H) \end{aligned}$$

Algorithm 21 AES-GCM Decryption

Input: (IV, A, C, t)
Output: (A, M)

$L \leftarrow L_A, L_M$, where L_A is the bitlength of A and L_M is the bitlength of M expressed as 64-bit integers.
$H \leftarrow \text{AES}_k(0^{128})$ ▷ Authentication
$t' \leftarrow \text{AES}_k(J_0) \oplus f_{A,C}(H)$ ▷ Authentication
if $t' \neq t$ **then**
    return Message Rejected
end if
$J_0 \leftarrow IV||0^{31}||1$ ▷ Decryption
for $i = 1, 2, \ldots, u$ **do** ▷ Decryption
    $J_i \leftarrow J_{i-1} + 1$
    $M_i \leftarrow \text{AES}_k(J_i) \oplus C_i$
end for
return (A, M)

Here are some features of AES-GCM:

1. Performs both authentication and encryption.
2. Supports authentication only (by using empty M).
3. Very fast implementations on Intel and AMD processors.
4. Encryption and decryption can be parallelized.
5. AES-GCM can be used in streaming mode.
6. The secret key can have bitlength 128, 192, or 256.
7. Security is justified by a security proof.

5.1.13 Why IV's Should not be Repeated

Suppose that Alice uses an IV twice. She sends

$$(IV, A_1, C_1, t_1), \quad (IV, A_2, C_2, t_2)$$

Suppose C_1, C_2 have the same block length. Then the adversary Eve computes

$$\begin{aligned} C_1 \oplus C_2 &= (M_1 \oplus \text{keystream}) \oplus (M_2 \oplus \text{keystream}) \\ &= M_1 \oplus M_2 \qquad \text{since keystreams are the same} \end{aligned}$$

So, Eve can compute M_1, M_2 and the keystream. This destroys the confidentiality of the messages since it only depends on M_1, M_2. Recall that

$$t_1 = \text{AES}_k(J_0) \oplus f_{A_1,C_1}(H), \quad t_2 = \text{AES}_k(J_0) \oplus f_{A_2,C_2}(H)$$

where $H = \text{AES}_k(0)$. Then

$$\begin{aligned} t_1 \oplus t_2 &= \text{AES}_k(J_0) \oplus f_{A_1,C_1}(H) \oplus \text{AES}_k(J_0) \oplus f_{A_2,C_2}(H) \\ &= f_{A_1,C_1}(H) \oplus f_{A_2,C_2}(H) \end{aligned}$$

Eve solves the polynomial equation

$$f_{A_1,C_1}(x) \oplus f_{A_2,C_2}(x) = t_1 \oplus t_2$$

for x (Fact: this can be done efficiently). So Eve can find H efficiently. Eve can also compute

$$\text{AES}_k(J_0) = t_1 \oplus f_{A_1,C_1}(H)$$

Now, for any plaintext M of the same length as M_1, Eve can compute

$$(IV, A', C, t)$$

which Bob will accept. So Alice and Bob completely lose authentication services.

5.1.14 Encryption in the cloud: AWS

5.1.15 AWS Global Infrastructure

AWS has 33 regions, 105 availiability zones (at least 3 in each region). Each availability zone is comprised of one or more data centres, which are physically separated from each other. There are 600+ edge locations (data centres).

5.1.16 AWS Security

AWS has a lot of high profile customers, such as Netflix, NASA, and the CIA. Security is needed for

1. data in transit
2. data in use
3. data at rest

So they have to encrypt everything, have physical data centre security, and hardware security modules (HSMs).

5.1.16.1 Data Centre Security

Data centres are located in non-descript undisclosed facilities, with secured entrances. There is 24/7 monitoring; there are professional security staffs, video surveillance, intrusion detection, and access log monitoring. They have planning for flooding, extreme weather, seismic activity, fires, power supply, securely destroying old storage devices, etc. All data flowing between data centres is automatically encrypted.

5.1.16.2 Hardware Security Modules (HSMs)

HSMs are expensive, tamper-resistant devices. They store the secret keys of customers. When operational, no AWS operator can access an HSM, e.g. via ssh, and no software updates are allowed. It has very limited API, e.g. generate key, delete key, generate random bytes. An analogy is that it plays the role of the SIM card, so people cannot get the key itself.

After reboot and in a non-operational state, there is no keying material on the HSM. Software can only be updated after multiple AWS employees have reviewed the code, and under quorum of multiple AWS operators with valid credentials.

6 Week 6 (Feb 12 - 16)

6.1 Message Authentication Codes (Continued)

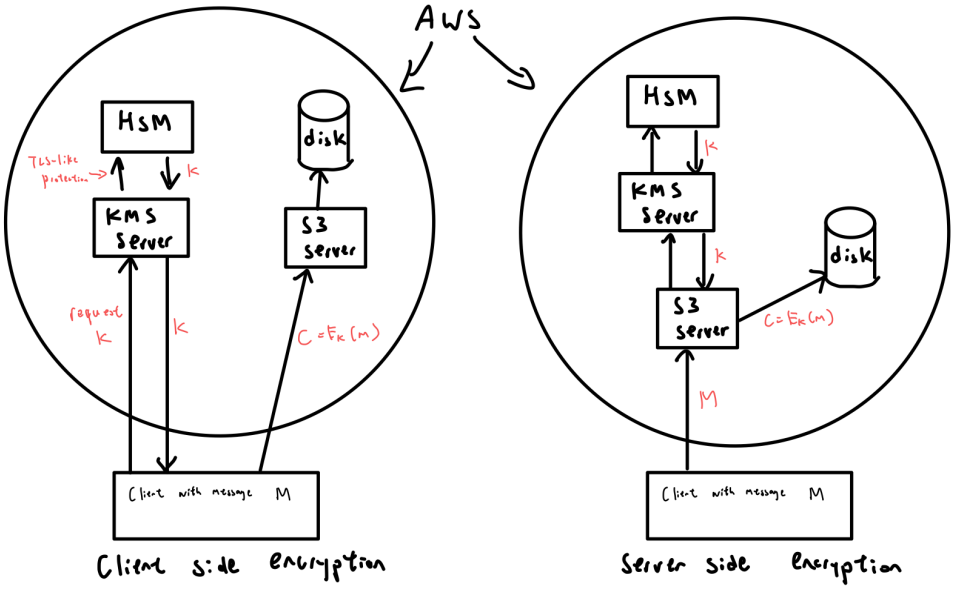
6.1.1 AWS Encryption

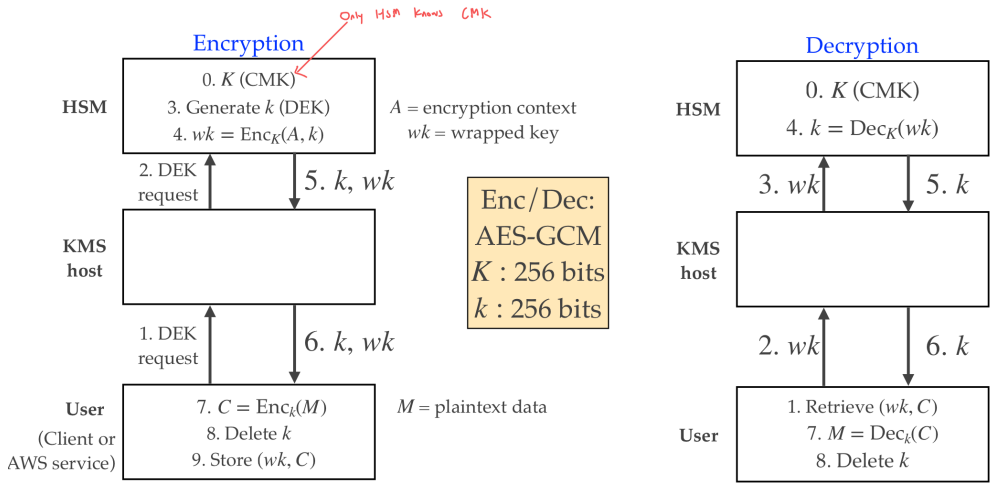
AWS Encryption has the following components:

- KMS: Key Management Service
- Data at rest: all data stored by all AWS services
- Client-side/Server-side encryption, where the AWS KMS generates and manages keys.

Each data item is encrypted with a one-time data encryption key (DEK). Consequently, there are a massive number of keys to manage (generation, storage, retrieval, access control, deletion, etc). This leads naturally to the notion of **envelope encryption**:

- Each client has a **Customer Main Key** (CMK).
- The CMK is used to encrypt/decrypt DEKs.
- DEKs are used to encrypt/decrypt customer data.
- DEKs are deleted after each usage.





For plaintext encryption, we can break a plaintext into chunks, each chunk encrypted with a different IV. The IV can be a counter if it is easy to maintain state. Otherwise, the IV is selected at random.

Algorithm 22 AWS Plaintext Encryption

Input: $M = (M_1, M_2, \ldots, M_u)$, where each M_i is a block of plaintext. $IV \in \{0,1\}^{96}$.
Output: (IV, C)

1: $J_0 \leftarrow IV||0^{31}||1$
2: **for** $i = 1$ to u **do**
3: $\quad C_i \leftarrow M_i \oplus \text{AES}_k(J_0 + 1)$
4: **end for**
5: **return** $(IV, (C_1, C_2, \ldots, C_u))$

Note that the IVs in GCM must fulfill the following uniqueness requirement: The probability that the authenticated encryption function ever will be invoked with the same IV and the same key on two (or more) distinct sets of input data shall be no greater than 2^{-32}. The Birthday paradox states that if B IV's are drawn at random (with replacement) from $\{0,1\}^{96}$, the probability that at least one string is selected two (or more) times is $\approx \frac{B^2}{2 \cdot 2^{96}}$. So, at most $B = 2^{32}$ chunks should be encrypted with the same DEK k when IV's are selected at random.

Now, recall that the wrapped key is $wk = \text{Enc}_K(A, k)$, where Enc = AES-GCM. We have

$$IV \in \{0,1\}^{96}, \quad J_0 = IV||0^{31}||1$$

A CMK k can be used simultaneously by several HSMs. State management between these HSMs should be minimized to reduce latency, so, IV's are selected at random. However, the uniqueness requirement above implies that K can be used to wrap at most 2^{32} DEKs. This can be problematic, since CMK's are only rotated once a year. So, only 136 DEKs can be encrypted per second (on average). A solution to this is the derive key mode (for key wrapping).

6.1.2 Derive Key Mode (for key wrapping)

Algorithm 23 Derive Key Mode for Key Wrapping

Input: Random nonce $N \in_R \{0,1\}^{128}$. key K
Output: Wrapped key wk

1: Derive a one-time 256-bit key $L \leftarrow \text{HMAC}_K(N)$
2: $wk \leftarrow \text{Enc}_L(A, k), N$ ▷ Enc is AES-GCM
3: **return** wk

A careful analysis shows that the system tolerates 2^{40} CMKs, each encrypting as many as 2^{50} DEKs.

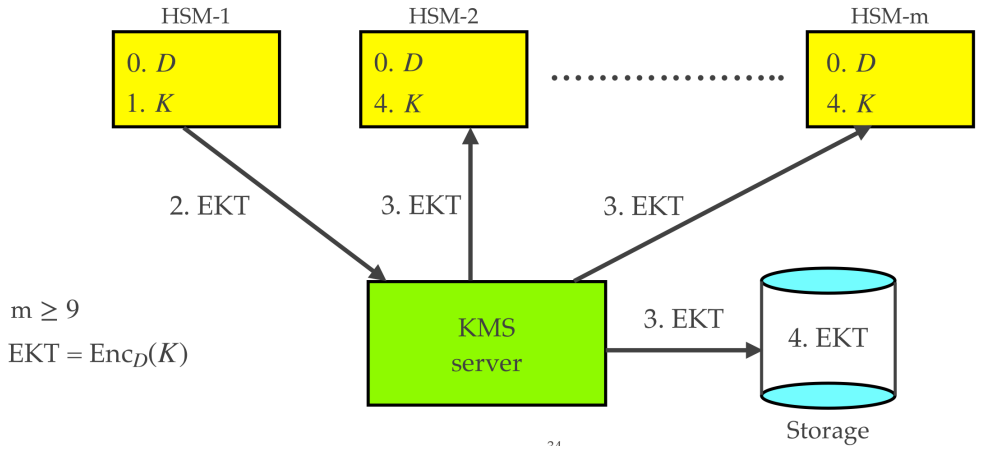
6.1.3 DynamoDB

Amazon's DynamoDB is a fully-managed proprietary NoSQL database. It is offered as an AWS service, providing single-digit millisecond latency at virtually any scale. Each table entry is encrypted with a unique DEK, and it handles about 90 million requests per second.

6.1.4 Protecting a CMK

A CMK must have high security, availability, and durability. Copies of a CMK in use are stored in memory (never on disk) of the HSMs in a domain - a collection of HSMs in a region, with at least three HSMs per availability zone. The HSMs in a domain do not communicate directly with each other. A CMK never leaves an HSM (in unencrypted form), instead, a CMK is encrypted with a domain key (DK), a 256-bit AES-GCM key that is shared by all HSMs in the domain.

The ciphertext, called an Exported Key Token, is stored in highly durable, low-latency storage. The exported key tokens are managed by the KMS.



From this figure, we can see that $m \geq 9$, this means that AWS is storing the CMK in at least 9 HSMs. Everything stored on the HSMs are in memory instead of disks.

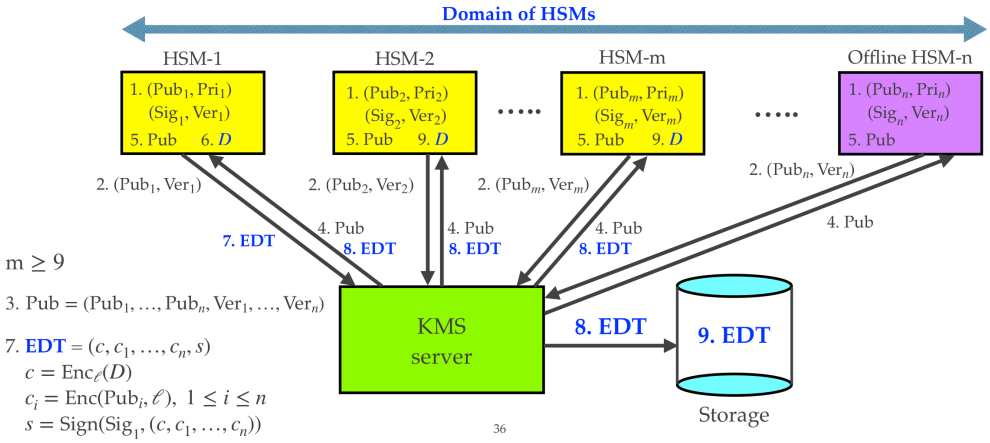
6.1.5 Protecting a Domain Key

A domain key is rotated daily. During rotation, encrypted CMKs are re-encrypted with the new domain key. One of the HSMs in a domain, say HSM-1, selects the new domain key, and encrypts it with a one-time AES-GCM key ℓ; let the resulting ciphertext be c.

The key ℓ is encrypted using public-key cryptography (elliptic curve cryptography, defined in next chapter), using the elliptic curve public keys of the other HSMs in the domain. So, let $c_1, c_2, \ldots, c_n$ be the ciphertexts of ℓ encrypted with the public keys of HSM-2, HSM-3, . . ., HSM-m. The list of ciphertexts

$$C = (c, c_1, c_2, \ldots, c_n)$$

is signed by HSM-1. The signed C, called the exported domain token, is distributed by KMS to the other (online) HSMs in the domain, and is also stored in highly durable storage. Among the HSMs in a domain are offline HSMs. The offline HSMs are stored in safes within monitored safe rooms in multiple independent geographical locations. Each safe requires at least one AWS security officer and one AWS KMS operator, from two independent teams, to obtain these materials.



Now, what if all HSMs in a domain lose power? The offline HSMs are used to recover the domain key.

1. The EDT is extracted from storage.
2. An AWS employee delivers the EDT physically to the offline HSM.
3. The EDT is used to recover the domain key D.
4. Restore the domain key to the online HSMs, but instead using the offline HSM as the starting point.

6.2 Public-key Cryptography

Recall that in Symmetric-key cryptography, communicating parties share some secret keying information beforehand. The shared secret keys can then be used to achieve confidentiality (e.g. using AES), or authentication (e.g. using HMAC), or both (using AES-GCM).

6.2.1 Key Establishment Problem

However, how can Alice and Bob establish the secret key k. One method is by point-to-point key distribution. Alice selects the key and sends it to Bob over a secured channel. The secured channel can be

- A trusted courier
- A face-to-face meeting
- Installation of an authentication key in a SIM card.

However, this is generally not practical for large-scale applications.

Another method is by using a trusted third party (TTP) T. Each user A shares a secret key K_{AT} with T for a symmetric-key encryption scheme E. To establish this key, A must visit T once. Then, T serves as a key distribution centre (KDC). So now suppose that we have communicating parties A and B

1. A sends T a request for a key to share with B
2. T selects a session key k, and encrypts it for A using k_{AT}
3. T encrypts k for B using k_{BT}

The drawbacks of using this method are that the TTP must be unconditionally trusted, the TTP is an attractive target for eavesdroppers, and the TTP must be online, and is therefore a potential bottleneck and critical reliability point. Moreover, in a network of n users, each user has to share a different secret key with every other user. Thus, each user has to store and manage $n-1$ different secret keys. The total number of secret keys is $\binom{n}{2} \approx \frac{n^2}{2}$. This is not ideal.

Another point is that non-repudiation (preventing an entity from denying previous actions or commitments, or denying being the source of a message) is impractical. This is because the TTP can forge messages on behalf of any user.

6.2.2 Public-key Cryptography

This is why we use public-key cryptography. Communicating parties share some authenticated (but non-secret) information. Thus, each entity A generates a key pair (P_A, S_A), where P_A is A's public key and S_A is A's secret key.

This has the following security requirement, it should be infeasible for an adversary to recover S_A from P_A. An example of a public/secret key pair are $S_A = (p, q)$, where p, q are randomly selected prime numbers, and $P_A = p \cdot q$.

Algorithm 24 Public-key Encryption

Input: Message m

1: Obtain an authentic copy of Bob's public key P_B.
2: $c \leftarrow E(P_B, m)$, where E is the encryption function.
3: Send c to Bob.

Algorithm 25 Public-key Decryption

Input: Ciphertext c

1: $m \leftarrow D(S_B, c)$, where D is the decryption function.

Now, we also need digital signatures so that Bob can verify that the message indeed came from Alice.

Algorithm 26 Public-key Message Signing

Input: Message m

1: $s \leftarrow \text{Sign}(S_A, m)$
2: Send (m, s) to Bob

Since s depends on m, other people cannot just take s to act as a signature for other messages.

Algorithm 27 Public-key Signature Verification

Input: Signature s

1: Obtain an authentic copy of Alice's public key P_A
2: **if** $\text{Verify}(P_A, m, s) = \text{accept}$ **then**
3: Accept
4: **else**
5: Reject
6: **end if**

Now, suppose that Alice generates a signed message (m, s). Then anyone who has an authentic copy of Alice's public key P_A can verify the authenticity of the signed message. This authentication feature cannot be achieved with a symmetric key MAC scheme. Digital signature schemes are widely deployed to sign software upgrades which are then broadcast to computers around the world.

The advantages of public-key cryptography are:

- No requirements for a secured channel
- Each user has only one key pair, which simplifies key management

- A signed message can be verified by anyone
- Facilitates the provision of non-repudiation services (with digital signatures)

However, one disadvantage of public-key cryptography is that it is slower than their symmetric-key counterparts.

6.2.3 Hybrid Schemes

Since public-key cryptography is slower than symmetric-key counterparts, we avoid using public-key cryptography to encrypt large amounts of data. In practice, we use a hybrid approach, use public key cryptography to obtain the secret key for symmetric-key cryptography, then use the key to do symmetric key decryption.

Algorithm 28 Hybrid Scheme for Encrypting and Signing a Message

Input: Message m

1: $s \leftarrow \text{Sign}(S_A, m)$
2: Select an AES secret key k
3: Obtain an authentic copy of Bob's public key P_B
4: $c_1 \leftarrow E(P_B, k)$
5: $c_2 \leftarrow AES_k(m, s)$
6: Send c_1, c_2 to Bob

Algorithm 29 Hybrid Scheme for Recovering m and Verifying its Authenticity

Input: Ciphertexts c_1, c_2

1: $k \leftarrow D(S_B, c_1)$
2: $(m, s) \leftarrow AES_k^{-1}(c_2)$
3: Obtain an authentic copy of Alice's public key P_A
4: **if** Verify$(P_A, m, s) ==$ accept **then**
5: Accept
6: **else**
7: Reject
8: **end if**

6.2.4 Algorithmic Number Theory

Recall the Fundamental Theorem of Arithmetic

Theorem 6.1 – Fundamental Theorem of Arithmetic
Every integer $n \geq 2$ has a unique prime factorization (up to ordering of the factors).

This motivates the following questions:

- Given an integer $n \geq 2$, how do we find its prime factorization **efficiently**? (This is hard)
- How do we **efficiently** verify an alleged prime factorization of an integer $n \geq 2$? (This is easy)
- Given an integer $n \geq 2$, how do we **efficiently** decide whether n is prime or composite? (This is easy)

An algorithm is a well-defined computational procedure (e.g. a Turing Machine) that takes a variable input and eventually halts with some output. For example, for an integer factorization algorithm, the input is a positive integer n, and the output is its prime factorization.

The efficiency of an algorithm is measured by the scarce resource it consumes, e.g. time, space, number of processors, number of chosen plaintext-ciphertext pairs.

Definition 6.1 – Input Size
The input size is the number of bits required to write down the input using a reasonable encoding.

For example, the size of a positive integer n is $\lfloor \log_2 n \rfloor + 1$ bits. This comes from the fact that $k = \log_2 n \iff n = 2^k$

Definition 6.2 – Running Time
The running time of an algorithm is an upper bound, as a function of the input size, of the worst case number of basic operations the algorithm executes over all inputs of a fixed size.

Definition 6.3 – Polynomial-Time
An algorithm is a polynomial-time (efficient) algorithm if its expected running time is $O(k^c)$, where k is the input size and c is a fixed positive integer.

Theorem 6.2
If $f(n) : \mathbb{Z}_+ \to \mathbb{R}_+$ and $g(n) : \mathbb{Z}_+ \to \mathbb{R}_+$ are functions from the positive integers to the positive real numbers, then

$$f(n) = O(g(n))$$

means that there exists a positive constant c and a positive integer n_0 such that

$$f(n) \leq cg(n)$$

for all $n \geq n_0$.

For example, $7.5n^3 + 1000n^2 - 99 = O(n^3)$.

Here are some basic integer operations and their running time. The input is two k-bit integers a and b, so the input size is $O(k)$ bits. FAK stands for fastest algorithm known.

Operation	Algorithm	Running time (Bit operations)	Remark
Addition	Elementary School	$O(K)$	$a + b$
Subtraction	Elementary School	$O(k)$	$a - b$
Multiplication	Elementary School	$O(k^2)$	FAK: $O(k \log k)$
Division	Elementary School	$O(k^2)$	$a = qb + r, 0 \leq r < b$
GCD	Euclidean Algorithm	$O(k^2)$	$\gcd(a, b)$, FAK: $O(k \log^2 k)$

Then, here are some basic modular operations and their running time. The input is a k-bit integer n, and integers $a, b, m \in [0, n-1]$, thus the input size is $O(k)$ bits.

Operation	Algorithm	Running time (Bit operations)	Remark
Addition	Elementary School	$O(K)$	$a + b \pmod n$
Subtraction	Elementary School	$O(k)$	$a - b \pmod n$
Multiplication	Elementary School	$O(k^2)$	$a \cdot b \pmod n$
Inversion	Extended Euclidean Algorithm	$O(k^2)$	$a^{-1} \pmod n$
Exponentiation	Repeated square-and-multiply	$O(k^3)$	$a^m \pmod n$

For calculating the GCD of a, b, we first find their prime factorizations

$$a = p_1^{a_1} p_2^{a_2} \dots p_\ell^{a_\ell}, \quad b = p_1^{b_1} p_2^{b_2} \dots p_\ell^{b_\ell}$$

Then

$$\gcd(a,b) = p_1^{\min a_1, b_1} \dots p_\ell^{\min(a_\ell, b_\ell)}$$

For Euclid's algorithm, we have

Algorithm 30 Euclid's Algorithm

Input: $a \geq b \geq 1$
Output: $\gcd(a,b)$

1: **while** $b \neq 0$ **do**
2: $\quad r \leftarrow a \pmod{b}$
3: $\quad a \leftarrow b$
4: $\quad b \leftarrow r$
5: **end while**
6: **return** a

For modular inversion, $x = a^{-1} \mod n$ means the integer $x \in [0, n-1]$ such that

$$ax \equiv 1 \pmod{n}$$

if such x exists. For example, $3^{-1} \mod 7 = 5$. Recall that $a^{-1} \mod n$ exists if and only if $\gcd(a,n) = 1$. The algorithm for findin $x = a^{-1} \mod n$ (when $\gcd(a,n) = 1$) is given as follows:

Algorithm 31 Modular Inversion

Input: a and n
Output: $x = a^{-1} \mod n$

1: Use Extended Euclidean Algorithm to find integers s, t such that

$$as + tn = 1$$

2: We have $as \equiv 1 \pmod{n}$, so $x = s \mod n$
3: **return** $s \mod n$.

7 Week 7 (Feb 26 - Mar 1)

7.1 Public-key Cryptography (Continued)

7.1.1 Algorithmic Number Theory (Continued)

Algorithm 32 Naive Modular Exponentiation 1

Input: A k bit integer n, and integers $a, m \in [0, n-1]$.
Output: $a^m \mod n$.

1: Compute $d = a^m$.
2: Return $d \mod n$.

The bitlength of d is

$$\log_2 d = \log_2 a^m = m \log_2 a = O(2^k k)$$

since $m \approx 2^k$. Hence the algorithm is not polytime.

Algorithm 33 Naive Modular Exponentiation 2

Input: A k bit integer n, and integers $a, m \in [0, n-1]$.
Output: $a^m \mod n$.

1: $A \leftarrow a$
2: **for** $i = 2, 3, \ldots, m$ **do**
3: $\quad A \leftarrow A \cdot a \mod n$
4: **end for**
5: Return A.

This algorithm is also not polytime.

Let the binary representation of m be

$$m = \sum_{i=0}^{k-1} m_i 2^i$$

where $m_i \in \{0, 1\}$. Then we have

$$\begin{aligned} a^m &= a^{\sum_{i=0}^{k-1} m_i 2^i} \\ &= \prod_{i=0}^{k-1} a^{m_i 2^i} \\ &\equiv \prod_{\substack{0 \le i \le k-1 \\ m_i = 1}} a^{2^i} \pmod{n} \end{aligned}$$

This suggests the following repeated **square-and-multiply** algorithm for computing $a^m \mod n$.

Algorithm 34 Repeated Square-and-Multiply

Input: A k bit integer n, and integers $a, m \in [0, n-1]$.
Output: $a^m \mod n$.

```
Write m in binary: m = Σ_{i=0}^{k-1} m_i 2^i
if m_0 = 1 then
    B ← a
else
    B ← 1
end if
A ← a
for i = 1, 2, ..., k − 1 do
    A ← A^2 mod n
    if m_i = 1 then
        B ← B · A mod n
    end if
end for
Return B
```

Since we perform at most k modular squaring and k modular multiplications, the worst case running time is $O(k^3)$ bit operations (which is polytime).

As an example, consider $a^{137} \mod n$. We write m in binary: $137 = 2^7 + 2^3 + 2^0$. Then we have

$$a^{137} = a^{2^7+2^3+2^0} = a^{2^7} \cdot a^{2^3} \cdot a^{2^0}$$

So we compute

$$a, a^2 \mod n, a^{2^2} \mod n, a^{2^3} \mod n, \ldots, a^{2^7} \mod n$$

Then we multiply:

$$a \cdot (a^{2^3} \mod n) \cdot (a^{2^7} \mod n)$$

to get $a^{137} \mod n$.

Another example, consider $99999^{1234} \mod 211103$.

1. Write the exponent in binary $m = 10011010010$
2. Initialize $A = 99999$ and $B = 1$
3. Compute the following table:

i	A	m_i	B
1	61994	1	61994
2	125921	0	61994
3	151911	0	61994
4	16373	1	44538
5	185422	0	44538
6	27989	1	108678
7	191991	1	35248
8	60354	0	35248
9	23051	0	35248
10	2350	1	80424

4. The final result is 80424.

7.2 RSA Public-key Cryptography

7.2.1 Basic RSA Encryption Scheme

For RSA key generation, each entity A does the following:

Algorithm 35 RSA Key Generation

Output: Public key (n, e), private key d.

1: Randomly select two large, distinct primes p and q of the same bitlength
2: Compute $n = pq$ ▷ Called the RSA modulus
3: Compute $\phi = \phi(n) = (p-1)(q-1)$
4: Select an arbitrary integer e with $1 < e < \phi$ with $\gcd(e, \phi) = 1$ ▷ Called the encryption exponent
5: Compute the integer d with $1 < d < \phi$ with $ed \equiv 1 \pmod{\phi}$. This also means that $d = e^{-1} \mod \phi$ ▷ Called the decryption exponent
6: Public key: (n, e)
7: Private key: d

Then using the public keys and private keys, we have the following basic RSA encryption and decryption algorithms. Let's say that B wants to send a message m to A.

Algorithm 36 RSA Encryption

Input: Message m
Output: Ciphertext c

1: Obtain an authenticated copy of A's public key (n, e)
2: Represent the message as an integer $m \in [0, n-1]$.
3: Compute the ciphertext $c = m^e \mod n$ using repeated square multiplication.
4: Send c to A.

Then A decrypts c:

Algorithm 37 RSA Decryption

Input: Ciphertext c
Output: Message m

1: Compute $m = c^d \mod n$

Let's prove the correctness of RSA for all $m \in [0, n-1]$. So if $c = m^e \mod n$, then $m = c^d \mod n$.

Proof. Suppose $c = m^e \mod n$. Raising both sides by d, we get

$$c^d = m^{ed} \mod n$$

So we will prove that $c^d = m^{ed} \equiv m \pmod{n}$ for $m \in [0, n-1]$. Since $ed \equiv 1 \pmod{\phi}$, we can write

$$ed = 1 + k\phi = 1 + k(p-1)(q-1), \quad \text{for some } k \in \mathbb{Z}$$

Since $ed > 1$, and $(p-1)(q-1) \geq 1$, we have $k \geq 1$. Then we consider the following cases:

- Case 1: Suppose $p \mid m$. That is, $m \equiv 0 \pmod{p}$. So we get

  $$m^{ed} \equiv 0^{ed} \equiv 0 \pmod{p}$$

  So

  $$m^{ed} \equiv 0 \pmod{p}$$

  This means that

  $$m^{ed} \equiv m \pmod{p}, \quad \text{since } m \equiv 0 \pmod{p}$$

- Case 2: Suppose $p \nmid m$. So we have $\gcd(m, p) = 1$ (since p is prime). By Fermat's Little Theorem, we have

$$m^{p-1} \equiv 1 \pmod{p}$$

Raise both sides to power $k(q-1)$ and multiply both sides by m, we get

$$\begin{aligned} m \cdot m^{k(p-1)(q-1)} &\equiv m \cdot 1 \pmod{p} \\ m^{1+k(p-1)(q-1)} &\equiv m \pmod{p} \\ m^{1+k\phi} &\equiv m \pmod{p} \\ m^{ed} &\equiv m \pmod{p} && \text{since } ed \equiv 1 \pmod{\phi} \end{aligned}$$

Similarly, we also have $m^{ed} \equiv m \pmod{q}$. So we get

$$m^{ed} = m + sp \text{ for some } s \in \mathbb{Z} \implies p \mid m^{ed} - m$$
$$m^{ed} = m + tq \text{ for some } t \in \mathbb{Z} \implies q \mid m^{ed} - m$$

Since p, q are distinct primes, we have $pq \mid m^{ed} - m$. So we get

$$\begin{aligned} m^{ed} &\equiv m \pmod{pq} \\ m^{ed} &\equiv m \pmod{n} && \text{since } n = pq \end{aligned}$$

□

7.2.1.1 Toy Example: RSA Key generation

Alice generates her public and private keys:

1. Select primes $p = 23, q = 37$.
2. Computes $n = pq = 851$ and $\phi = \phi(n) = (p-1)(q-1) = 792$.
3. Selects $e = 631$. This satisfies $\gcd(631, 792) = 1$
4. Solves $631d \equiv 1 \pmod{792}$:

   Since $\gcd(631, 792) = 1$, we can use Extended Euclidean Algorithm to find d, t such that $631d + 792t = 1$. Rearranging, we get

   $$631d = 1 - 792t$$

   So we have

   $$631d \equiv 1 \pmod{792}$$

   We get $d \equiv -305 \equiv 487 \pmod{792}$. So we set $d = 487$.
5. Alice's public key is $(n = 851, e = 631)$ and her private key is $d = 487$.

Now Bob wants to send a message $m = 13$ to Alice. Bob does the following:

1. Obtain Alice's public key $(n = 851, e = 631)$.
2. Compute $c = 13^{631} \mod 851$ using repeated square-and-multiply.
   (a) Write $e = 631$ in binary:
   $$e = 2^9 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0$$
   (b) Compute successive squaring $(i, m^{2^i} \mod n)$ of $m = 13 \mod n$:
   $$(0, 13), (1, 169), (2, 478), (3, 416), (4, 303), (5, 752), (6, 440), (7, 423), (8, 219), (9, 305)$$
   (c) Multiply together the square m^{2^i} for which the ith bit of the binary representation of 631 is 1:
   $$13^{631} \equiv 305 \cdot 440 \cdot 752 \cdot 303 \cdot 478 \cdot 169 \cdot 13 \equiv 616 \pmod{851}$$
3. Bob sends the ciphertext $c = 616$ to Alice.

Then Alice decrypts $c = 616$. Alice uses her private key $d = 487$ to compute $m = 616^{487} \mod 851$ and she gets $m = 13$.

7.2.2 Basic RSA Signature Scheme

To sign a message m, A does the following:

Algorithm 38 Basic RSA Signature Scheme

Input: Message m
Output: Signed message (m, s)

1: Compute $M = H(m)$, where H is a hash function.
2: Compute the signature $s = M^d \mod n$
3: Return signed message (m, s)

Then B verifies the signed message (m, s):

Algorithm 39 Basic RSA Signature Verification

Input: Message (m, s)

1: Obtain an authenticated copy of A's public key (n, e)
2: Compute $M = H(m)$
3: Compute $M' = s^e \mod n$
4: Accept (m, s) if and only if $M = M'$

7.2.3 Case Study: QQ Browser Encryption

QQ Browser is a web browser used in China. The browser makes a series of WUP requests to the QQ server. The requests contains personal user data, so it should be encrypted before sending to the server. To encrypt a WUP request m, the client does the following:

Algorithm 40 QQ Browser Client Encryption v1

Input: Message m

1: Randomly generate a 128-bit AES session key $k = i||j$ where

$$i, j \in [10000000, 99999998]$$

2: Encrypt k with the server's RSA public key (n, e), $c_1 = k^e \mod n$
3: Encrypt m with the AES session key k, $c_2 = \text{AES-ECB}_k(m)$.
4: Send (c_1, c_2) to the server.

The server decrypts the message:

Algorithm 41 QQ Server Decryption v1

Input: Ciphertext (c_1, c_2)

1: Decrypt c_1 using its RSA private key d

$$k = c_1^d \mod n$$

2: Decrypt c_2 using k

$$m = \text{AES-ECB}_k^{-1}(c_2)$$

3: Generate a WUP response m' that contains user information.
4: Encrypt it using k', which is embedded in all QQ browsers (so it is publicly available).

$$c' = \text{TEA-CBC}_{k'}(m')$$

where TEA stands for Tiny Encryption Algorithm.
5: Send c' to the client.

The client decrypts the response:

Algorithm 42 QQ Browser Client Decryption v1

Input: Ciphertext c'

1: Decrypt c' using the publicly available k'.

$$m' = \text{TEA-CBC}_{k'}^{-1}(c')$$

This implementation is not good, here are some problems:

1. i and j, used to generate the AES session key, are integers in the interval $[10000000, 99999998]$. This is a very small interval, the number of possible keys are

   $$2^{26.4} \times 2^{26.4} \approx 2^{52.8}$$

   instead of 2^{128}. This makes exhaustive key search feasible.
2. The server's RSA public key can be easily factored (i.e. the servers public key n is not large enough).
3. The server's response uses the fixed key k' that is hard coded in all QQ browsers. This means that anyone can decrypt the server's response.
4. They are using ECB mode, which is not good.

Therefore, to combat this, QQ made a version 2.

Algorithm 43 QQ Browser Client Encryption v2

Input: Message m

1: Randomly generate a 128-bit AES session key k
2: Encrypt k with the server's RSA public key (n, e), $c_1 = k^e \mod n$, where $e = 65537$ and n is a 1024-bit RSA modulus.
3: Encrypt m with the AES session key k, $c_2 = \text{AES-ECB}_k(m)$.
4: Send (c_1, c_2) to the server.

The server decrypts the message:

Algorithm 44 QQ Server Decryption v2

Input: Ciphertext (c_1, c_2)

1: Decrypt c_1 using its RSA private key d

$$K = c_1^d \mod n$$

Let k be the 128 least significant bits of K
2: Decrypt c_2 using k

$$m = \text{AES-ECB}_k^{-1}(c_2)$$

3: **if** m is a properly formatted WUP request **then**
4: Generate a WUP response m' that contains user information.
5: Encrypt it using k

$$c' = \text{AES-ECB}_k(m')$$

6: Send c' to the client.
7: **else**
8: Don't respond
9: **end if**

The client decrypts the response:

Algorithm 45 QQ Browser Client Decryption v2

Input: Ciphertext c'

1: Decrypt c' using the session key k.

$$m' = \text{AES-ECB}_k^{-1}(c')$$

This version is better, however, in the server decryption algorithm, the integer K is represented as a 1024-bit number, of which the least significant 128-bits are taken to be k. However, the QQ server did not check that the remaining $1024 - 128 = 896$ bits of K are all 0.

This flaw can be exploited using a **restricted chosen-ciphertext attack**.

Algorithm 46 Restricted Chosen-ciphertext Attack

1: Adversary intercepts a ciphertext (c_1, c_2)
2: She sends a carefully-chosen modification $\hat{c} = (\hat{c_1}, \hat{c_2})$ to the QQ server
3: Depending on whether the server responds or not, the adversary learns one bit of the secret key k.
4: Repeat the above to learn all 128 bits of k.

This attack requires 128 interactions with the QQ server and very little computation. Note that the RSA private key d is not computed.

7.2.4 Security of RSA Encryption

- Security of RSA key generation: If an adversary can factor n, she can compute d from (n, e). It has been proven that any efficient method for computing d from (n, e) is equivalent to factoring n.
- Security of Basic RSA encryption: A basic notion of security is that it should be computationally infeasible to compute m from c. This is known as the RSA problem.
- RSA Problem (RSAP): Given an RSA public key (n, e) and $c = m^e \mod n$, where $m \in_R [0, n-1]$, compute m.

The only effective method known for solving RSAP is to factor n (and thereafter compute d and then m). Henceforth, we shall assume that RSAP is **intractable**. So, we are working on the assumption that if factoring is hard, then the RSA problem is hard.

7.2.5 Dictionary Attack on Basic RSA Encryption

The dictionary attack is as follows. Suppose that the plaintext m is chosen from a relatively small (and known) set M of messages. Then, given a target ciphertext c, the adversary can encrypt each $m \in M$ until c is obtained. So, the adversary can simply create a dictionary of all (m, c) pairs. Then when receives c', she can look up the dictionary to find the corresponding m'.

A countermeasure is to append a randomly selected 128-bit string (called a **salt**) to m prior to encryption.

$$m' = \text{salt}||m$$

Note that m is now encrypted to one of 2^{128} possible ciphertexts, so a dictionary attack is infeasible.

7.2.6 Chosen-ciphertext Attack on Basic RSA Encryption

Suppose that an adversary E has a target ciphertext c that was encrypted for A. Suppose also that E can induce A to decrypt any ciphertext for E, except for c itself (We say that E has a decryption oracle). Then E can decrypt c as follows.

Algorithm 47 Chosen-ciphertext Attack

Input: Ciphertext c

1: Select arbitrary $x \in [2, n-1]$ with $\gcd(x, n) = 1$

Note that if $\gcd \neq 1$, then $\gcd = p, q, n$ since n is prime. It cannot be n since $x < n$. If it is p or q, then we already have the factorization of n. So let's assume that $\gcd = 1$.

2: Compute $\hat{c} = cx^e \mod n$, where (n, e) is A's public key. ▷ Note that $\hat{c} \neq c$, unless $\gcd \neq 1$

3: Obtain the decryption $\hat{m}$ of $\hat{c}$ from the decryption oracle.

So we obtain

$$\hat{m} \equiv \hat{c}^d \equiv (cx^e)^d \equiv c^d x^{ed} \equiv mx \pmod{n}$$

4: Compute $m = \hat{m}x^{-1} \mod n$ using the Extended Euclidean Algorithm.

7.2.7 Countermeasure to the Chosen-ciphertext Attack

Add some prescribed formatting to m prior to encryption. After decrypting the ciphertext c, if the plaintext is not properly formatted, then A rejects c (so the decryption oracle does not return a plaintext).

In summary, an RSA encryption scheme should incorporate bold salting and formatting.

7.2.8 Security Definition of Public-key Encryption

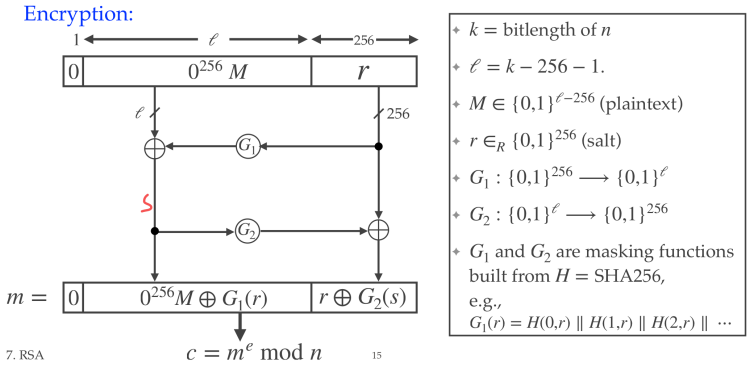
Definition 7.1 – Secure Public-key Encryption
A public-key encryption scheme is secure if it is semantically secure against chosen-ciphertext attacks by a computationally bounded adversary.

Note that we do not need chosen-plaintext attacks, since encryption is a public operation which uses the public key. Thus, to break a public-key encryption scheme, the adversary E has to accomplish the following:

1. E is given a public key and challenge ciphertext c
2. E has a decryption oracle, to which she can present any ciphertexts for decryption except for c itself.
3. After a feasible amount of computation, E should learn **something** about the plaintext m that corresponds to c (other than its length).

7.2.9 RSA Optimal Asymmetric Encryption Padding (OAEP)

Let M be the plaintext, and m the salted and formatted plaintext. Then the RSA-OAEP encryption scheme is as follows:



And for decryption:

Decryption. To decrypt c, do the following:

1. Compute $m = c^d \bmod n$.
2. Parse m: 

3. Compute $r = G_2(s) \oplus t$.
4. Compute $G_1(r) \oplus s =$ 

5. If $a = 0^{256}$, then output $M = b$;
   else reject c.

We have the following theorem that proves the security of RSA-OAEP:

Theorem 7.1
Suppose that RSAP (RSA problem) is intractable. Suppose that G_1 and G_2 are random functions. Then RSA-OAEP is a secure public-key encryption scheme.

7.2.10 Status of Integer Factorization

First let's define Big-O and Little-o notation. Let $f(n), g(n) : \mathbb{Z}_+ \to \mathbb{R}_+$ be functions from the positive integers to the positive real numbers.

Definition 7.2 – Big-O
We write $f(n) = O(g(n))$ if there exists a positive constant c and a positive integer n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$

For example, $3n^3 + 4n^2 + 79 = O(n^3)$. We have

$$f(n) = 3n^3 + 4n^2 + 79, \quad g(n) = n^3$$

We have

$$f(n) \leq 4g(n)$$

for $n \geq 7$.

Definition 7.3 – Little-o
We write $f(n) = o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

For example, $\frac{1}{n} = o(1)$ since

$$\lim_{n\to\infty} \frac{\frac{1}{n}}{1} = 0$$

Then, let's talk about polynomial-time and exponential-time algorithms.

Definition 7.4 – Polynomial-time Algorithm
One whose worst-case running time is of the form $O(n^c)$, where n is the **input size** and c is a constant.

Definition 7.5 – Exponential-time Algorithm
One whose worst-case running time is not of the form $O(n^c)$ for any constant c.

In this course, **fully exponential-time** functions are of the form 2^{cn}, where c is a constant. For example, $O(2^{n/2})$.

Subexponential-time algorithms is one whose worst-case running time function is of the form $2^{o(n)}$, and not of the form $O(n^c)$ for any constant c. For example, $O(2^{\sqrt{n}})$.

7.2.10.1 Example: Trial Division

Consider the following algorithm (trial division) for factoring an RSA modulus n. We trial divide n by the primes $2, 3, 5, \ldots, \lfloor\sqrt{n}\rfloor$. If any of these, say ℓ, divides n, then stop and output the factor ℓ of n.

The running time of this method is at most $\sqrt{n}$ trial divisions, which is $O(\sqrt{n})$. However, is this a polynomial-time algorithm for factoring RSA moduli? No! n is the input in this case, not the input size.

Let A be an algorithm whose input is an integer n. The input size is then $O(\log n)$. That is, $n = 2^{\log n}$. If the expected running time of A is of the form

$$L_n[\alpha, c] = O\left(e^{(c+o(1))(\ln n)^{\alpha}(\ln\ln n)^{1-\alpha}}\right)$$

where c is a positive constant, α a constant satisfying $0 < \alpha < 1$, then A is a **subexponential-time** algorithm.

- If $\alpha = 0$, then $L_n[0, c] = O((\ln n)^{c+o(1)})$, which is polytime.
- If $\alpha = 1$, then $L_n[1, c] = O(n^{c+o(1)})$, which is fully exponential time.

8 Week 8 (Mar 4 - Mar 8)

8.1 RSA Public-key Cryptography (Continued)

8.1.1 Equivalent Security Levels

Recall that a cryptographic scheme has a security level of ℓ bits if the fastest known attack known on the scheme takes approximately 2^ℓ operations. Then we have the following table to compare the security levels of different RSA modulus bitlength n:

Security (bits)	Block Cipher	Hash Function	Bit length n (so RSA modulus = 2^n)
80	SKIPJACK	SHA-1	1024
112	Triple-DES	SHA-224	2048
128	AES small	SHA-256	3072
192	AES medium	SHA-384	7680
256	AES large	SHA-512	15360

8.1.2 RSA Encryption Summary

Factoring is **believed** to be a hard problem, however we have no proof or theoretical evidence that factoring is indeed hard. In fact, factoring is known to be easy on a quantum computer. A quantum computer can use Shor's algorithm to factor n in $O((\log n)^2)$ operations. However, no large-scale quantum computers exist yet. In the table above, since using a 1024-bit RSA modulus has 80 bits of security, it is considered risky. Most application use 2048-bit or 3072-bit RSA modulus for security.

8.1.3 RSA Signature Scheme

8.1.3.1 Basic RSA Signature Scheme

The RSA signature scheme is similar to the RSA encryption scheme. The key generation is exactly the same

Algorithm 48 RSA Key Generation

Output: Public key (n, e), private key d.

1: Randomly select two large, distinct primes p and q of the same bitlength
2: Compute $n = pq$ ▷ Called the RSA modulus
3: Compute $\phi = \phi(n) = (p-1)(q-1)$
4: Select an arbitrary integer e with $1 < e < \phi$ with $\gcd(e, \phi) = 1$ ▷ Called the encryption exponent
5: Compute the integer d with $1 < d < \phi$ with $ed \equiv 1 \pmod{\phi}$. This also means that $d = e^{-1} \mod \phi$ ▷ Called the decryption exponent
6: Public key: (n, e)
7: Private key: d

Then to sign a message m, we have

Algorithm 49 RSA Signature Generation

Input: Message m
Output: Signature s

1: Compute $M = H(n)$, where H is a hash function
2: Compute $s = M^d \mod n$ ▷ So $s^e \equiv M^{ed} \equiv M \pmod{n}$
3: **return** s

We can see that signing is equivalent to finding the eth root of M modulo n. This is because when we compute $s = M^d \mod n$, we have $s^e \equiv M^{ed} \equiv M \pmod{n}$. So we are finding a value s such that when raised to a power of e, it is equivalent to M modulo n. This is the same as finding the eth root of M modulo n.

Then to verify a signature s on a message m, we have

Algorithm 50 RSA Signature Verification

Input: Message m, signature s
Output: Acceptance or rejection

1: Obtain an authentic copy of A's public key (n, e)
2: Compute $M = H(m)$
3: Compute $M' = s^e \mod n$
4: **if** $M = M'$ **then**
5: **return** Accept
6: **else**
7: **return** Reject
8: **end if**

The security of the basic RSA signature scheme requires that the RSA problem (RSAP) be intractable. Otherwise E could forge A's signature as follows:

1. Select arbitrary m
2. Compute $M = H(m)$
3. Solve $s^e \equiv M \pmod n$ for s (finding the eth root of M modulo n)
4. s is a valid signature for m

For the hash function used, we require the following properties:

1. It has preimage resistance. If H is not preimage resistant, and the range of H is $[0, n-1]$, then signatures can be forged as follows:
   (a) Select $s \in_R [0, n-1]$, and compute $M = s^e \mod n$
   (b) Find m such that $H(m) = M$. This is efficient since H is not preimage resistant
   (c) Then s is a valid signature for m
2. It is 2nd preimage resistance. If H is not 2nd preimage resistant, then signatures can be forged as follows:
   (a) Suppose that (m, s) is a valid signed message
   (b) Find an m' with $m' \neq m$ such that $H(m') = H(m)$. This is efficient since H is not 2nd preimage resistant
   (c) Then s is a valid signature for m'
3. It is collision resistant. If H is not collision resistant, then signatures can be forged as follows:
   (a) Select m_1, m_2 with $m_1 \neq m_2$ and $H(m_1) = H(m_2)$. This is efficient since H is not collision resistant
   (b) Induce A to sign m_1 to obtain $s = H(m_1)^d \mod n$
   (c) Then s is a valid signature for m_2

Then for the adversary E, he has the following goals:

1. Total break: E recovers A's private key, or a method to systematically forge A's signatures. (Strongest goal)
2. Existential forgery: E forges A's signature for a single message of E's choosing; E might not have any control over the content or structure of this message. (Weakest goal)

And he might have the following attacks:

1. Key-only attack: The only information E has is A's public key. (Weakest attack)

2. Known-message attack: E knows some message-signature pairs.
3. Chosen-message attack: E has access to a signing oracle which it can use to obtain A's signature on some messages of its choosing. (Strongest attack)

Notice that this is the same as an attack on MAC schemes, except for MAC schemes we do not have a public key. As always, we choose the weakest goal with the strongest capability. This motivates the following security definition:

Definition 8.1 – Security of Signature Scheme
A signature scheme is secure if it is existentially unforgeable by a computationally bounded adversary who launches a chosen-message attack.

So we can assume that the adversary has access to a signing oracle. Her goal is to compute a single valid message-signature pair for any message (of the adversary's choosing) that was not previously given to a signing oracle. The question is, is the basic RSA signature scheme secure? If H is SHA-256, then it is not secure. If H is full domain, then it is secure.

8.1.3.2 Full Domain Hash RSA (RSA-FDH)

The RSA-FDH signature scheme is the same as the basic signature scheme, except that the hash function is

$$H : \{0,1\}^* \to [0, n-1], \quad \text{where } n \text{ is the RSA modulus}$$

In practice, one could define

$$H(m) = \text{SHA-256}(1, m)||\text{SHA-256}(2, m)||\cdots||\text{SHA-256}(t, m)$$

Theorem 8.1
If RSAP is intractable and H is a random function, then RSA-FDH is a secure signature scheme.

8.1.3.3 PKCS # 1 v1.5 RSA Signature Scheme

PKCS stands for Public Key Cryptographic Standards. The scheme is as follows.

Algorithm 51 PKCS # 1 v1.5 RSA Signature Generation

Input: Message m
Output: Signature s

1: Compute $h = H(m)$, where H is a hash function from an approved list.
2: Format h, where k is the byte length of n (e.g. $k = 384$ for a 3072-bit RSA modulus)

$$M(k \text{ bytes}) = 00 \parallel 01 \parallel FF \parallel \cdots \parallel FF \parallel 00 \parallel \underbrace{\text{hash name}}_{\text{15 bytes}} \parallel \underbrace{h}_{\text{20 bytes for SHA-1}}$$

3: Compute $s = M^d \mod n$
4: **return** s

Note that in the formatting, we have $00 \parallel 01 \parallel FF \parallel \cdots \parallel FF \parallel 00$ as a padding, this makes the verification easier since we can check see where the padding starts and ends. In the middle, we use all $FF = 0\text{x}11111111$ to make M as large as possible. Then for signature verification, we have the following

Algorithm 52 PKCS # 1 v1.5 RSA Signature Verification

Input: Message m, signature s
Output: Acceptance or rejection

1: Obtain an authentic copy of A's public key (n, e)
2: Compute $M = s^e \mod n$, and write M as a string of k bytes.
3: Check the formatting, the first byte is 00, the second byte is 01. Then we have consecutive FFs, and then a 00.
4: Obtain the hash name from the next 15 bytes after the padding. Say $H =$ SHA-1.
5: Let h be the next 20 bytes after the hash name.
6: Compute $h' = H(m)$
7: **if** $h = h'$ **then**
8: **return** Accept
9: **else**
10: **return** Reject
11: **end if**

This is secure on paper, however, when implemented, it is not secure. Since the implementation does not check that there is nothing to the right of h, an attacker can exploit it with the following attack that does not need any signing oracle.

Algorithm 53 Bleichenbacher's Attack

Input: Any message m
Output: A valid signature s for m
Assumptions:

- $e = 3$
- n is 3072 bits (384 bytes) long
- H is SHA-1
- The verifier does not check that there are no bytes to the right of h

1: $N =$ random number not divisible by 3
2: **while** N not divisible by 3 **do**
3: Modify m slightly ▷ For the first iteration, don't modify m
4: Hash the message: $h \leftarrow H(m)$
5: Let

$$D(288 \text{ bits}) = \underbrace{00}_{8 \text{ bits}} \| \underbrace{\text{hash name}}_{120 \text{ bits}} \| \underbrace{h}_{160 \text{ bits}}$$

6: Let $N = 2^{288} - D$
7: **end while**
8: Let $s = 2^{1019} - 2^{34}\frac{N}{3}$ ▷ $\frac{N}{3}$ is a positive integer, since N is divisible by 3
9: **return** s

Let's prove that this attack works.

Claim 8.1
The faulty verifier will accept (m, s)

Proof. Recall that $(x-y)^3 = x^3 - 3x^2y + 3xy^2 - y^3$. The verifier first computes

$$\begin{aligned}
M &= s^e \mod n \\
&= \left(2^{1019} - 2^{34}\frac{N}{3}\right)^3 \mod n \\
&= 2^{3057} - 2^{2072}N + \underbrace{2^{1087}\frac{N^2}{3} - \left(2^{34}\frac{N}{2}\right)^3}_{\text{garbage}} \mod n \\
&= 2^{3057} - 2^{2072}(2^{288} - D) + \text{garbage} \mod n && \text{since } N = 2^{288} - D \\
&= 2^{3057} - 2^{2360} + 2^{2072}D + \text{garbage} \mod n
\end{aligned}$$

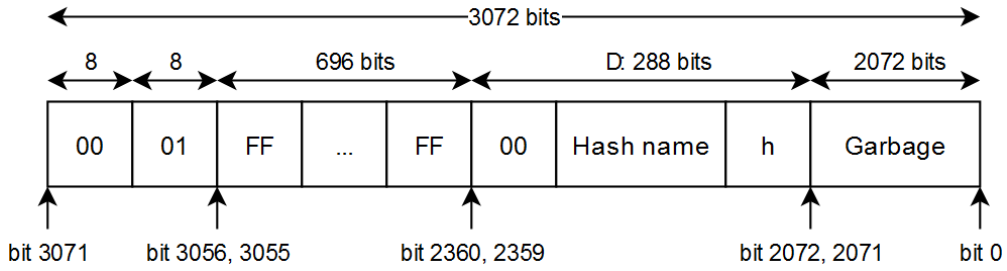
Now, notice that

$$\text{garbage} = 2^{1087}\frac{N^2}{3} - \left(2^{34}\frac{N}{2}\right)^3$$

and we have $0 \leq \text{garbage} < 2^{2072}$. Then, since the leading term of M is 2^{3057}, we have $M < n = 3^{3072}$, so the $\mod n$ takes no effect. Thus we write

$$\begin{aligned}
M &= 2^{3057} - 2^{2360} + 2^{2072}D + \text{garbage} \\
&= 2^{2360}(2^{697} - 1) + 2^{2072}D + \text{garbage}
\end{aligned}$$

Counting bits of M, we see that M is the following:



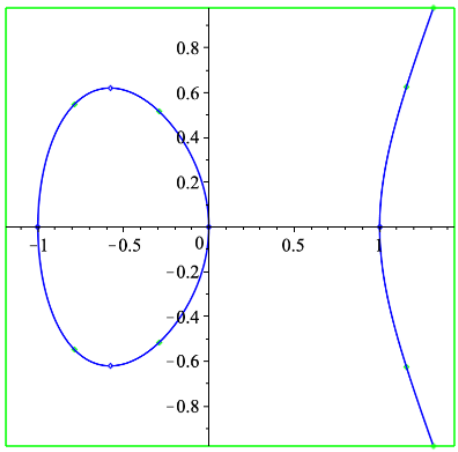
Thus the verifier extracts h, and checks that $h = H(m)$. So (m, s) is accepted. □

Therefore, the attack takes advantage of the fact that the verifier does not fully validate the structure of the decoded message M after it has been exponentiated with the public key during verification. The attack cleverly constructs a signature that, when processed by the faulty verifier, appears to have the correct padding and hash value but actually includes additional, unverified content. This is done by crafting a value that, when cubed, matches the expected prefix and the correct hash value for the message.

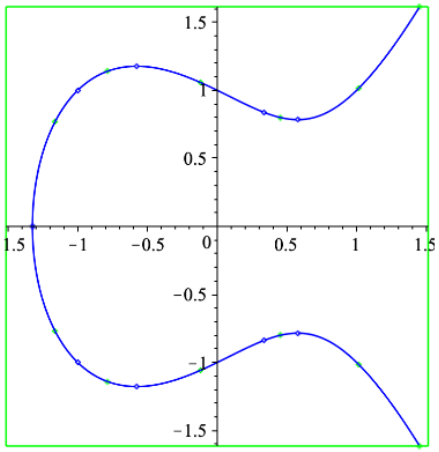
The garbage portion of the constructed message M after the RSA operation is where the attack exploits the verification process. The verifier's failure to check the entirety of the message allows this garbage data to exist without affecting the outcome of the verification. Since the verifier only checks for the correct beginning padding sequence and the expected hash value, it neglects to validate what comes after, assuming the signature is valid if those two checks pass.

8.2 Elliptic Curve Cryptography (ECC)

Here are some elliptic curves over $\mathbb{R}$.

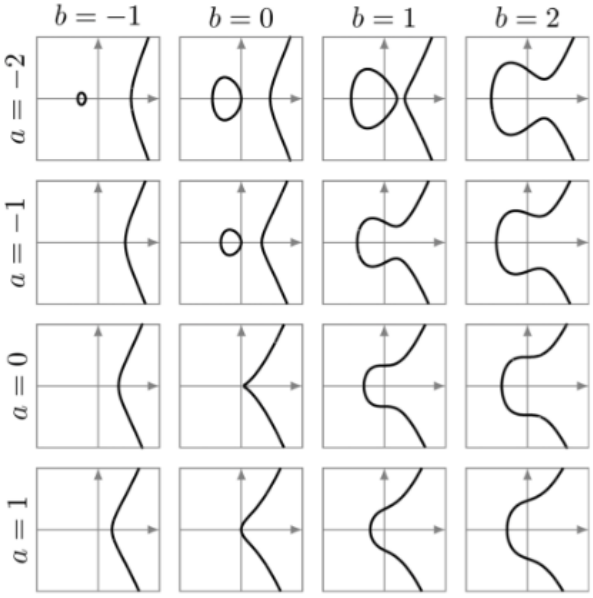


$E/\mathbb{R} : Y^2 = X^3 - X$



$E/\mathbb{R} : Y^2 = X^3 - X + 1$

As we can see from the left graph, there are 3 roots, $X = 0, 1, -1$. And here are some general forms of elliptic curves:



$E/\mathbb{R} : Y^2 = X^3 + aX + b$

However, for cryptography, we cannot use these because these are over $\mathbb{R}$, and a computer cannot represent real numbers exactly. Instead, we use elliptic curves over $\mathbb{F}_p$, where p is a prime. Let's first define formally what an elliptic curve is.

Let the field $F = \mathbb{R}$ or $F = \mathbb{Z}_p$ (where $p \geq 5$ is prime).

Definition 8.2 – Elliptic Curve
An elliptic curve E over F is defined by an equation

$$E/F : Y^2 = X^3 + aX + b$$

where $a, b \in F$ with $4a^3 + 26b^2 \neq 0$.

The condition where $4a^3 + 26b^2 \neq 0$ is just here so that it doesn't have repeated roots, we do not have to pay too much attention to this condition.

Some examples include

- $E/\mathbb{R} : Y^2 = X^3 - X$
- $E/\mathbb{R} : Y^2 = X^3 - X + 1$
- $E/\mathbb{Z}_{89} : Y^2 = X^3 - 2X + 1$
- $E/\mathbb{Z}_{11} : Y^2 = X^3 + X + 6$

Definition 8.3 – Set of F-rational points
The set of F-rational points on E is

$$E(F) = \{(x, y) \in F \times F : y^2 = x^3 + ax + b\} \cup \{\infty\}$$

where ∞ is a special point call the point at infinity.

The point at infinity is special, it is defined to be a point where all vertical lines must intersect.

Example 8.1
Consider the elliptic curve

$$E/\mathbb{Z}_{11} : Y^2 = X^3 + X + 6$$

The set of $\mathbb{Z}_{11}$-rational points on E is

$$E(\mathbb{Z}_{11}) = \{\infty, (2,4), (2,7), (3,5), (3,6), (5,2), (5,9), (7,2), (7,9), (8,3), (8,8), (10,2), (10,9)\}$$

and the cardinality is $\#E(\mathbb{Z}_{11}) = 13$.

Let's show how we can find these points. First, the elements of $\mathbb{Z}_{11}$ are $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Addition and multiplication is defined modulo 11. So for example,

$$\frac{7}{9} = 7 \cdot 9^{-1} = 7 \cdot 5 = 2$$

To find the points in the set, we can just plug in all the values of x into the equation and see if there is a corresponding y. Let's first find all the squares in $\mathbb{Z}_{11}$.

$$\begin{aligned} 0^2 &= 0 \\ 1^2 &= 1 \\ 2^2 &= 4 \\ 3^2 &= 9 \\ 4^2 &= 5 \\ 5^2 &= 3 \end{aligned} \qquad \begin{aligned} 6^2 &= (-5)^2 = 3 \\ 7^2 &= (-4)^2 = 5 \\ 8^2 &= (-3)^2 = 9 \\ 9^2 &= (-2)^2 = 4 \\ 10^2 &= (-1)^2 = 1 \end{aligned}$$

Then we try all points x to see if there is a corresponding y. Plugging in $x = 0$, we get $y^2 = 6$. But there does not exist a y such that $y^2 = 6$. So $(0, y) \notin E(\mathbb{Z}_{11})$ for all $y \in \mathbb{Z}_{11}$. This is the same for $x = 1$. If we try $x = 2$, we get

$$y^2 = 16 = 5$$

In this case we have $4^2 = 7^2 = 5$, so we have two points $(2, 4)$ and $(2, 7)$. We can continue this process to find all the points.

Let $E/\mathbb{Z}_p : Y^2 = X^3 + aX + b$ be an elliptic curve. Then $\#E(\mathbb{Z}_p)$ is finite. It is easy to see that

$$1 \leq \#E(\mathbb{Z}_p) \leq 2p + 1$$

this is because there are p choices for x, and for each x, it gives either 0, 1, or 2 points, so we have the term $2p$. We add a 1 to also account for the point at infinity. In fact, the following is true (and a proof will not be provided)

Theorem 8.2 – Hasse's Theorem
Let E be an elliptic curve defined over $\mathbb{Z}_p$. Then

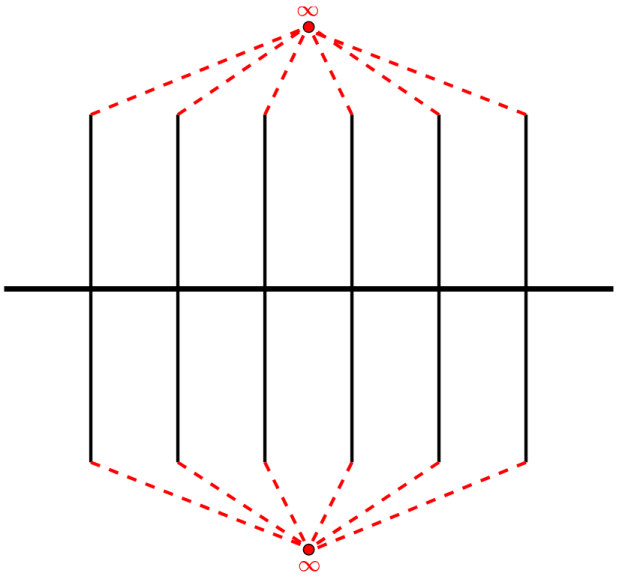
$$(\sqrt{p} - 1)^2 \leq \#E(\mathbb{Z}_p) \leq (\sqrt{p} + 1)^2$$

Hence, $\#E(\mathbb{Z}_p) \approx p$. In fact, there is an efficient (polynomial-time) algorithm for computing $\#E(\mathbb{Z}_p)$.

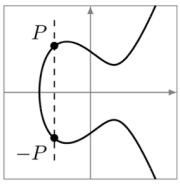
8.2.1 Point Addition

There is a natural way to add two points in $E(F)$ to get a third point in $E(F)$, and it is not pointwise addition.

Let E be an elliptic curve defined over $\mathbb{R}$. We think of ∞ as an imaginary point through which every vertical line passes (in either direction).

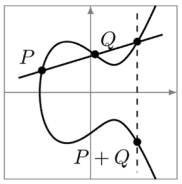


The geometric rule for point addition is the following:
Let $P, Q \in E(\mathbb{R})$. Let ℓ denote the straight line through P and Q. Let $T \in E(\mathbb{R})$ be the third point of intersection of ℓ with the elliptic curve. Then $P + Q$ is the reflection of T in the X-axis.



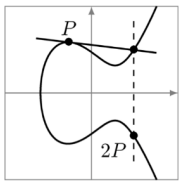
Inverse element $-P$

A2



Addition $P + Q$
"Chord rule"

A3



Doubling $P + P$
"Tangent rule"

A4

For case A2, the two points are P and $-P$. So we have ℓ a vertical line. Since every vertical line intersects with ∞, we have $P + (-P) = \infty$.

For case A4, the two points are P and P. So, ℓ is a tangent line at P. We consider this tangent line to intersect with the curve at point P two times. Then we take the reflection of the third point of intersection in the X-axis.

Algebraically, the addition rules are defined as follows:

Definition 8.4 – Addition Rules
Let E be an elliptic curve defined over F.

A1: $P + \infty = \infty + P = P$ for all $P \in E(F)$.

A2: If $P = (x, y) \in E(F)$, then $-P = (x, -y)$, also $-\infty = \infty$. Then, their sum is

$$P + (-P) = -P + P = \infty, \quad \text{for all } P \in E(F)$$

A3: Let $P = (x_1, y_1), Q = (x_2, y_2) \in E(F)$ with $P \neq \pm Q$. Then $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = -y_1 + \lambda(x_1 - x_3), \quad \lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

A4: Let $P = (x_1, y_1) \in E(F)$ with $P \neq -P$. Then $P + P = (x_3, y_3)$, where

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = -y_1 + \lambda(x_1 - x_3), \quad \lambda = \frac{3x_1^2 + a}{2y_1}$$

Let's show the derivation of A3 for adding two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. Let ℓ be the line through P and Q. Then ℓ has the equation

$$\ell : y = y_1 + \lambda(x - x_1), \quad \lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

Substitute into the equation $Y^2 = X^3 + aX + b$ to get

$$\begin{aligned} [y_1 + \lambda(x - x_1)]^2 &= x^3 + ax + b \\ x^3 + ax + b - [y_1 + \lambda(x - x_1)]^2 &= 0 \end{aligned}$$

We know that x_1, x_2 are solutions in F. Since this is degree 3, there is a third solution x_3. We can write

$$x^3 + ax + b - [y_1 + \lambda(x - x_1)]^2 = (x - x_1)(x - x_2)(x - x_3)$$

Equating coeffecients of x^2 on both sides, we get

$$\begin{aligned} -\lambda^2 &= -x_1 - x_2 - x_3 \\ x_3 &= \lambda^2 - x_1 - x_2 \end{aligned}$$

The Y-coordinate of the third point of intersection of λ with E is

$$y_3 = y_1 + \lambda(x_3 - x_1)$$

So, we have

$$P + Q = (x_3, -y_3) = (\lambda^2 - x_1 - x_2, -y_1 + \lambda(x_1 - x_3))$$

Notice that we have $-y_3$ for the Y-coordinate because we take the reflection of the third point of intersection in the X-axis.

Considering the case A4, we can find the slope of the tangent line to E at $P = (x_1, y_1)$ where

$$E : Y^2 = X^3 + aX + b$$

by taking the derivative of Y with respect to X. Then we have

$$2Y\frac{dY}{dX} = 3X^2 + a \implies \frac{dY}{dX} = \frac{3X^2 + a}{2Y}$$

Example 8.2
Consider the elliptic curve

$$E/\mathbb{Z}_{11} : Y^2 = X^3 + X + 6$$

The set of $\mathbb{Z}_{11}$-rational points on E is

$$E(\mathbb{Z}_{11}) = \{\infty, (2,4), (2,7), (3,5), (3,6), (5,2), (5,9), (7,2), (7,9), (8,3), (8,8), (10,2), (10,9)\}$$

and the cardinality is $\#E(\mathbb{Z}_{11}) = 13$. Let's do some point addition.

$$(2,4) + (2,7) = \infty \quad \text{A2}$$
$$(2,4) + (8,3) = (5,2) \quad \text{A3}$$
$$(2,4) + (2,4) = (5,9) \quad \text{A4}$$

8.2.2 Abelian Group

In fact, $(E(F), +)$ is an abelian group. In other words, the addition rule satisfies the following properties:

- P1: $P + \infty = P$ for all $P \in E(F)$ (identity)
- P2: For each $P \in E(F)$, there exists $Q \in E(F)$ such that $P + Q = \infty$ (inverse)
- P3: $P + Q = Q + P$ for all $P, Q \in E(F)$ (commutative)
- P4: $(P + Q) + R = P + (Q + R)$ for all $P, Q, R \in E(F)$ (associative)

9 Week 9 (Mar 11 - Mar 15)

9.1 Elliptic Curve Cryptography (Continued)

9.1.1 Elliptic Curve Discrete Logarithm Problem

First, let's define the multiplication of a point with a natural number. Note that multiplication of a point with a point is not defined.
Let

$$E : Y^2 = X^3 + aX + b$$

be an elliptic curve defined over $F = \mathbb{Z}_p$. Let $n = \#E(\mathbb{Z}_p)$ and suppose that n is prime. Recall that $n \approx p$ by Hasse's Theorem.

Definition 9.1 – Point Multiplication
Let $P \in E(\mathbb{Z}_p)$ and let $k \in \mathbb{N}$. Then

$$kP = \underbrace{P + P + \cdots + P}_{k}$$

Also, $0P = \infty$, and $(-k)P = -(kP)$. This is called point multiplication.

Theorem 9.1 – Generator
Suppose $n = \#E(\mathbb{Z}_p)$ is prime, and let $P \in E(\mathbb{Z}_p)$ be some point with $P \neq \infty$. Then

1. $nP = \infty$
2. The points $\infty, P, 2P, 3P, \ldots, (n-1)P$ are distinct, and so

$$E(\mathbb{Z}_p) = \{\infty, P, 2P, 3P, \ldots, (n-1)P\}$$

P is called a generator of $E(\mathbb{Z}_p)$. Note that due to the cyclic nature of the group, we have

$$kP = (k \mod n)P, \quad \text{for all } k \in \mathbb{Z}$$

Example 9.1
As an example, let's consider the elliptic curve

$$E/\mathbb{Z}_{23} : Y^2 = X^3 + X + 4$$

over $\mathbb{Z}_{23}$. We have $\#E(\mathbb{Z}_{23}) = 29$. $P = (0, 2)$ is a generator of $E(\mathbb{Z}_{23})$ since we have the following

$1P = (0, 2)$	$2P = (13, 12)$	$3P = (11, 9)$	$4P = (1, 12)$	$5P = (7, 20)$	$6P = (9, 11)$
$7P = (15, 6)$	$8P = (14, 5)$	$9P = (4, 7)$	$10P = (22, 5)$	$11P = (10, 5)$	$12P = (17, 9)$
$13P = (8, 15)$	$14P = (18, 9)$	$15P = (18, 14)$	$16P = (8, 8)$	$17P = (17, 14)$	$18P = (10, 18)$
$19P = (22, 18)$	$20P = (4, 16)$	$21P = (14, 18)$	$22P = (15, 17)$	$23P = (9, 12)$	$24P = (7, 3)$
$25P = (1, 11)$	$26P = (11, 14)$	$27P = (13, 11)$	$28P = (0, 21)$	$29P = \infty$	

We can see that all the points are distinct, and $29P = \infty$, so P is a generator of $E(\mathbb{Z}_{23})$.

With this, we can define the **Elliptic Curve Discrete Logarithm Problem (ECDLP)**.

Definition 9.2 – Elliptic Curve Discrete Logarithm Problem (ECDLP)
Given $E, p, n = \#E(\mathbb{Z}_p), P \in E(\mathbb{Z}_p)$ with $P \neq \infty$ and $Q \in E(\mathbb{Z}_p)$, find the integer $\ell \in [0, n-1]$ such that

$$Q = \ell P$$

The integer ℓ is called the **discrete logarithm** of Q to the base P, written $\ell = \log_P Q$.

Using the previous example, if $P = (0, 2)$ and $Q = (10, 18)$, we determine that $\log_P Q = 18$, that is, $Q = 18P$.

We now introduce algorithms for solving the ECDLP.

1. Brute force. Compute $P, 2P, 3P, \ldots$ until Q is encountered. The running time of this attack is $O(n)$ point additions, or $O(p)$ point additions since $n \approx p$. This is a fully exponential-time attack, since an ECDLP instance has size $O(\log p)$ bits.

2. Shanks' Algorithm. Let $m = \lceil \sqrt{n} \rceil$. By the division algorithm there exist unique integers q, r with

   $$\ell = qm + r, \quad \text{where } 0 \leq r < m \text{ and } 0 \leq q < m$$

   Then $\ell - qm = r$. We multiply both sides by P to get

   $$\ell P - qmP = rP$$

   let $M = mP$. Since $Q = \ell P$, we can rewrite this as

   $$Q - qM = rP$$

   The equation $Q - qM = rP$ suggests the following ECDLP algorithm

Algorithm 54 Shanks' Algorithm

Input: $E, p, n = \#E(\mathbb{Z}_p), P \in E(\mathbb{Z}_p)$ with $P \neq \infty, Q \in E(\mathbb{Z}_p)$

Output: $\ell = \log_P Q$

```
for r = 0, ..., m − 1 do
    Compute (rP, r) and store in a table
end for
Compute M = mP
for q = 0, ..., m − 1 do
    Compute R = Q − qM and look it up in the table.
    if R = rP for some r then
        return ℓ = qm + r
    end if
end for
```

   The run time is $O(m) = O(\sqrt{p})$ point additions, with space requirements $O(\sqrt{p})$ points. Notice that this looks like the meet-in-the-middle attack for double DES.

3. Pollard's Algorithm. The running time is $\approx \sqrt{p}$ point additions (just remember this), and uses negligible storage. This is similar to the VW parallel collision search.

4. Special cases. The ECDLP can be solved in polytime if $n = p$. Morevoer, the ECDLP can be solved in subexponential time if $n \mid (p^c - 1)$ for a small integer c (e.g. $c \leq 100$).

5. Shor's Algorithm. The ECDLP can be solved in time $O((\log p)^2)$ on a quantum computer.

In summary, Pollard's algorithm is the fastest algorithm known for solving the ECDLP on a classical computer.

9.1.2 Elliptic Curve Cryptography (ECC)

Why do we use ECC? For RSA, security is based on the hardness of integer factorization. The fastest attacks known take **subexponential time**. On the other hand, security of ECC is based on the hardness of the ECDLP. The fastest attacks known take **fully exponential time**. Thus, ECC systems can use smaller parameters than their RSA counterparts while achieving the same security level.

Security Level	Bitlength of RSA modulus n	Bitlength of ECC field modulus p
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Next, we will talk about the different elliptic curves used in practice.

9.1.2.1 P-256 Elliptic Curve

P-256 is chosen where

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

with

$$E/\mathbb{Z}_p : Y^2 = X^3 - 3X + b$$

where $b = 41058363725152142129326129780047268409114441015993725554835256314039467401291$. We have $n = \#E(\mathbb{Z}_p)$ is prime. And the generator is a point P. It is used in applications that require 128-bit security level since $\sqrt{p} \approx 2^{128}$.

9.1.2.2 P-384 Elliptic Curve

P-384 is chosen where

$$p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$$

with

$$E/\mathbb{Z}_p : Y^2 = X^3 - 3X + b$$

where $b = 27580193559959705877849011840389048093056905856361568521428707301988689241309860865136260764883745107765439761230575$
We have $n = \#E(\mathbb{Z}_p)$ is prime. And the generator is a point P. It is used in applications that require 192-bit security level since $\sqrt{p} \approx 2^{192}$.

9.1.2.3 P-521 Elliptic Curve

P-521 is chosen where

$$p = 2^{521} - 1$$

with

$$E/\mathbb{Z}_p : Y^2 = X^3 - 3X + b$$

where b is some fixed number. We have $n = \#E(\mathbb{Z}_p)$ is prime. And the generator is a point P. It is used in applications that require 256-bit security level since $\sqrt{p} \approx 2^{256}$.

9.1.2.4 Curve25519 Elliptic Curve

Curve25519 is chosen where

$$p = 2^{255} - 19$$

with

$$E/\mathbb{Z}_p : Y^2 = X^3 + 486662X^2 + X$$

Note that the curve equation is in Montgomery form. We have $\#E(\mathbb{Z}_p) = 8n$ where n is some 253-bit prime.

9.1.2.5 SM2 Elliptic Curve

SM2 is a standard for ECC in China. It is chosen where

$$p = 2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$$

with

$$E/\mathbb{Z}_p : Y^2 = X^3 - 3X + b$$

for some fixed b. We have $n = \#E(\mathbb{Z}_p)$ is prime. SM2 is used in applications that require 128-bit security level since $\sqrt{p} \approx 2^{128}$.

9.1.3 CNSA 1.0: Commercial National Security Algorithm Suite

This standard is established by the NSA. It is used for protecting unclassificed, sensitive but unclassified, and classified data.

- Encryption: AES-256
- Hashing: SHA-384
- Signature: ECDSA with P-384 and RSA-PSS with $\geq$ 3072-bit modulus n
- Key Establishment: ECDH with P-384 and RSA encryption with $\geq$ 3072-bit modulus n

9.1.4 Modular Reduction

The primes

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

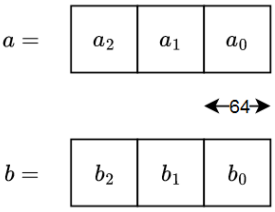
and

$$p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$$

used in P-256 and P-384 were chosen because the operation of reduction modulo p can be easily implemented on a 32-bit machine without doing a costly long division. To illustrate this technique, let's consider the reduction modulo the prime $p = 2^{192} - 2^{64} - 1$ on a 64-bit machine. A 64-bit machine refers to a computer with built-in instructions for addition, subtraction, and multiplication of 64-bit integers (note that these are regular addition, instead of the bitwise XOR that we are used to, so it may contain a carry bit).

Suppose that $a, b \in [0, p-1]$. We wish to compute $c = a \times b \mod p$. We have

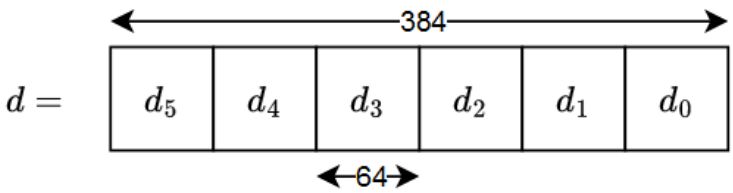
$$a = a_2 2^{128} + a_1 2^{64} + a_0$$
$$b = b_2 2^{128} + b_1 2^{64} + b_0$$



where $a_0, a_1, a_2, b_0, b_1, b_2$ are 64-bit integers. We first compute $d = a \times b$ using ordinary base-2^{64} integer multiplication. Let

$$d = d_5 2^{320} + d_4 2^{256} + d_3 2^{192} + d_2 2^{128} + d_1 2^{64} + d_0$$

where $d_0, d_1, d_2, d_3, d_4, d_5$ are 64-bit integers. We then reduce d modulo p.



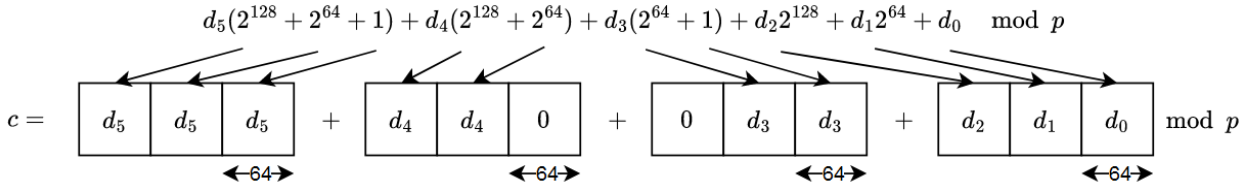
We have

$$
\begin{aligned}
2^{192} &\equiv 2^{64} + 1 \pmod{p} \\
2^{256} &\equiv 2^{192} \times 2^{64} \equiv \left(2^{64} + 1\right) 2^{64} \equiv 2^{128} + 2^{64} \pmod{p} \\
2^{320} &\equiv 2^{256} \times 2^{64} \equiv 2^{192} + 2^{128} \equiv 2^{128} + 2^{64} + 1 \pmod{p}
\end{aligned}
$$

Hence,

$$
\begin{aligned}
c &= d \mod p \\
&= d_5 2^{230} + d_4 2^{128} + d_3 2^{64} + d_2 + d_1 2^{192} + d_0 2^{128} + d_0 2^{64} \mod p \\
&= d_5(2^{128} + 2^{64} + 1) + d_4(2^{128} + 2^{64}) + d_3(2^{64} + 1) + d_2 2^{128} + d_1 2^{64} + d_0 \mod p
\end{aligned}
$$

So we have



Note that c is probably only slightly larger than p, so we can just subtract p from c until $c < p$.
So we have the following algorithm for computing $c = a \times b \mod p$.

Algorithm 55 Modular Multiplication

Input: $a, b \in [0, p-1]$
Output: $c = a \times b \mod p$

1: Compute $d = a \times b = (d_5, d_4, d_3, d_2, d_1, d_0)$, where each d_i is a 64-bit integer.
2: Define the 192-bit integers

$$
\begin{aligned}
t_1 &\leftarrow (d_2, d_1, d_0) \\
t_2 &\leftarrow (0, d_3, d_3) \\
t_3 &\leftarrow (d_4, d_4, 0) \\
t_4 &\leftarrow (d_5, d_5, d_5)
\end{aligned}
$$

3: Compute $c = t_1 + t_2 + t_3 + t_4$
4: **if** $c \geq p$ **then**
5: Repeatedly subtract p from c until $c < p$
6: **end if**
7: **return** c

Note that in this algorithm, no full long division is performed. Moreover, in step 3, we have $0 \leq c < 4p$, so at most three subtractions by p are needed.

9.1.5 Elliptic Curve Diffie-Hellman (ECDH)

ECDH is the elliptic curve analogue of a key agreement protocol. The objective is for two communicating parties to agree on a shared secret key k. They can then use k in a symmetric-key scheme such as AES-GCM.

The parameters of ECDH are

- Elliptic curve E, defined over $\mathbb{Z}_p$ with $n = \#E(\mathbb{Z}_p)$ prime

- A base point (generator) $P \in E(\mathbb{Z}_p)$, with $P \neq \infty$
- A key derivation function KDF.

Now, here is an unauthenticated ECDH protocol.

Algorithm 56 Unauthenticated ECDH

Input: Elliptic curve E, defined over $\mathbb{Z}_p$ with $n = \#E(\mathbb{Z}_p)$ prime, base point $P \in E(\mathbb{Z}_p)$, with $P \neq \infty$, key derivation function KDF
Output: Both Alice and Bob gets shared secret key k

1: Alice selects $x \in_R [1, n-1]$ and compute $X = xP$. ▷ X is Alice's public key and x is Alice's private key
2: Alice sends X to Bob.
3: Bob selects $y \in_R [1, n-1]$ and compute $Y = yP$. ▷ Y is Bob's public key and y is Bob's private key
4: Bob sends Y to Alice.
5: Bob computes $K = yX$ and $k = \text{KDF}(K)$. ▷ k is their shared secret key
6: Alice computes $K = xY$ and $k = \text{KDF}(K)$. ▷ k is their shared secret key

We can show that both Alice and Bob computes the same k since

$$xY = x(yP) = (xy)P = y(xP) = yX$$

An eavesdropper sees the public keys X, Y, and the domain parameters. Her goal is then to compute $K = xY = yX$. The fastest way known to accomplish this is to first compute x or y; this is an instance of the ECDLP. Now, since X and Y are not authenticated, EDCH is vulnerable to the Malicious-Intruder-in-the-Middle (MITM) attack.

Algorithm 57 Malicious-Intruder-in-the-Middle (MITM) Attack on ECDH

1: Alice selects $x \in_R [1, n-1]$ and compute $X = xP$. ▷ X is Alice's public key and x is Alice's private key
2: Alice attempts to send X to Bob, but gets intercepted by Eve ▷ Eve is the eavesdropper
3: Eve Selects x', y' and compute

$$X' \leftarrow x'P \tag{1}$$
$$Y' \leftarrow y'P \tag{2}$$

4: Eve sends X' to Bob. Bob mistakenly thinks that she received X' from Alice.
5: Bob selects $y \in_R [1, n-1]$ and compute $Y = yP$. ▷ Y is Bob's public key and y is Bob's private key
6: Bob attempts to send Y to Alice, but gets intercepted by Eve.
7: Eve sends Y' to Alice. Alice mistakenly thinks that she received Y' from Bob.
8: Bob computes

$$K_2 \leftarrow yX' \tag{3}$$
$$k_2 \leftarrow KDF(K_2) \tag{4}$$

9: Alice computes

$$K_1 \leftarrow xY' \tag{5}$$
$$k_1 \leftarrow KDF(K_1) \tag{6}$$

Now, suppose that Alice sends $c = E_{k_1}(m)$ to Bob. Eve intercepts c, computes

$$m = E_{k_1}^{-1}(c), \quad c' = E_{k_2}(m)$$

and sends c' to Bob. Bob computes $m = E_{k_2}^{-1}(c')$. So Eve is able to decrypt and re-encrypt the message without Alice or Bob knowing. This is a classic MITM attack. The whole idea is that Eve replaces X and Y with her own. The solution to this is to use authenticated ECDH.

Alice sends Bob X, and her ECDSA signature on X, and a certificate for the ECDSA public key. Bob then verifies the certificate, and then uses Alice's public key to verify her signature on X. Similarly, Bob signs Y. As a result, the MITM attack is thwarted.

9.1.6 Elliptic Curve Digital Signature Algorithm (ECDSA)

For ECDSA, we have the following domain parameters

- Elliptic curve E, defined over $\mathbb{Z}_p$ (e.g. P-256) with $n = \#E(\mathbb{Z}_p)$ prime
- A generator $P \in E(\mathbb{Z}_p)$
- A collision-resistant hash function H whose output has the same bitlength as n (e.g. $H =$ SHA-256 for P-256)

Algorithm 58 ECDSA Key Generation

Input: Domain parameters described above
Output: Public key A, private key a for Alice

1: Select $a \in_R [1, n-1]$ and compute $A = aP$
2: Alice's public key is A, her private key is a.

Note that computing a from A is an instance of the ECDLP. Then we have the following ECDSA signature algorithm.

Algorithm 59 ECDSA Signature Generation

Input: Message M, and domain parameters above
Output: Alice's signature on M, (r, s).

1: Compute $m = H(M)$ and interpret m as an integer
2: Select a per-message secret key $k \in_R [1, n-1]$
3: Compute $R = kP$. Let $r = x(R) \mod n$ be the x-coordinate of R modulo n. Check that $r \neq 0$. ▷ $x(R)$ is the x-coordinate of R
4: Compute $s = k^{-1}(m + ar) \mod n$ and check that $s \neq 0$ ▷ k^{-1}, a are private
5: Alice's signature on M is (r, s). ▷ $256 + 256 = 512$ bits

Note that a fresh random k should be generated each time Alice signs a message. Moreover, the x coordinate of a point uniquely defines a point, since the curve is symmetric about the x-axis. So a given x point has at most two y points.

Then we have the following algorithm for signature verification. To verify Alice's signature (r, s) on M

Algorithm 60 ECDSA Signature Verification

Input: Alice's signature on M, (r, s).
Output: Accept or Reject

1: Obtain an authentic copy of Alice's public key A
2: Check that r, s are integers in $[1, n-1]$
3: Compute $m = H(M)$
4: Compute $u_1 = ms^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$
5: Compute $V = u_1 P + u_2 A$ and verify that $V \neq \infty$
6: Compute $v = x(V) \mod n$
7: **if** $v = r$ **then**
8: Accept the signature (r, s)
9: **else**
10: Reject the signature (r, s)
11: **end if**

Let's see why this makes sense.

Proof. Given M, (r, s) and A. We need to check if $s = k^{-1}(m + ar) \mod n$. To the verifier, both k and a are unknown, so he is unable to verify it directly. We have

$$\begin{aligned}
s &= k^{-1}(m + ar) \mod n &&\iff \\
k &\equiv s^{-1}(m + ar) \pmod n &&\iff \\
k &\equiv \underbrace{s^{-1}m}_{u_1} + \underbrace{(s^{-1}r)}_{u_2} a \pmod n &&\iff \\
kP &= (u_1 + u_2 a)P &&\iff \\
R &= u_1 P + u_2 aP &&\iff \\
R &= u_1 P + u_2 A
\end{aligned}$$

Now, u_1 is known, since $m = H(M)$ for some hash function H, and s is known. P is known, since it is a domain parameter. u_2 is known, since r and s are known. A is known, since it is Alice's public key. So R is known. Now, let V be the right hand side of the equation, so we have

$$\begin{aligned}
R &= V &&\implies \\
x(R) &= x(V) &&\implies \\
x(R) \mod n &= x(V) \mod n &&\iff \\
r &= x(V) \mod n
\end{aligned}$$

□

Now, the security of ECDSA is believed to be secure, assuming that the ECDLP is **intractable** and H is a **secure** hash function. For ECDSA (with 256-bit p) and RSA (with 3072-bit n),

1. In practice, RSA is used with $e = 3$ or $e = 2^{16} + 1 = 65537$. So, RSA signature **verification** ($s^e \mod n$) is generally faster than ECDSA signature **verification** ($V = u_1 P + u_2 A$).
2. RSA signature **generation** ($H(M)^d \mod n$) is generally slower than ECDSA signature **generation** ($R = kP$).
3. RSA signatures are 3072 bits in length, whereas ECDSA signatures are 512 bits in length.

9.1.6.1 ECDSA Notes

If an adversary learns a signed message $(M, (r, s))$ and the k that was used, then she can compute

$$a = (ks - m)r^{-1} \mod n$$

that is, she can compute Alice's private key.

If Alice signs two messages, M_1, M_2, with the same k, then the adversary who learns the two signed messages $(M_1, (r_1, s_1))$ and $(M_2, (r_2, s_2))$ can do the following. Since both comes from the same k value, we have $r_1 = r_2$. So,

$$\begin{aligned}
ks_1 &\equiv m_1 + ar_1 \pmod n \\
ks_2 &\equiv m_2 + ar_1 \pmod n
\end{aligned}$$

So the attacker has two linear congruences and two unknowns, thus she can just solve it to get a and k.
In 2011, it was discovered that Sony used the same k to sign many games. This resulted in people being able to compute the private key and sign their own games.

9.1.6.2 Deterministic ECDSA

Alice has a second secret key b. Alice selects

$$k = H(b, M)$$

Since H is a hash function, every message M results in a different k. This ensures that k is different for each message. This is called **deterministic ECDSA**.

10 Week 10 (Mar 18 - Mar 22)

10.1 Bluetooth Security

There are five basic services specified in the Bluetooth standard

1. Confidentiality: prevent information compromise caused by eavesdropping by ensuring that only authorized devices can access and view transmitted data.
2. Data integrity: verifying that messages sent between two bluetooth devices have not been altered in transit.
3. Entity authentication: verifying the identity of communicating devices based on their bluetooth addresses.
4. Authorization: allowing the control of resources by ensuring that a device is authorized to use a service before permitting it to do so.
5. Pairing: creating one or more shared keys and the storing of these keys for use in subsequent connections in order to form a trusted device pair.

We look at the **Secure Connections** security mechanism for pairing. There are four association models depending on the I/O capabilities of the devices.

1. Numeric Comparison: both devices have displays for 6-digit numbers, and each device has either an OK/Reject confirmation button or a keyboard
2. PassKey Entry: when one device has a keyboard but not display, and the other device has a display. Or, when one device has a display only and the other device a keyboard and display.
3. Just Works: when one device has a display and the other device has no display or keyboard.
4. Out-of-Band: designed for devices that support a common additional wireless (e.g. NFC) or wired technology for the purpose of device discovery and cryptographic value exchange.

We will look at numeric comparison. Suppose that we have Alice, an iPhone, and Bob, a Macbook. Both are owned by the same owner. For numeric comparison, there are five phases:

1. Phase 1: Public key exchange

   Purpose: establish an (unauthenticated) shared secret
2. Phase 2: Authentication stage 1

   Purpose: provide some protection against active MITM attacks
3. Phase 3: Authentication stage 2

   Purpose: Confirm that both devices have successfully completed the exchange and confirm the Bluetooth addresses
4. Phase 4: Link key calculation

   Purpose: Compute the long term link key
5. Phase 5: Authentication and encryption

   Purpose: Check that the communicating devices hold the same link key (and thus are paired) and then derive a shared secret encryption key K_E, which is used to encrypt and authenticate data for that session.

For the following phases, we have the following notation:

- P-256 elliptic curve parameters, p (the prime number), E (the curve), n (the number of points on the curve), P (the generator)
- Alice: the **initiating** device
- Bob: the **responding** device

- A, B: Alice's and Bob's 48-bit Bluetooth addresses
- N_A, N_B: Alice's and Bob's randomly selected 128-bit nonces
- C_B: Bob's commitment (to his nonce)
- V_A, V_B: Alice's and Bob's 6-digit verification values
- E_A, E_B: Alice's and Bob's exchange confirmation values
- LK: link key
- S_A, S_B: Alice's and Bob's signed responses
- ACO: authenticated ciphering offset
- K_E: encryption key

10.1.1 Phase 1: Public key exchange

Purpose: establish an (unauthenticated) shared secret

Algorithm 61 Public key exchange

1: Alice selects $x \in_R [1, n-1]$, computes $X = xP$, and sends X to Bob.
2: Bob selects $y \in_R [1, n-1]$, computes $Y = yP$, and sends Y to Alice.
3: Bob sets K to be the x-coordinate of yX.
4: Alice sets K to be the x-coordinate of xY.
5: The shared secret is K (256 bits).

10.1.2 Phase 2: Authentication stage 1

Purpose: provide some protection against active MITM attacks

Algorithm 62 Authentication stage 1

1: Bob selects **nonce** $N_B \in_R \{0,1\}^{128}$, computes a commitment

$$C_B = \text{HMAC-SHA-256}_{N_B}(Y, X)$$

and sends C_B to Alice.
2: Alice selects **nonce** $N_A \in_R \{0,1\}^{128}$, and sends N_A to Bob.
3: Bob sends N_B to Alice.
4: Alice checks if $C_B = \text{HMAC-SHA-256}_{N_B}(Y, X)$.
5: Alice computes the **verification value**

$$V_A = \big(\text{SHA-256}(X, Y, N_A, N_B) \mod 2^{32}\big) \mod 10^6$$

and displays this value. The purpose of modulo 2^{32} is to get the 32 least significant (rightmost) bits, and the modulo 10^6 is to get a 6-digit number.
6: Bob computes the **verification value**

$$V_B = \big(\text{SHA-256}(X, Y, N_A, N_B) \mod 2^{32}\big) \mod 10^6$$

and displays this value.
7: **if** $V_A = V_B$ **then**
8: The owner presses the OK button on both devices
9: **end if**

The use of the commitment C_B ensures that Alice and Bob each select their nonces before seeing the other party's nonce. Failure in step 4 indicates the presence of an attacker, or some transmission error. An active MITM attack will result in two 6-digit verification values V_A, V_B being different with probability 0.999999.

10.1.3 Phase 3: Authentication stage 2

Purpose: Confirm that both devices have successfully completed the exchange and confirm the Bluetooth addresses

Algorithm 63 Authentication stage 2

1: Alice computes

$$\text{HMAC-SHA-256}_K(N_A, N_B, A, B)$$

and takes the 128 most significant bits. This is the **exchange confirmation value** E_A. She sends E_A to Bob.

2: Bob verifies that the 128 most significant bits of

$$\text{HMAC-SHA-256}_K(N_A, N_B, A, B)$$

are equal to E_A.

3: Bob computes

$$\text{HMAC-SHA-256}_K(N_B, N_A, B, A)$$

and takes the 128 most significant bits. This is the **exchange confirmation value** E_B. He sends E_B to Alice.

4: Alice verifies that the 128 most significant bits of

$$\text{HMAC-SHA-256}_K(N_B, N_A, B, A)$$

are equal to E_B.

10.1.4 Phase 4: Link key calculation

Purpose: Compute the long term link key

Algorithm 64 Link key calculation

1: Alice computes

$$\text{HMAC-SHA-256}_K(N_A, N_B, \text{0x62746C6B}, A, B)$$

and takes the 128 most significant bits. This is the **link key** LK.

2: Bob computes

$$\text{HMAC-SHA-256}_K(N_A, N_B, \text{0x62746C6B}, A, B)$$

and takes the 128 most significant bits. This is the **link key** LK.

Note that the link key is the long term authentication key, it is used to maintain the pairing. The nonces N_A, N_B ensure the freshness of the link key, even if long term ECDH values X, Y are used by both sides.

10.1.5 Phase 5: Authentication and encryption

Purpose: Check that the communicating devices hold the same link key (and thus are paired) and then derive a shared secret encryption key K_E, which is used to encrypt and authenticate data for that session.

Algorithm 65 Authentication and encryption

1: Alice generates $R_A \in_R \{0,1\}^{128}$ and sends (A, R_A) to Bob.
2: Bob generates $R_B \in_R \{0,1\}^{128}$ and sends (B, R_B) to Alice.
3: Alice and Bob compute:

1. **Device authentication key** h, which is the 128 most significant bits of

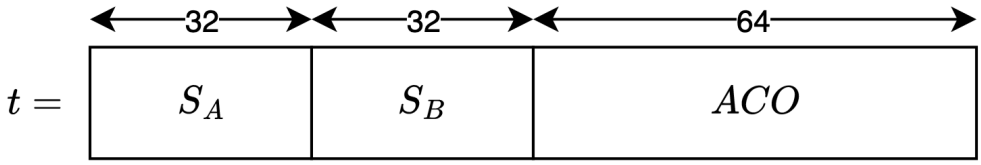
$$\text{HMAC-SHA-256}_h(R_A, R_B)$$

2. **Device authentication confirmation values**

$$t = \text{HMAC-SHA-256}_h(R_A, R_B)$$

and let

$$S_A = \text{leftmost 32 bits of } t, \quad S_B = \text{next 32 bits of } t, \quad ACO = \text{next 64 bits of } t$$



4: Alice sends the signed response S_A to Bob.
5: Bob sends the signed response S_B to Alice.
6: Alice and Bob compare the signed responses they received with the values they computed.

Then, the encryption key K_E is the 128 most significant bits of

$$\text{HMAC-SHA-256}_{LK}(A, B, ACO)$$

For smaller ℓ-bit keys, take the ℓ-most significant bits, and set the remaining bits to 0. The encryption key K_E derived may vary in length in single byte increments from 1 byte to 16 bytes. The length is set during a **negotiation process** that occurs between two communicating devices.

Then, encryption is performed using AES-GCM (CBC-MAC then AES-CTR encryption) in Bluetooth versions 4.1 and later.

10.1.6 KNOB attack

This attack is based on the negotiation protocol. There is a negotiation protocol used by two bluetooth devices to agree on the length of K_E. Since this negotiation is neither authenticated nor encrypted, teh adversary can cause the devices to agree to produce an 8-bit key K_E. As a result, this key can be brute forced in real time.

10.2 Key Management

Key management is the set of techniques and procedures supporting the establishment and maintenance of keying relationships between authorized parties. We will consider the management of public keys that are used for public-key encryption, ECDH key agreement, and for the verification of digital signatures.

One techniques for distributing public keys over the internet is by offline **certification authorities** (CAs).

10.2.1 Certification Authorities (CAs)

A CA issues certificates that bind an entity's identity A to its public key. A's certificate Cert_A consists of

- Data part D_A: contains A's identity, her public key, and other information such as validity period.
- Signature part S_{CA}: The CA's signature on D_A.

Another party B then obtains an authentic copy of A's public key as follows:

1. Obtain an authentic copy of CA's public key
2. Obtain Cert_A from the CA (over an unsecured channel)
3. Verify the CA's signature S_{CA} on D_A using the CA's public key

So, the CA does not have to be trusted with users' private keys. However, the CA has to be trusted to not create fraudulent certificates.

10.2.2 Public Key Infrastructure (PKI)

PKI is a collection of technologies and processes for managing public keys, their corresponding private keys, and their use in applications. Some components of a PKI include

- Certificate format
- The certification process
- Certification distribution
- Certificate policy: Details of the intended use and scope of a particular certificate
- Certificate practices statement (CPS): practices and policies followed by a CA
- Certificate revocation

10.2.3 Transport Layer Security (TLS)

It is originally named SSL. The main components of TLS are:

1. Handshake protocol: Allows the server to authenticate itself to the client, and then negotiate cryptographic keys.
2. Record protocol: Used to encrypt and authenticate transmitted data for the remainder of the session.

10.2.3.1 Handshake protocol

There are four phases in the handshake protocol:

1. Phase 1: Establish security capabilities. This negotiates protocol version, cryptographic algorithms, security levels, etc.
2. Phase 2: Server authentication and key exchange. The server sends its certificate, and key exchange parameters (if any)
3. Phase 3: Client authentication and key exchange. Client sends its certificate (if available) and key exchange parameters
4. Phase 4: Finish. Both parties send a finished message to confirm that the handshake is complete.

There are two main establishment schemes in the handshake protocol:

1. RSA key transport: The shared secret key k is selected by the **client** and encrypted with the server's RSA public key. (Not allowed in latest version of TLS)
2. Elliptic Curve Diffie-Hellman (ECDH): The server selects a one time ECDH public key $X = xP$ and signs it with its RSA or ECDSA signature key. The client also selects a one-time ECDH public key $Y = yP$. The session key k is then

$$k = \text{KDF}(xyP), \quad \text{where KDF is a key derivation function}$$

10.2.3.2 Record Protocol

Now, suppose that the client and server share a **MAC secret key** and a **session encryption key**. Then the data exchanged for the remainder of the session is encrypted and authenticated.

For TLS 1.2, they use RC4, ChaCha20, Triple-DES, AES for **Symmetric-key encryption**. For MAC, they use HMAC-MD5, HMAC-SHA-1, HMAC-SHA-256, ... For the authenticated encryption, they use MAC-then-encrypt.

For TLS 1.3, they removed support for RC4, Triple-DES, CBC-mode, MAC-then-encrypt, MD5, SHA-1, RSA key transport. Moreover, they mandated the use of AES-GCM, and all public key exchanges use ECDH. The elliptic curves they use include P-256, Curve25519, and P-384.

10.2.4 RSA key transport vs. ECDH

Recall that for RSA key transport, the session key k is selected by the client and encrypted with the server's RSA public key, let's call the resulting cipher text $c = E_{\text{RSA}}(k)$. The session key k is then used to encrypt and authenticate all data exchanged for the remainder of the session (e.g. using AES-GCM).

One drawback is **forward secrecy** is not provided. That is, suppose that an eavesdropper has access to $c = E_{\text{RSA}}(k)$, and the encrypted data. Suppose that at a future point in time, the eavesdropper is able to get the server's private key (e.g. through law enforcement), then the eavesdropper is able to decrypt c to obtain k, then using k to decrypt the encrypted data.

Now, why can't the server just use a new private key every time? This is because the server will then have to get a new certificate every time, which is costly and time-consuming. In contrast, forward secrecy can be provided if ECDH is used to establish k.

Let's outline how ECDH achieves forward secrecy. When a web browser (Alice) visits a secured website (Bob).

1. Alice selects $x \in_R [1, n-1]$ and sends $X = xP$ to Bob. X is Alice's public key and x is her private key.
2. Bob selects $y \in_R [1, n-1]$, signs $Y = yP$ with its RSA signing key, and sends Y with its signature to Alice. Y is Bob's public key and y is his private key.
3. Alice verifies the signature using Bob's RSA public key.
4. Both Alice and Bob compute the session key
$$k = \text{KDF}(xyP)$$
5. Alice **deletes** her private key x and Bob **deletes** his private key y.
6. Alice and Bob uses k to authenticate/encrypt data with AES-GCM.
7. At the end of the session, Alice and Bob **deletes** k.

Thus, k is used only for one session. Now, even if the eavesdropper gets a hold of the encrypted data, the public keys X, Y, and the signature, the eavesdropper(e.g. law enforcement) will not be able to obtain the private key x, y since they have been deleted. Thus, they will not be able to obtain the session key k as well. This is how ECDH provides forward secrecy.

Compared to RSA key transport, the private key is used multiple times, so if the private key is compromised, all past sessions are compromised.