CO 487/687 Course Notes

Applied Cryptography

Haochen Wu

University of Waterloo
Winter 2021

Contents

List of Algorithms

List of Figures

Chapter 1 Introduction

Cryptography is about securing communications in the presence of malicious adversaries. The adversary is **malicious**, **powerful** and **unpredictable**.

Definition 1.1. Symmetric-key cryptography: The **client** and **server** a priori share some **secret** information k, called a **key**.

Definition 1.2. Public-key cryptography: Communicating parties a priori share some **authenticated** (but non-secret) information.

Definition 1.3. Cybersecurity, also known as **information security**, 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, loss or use.

1.1 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
- Data origin authentication: Corroborating the source of data
- Non-repudiation: Preventing an entity from denying previous commitments or actions

Chapter 2 Symmetric Key Encryption

2.1 Basic Concepts

Definition 2.1. 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, $E_k : M \to C$, $\forall k \in K$
- a family of decryption functions, $D_k : C \to M$, $\forall k \in K$, such that $D_k(E_k(m)) = m$ for all $m \in M, k \in K$.

To use a SKES to Achieve Confidentiality:

1. Alice and Bob agree on a **secret key** $k \in K$ by communicating over the **secure 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 $m = D_k(c)$.

2.1.1 Security Definitions

Computational Power of the Adversary:

- Information-theoretic security: Eve has infinite computational resources.
- Complexity-theoretic security: Eve is a 'polynomial-time Turing machine'.
- Computational security: Eve has 36,768 Intel E5-2683 V4 cores running at 2.1 GHz at her disposal.

Adversary's Interaction:

- 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.
- Active attacks

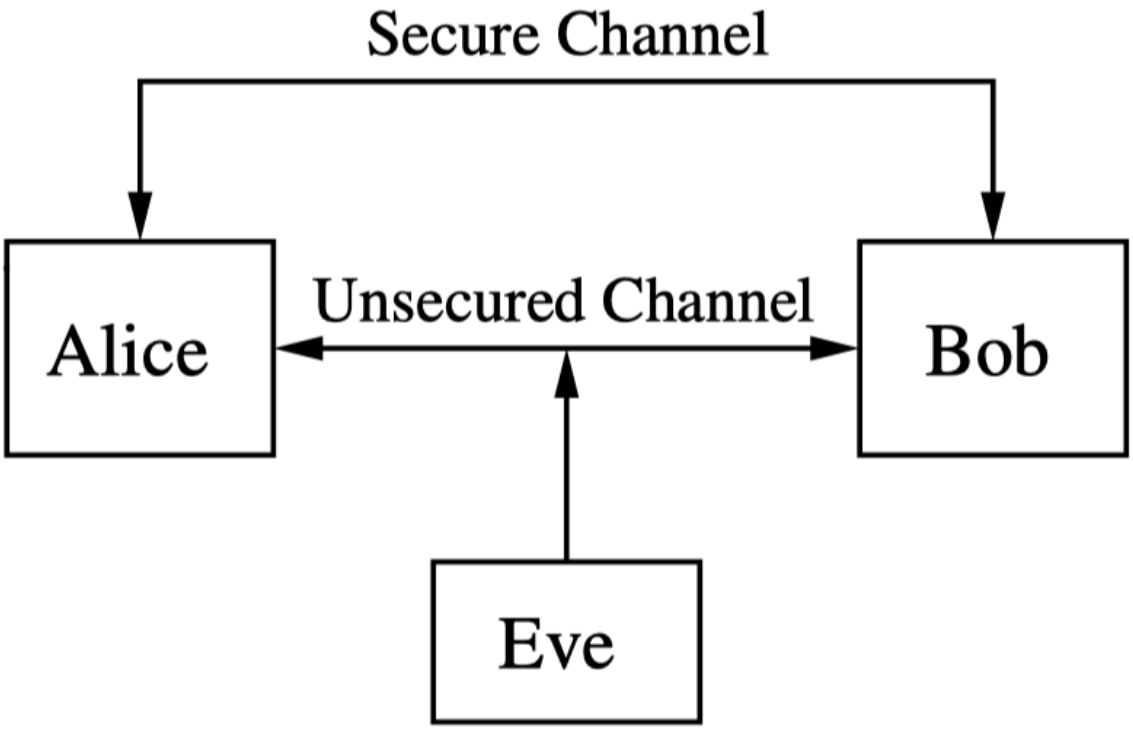


Figure 1: Use a SKES to Achieve Confidentiality

  - Chosen-plaintext attack: The adversary can also choose some plaintext and obtains the corresponding ciphertext.

- Other attacks
  - Clandestine attacks: bribery, blackmail, etc.
  - Side-channel attacks: monitor the encryption and decryption equipment (timing attacks, power analysis attacks, electromagnetic-radiation analysis, etc.)

Adversary's Goal:

- Recover the secret key.
- Systematically recover plaintext from ciphertext (without necessarily learning the secret key).
- Learn some partial information about the plaintext from the ciphertext (other than its length).

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

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**.

Definition 2.2. A symmetric-key encryption scheme is said to be **secure** if it is semantically secure against chosen- plaintext attack by a computationally bounded adversary.

To break a symmetric-key encryption scheme, the adversary has to accomplish the following:

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

Desirable Properties of a SKES:

1. Efficient algorithms should be known for computing E_k and D_k (i.e. for encryption and decryption).
2. The secret key should be small (but large enough to render exhaustive key search infeasible).
3. The scheme should be **secure**.
4. The scheme should be secure even against the designer of the system.

Definition 2.3. 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.

As of the year 2021, a security level of 128 bits is desirable in practice.

2.1.2 The Simple Substitution Cipher

Definition 2.4. The Simple Substitution Cipher is defined as follows:

- M = all English messages.
- C = all encrypted messages.
- K = all permutations of the English alphabet.
- $E_k(m)$: Apply permutation k to m, one letter at a time.
- $D_k(c)$: Apply inverse permutation k^{-1} to c, one letter at a time.

Clearly, it is totally insecure against a chosen-plaintext attack.

For ciphertext-only attack:

- Given sufficient amounts of ciphertext c, decrypt c using each possible key until c decrypts to a plaintext message which "makes sense".
- In principle, 30 characters of ciphertext are sufficient on average to yield a unique plaintext that is a sensible English message. In practice, a few hundred characters are needed.
- In terms of exhaustive key search, the number of keys to try is $26! \approx 4 \times 10^{26} \approx 2^{88}$
- If the adversary uses 10^6 computers, each capable of trying 10^9 keys per second, then exhaustive key search takes about 10^4 years. So, exhaustive key search is infeasible.
- However, we can do a simple frequency analysis to break this scheme.

2.1.3 Work Factor

Definition 2.5. For this course, we consider:

- 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 2.6. The **Landauer limit** from thermodynamics suggests that exhaustively trying 2^{128} symmetric keys would require $\gg 3000$ gigawatts of power for one year (which is $\gg 100\%$ of the world's energy production).

2.1.4 Polyalphabetic Ciphers

Since the simple substitution cipher is totally broken even against ciphertext-only attacks, we can consider the following idea: use several permutations, so a plaintext letter is encrypted to one of several possible ciphertext letters

For example, **Vigenere cipher** encrypts in the following way:

However, it is still totally insecure under a chosen-plaintext attack or a ciphertext-only attack

2.1.5 Convention and Notations

- From now on, unless otherwise stated, messages and keys will be assumed to be **bit strings**
- $\oplus$ is is bitwise exclusive-or (XOR), or equivalently, bitwise addition modulo 2.

Example: Vigenère cipher:

- **Secret key** is an English word having no repeated letters
  e.g. $k =$ CRYPTO.
- Example of **encryption**:

$m =$	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

- Here, A=0, B=1, . . ., Z=25; addition of letters is mod 26.
- **Decryption** is subtraction modulo 26: $m = c - k$.
- Frequency distribution of ciphertext letters is flatter (than for a simple substitution cipher).

– 53

Figure 2: Vigenere Cipher

2.2 One Time Pad

Using the idea from above, we can do a one-time pad, which is invented Vernam in 1917 for the telegraph system. The key a **random** string of letters, and the key is as long as the plaintext.

Example of encryption:

$m =$	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	

Figure 3: Example of One Time Pad

Note that The key should not be re-used:

- If $c_1 = m_1 + k$ and $c_2 = m_2 + k$, then $c_1 - c_2 = m_1 - m_2$

- Thus $c_1 - c_2$ depends only on the plaintext (and not on the key) and hence can leak information about the plaintext.
- In particular, if m_1 is known, then m_2 can be easily computed.

The encryption of one-time pad would be $c = m \oplus k$, and decryption would be $m = c \oplus k$

Definition 2.7. Perfect secrecy: The one-time pad is **semantically secure** against ciphertext-only attack by an adversary with infinite computational resources.

Remarks:

- This can be proven formally using concepts from information theory [Shannon 1949].
- The bad news: Shannon (1949) proved that if plaintexts are m-bit strings, then any symmetric-key encryption scheme with perfect secrecy must have $|K| \geq 2^m$.
- So, perfect secrecy (and the one-time pad) is fairly useless in practice.

2.3 Stream Ciphers

The basic idea is that instead of using a **random** key in the one-time pad, we can use a "**pseudorandom**" key.

Definition 2.8. A **pseudorandom bit generator (PRBG)** is a deterministic algorithm that takes as input a (random) **seed**, and outputs a longer "**pseudorandom**" sequence called the **keystream**.

The security of this depends on the quality of the PRBG. We have several requirements on that:

- The keystream should be "indistinguishable" from a random sequence (the **indistinguishability** requirement).
- If an adversary knows a portion c_1 of ciphertext and the corresponding plaintext m_1 , then she can easily find the corresponding portion $k_1 = c_1 \oplus m_1$ of the keystream. Thus, given portions of the keystream, it should be **infeasible** to learn **any information** about the rest of the keystream (the **unpredictability** requirement).

2.4 The RC4 Stream Cipher

It is designed by Ron Rivest in 1987:

- Pros: Extremely simple; extremely fast; variable key length. No catastrophic weakness has been found.
- Cons: Design criteria are proprietary; not much public scrutiny until the year 2001

Using a PRBG for encryption (a stream cipher):

- The seed is the secret key shared by Alice and Bob.

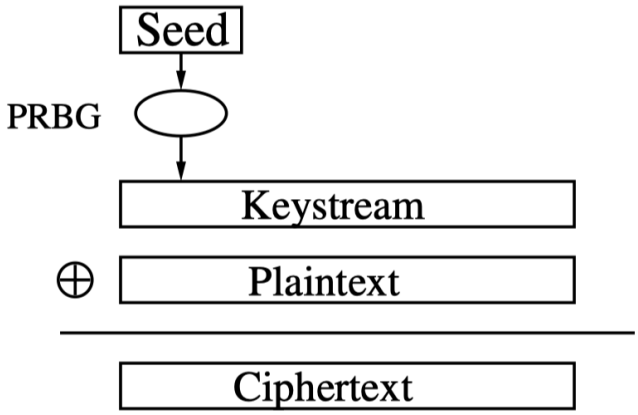


Figure 4: Encryption of PRBG

- RC4 has two components: (i) a **key scheduling algorithm**, and (ii) **a keystream generator**.

In the following, $K[i], \bar{K}[i]$ and $S[i]$ are 8-bit integers (bytes).

Algorithm 1: RC4 Key Scheduling Algorithm

Input: Secret key $K[0], K[1], ..., K[d-1]$. (Keylength is $8d$ bits.)
Output: 256-long array: $S[0], S[1], ..., S[255]$.

for $i = 0$ *to* 255 **do**
    $S[i] \leftarrow i$
    $\bar{K}[i] \leftarrow K[i \mod d]$
$K \leftarrow 0$
for $i = 0$ *to* 255 **do**
    $j \leftarrow (\bar{K}[i] + S[i] + j) \mod 256$
    Swap $(S[i], S[j])$

The idea is that S is a "random-looking" permutation of $\{0, 1, 2, ..., 255\}$ that is generated from the secret key.

Algorithm 2: RC4 Keystream Generator

Input: 256-long byte array: $S[0], S[1], ..., S[255]$ produced by the RC4 Key Scheduling Algorithm.

Output: Keystream

```
1 i ← 0; j ← 0
2 while keystream bytes are required do
3 |   i ← (i + 1)  mod 256
4 |   j ← (S[i] + j)  mod 256
5 |   Swap(S[i], S[j])
6 |   t ← (S[i] + S[j])  mod 256
7 |   Output S[t]
```

For encryption, the keystream bytes are XORed with the plaintext bytes to produce ciphertext bytes.

2.4.1 Wired Equivalent Privacy (WEP)

It was ratified in September 1999. WEP's goal is (only) to protect link-level data during wireless transmission between mobile stations and access points.

Main Security Goals of WEP:

- **Confidentiality**: Prevent casual eavesdropping. Here, RC4 is used for encryption.
- **Data Integrity**: Prevent tampering with transmitted messages. Here, An "integrity checksum" is used.
- **Access Control**: Protect access to a wireless network infrastructure. Here, we discard all packets that are not properly encrypted using WEP.

Description of WEP Protocol:

- Mobile stations share a secret key k with access point:
  - k is either 40 bits or 104 bits in length.
  - The standard does not specify how the key is to be distributed.
  - In practice, one shared key per LAN is common; this key is manually injected into each access point and mobile station; the key is not changed very frequently.
- Messages are divided into **packets** of some fixed length (e.g. 1500 bytes).
- WEP uses a per-packet 24-bit **initialization vector (IV)** v to process each packet. WEP does not specify how the IVs are managed. In practice:
  - A random IV is generated for each packet; or
  - The IV is set to 0 and incremented by 1 for each use.

Algorithm 3: To send a packet m, an entity does the following

1 Select a 24-bit IV v
2 Compute a 32-bit checksum: $S = CRC(m)$.
3 802.11 specifies that a CRC-32 checksum be used. CRC-32 is **linear**. That is, for any two messages m_1 and m_2 of the same bitlength, $CRC(m_1 \oplus m_2) = CRC(m_1) \oplus CRC(m_2)$.
4 Compute $c = (m \| S) \oplus RC4(v \| k)$.
5 $\|$ or , denotes concatenation.
6 $(v \| k)$ is the key used in the RC4 stream cipher
7 Send (v, c) over the wireless channel

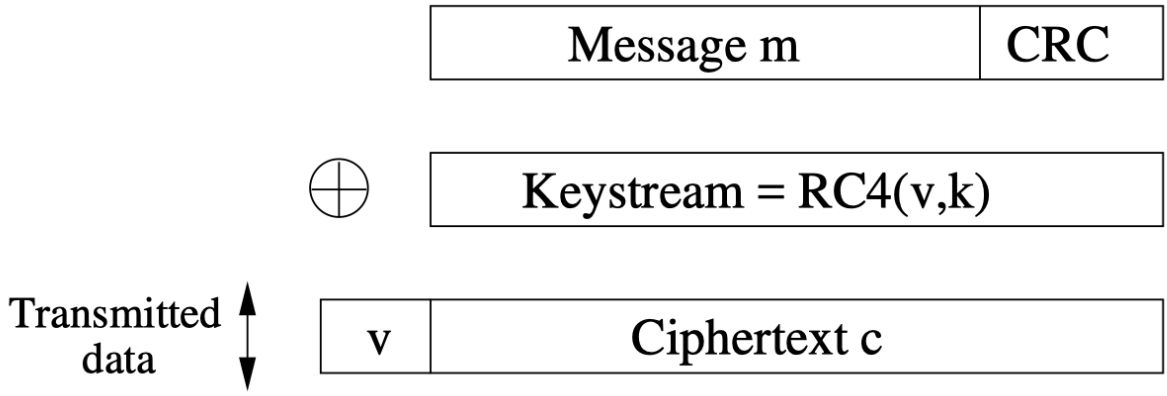


Figure 5: Description of WEP Protocol

Algorithm 4: The receiver of (v, c) does the following

1 Compute $(m \| S) = c \oplus RC4(v \| k)$.
2 Compute $S' = CRC(m)$; reject the packet if $S' \neq S$.

As proved by Borisov, Goldberg, and Wagner in 2021, none of confidentiality, data integrity, and access control are achieved. There are several problems:

1. **IV Collision**:
   - Suppose that two packets (v, c) and (v, c') use the same IV v. Let m, m' be the corresponding plaintexts. Then $c \oplus c' = (m \| S) \oplus (m' \| S')$. Thus, the eavesdropper can compute $m \oplus m'$.
   - If m is known, then m' is immediately available.
   - If m is not known, then one may be able to use the expected distribution of m and m' to discover information about them. This is because some contents of network traffic is predictable.

- To finding IV Collisions:
  - Since there are only 2^{24} choices for the IV, collisions are guaranteed after enough time — a few days on a busy network (5 Mbps).
  - If IVs are randomly selected, then one can expect a collision after about 2^{12} packets. This result comes from the 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 a second time (called a **collision**) is approximately $\sqrt{\pi n/2} \approx \sqrt{n}$
  - Collisions are more likely if keys k are long-lived and the same key is used for multiple mobile stations in a network.
- So, WEP does not provide a high degree of confidentiality.

2. **Checksum is Linear**:
   - CRC-32 is used to check integrity. This is fine for random errors, but not for deliberate ones.
   - It is easy to make controlled changes to (encrypted) packets:
     - Suppose (v, c) is an encrypted packet.
     - Let $c = RC4(v\|k) \oplus (m\|S)$, where k, m, S are unknown.
     - Let $m' = m \oplus \Delta$, where Δ is a bit string. (The 1's in Δ correspond to the bits of m an attacker wishes to change.)
     - Let $c' = c \oplus (\Delta\|CRC(\Delta))$.
     - Then (v, c') is a valid encrypted packet for m'.
   - So, WEP does not provide data integrity.
3. **Integrity Function is Unkeyed**:
   - Suppose that an attacker learns the plaintext m corresponding to a single encrypted packet (v, c).
   - Then, the attacker can compute the RC4 keystream

$$RC4(v\|k) = c \oplus (m\|CRC(m))$$

   - Henceforth, the attacker can compute a valid encrypted packet for any plaintext m' of her choice: (v, c'), where $c' = RC4(v\|k) \oplus (m'\|CRC(m'))$.
   - So, WEP does not provide access control.

2.4.2 The Fluhrer-Mantin-Shamir Attack of WEP

For this attack, we assume that:

- The same 104-bit key k is used for a long period of time. [Most products do this.]
- The IV is incremented for each packet, or a random IV is selected for each packet. [Most products do this.]
- The first plaintext byte of each packet (i.e. the first byte of each m) is known to the attacker. [Most wireless protocols prepend the plaintext with some header bytes which are non-secret.]

To perform the attack, a passive adversary who can collect about 5,000,000 encrypted packets can very easily recover k (and thus totally break the system).

In practice:

- We can buy a $100 wireless card and hack drivers to capture (encrypted) packets.
- On a busy wireless network (5Mbps), 5 million packets can be captured in a few hours, and then k can be immediately computed.

2.4.3 Lessons Learned from WEP's Insecurity

1. **Details matter**: RC4 was considered to be a secure stream cipher. However, it was improperly used in WEP and the result was a highly insecure communications protocol. Do not assume that "obvious" ways of using cryptographic functions are secure.
2. **Attacks only get better; they never get worse**:
   - **Moore's law (1965)**: computers get twice as fast every two years.
   - Known attacks are constantly being tweaked and improved.
   - New attacks are constantly being invented.
3. **Designing security is hard**:
   - Designing cryptographic protocols is complicated and difficult.
   - Clearly state your security objectives.
   - Hire cryptography experts to design your security. Programming or engineering experts are not good enough.
   - Make your protocols available for public scrutiny.
4. **There is a big demand in industry for "security engineers"**:
   - People who have a **deep understanding of applied cryptography** and have **excellent programming skills**.

2.5 ChaCha20 Stream Cipher

The **ChaCha20 stream cipher** is conceptually simple, word-oriented, and uses only simple arithmetic operations (integer addition modulo 2 32 , xor, and left rotations). It is designed

by Dan Bernstein in 2008. It is extremely fast in software, and does not require any special hardware. To date, no security weaknesses have been found.

ChaCha20 Initial State

Notation:

256-bit key: $k = (k_1, k_2, k_3, \ldots, k_8)$ 96-bit nonce: $n = (n_1, n_2, n_3)$

128-bit constant: $f = (f_1, f_2, f_3, f_4)$ 32-bit counter: c

The inititial state is:

f_1	f_2	f_3	f_4
k_1	k_2	k_3	k_4
k_5	k_6	k_7	k_8
c	n_1	n_2	n_3

=

S_1	S_2	S_3	S_4
S_5	S_6	S_7	S_8
S_9	S_{10}	S_{11}	S_{12}
S_{13}	S_{14}	S_{15}	S_{16}

- A hexadecimal digit is a 4-bit number:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

- $f_1 = \texttt{0x61707865}$, $f_2 = \texttt{0x3320646e}$,
  $f_3 = \texttt{0x79622d32}$, $f_4 = \texttt{0x6b206574}$.
- A nonce (or IV) is a non-repeating quantity (either a counter, or a randomly-generated string).

Figure 6: ChaCha20 Initial State

ChaCha20 Quarter Round Function

Notation

$\oplus$: XOR

$\boxplus$: integer addition modulo 2^{32}

$\lll t$: left-rotate by t bit positions

QR: Quarter Round Function

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

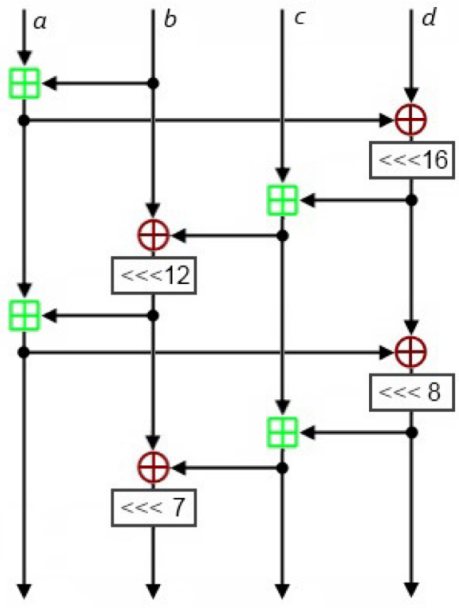
$a \leftarrow a \boxplus b,\ d \leftarrow d \oplus a,\ d \leftarrow d \lll 16$

$c \leftarrow c \boxplus d,\ b \leftarrow b \oplus c,\ b \leftarrow b \lll 12$

$a \leftarrow a \boxplus b,\ d \leftarrow d \oplus a,\ d \leftarrow d \lll 8$

$c \leftarrow c \boxplus d,\ b \leftarrow b \oplus c,\ b \leftarrow b \lll 7$

Output: a, b, c, d



Source: Wikipedia

Figure 7: ChaCha20 Quarter Round Function

Algorithm 5: ChaCha20 Keystream Generator

Select a nonce n and initialize the counter c
while *keystream bytes are required* **do**
    Create the initial state S
    Make a copy S' of S
    Update S by repeating the following 10 times:
        $QR(S_1, S_5, S_9, S_{13})$
        $QR(S_2, S_6, S_{10}, S_{14})$
        $QR(S_3, S_7, S_{11}, S_{15})$
        $QR(S_4, S_8, S_{12}, S_{16})$
        $QR(S_1, S_6, S_{11}, S_{16})$
        $QR(S_2, S_7, S_{12}, S_{13})$
        $QR(S_3, S_8, S_9, S_{14})$
        $QR(S_4, S_5, S_{10}, S_{15})$
    Output $S \oplus S'$ (64 keystream bytes).
    Increment the counter.

For encryption, the keystream bytes are XORed with the plaintext bytes to produce ciphertext bytes. The nonce is included in the ciphertext.

2.6 Block Ciphers

Definition 2.9. A **block cipher** is a SKES 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.

The Data Encryption Standard (DES):

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

Some Desirable Properties of Block Ciphers (Design principles described by Claude Shannon in 1949):

- Security:
  - **Diffusion**: each ciphertext bit should depend on all plaintext bits.
  - **Confusion**: the relationship between key and ciphertext bits should be complicated.

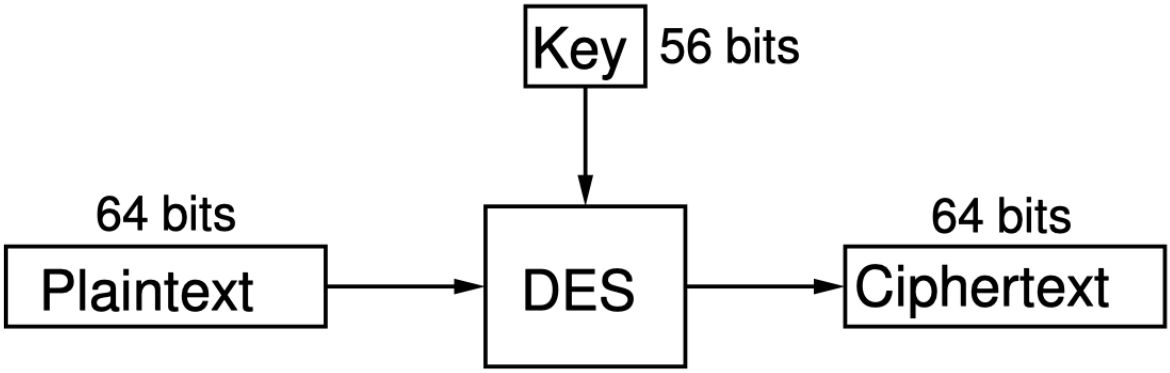


Figure 8: Data Encryption Standard (DES)

  - **Key length**: should be small, but large enough to preclude exhaustive key search.
- Efficiency:
  - Simplicity (easier to implement and analyze).
  - High encryption and decryption rate.
  - Suitability for hardware or software.

DES has several problems:

1. **Small Key Size**:
   - Exhaustive search on key space takes 2^{56} steps and can be easily parallelized
   - DES challenges from RSA Security (3 known PT/CT pairs):
     - June 1997: Broken in 3 months by Internet search.
     - 1999: Broken in 56 hours by DeepCrack machine (1800 chips; $250,000).
     - 2006 Broken in 153 hours by COPACOBANA machine ($10,000).
     - 2012 Broken in 11.5 hours by crack.sh ($200).
   - RC5 64-bit challenge:
     - July 2002: Broken in 1757 days.
     - Participation by 331,252 individuals.
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 $\approx 2^{32}$ (by the birthday paradox). Hence the ciphertext reveals **some** information about the plaintext.

- Small block length is also damaging to some authentication applications

2.6.1 Double-DES

Since the only (substantial) weaknesses known in DES are the obvious ones: small key length and small block length, we may consider perform multiple encryption on DES.

For Double-DES:

- The key is $k = (k_1, k_2), k_1, k_2 \in_R \{0,1\}^{56}$.

  $[k \in_R K$ means that k is chosen **uniformly** and **independently at random** from $K]$
- Encryption: $c = E_{k_2}(E_{k_1}(m))$, in which (E = DES encryption, E^{-1} = DES decryption)
- Decryption: $m = E_{k_1}^{-1}(E_{k_2}^{-1}(c))$.
- The Double-DES key length is $\ell = 112$, so exhaustive key search takes 2^{112} steps (infeasible).
- Note that Block length is unchanged.

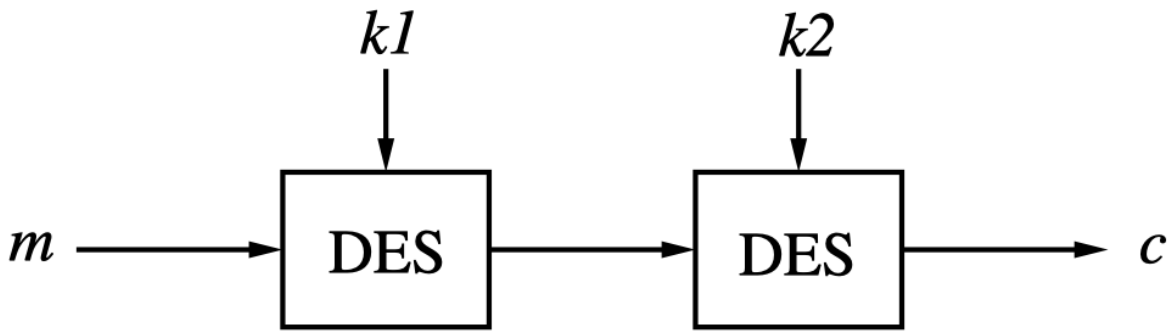


Figure 9: Double-DES Encryption

However, we can attack Double DES in a more efficient way. The main idea is to see that

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

Algorithm 6: Meet-In-The-Middle Attack on Double-DES

Input: 3 known PT/CT pairs $(m_1, c_1), (m_2, c_2), (m_3, c_3)$
Output: The secret key (k_1, k_2)

```
for each h_2 ∈ {0,1}^56 do
    Compute E_{h_2}^{-1}(c_1), and store [E_{h_2}^{-1}(c_1), h_2] in a table sorted by first component.
for each h_1 ∈ {0,1}^56 do
    Compute E_{h_1}(m_1)
    Search for E_{h_1}(m_1) in the table. (We say that E_{h_1}(m_1) matches table entry
     [E_{h_2}^{-1}(c_1), h_2] if E_{h_2}^{-1}(c_1) = E_{h_1}(m_1)
    for each match [E_{h_2}^{-1}(c_1), h_2] in the table do
        if E_{h_2}(E_{h_1}(m_2)) = c_2) then
            if E_{h_2}(E_{h_1}(m_3)) = c_3 then
                Output (h_1, h_2) and STOP
```

Analysis on this algorithm:

- First, see how to calculate FK in the following section.
- Let E be the DES encryption function, so Double-DES encryption is $c = E_{k_2}(E_{k1}(m))$.
- If $\ell = 112, L = 64, t = 3$, then $FK \approx 1/2^{80} \approx 0$ Thus if a Double-DES key (h_1, h_2) is found for which $E_{h_2}(E_{h_1}(m_i)) = c_i$ for $i = 1, 2, 3$, then with very high probability we have $(h_1, h_2) = (k_1, k_2)$.
- If $\ell = 112, L = 64, t = 1$, then $FK \approx 2^{48}$. Thus the expected number of Double-DES keys (h_1, h_2) for which $E_{h_2}(E_{h_1}(m_1)) = c_1$ is $\approx 2^{48}$.
- If $\ell = 112, L = 64, t = 2$, then $FK \approx 1/2^{16}$. Thus the expected number of Double-DES keys (h_1, h_2) for which $E_{h_2}(E_{h_1}(m_1)) = c_1$ and $E_{h_2}(E_{h_1}(m_2)) = c_2$ is $\approx 2^{-16}$.
- So, the number of DES operations is $\approx 2^{56} + 2^{56} + 2 \cdot 2^{48} \approx 2^{57}$
- Space requirements: $2^{56}(64 + 56)$ bits $\approx 1,080,863$ Tbytes.

Time-memory tradeoff: The attack can be modified to decrease the storage requirements at the expense of time:

- Time: 2^{56+s} steps
- memory: 2^{56-s} units
- $1 \leq s \leq 55$.

2.6.2 False Keys

What is the number of known plaintext/ciphertext pairs needed for unique key determination?

Let's formulate this question in the following way: Let E be a block cipher with key space $K = \{0,1\}^\ell$, and plaintext and ciphertext space $\{0,1\}^L$.

Let $k' \in K$ be the secret key chosen by Alice and Bob, and let (m_i, c_i), $1 \leq i \leq t$, be known plaintext/ciphertext pairs, where the plaintext m_i are distinct.

(Note that $c_i = E_{k'}(m_i)$ for all $1 \leq i \leq t$)

Then how large should t be to ensure (with probability very close to 1) that there is only one key $k \in K$ such that $E_k(m_i) = c_i$ for all $1 \leq i \leq t$?

The simple answer is that select t such that $FK \approx 0$

How do we calculate FK?

- For each $k \in K$, the encryption function $E_k : \{0,1\}^L \to \{0,1\}^L$ is a permutation.
- We make the **heuristic** assumption that for each $k \in K$, E_k is a random function (i.e., a randomly selected function). This assumption is certainly false since E_k is not random, and because a random function is almost certainly not a permutation. Nonetheless, it turns out that the assumption is good enough for our analysis.
- Now, fix $k \in K$, $k \neq k'$. Then the probability that $E_k(m_i) = c_i$ for all $1 \leq i \leq t$ is

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

- Thus 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

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

2.6.3 Storage Units

- 10^3 bytes = 1 Kbyte (kilo) $\approx 2^{10}$ bytes
- 10^3 Kbyte = 1 Mbyte (mega) $\approx 2^{20}$ bytes
- 10^3 Mbyte = 1 Gbyte (giga) $\approx 2^{30}$ bytes
- 10^3 Gbyte = 1 Tbyte (tera) $\approx 2^{40}$ bytes
- 10^3 Tbyte = 1 Pbyte (peta) $\approx 2^{50}$ bytes
- 10^3 Pbyte = 1 Ebyte (exa) $\approx 2^{60}$ bytes

2.6.4 Triple-DES

Since:

- Double-DES has the same effective key length as DES.

- Double-DES is not much more secure than DES.

We can consider have Triple-DES:

- Key is $k = (k_1, k_2, k_3)$, $k_1, k_2, k_3 \in_R \{0,1\}^{56}$
- Encryption: $c = E_{k_3}(E_{k_2}(E_{k_1}(m)))$, in which ($E$ = DES encryption, E^{-1} = DES decryption)
- Decryption: $m = E_{k_1}^{-1}(E_{k_2}^{-1}(E_{k_3}^{-1}(c)))$.
- Key length of Triple-DES is $\ell = 168$, so exhaustive key search takes 2^{168} steps (infeasible).
- Meet-in-the-middle attack takes $\approx 2^{112}$ steps.
- So, the effective key length of Triple-DES against exhaustive key search is ≈ 112 bits.

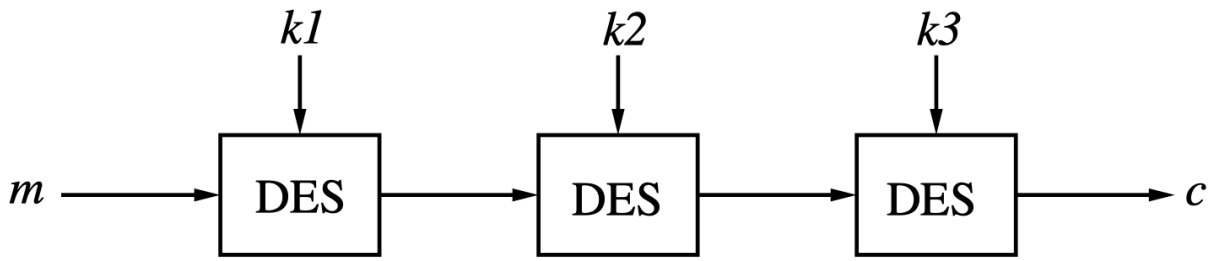


Figure 10: Triple DES Encryption

2.7 The Advanced Encryption Standard (AES)

- Key lengths: 128, 192 and 256 bits.
- Block length: 128 bits.
- Efficient on both hardware and software platforms.
- Availability on a worldwide, non-exclusive, royalty-free basis.

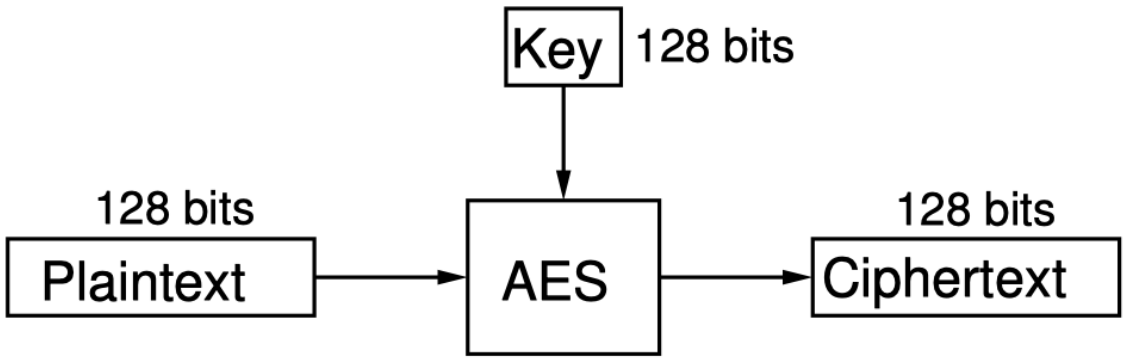


Figure 11: AES Encryption

Definition 2.10. A **substitution-permutation network (SPN)** is an iterated block cipher where a round consists of a **substitution** operation followed by a **permutation** operation.

Components of a SPN cipher:

- n: block length
- ℓ: key length
- h: the number of rounds
- A fixed invertible function $S : \{0,1\}^b \to \{0,1\}^b$, where b is a divisor of n
- A fixed permutation P on $\{1, 2, ..., n\}$
- A key scheduling algorithm that determines subkeys $k_1, k_2, ..., k_h, k_{h+1}$ from a key k

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

Decryption is just the reverse of encryption.

Algorithm 7: SPN Encryption

$A \leftarrow$ plaintext
for $i = 1...h$ **do**
    $A \leftarrow A \oplus k_i$ (XOR)
    $A \leftarrow S(A)$ (Substitution)
    $A \leftarrow P(A)$ (Permutation)
$A \leftarrow A \oplus k_{h+1}$
ciphertext $\leftarrow A$

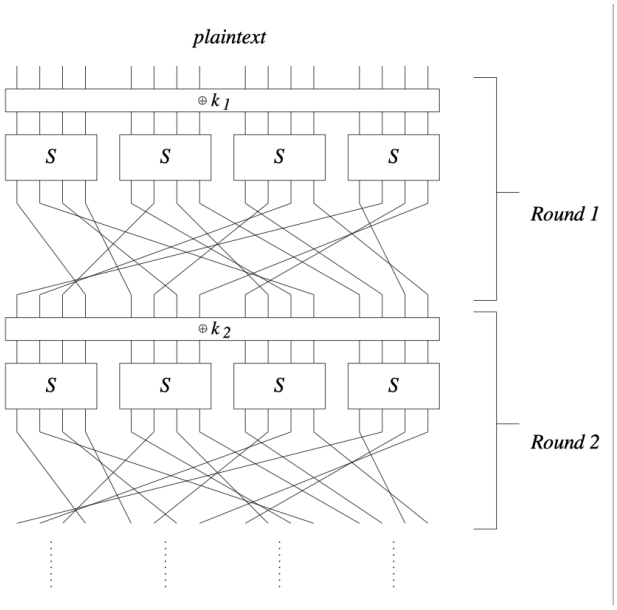


Figure 12: SPN Encryption

- AES is an SPN, where the permutation operation is replaced by two invertible linear transformations.
- All operations are **byte** oriented (e.g., $b = 8$ so the S-box maps 8-bits to 8-bits). This allows AES to be efficiently implemented on software platforms.
- The **block length** of AES is $n = 128$ bits.
- Each **subkey** is 128 bits.
- AES accepts three different key lengths. The number of rounds h depends on the key length:

cipher	key length ℓ	h
AES-128	128	10
AES-192	192	12
AES-256	256	14

AES Round Operations:

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

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$\leftarrow$ plaintext

- After h rounds are completed, a final subkey is XOR-ed with **State**, the result being the ciphertext.
- The AES round function uses four **invertible** operations:
  1. `AddRoundKey` (key mixing). This step is bitwise-XOR each byte of **State** with the corresponding byte of the subkey.

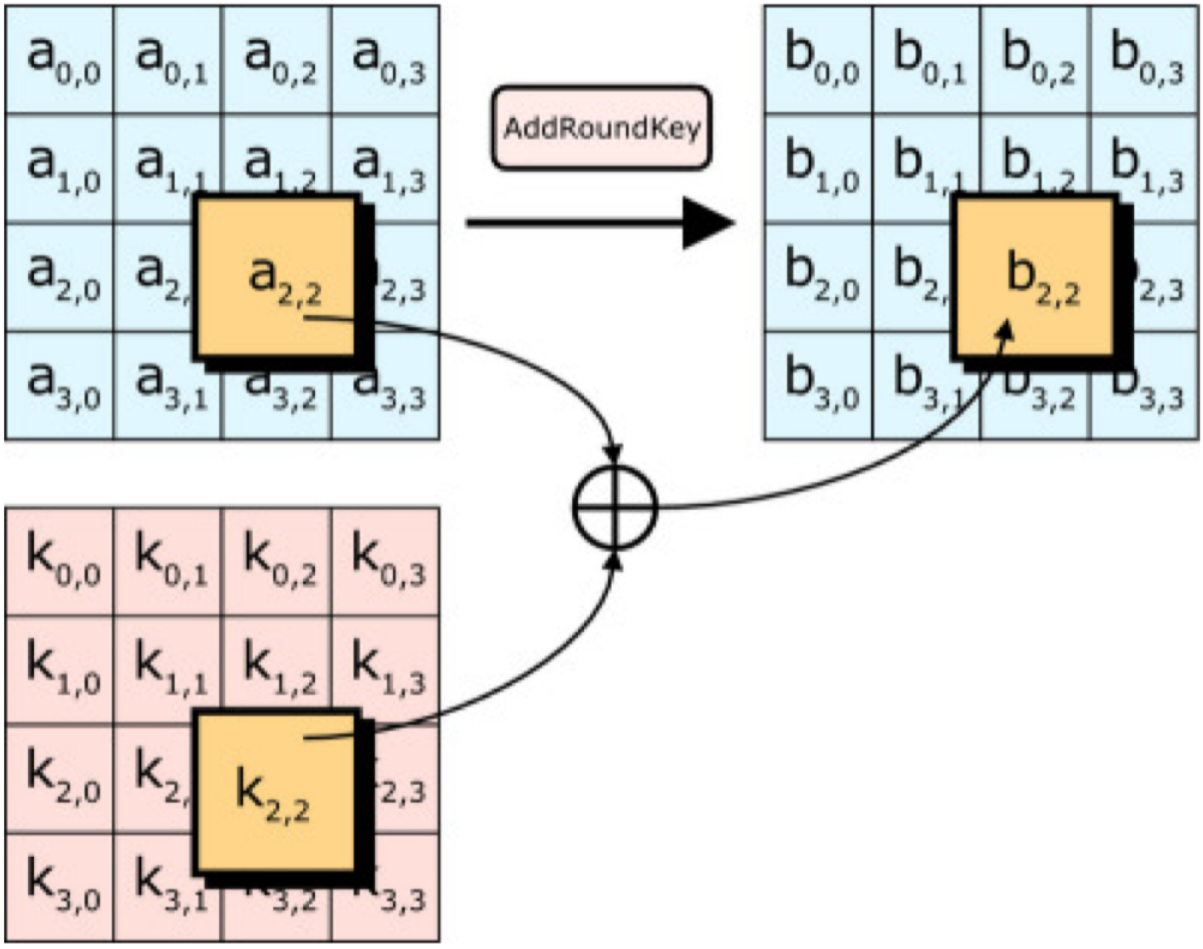


Figure 13: AES Add Round Key

2. `SubBytes` (S-box). This step is take each byte in **State** and replace it with the output of the S-box. Here, $S : \{0,1\}^8 \to \{0,1\}^8$ is a fixed, public, invertible, and non-linear function.

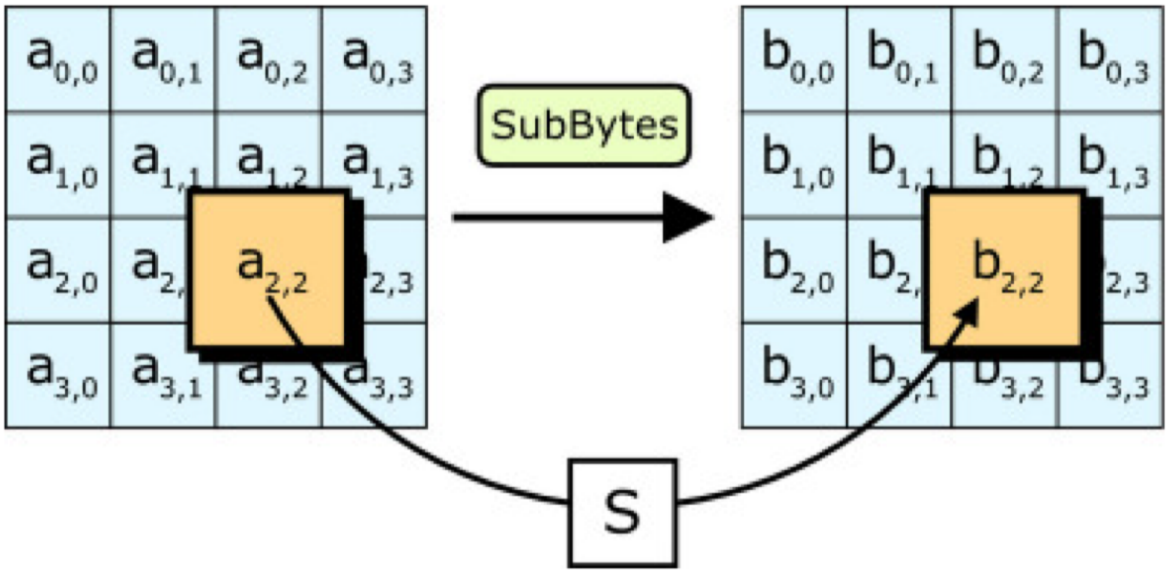


Figure 14: AES Substitute Bytes

3. `ShiftRows` (permutation). This step is permute the bytes of State by applying a **cyclic shift** to each row

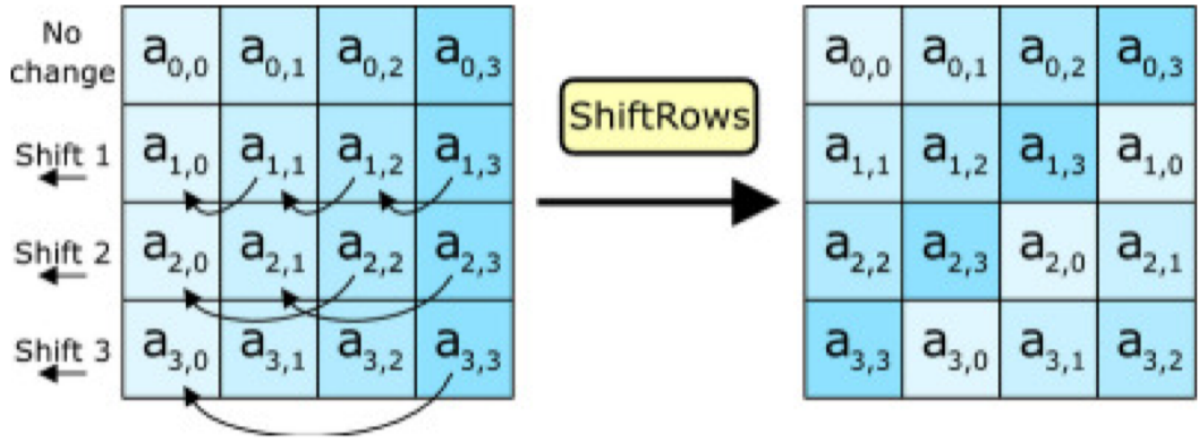


Figure 15: AES Shift Rows

4. `MixColumns` (linear transformation).

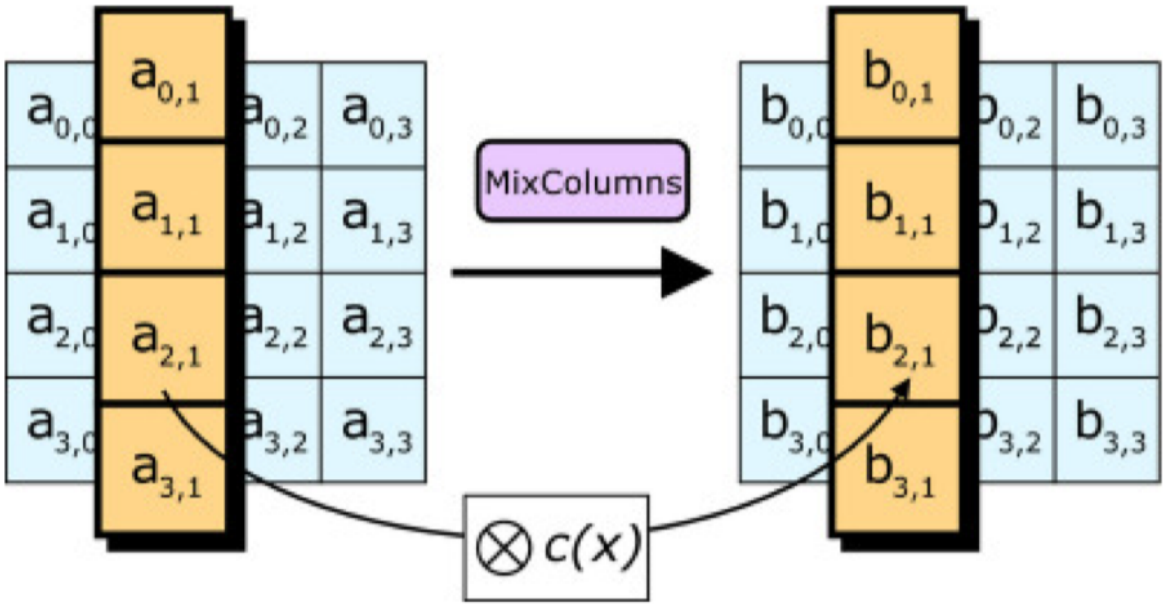


Figure 16: AES Mix Columns

2.7.1 Operation of AES

Below we show some detailed explanation on the above operations. First, we need to define

Definition 2.11. The Finite Field $GF(2^8)$: The elements of the finite field $GF(2^8)$ are the polynomials of degree at most 7 in $\mathbb{Z}_2[y]$, with addition and multiplication performed modulo the irreducible polynomial $f(y) = y^8 + y^4 + y^3 + y + 1$.

Then, we interpret an 8-bit string $a = a_7a_6a_5a_4a_3a_2a_1a_0$ as coefficients of the polynomial $a(y) = a_7y^7 + a_6y^6 + a_5y^5 + \cdots + a_1y + a_0$, and vice versa.

We define the following operations:

- **Addition**: add coefficients with respect to each power of y. Perform modulo 2 on addition.
- **Multiplication**: multiply two polynomials together, finds the remainder upon division by $f(y) = y^8 + y^4 + y^3 + y + 1$.
- **Inversion**: Find another polynomial in which the result of multiplication is 01.

Here is how S-box is calculated:

1. Let $p \in \{0,1\}^8$, and consider p as an element of $GF(2^8)$.
2. Let $q = p^{-1}$ if $p \neq 0$, and $q = p$ if $p = 0$.

3. Define $q = (q_7q_6q_5q_4q_3q_2q_1q_0)$
4. Compute

$$r = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \mod 2$$

5. Then $S(p) = r = (r_7r_6r_5r_4r_3r_2r_1r_0)$.

Here is a look up table for AES S-box. In the following example, $S(a8) = c2$.

	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

Figure 17: AES SBox

Here is how Mix Columns is calculated:

1. Read column i of State as a polynomial:

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

   Essentially, we interpret the coefficients as elements of the finite field $GF(2^8)$.

2. Multiply this polynomial with the constant polynomial $c(x) = \texttt{03}\cdot x^3 + \texttt{01}\cdot x^2 + \texttt{01}\cdot x + \texttt{02}$ and reduce modulo $x^4 - 1$. This gives a new polynomial: $b_{0,i} + b_{1,i}x + b_{2,i}x^2 + b_{3,i}x^3$

Equivalently,

Algorithm 8: The $\otimes c(x)$ Operation of AES

Let $a(x) = a_0 + a_1x + a_2x^2 + a_3x^3$, where each $a_i \in GF(2^8)$
Let $c(x) = \texttt{03} \cdot x^3 + \texttt{01} \cdot x^2 + \texttt{01} \cdot x + \texttt{02}$ where `01, 02, 03` are elements in $GF(2^8)$ (written in hexadecimal)
To compute $a(x) \otimes c(x)$:
    Compute $d(x) = a(x) \times c(x)$ (polynomial multiplication, where coefficient arithmetic is in $GF(2^8)$).
    Divide by $x^4 - 1$ to find the remainder polynomial $r(x)$ (equivalently, replace x^4 by 1, x^5 by x, and x^6 by x^2). Then $a(x) \otimes c(x) = r(x)$.

Finally, we get to encrypt using AES:

Algorithm 9: AES Encryption

From the key k derive $h + 1$ subkeys $k_0, k_1, ..., k_h$.
$State \leftarrow$ plaintext
$State \leftarrow State \oplus k_0$
for $i = 1...h - 1$ **do**
    $State \leftarrow$ `SubBytes`$(State)$
    $State \leftarrow$ `ShiftRows`$(State)$
    $State \leftarrow$ `MixColumns`$(State)$
    $State \leftarrow State \oplus k_i$
$State \leftarrow$ `SubBytes`$(State)$
$State \leftarrow$ `ShiftRows`$(State)$
$State \leftarrow State \oplus k_h$
ciphertext $\leftarrow State$

Note that `InvMixcolumns` is multiplication by $d(x) = \texttt{0e} + \texttt{09}x + \texttt{0d}x^2 + \texttt{0b}x^3$ modulo $x^4 - 1$.

As for AES Key Schedule:

- For 128-bit keys, AES has 10 rounds, so we need 11 subkeys.
- The first subkey $k0 = (r_0, r_1, r_2, r_3)$ is the actual AES key.
- The second subkey is $k_1 = (r_4, r_5, r_6, r_7)$.

Algorithm 10: AES Decryption

From the key k derive $h+1$ subkeys $k_0, k_1, ..., k_h$.
$State \leftarrow$ ciphertext
$State \leftarrow State \oplus k_h$
$State \leftarrow$ `InvShiftRows`$(State)$
$State \leftarrow$ `InvSubBytes`$(State)$
for $i = h-1...1$ **do**
    $State \leftarrow State \oplus k_i$
    $State \leftarrow$ `InvMixColumns`$(State)$
    $State \leftarrow$ `InvShiftRows`$(State)$
    $State \leftarrow$ `InvSubBytes`$(State)$
$State \leftarrow State \oplus k_0$
plaintext $\leftarrow State$

- The third subkey is $k2 = (r_8, r_9, r_{10}, r_{11})$.
- $\cdots$
- The eleventh subkey is $k_{10} = (r_{40}, r_{41}, r_{42}, r_{43})$.

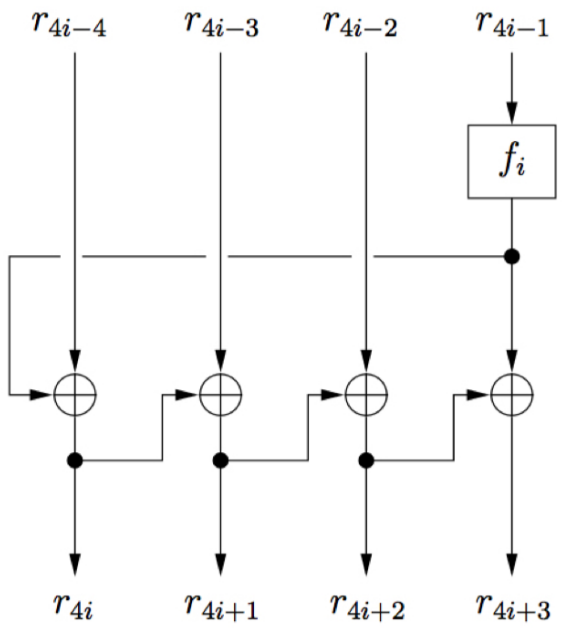
The functions $f_i : \{0,1\}^{32} \to \{0,1\}^{32}$ are defined as follows:

1. The input is divided into 4 bytes: (a, b, c, d).
2. Left-rotate the bytes: (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 are defined as follows:

i	ℓ_i	i	ℓ_i	i	ℓ_i	i	ℓ_i	i	ℓ_i
1	0x01	2	0x02	3	0x04	4	0x08	5	0x10
6	0x20	7	0x40	8	0x80	9	0x1b	10	0x36



Generating k_i, $1 \leq i \leq 10$.

Figure 18: AES Key Schedule for 128-bit keys

2.8 Performance of SKES

Speed benchmarks (2017) for software implementations on an Intel Core i9 2.9 GHz 6-core Coffee Lake (8950HK) using OpenSSL 1.1.1d.

Algorithm	block length (bits)	key length (bits)	speed (Mbytes/sec)
RC4	-	128	644
ChaCha20	-	256	1846
DES	64	56	98
Triple-DES	64	168	39
AES (software)	128	128	254
AES (AES-NI)	128	128	1632

2.9 Block Cipher Modes of Operation

How should we use a block cipher $E_k : \{0,1\}^L \to \{0,1\}^L$ to encrypt m?

We have:

- Electronic Codebook (ECB) Mode:
  - Encrypt blocks independently, one at a time: $c = c_1, c_2, ..., c_t$, where $c_i = E_k(m_i)$.

- Decryption: $m_i = E_k(c_i), i = 1, 2, ..., t$.
- Drawback: Identical plaintexts result in identical ciphertexts (under the same key), and thus ECB encryption is not (semantically) secure against chosen-plaintext attacks.

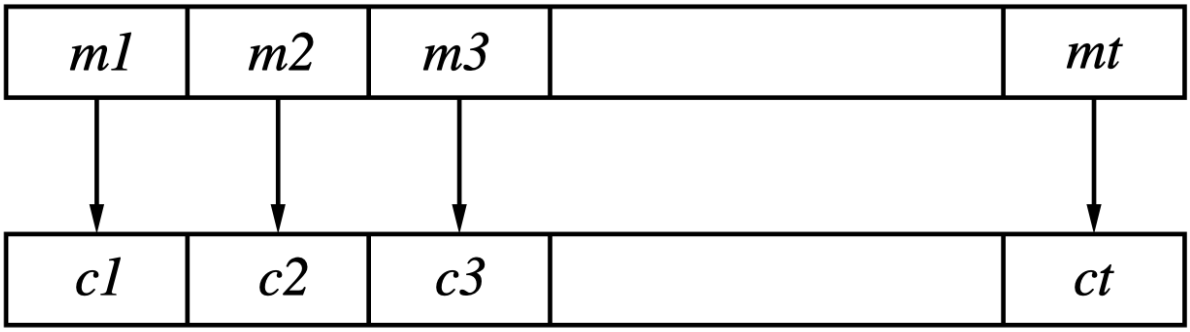


Figure 19: Electronic Codebook (ECB) Mode

- Cipher Block Chaining (CBC) Mode:
  - Encryption: Select $c_0 \in_R \{0,1\}^L$ (c_0 is a random non-secret IV). Then compute $c_i = E_k(m_i \oplus c_{i-1}), i = 1, 2, ..., t$.
  - Ciphertext is $(c_0, c_1, c_2, ..., c_t)$
  - Decryption: $m_i = E_k(c_i) \oplus c_{i-1}, i = 1, 2, ..., t$.
  - Identical plaintexts with different IVs result in different ciphertexts and for this reason CBC encryption is (semantically) secure against chosen-plaintext attacks (for a well chosen block cipher E)

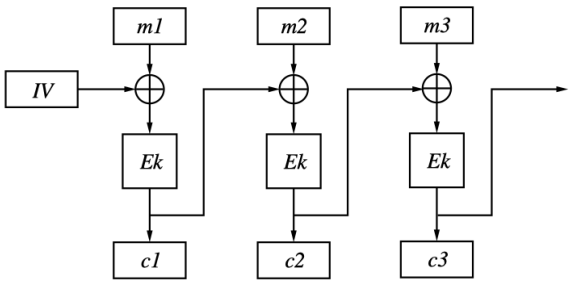


Figure 20: Cipher Block Chaining (CBC) Mode

Chapter 3 Hash Function

3.1 Definitions and Terminology

Definition 3.1. A **hash function** is a mapping H such that:

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

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

  (L is large, for example, $L = 2^{64}$; n is small, for example $n = 256$)

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

Notes:

- H is called an n-bit hash function.
- $H(x)$ is called the **hash** or **message digest** of x.
- The description of a hash function is public. There are no secret keys.
- For simplicity, we will usually write $\{0,1\}^*$ instead of $\{0,1\}^{\leq L}$

3.2 Hash Functions from Block Ciphers

This one is called **Davies-Meyer hash function**. We let E_k be an m-bit block cipher with n-bit key k, and let IV be a fixed m-bit initializing value.

Algorithm 11: Davies-Meyer hash function

1 Break up $x\|1$ into n-bit blocks: $x = x_1, x_2, ..., x_t$, padding out the last block with 0 bits if necessary.
2 Define $H_0 = IV$
3 **for** $i = 1...t$ **do**
4 Compute $H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1}$
5 Define $H(x) = H_t$

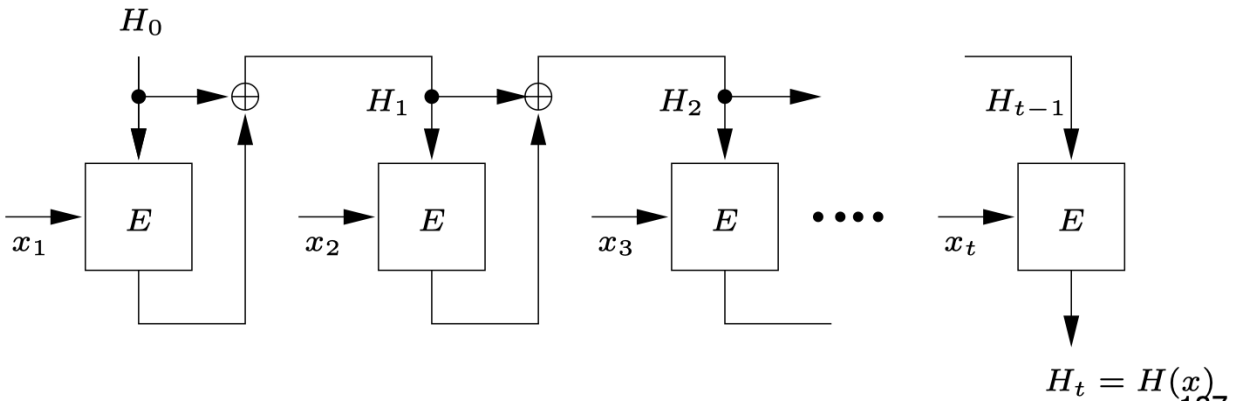


Figure 21: Davies Meyer Hash Function

3.3 Preimage Resistance, 2nd Preimage Resistance, and Collision Resistance

Definition 3.2. Preimage resistance: Given a hash value $y \in_R \{0,1\}^n$, it is computationally infeasible to find (with non-negligible probability of success) **any** $x \in \{0,1\}^*$ such that $H(x) = y$.

x is called a **preimage** of y

$x \in_R S$ means that x is chosen independently and uniformly at random from S.

This could be useful in Password protection on a multi-user computer system:

- Server stores $(userid, H(password))$ in a password file. Thus, if an attacker gets a copy of the password file, she does not learn any passwords.
- Requires preimage-resistance

Definition 3.3. 2nd preimage resistance: Given an input $x \in_R \{0,1\}^*$, it is computationally infeasible to find (with non-negligible probability of success) a second input $x' \in \{0,1\}^*$, $x \neq x'$, such that $H(x') = H(x)$.

This could be useful in **Modification Detection Codes (MDCs)**:

- To ensure that a message m is not modified by unauthorized means, one computes $H(m)$ and protects $H(m)$ from unauthorized modification.
- For example, virus protection.
- Requires 2nd preimage resistance.

Definition 3.4. Collision resistance: It is computationally infeasible to find (with non-negligible probability of success) two distinct inputs $x, x' \in \{0,1\}^*$ such that $H(x) = H(x')$.

The pair (x, x') is called a collision for H.

This could be useful in **Message digests for digital signature schemes**:

- For reasons of efficiency, instead of signing a (long) message, the (much shorter) message digest is signed.
- Requires preimage-resistance, 2nd preimage resistance, and collision resistance.
- To see why collision resistance is required: Suppose that the legitimate signer Alice can find two messages x_1 and x_2 , with $x_1 \neq x_2$ and $H(x_1) = H(x_2)$.
- Alice can sign x_1 and later claim to have signed x_2.

Relationship between PR, 2PR, and CR:

- CR implies 2PR
- 2PR does not guarantee CR
- CR does not guarantee PR
- if H is "somewhat" random, i.e. all hash values have roughly the same number of preimages, then, CR implies PR
- PR does not guarantee 2PR
- 2PR implies PR (for somewhat uniform H)

Proposition 3.5. Collision Resistance implies 2nd Preimage Resistance

Proof 3.6. We prove the contrapositive. Suppose H is not 2PR.

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

Proposition 3.7. 2nd Preimage Resistance does not guarantee Collision Resistance

Proof 3.8. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is 2PR. Consider $\bar{H} : \{0,1\}^* \to \{0,1\}^n$ defined by:

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

Suppose that $\bar{H}$ is not 2PR. So, given $x \in_R \{0,1\}^*$, we can efficiently find $x' \in \{0,1\}^*$, $x \neq x'$, with $\bar{H}(x') = \bar{H}(x)$. With probability essentially 1, we can assume that $x \neq 0, 1$. Hence, $\bar{H}(x) = H(x)$. Now, if $x' \neq 1$, then $\bar{H}(x') = H(x') = H(x)$; while if $x' = 1$, then $\bar{H}(x') = \bar{H}(1) = H(0) = H(x)$. In either case, we have found a second preimage for x with respect to H. So, $\bar{H}$ must be 2PR.

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

Proposition 3.9. Collision Resistance does not guarantee Preimage Resistance

Proof 3.10. Suppose that $H : \{0,1\}^* \to \{0,1\}^n$ is CR.

Consider $\bar{H} : \{0,1\}^* \to \{0,1\}^{n+1}$ defined by

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

Then, $\bar{H}$ is CR (since H is). And, $\bar{H}$ is not PR since preimages can be efficiently found for at least half of all $y \in \{0,1\}^{n+1}$ □

Proposition 3.11. if H is "somewhat" random, i.e. all hash values have roughly the same number of preimages, then, Collision Resistance implies Preimage Resistance

Proof 3.12. We prove the contrapositive.

Suppose H is not PR. Select $x \in_R \{0,1\}^*$ and compute $y = H(x)$. Since H is not PR, we can efficiently find a preimage x' of y.

Since H is somewhat uniform, we expect that y has many preimages, and thus $x' \neq x$ with very high probability. Thus, (x, x') is a collision for H that we have efficiently found, so H is not CR. □

From now on, we shall assume that hash functions are somewhat uniform.

Proposition 3.13. Preimage Resistance does not guarantee 2nd Preimage Resistance

Proof 3.14. Suppose $H : \{0,1\}^* \to \{0,1\}^n$ is PR.

Define $\bar{H} : \{0,1\}^* \to \{0,1\}^n$ by

$$\bar{H}(0, x_2, ..., x_t)$$

Then, $\bar{H}$ is PR, but $\bar{H}$ is 2PR. □

Proposition 3.15. For "somehwhat" uniform H. 2nd Preimage Resistance implies Preimage Resistance

We may refer to the following figure as a summary.

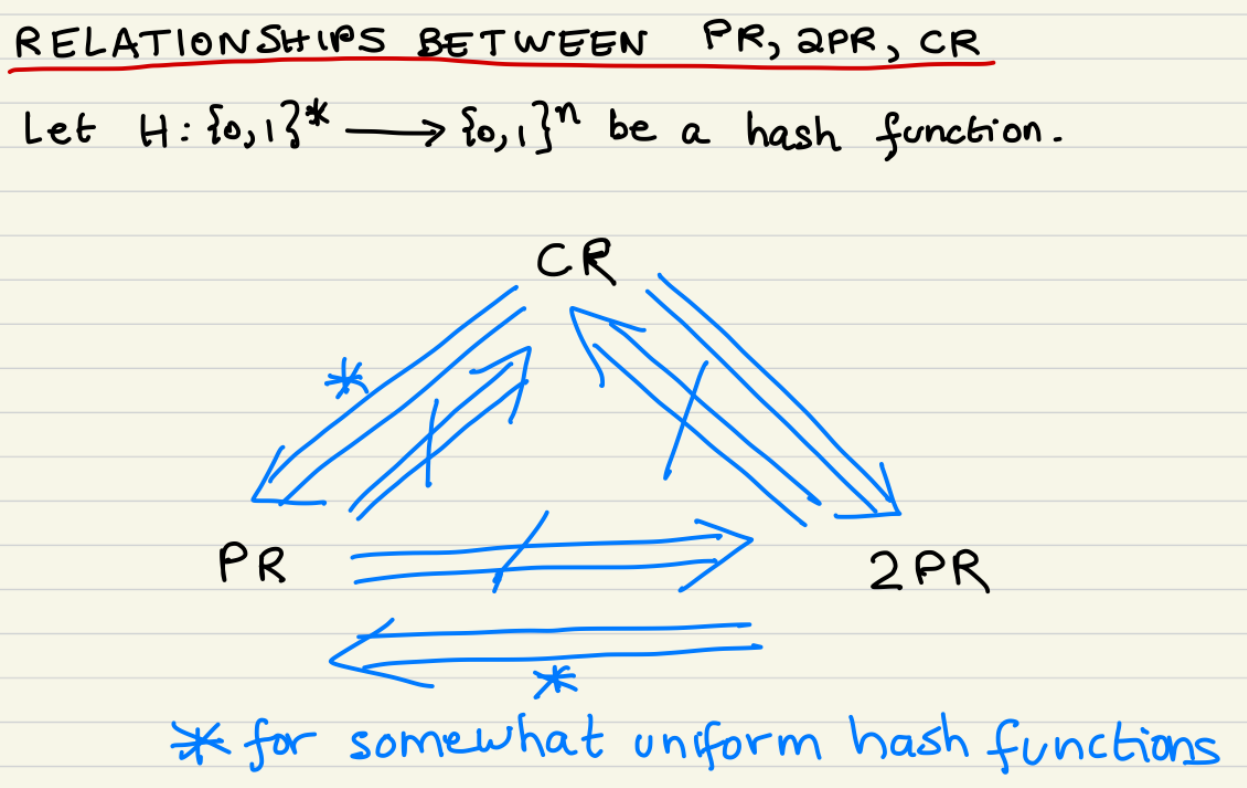


Figure 22: Relationship between PR, 2PR, and CR

3.4 Generic Attack

Definition 3.16. A **generic** attack on a hash function $H : \{0,1\}^* \to \{0,1\}^n$ does not exploit any properties the specific hash function may have.

In the **analysis** of a generic attack, we view H as a **random function** in the sense that for each $x \in \{0,1\}^*$, we assume that the value $y = H(x)$ was chosen 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 stored.

Algorithm 12: Generic Attack for Finding Preimages

1 Given $y \in_R \{0,1\}^n$, repeatedly select arbitrary $x \in \{0,1\}^*$ until $H(x) = y$

- Expected number of steps is $\approx 2^n$
- This attack is infeasible if $n \geq 128$
- It has been proven that this generic attack for finding preimages is optimal, i.e., no faster generic attack exists

Algorithm 13: Generic Attack for Finding Collisions

1 Select arbitrary $x \in \{0,1\}^*$ and store $(H(x), x)$ in a table sorted by first entry.
Continue until a collision is found.

- Expected number of steps is $\sqrt{\pi 2^n/2} \approx \sqrt{2^n}$ (by birthday paradox)
- This attack is infeasible if $n \geq 256$.
- It has been proven that this generic attack for finding collisions is optimal in terms of the number of hash function evaluations
- Expected number of space required is $\sqrt{\pi 2^n/2} \approx \sqrt{2^n}$

3.5 Van Oorschot and Wiener Parallel Collision Search

- Expected number of steps: $\sqrt{2^n}$
- Expected space required: negligible.
- Easy to parallelize: m-fold speedup with m processors.
- Conclusion: If collision resistance is desired, then use an n-bit hash function with $n \geq 256$

We want to find a collision for $H : \{0,1\}^* \to \{0,1\}^n$, assuming that H is a random function.

In terms of notation, let $N = 2^n$. Let's define a sequence $\{x_i\}_{i \geq 0}$ by $x_0 \in_R \{0,1\}^n$, and $x_i = H(x_{i-1})$ for $i \geq 1$.

Let j be the smallest index from which $x_j = x_i$ for some $i < j$; such j must exist. Then $x_{j+\ell} = x_{i+\ell}$ for all $\ell \geq 1$. By the birthday paradox, $E[j] \approx \sqrt{\frac{\pi N}{2}} \approx \sqrt{N}$. In fact, $E[i] \approx \frac{1}{2}\sqrt{N}$ and $E[j-i] \approx \frac{1}{2}\sqrt{N}$.

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

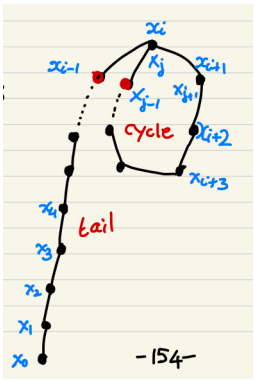


Figure 23: Main Idea of VW Search

However, storage would be a problem (too many points to store). VW's idea is to store only **distinguished points**.

We can select an easily testable distinguishing property for elements of $\{0,1\}^n$ (for example, leading 32 bits are all 0). Let Θ be the proportion of elements of $\{0,1\}^n$ that are distinguished.

In VW's algorithm, we compute $x_0, x_1, x_2, \ldots$ and only store the points in the sequence that are distinguished.

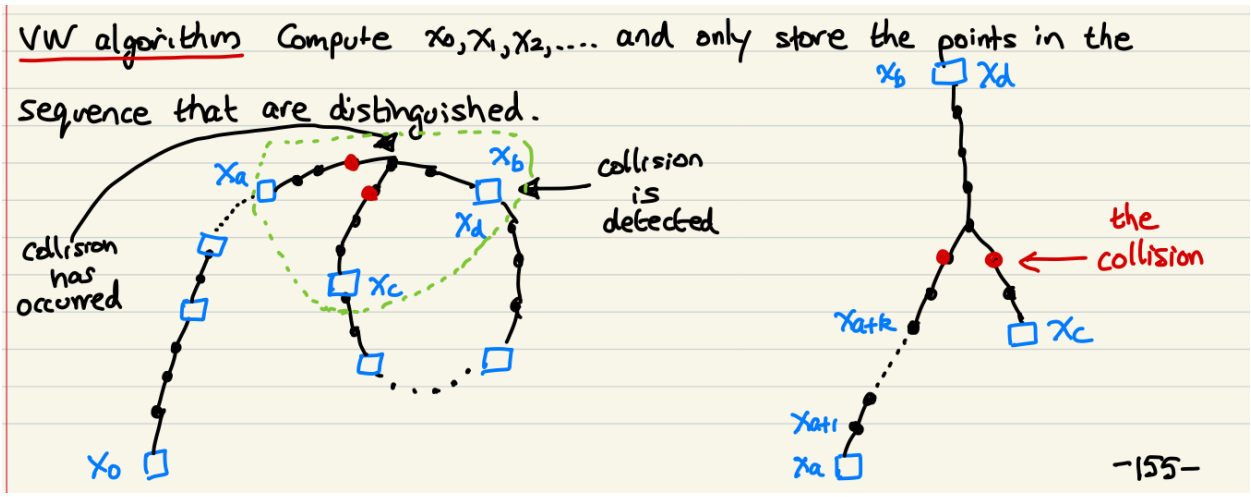


Figure 24: VW's Search Flow

The analysis of the algorithm is as follows:

- Stage 1: the expected number of H-evalutations is

$$\underbrace{\sqrt{\frac{\pi N}{2}}}_{\text{collision occurs}} + \underbrace{\frac{1}{\Theta}}_{\text{collision is detected}} \approx \sqrt{N} + \frac{1}{\Theta}$$

Algorithm 14: VW Collision Finding

Stage 1: (Detecting a collision)
Select $x_0 \in_R \{0,1\}^n$
Store $(x_0, 0, -)$ in a sorted table
$LP \leftarrow x_0$ (LP means last point stored)
for $d = 1, 2, ...$ **do**
    Compute $x_d = H(x_{d-1})$
    if x_d *is distinguished* **then**
        if x_d *is already in table, Say* $x_d = x_b$ *where* $b < d$ **then**
            Go to Stage 2
        Store (x_d, d, LP) in table
        $LP \leftarrow x_d$
Stage 2: (Finding a collision)
Set $\ell_1 \leftarrow b - a$, $\ell_2 \leftarrow d - c$
Suppose $\ell_1 \geq \ell_2$, set $k \leftarrow \ell_1 - \ell_2$
Compute $x_{a+1}, x_{a+2}, ..., x_{a+R}$
for $m = 1, 2, 3...$ **do**
    Compute (x_{a+k+m}, x_{c+m})
Until $x_{a+k+m} = x_{c+m}$
The collision is $(x_{a+k+m-1}, x_{c+m-1})$

- Stage 2: the expected number of H-evalutations is $\leq \frac{3}{\Theta}$
- Overall the expected running time is $\sqrt{N} + \frac{4}{\Theta}$
- Storage: $\approx \Theta\sqrt{N}(3n)$ (since each table entry has bit length $3n$)
- For example, let $n = 128$. Take $\Theta = 1/2^{32}$. Then, the expected runtime of VW collision search is 2^{64} H-evaluations, which is feasible, and the expected storage is 241 Gbytes.

We can parallelized this algorithm by running a copy of VW on each of the m processors. Report distinguished points to a central server.

The expected runtime would be $\sqrt{N}/m + \frac{4}{\Theta}$. The expected storage is $\Theta\sqrt{N}(3n)$ bits.

This gives us a factor m speedup. There are no communication between processors, and there are only occasional communication with central server.

3.6 Iterated Hash Function

For iterated hash functions, we have:

- Fixed **initializing value** $IV \in \{0,1\}^n$.

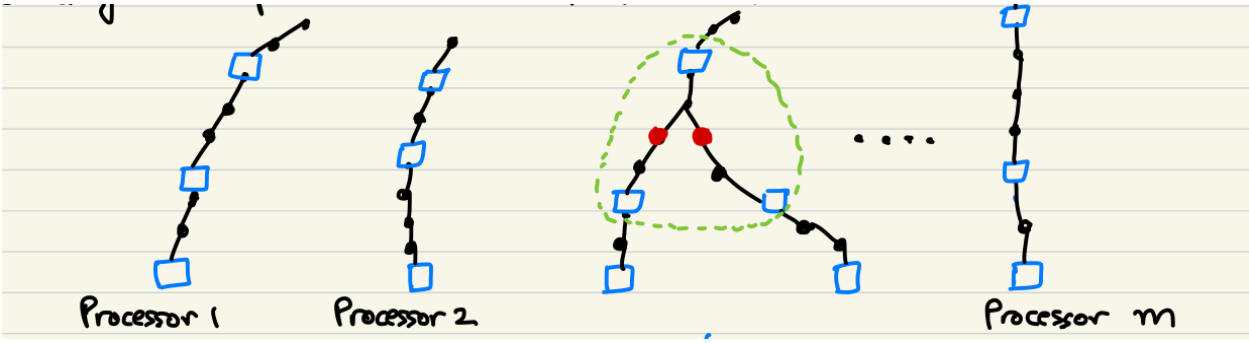


Figure 25: Parallelizing VW Search

- A **compression function** $f : \{0,1\}^{n+r} \to \{0,1\}^n$, which is efficiently computable.

Algorithm 15: Iterated Hash Function

Input: x has bitlength $b < 2^r$

Output: $H(x)$

1 Break up x into r-bit blocks $\bar{x} = x_1, x_2, ..., x_t$, padding out the last block with 0 bits if necessary.

2 Define x_{t+1}, the **length block**, to hold the right-justified binary representation of b

3 Define $H_0 = IV$ **for** $i = 1, 2, ..., t+1$ **do**

4 Compute $H_i = f(H_{i-1}, x_i)$

5 Output $H(x) = H_{t+1}$

The H_i's are called chaining variables.

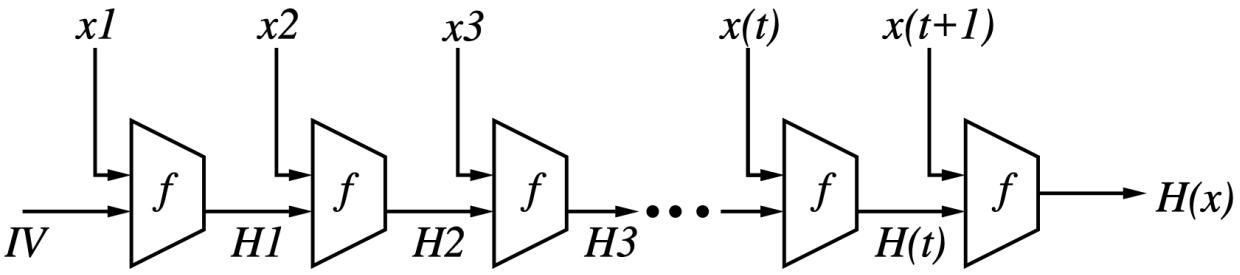


Figure 26: Iterated Hash Functions

Theorem 3.17. Merkle's Theorem: If the compression function f is collision resistant, then the hash function H is also collision resistant.

Proof 3.18. Suppose that H is not collision resistance, then we can efficiently find $x, x' \in \{0,1\}^*$ with $x \neq x'$, $H(x) = H(x')$. Let $\bar{x} = x_1, x_2, ..., x_t$ where $b = bitlength(x)$, $x_{t+1} =$ length block, and $\bar{x'} = x'_1, x'_2, ..., x'_t$ where $b' = bitlength(x')$, $x'_{t+1} =$ length block.

We can efficiently compute

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

Since $H(x) = H(x')$, we have $H_{t+1} = H'_{t'+1}$

Now, if $b \neq b'$, then $x_{t+1} \neq x'_{t'+1}$. Then, $(H_t, x_{t+1}), (H'_{t'}, x'_{t'+1})$ is a collision for f that we have efficiently found. Hence, f is not collision resistant.

Suppose that $b = b'$. Then $t = t'$ and $x_{t+1} = x'_{t'+1}$. Let i be the largest index, $0 \leq i \leq t$ for which

$$(H_i, x_{i+1}) \neq (H'_i, x'_{i+1})$$

Such an i must exist, since $x \neq x'$. 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. Hence f is not collision resistant. □

Merkle's theorem reduces the problem of designing collision-resistant hash functions to that of designing collision-resistant compression functions.

3.7 Provable Security

- A major theme of cryptographic research is to formulate precise security definitions and assumptions, and then prove that a cryptographic protocol is **secure**.
- A **proof of security** is certainly desirable since it rules out the possibility of attacks being discovered in the future.
- However, it isn't always easy to assess the practical security assurances (if any) that a security proof provides:
  - The assumptions might be unrealistic, or false, or circular.
  - The security proof might be fallacious.
  - The security model might not account for certain kinds of realistic attacks.

- The security proof might be asymptotic.
- The security proof might have a large tightness gap.

3.8 MDx-Family of Hash Functions

- **MDx** is a family of iterated hash functions.
- **MD4** was proposed by Ron Rivest in 1990.
- MD4 has **128-bit outputs**.
- Wang et al. (2004) found collisions for MD4 **by hand**.
- Leurent (2008) discovered an algorithm for finding MD4 preimages in 2^{102} steps.

MD5 Hash Function:

- MD5 is a strengthened version of MD4.
- Designed by Ron Rivest in 1991.
- MD5 has 128-bit outputs.
- Wang and Yu (2004) found MD5 collisions in 2^{39} steps.
- MD5 collisions can now be found in 2^{24} steps (in a few seconds on a laptop computer).
- Sasaki & Aoki (2009) discovered a method for finding preimages for MD5 in $2^{123.4}$ steps.
- So, MD5 should not be used if collision resistance is required, but is probably okay as a preimage-resistant hash func- tion.

3.9 SHA-Family of Hash Functions

SHA-1:

- Secure Hash Algorithm (**SHA**) was designed by NSA and published by NIST in 1993 (FIPS 180).
- **160-bit** iterated hash function, based on MD4.
- Slightly modified to **SHA-1** (FIPS 180-1) in 1994 in order to fix an (undisclosed) security weakness.
- Wang et al. (2005) discovered a collision-finding algorithm for SHA-1 that takes 2^{63} steps.
- No preimage or 2nd preimage attacks (that are faster than the generic attacks) are known for SHA-1

SHA-2 Family:

- In 2001, NSA proposed variable output-length versions 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) and **512** bits (SHA-512).
- As of 2020: No weaknesses in any of these hash functions have been found.
- The security levels of these hash functions against collision-finding attacks is the same as the security levels of Triple-DES, AES-128, AES-192 and AES-256 against exhaustive key search attacks.
- The SHA-2 hash functions are standardized in FIPS 180-2.

SHA-256:

- It is an iterated hash function (Merkle meta-method).
- $n = 256, r = 512$
- Compression function is $f : \{0,1\}^{256+512} \to \{0,1\}^{256}$.
- Input: bitstring x of arbitrary bitlength $b \geq 0$
- Output: 256-bit hash value $H(x)$ of x.

Here is a table for SHA-256 Notation:

A, B, C, D, E, F, G, H	32 bits words
$+$	addition modulo 2^{32}
$\bar{A}$	bitwise complement
$A \gg s$	shift A right through s positions
$A \hookrightarrow s$	rotate A right through s positions
AB	bitwise AND
$A \oplus B$	bitwise exclusive-OR
$f(A, B, C)$	$AB \oplus \bar{A}C$
$g(A, B, C)$	$AB \oplus AC \oplus BC$
$r_1(A)$	$(A \hookrightarrow 2) \oplus (A \hookrightarrow 13) \oplus (A \hookrightarrow 22)$
$r_2(A)$	$(A \hookrightarrow 6) \oplus (A \hookrightarrow 11) \oplus (A \hookrightarrow 25)$
$r_3(A)$	$(A \hookrightarrow 7) \oplus (A \hookrightarrow 18) \oplus (A \gg 3)$
$r_4(A)$	$(A \hookrightarrow 17) \oplus (A \hookrightarrow 19) \oplus (A \gg 10)$

Here is a table for SHA-256 Constants:

$h_1 =$ `0x6a09e667`	$h_2 =$ `0xbb67ae85`
$h_3 =$ `0x3c6ef372`	$h_4 =$ `0xa54ff53a`
$h_5 =$ `0x510e527f`	$h_6 =$ `0x9b05688c`
$h_7 =$ `0x1f83d9ab`	$h_8 =$ `0x5be0cd19`

These words were obtained by taking the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers.

Here is a table for SHA-256 Per-round integer additive constants:

$$\begin{array}{ll} y_0 = \texttt{0x428a2f98} & y_1 = \texttt{0x71374491} \\ y_2 = \texttt{0xb5c0fbcf} & y_3 = \texttt{0xe9b5dba5} \\ \vdots & \vdots \\ y_{62} = \texttt{0xbef9a3f7} & y_{63} = \texttt{0xc67178f2} \end{array}$$

These words were obtained by taking the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers.

SHA-256 Preprocessing:

- Pad x (with 1 followed by as few 0's as possible) so that its bitlength is 64 less than a multiple of 512.
- Append a 64-bit representation of $b \mod 2^{64}$.
- The formatted input is $x_0, x_1, ..., x_{16m-1}$, where each x_i is a 32-bit word.
- Initialize chaining variables

$$(H_1, H_2, ..., H_8) \leftarrow (h_1, h_2, ..., h_8)$$

SHA-256 Processing:

- For each i from 0 to $m-1$ do the following:
- Copy the i-th block of sixteen 32-bit words into temporary storage: $X_j \leftarrow x_{16i+j}$, $0 \leq j \leq 15$.
- Expand the 16-word block into a 64-word block: For j from 16 to 63 do:

$$X_j \leftarrow r_4(X_{j-2}) + X_{j-7} + r_3(X_{j-15}) + X_{j-16}$$

- Initialize working variables:

$$(A, B, C, ..., H) \leftarrow (H_1, H_2, ..., H_8)$$

- For j from 0 to 63 do:
  - $T_1 \leftarrow H + r_2(E) + f(E, F, G) + y_j + X$
  - $T_2 \leftarrow r_1(A) + g(A, B, C)$
  - $H \leftarrow G, G \leftarrow F, F \leftarrow E, E \leftarrow D + T_1$
  - $D \leftarrow C, C \leftarrow B, B \leftarrow A, A \leftarrow T_1 + T_2$
- Update chaining values:

$$(H_1, H_2, ..., H_8) \leftarrow (H_1 + A, H_2 + B, ..., H_8 + H)$$

Then output SHA-256$(x) = H_1 \| H_2 \| H_3 \| H_4 \| H_5 \| H_6 \| H_7 \| H_8$

3.10 Performance

Speed benchmarks (2017) for software implementations on an Intel Core i9 2.9 GHz 6-core Coffee Lake (8950HK) using OpenSSL 1.1.1d.

Algorithm	block length (bits)	key length (bits)	digest length (bits)	speed (Mbytes/sec)
RC4	-	128	-	644
ChaCha20	-	256	-	1846
DES	64	56	-	98
Triple-DES	64	168	-	39
AES (software)	128	128	-	254
AES (AES-NI)	128	128	-	1632
MD5	512	-	128	832
SHA-1	512	-	160	932
SHA-256	512	-	256	419
SHA-512	1024	-	512	550

Chapter 4 Message Authentication Code Schemes

Definition 4.1. A **message authentication code (MAC) scheme** is a family of functions $MAC_k : \{0,1\}^* \to \{0,1\}^n$ parameterized by an ℓ-bit key k, where each function MAC_k can be efficiently computed.

$t = MAC_k(x)$ is called the **MAC** or **tag** of x with key k.

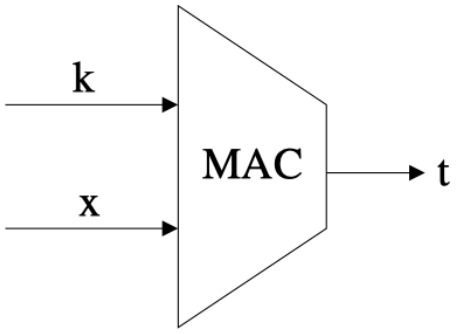


Figure 27: Message Authentication Code (MAC)

MAC schemes are used for providing (symmetric-key) **data integrity** and **data origin authentication**.

To provide data integrity and data origin authentication:

1. Alice and Bob establish a secret key $k \in \{0,1\}^\ell$.
2. Alice computes tag $t = MAC_k(x)$ and sends (x, t) to Bob.
3. Bob verifies that $t = MAC_k(x)$.

Note that there is no confidentiality or non-repudiation. Also, to avoid replay, we should add a timestamp or sequence number.

Let k be the secret key shared by Alice and Bob.

The adversary does not know k, but is allowed to obtain (from Alice or Bob) tags for messages of her choosing. The adversary's goal is to obtain the tag of **any** new message, i.e., a message whose tag she did not already obtain from Alice or Bob.

Definition 4.2. A MAC scheme is **secure** if given some tags $MAC_k(x_i)$ for x_i's of one's own choosing, it is computationally infeasible to compute (with non-negligible probability of success) a message-tag pair $(x, MAC_k(x))$ for any new message x.

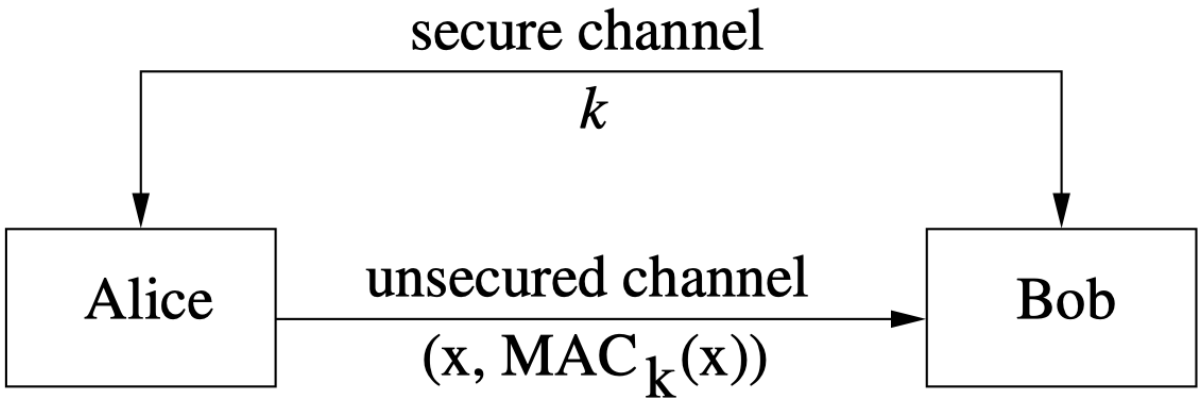


Figure 28: Applications of MAC Schemes

More concisely, a MAC scheme is **secure** if it is **existentially unforgeable against chosen-message attack**

An ideal MAC scheme has the following property:

- For each key $k \in \{0,1\}^\ell$, the function $MAC_k : \{0,1\}^* \to \{0,1\}^n$ is a random function.
- Ideal MAC schemes are useless in practice. However, when analyzing a generic attack on a MAC scheme, it is reasonable to assume that the MAC scheme is ideal.

4.1 Generic Attacks on MAC schemes

This attack is simple, we just guess the MAC of a message $x \in \{0,1\}^*$:

Algorithm 16: Generic Attack 1 on MAC schemes

1 Select $y \in_R \{0,1\}^n$ and guess that $MAC_k(x) = y$.

If we assume that MAC k is a random function, the probability of success is $1/2^n$. Also, guesses cannot be directly checked. MAC guessing is infeasible if $n \geq 128$.

Also, we can perform exhaustive key search:

Algorithm 17: Generic Attack 2 on MAC schemes

1 Given r known message-tag pairs $(x_1, t_1), ..., (x_r, t_r)$, one can check whether a guess k of the key is correct by verifying that $MAC_k(x_i) = t_i$, for $i = 1, 2, ..., r$.

Assuming that the MAC_k's are random functions, the expected number of keys for which the tags verify is $1 + FK = 1 + (2^\ell - 1)/2^{nr}$. The expected number of steps is $\approx 2^\ell$. Also,

exhaustive search is infeasible if $\ell \geq 128$.

4.2 MACs Based on Block Ciphers

- Let E be an n-bit block cipher with key space $\{0,1\}^\ell$
- Assumption: Suppose that plaintext messages all have lengths that are multiplies of n.
- To compute $MAC_k(x)$:
  1. Divide x into n-bit blocks $x_1, x_2, ..., x_r$
  2. Compute $H_1 = E_k(x_1)$.
  3. For $2 \leq i \leq r$, compute $H_i = E_k(H_{i-1} \oplus x_i)$.
  4. Then $MAC_k(x) = H_r$.

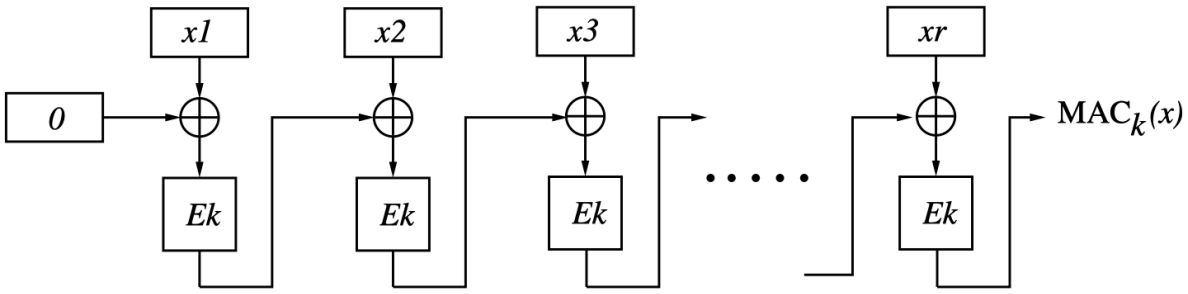


Figure 29: MACs Based on Block Ciphers

Theorem 4.3. Suppose that E is an "ideal" encryption scheme. (That is, for each $k \in \{0,1\}^\ell$, $E_k : \{0,1\}^n \to \{0,1\}^n$ is a "random" permutation) Then $CBC - MAC$ with **fixed-length inputs** is a secure MAC scheme.

CBC-MAC (described above) is **not secure** if variable length messages are allowed.

Here is a chosen-message attack on CBC-MAC:

1. Select an arbitrary n-bit block x_1.
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 one-block message t_1 (so $t_2 = E_k(t_1)$).
4. Then t_2 is the tag of the 2-block message $(x_1, 0)$ (since $t_2 = E_k(0 \oplus E_k(x_1)) = E_k(E_k(x_1)) = E_k(t_1)$).

4.2.1 Encrypted CBC-MAC (EMAC)

One countermeasure for **variable-length** messages is **Encrypted CBC-MAC**: We encrypt the last block under a second key s: $EMAC_{k,s}(x) = E_s(H_r)$, where $H_r = CBC - MAC_k(x)$.

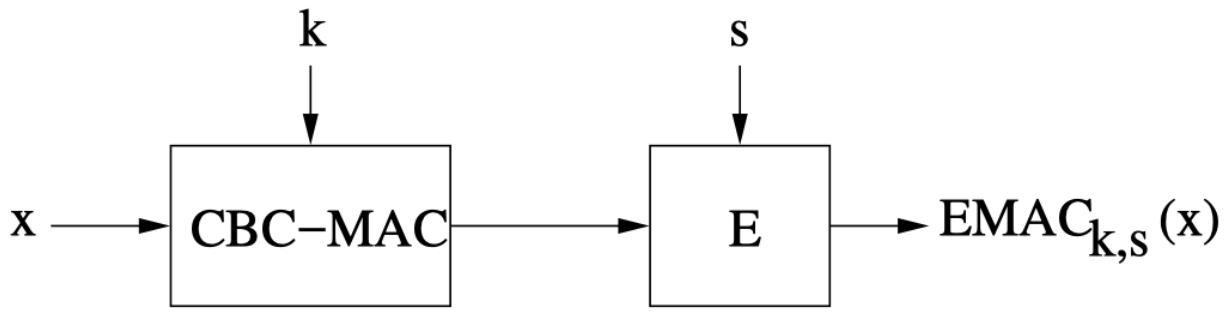


Figure 30: Encrypted CBC-MAC (EMAC)

Theorem 4.4. Suppose that E is an "ideal" encryption scheme. Then EMAC is a secure MAC scheme (for inputs of any length).

4.3 MACs Based on Hash Functions

Hash functions were not originally designed for message authentication; in particular they are not "keyed" primitives.

How to use them to construct secure MACs?

- Let H be an iterated n-bit hash function (without the length-block).
- Let $n+r$ be the input blocklength of the compression function $f : \{0,1\}^{n+r} \to \{0,1\}^n$.
- Let $k \in \{0,1\}^n$
- Let K denote k padded with $(r-n)0$'s. (So K has bitlength r)

We can consider Secret Prefix Method: we define

$$MAC_k(x) = H(K, x)$$

This is **insecure**. Here is a **length extension attack**:

1. Suppose that $(x, MAC_k(x))$ is known
2. Suppose that the bitlength of x is a multiple of r.
3. Then $MAC_k(x\|y)$ can be computed for any y (without knowledge of k).

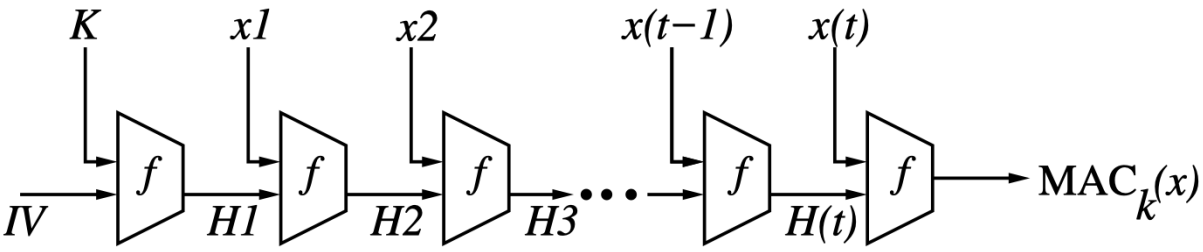


Figure 31: Secret Prefix Method

Also insecure if a length block is postpended to $K\|x$ prior to application of H.

Secret Suffix Method: we define

$$MAC_k(x) = H(x, K)$$

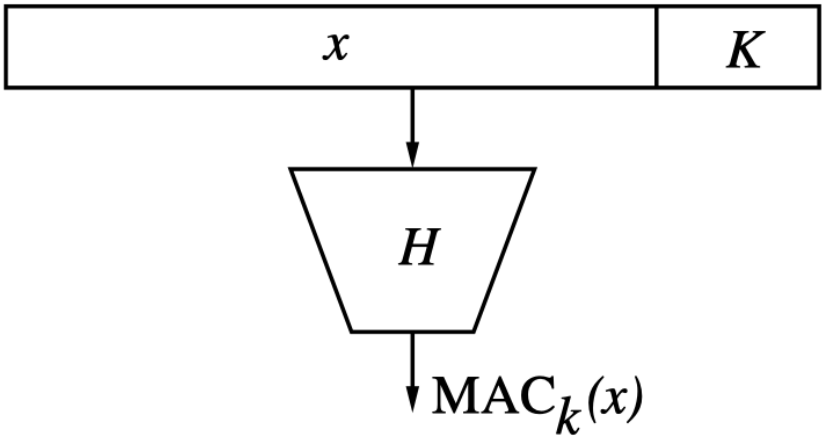


Figure 32: Secret Suffix Method

- The attack on the secret prefix method does not work here.
- Suppose that a collision (x_1, x_2) can be found for H (i.e., $H(x_1) = H(x_2)$). We assume that x_1 and x_2 both have bitlengths that are multiples of r. Thus $H(x_1, K) = H(x_2, K)$, and so $MAC_k(x_1) = MAC_k(x_2)$. Then the MAC for x_1 can be requested, giving the MAC for x_2. Hence if H is **not collision resistant**, then the secret suffix method MAC is **insecure**.

Envelope Method: we define

$$MAC_k(x) = H(K, x, K)$$

- The MAC key is used both at the start and end of the MAC computation.

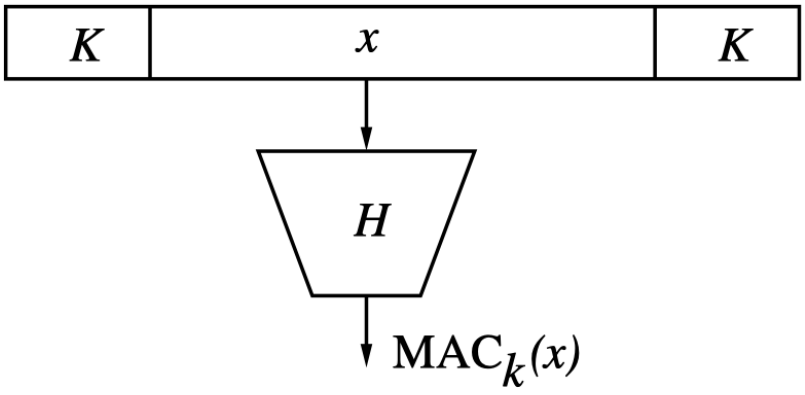


Figure 33: Envelope Method

- The envelope method appears to be secure (i.e., no serious attacks have been found).

$HMAC$: "Hash-based" MAC, developed by Bellare, Canetti and Krawczyk (1996).

We define two r-bit strings (in hexadecimal notation): ipad = `0x36`, opad = `0x5C`; each repeated $r/8$ times. Then, we define

$$HMAC_k(x) = H(K \oplus opad, H(K \oplus ipad, x))$$

Theorem 4.5. 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 algorithm.

4.4 Key Derivation Functions

HMAC is commonly used as a **key derivation function** (KDF).

- Suppose that Alice has a secret key k, and wishes to derive several session keys (e.g., to encrypt data in different communication sessions).
- Alice computes $sk_1 = HMAC_k(1), sk_2 = HMAC_k(2), sk_3 = HMAC_k(3), ...$
- Without knowledge of k, an adversary is unable to learn anything about any particular session key sk_j , even though it may have learnt some other session keys.

4.5 GSM

Global standards for mobile communications:

- **2G, 2.5G**: GSM (Global System for Mobile Communication)
- **3G**: UMTS (Universal Mobile Telecommunications System)

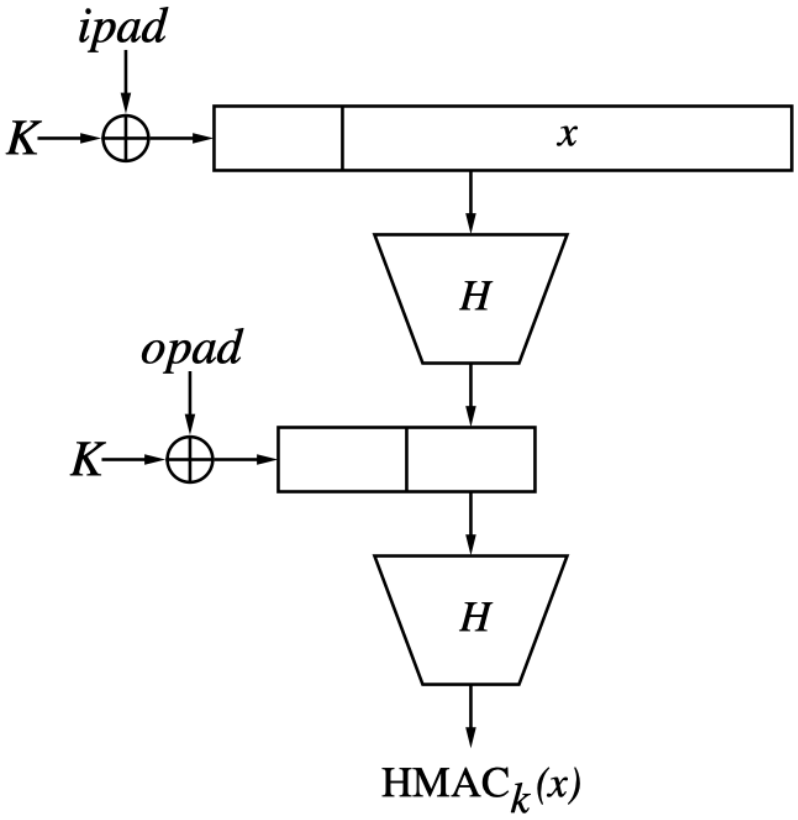


Figure 34: "Hash-based" MAC

- **4G**: LTE (Long Term Evolution)

GSM security is notable since it uses only symmetric-key primitives.

In terms of GSM security, we consider the following cryptographic ingredients:

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

Setup:

- A SIM card manufacturer 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.
- When a user purchases cell phone service, she gets the SIM card which she installs in her phone.
- A different key k is chosen for each user.

Objectives:

- **Entity authentication**: Cell phone service provider needs to be assured that entities accessing its service are legitimate subscribers.
- **Confidentiality**: Users need the assurance that their cell phone communications are private.

Algorithm 18: How GSM Works

1 **Alice**: cell phone user
2 **Bob**: cell phone service provider
3 Alice sends an authentication request to Bob
4 Bob selects a **challenge** $r \in_R \{0,1\}^{128}$, and sends r to Alice.
5 Alice's SIM card uses k to compute the **response** $t = MAC_k(r)$. Alice sends t to Bob.
6 Bob retrieves Alice's key k from its database, and verifies that $t = MAC_k(r)$.
7 Alice and Bob compute an **encryption key** $K_E = KDF(r)$, and thereafter use the encryption algorithm Enc_{K_E} to encrypt and decrypt messages for each other for the remainder of the session.

One drawback with using only symmetric-key crypto is that the SIM card manufacturer and the cell phone service providers have to securely maintain a large database of SIM keys k.

Chapter 5 Authenticaed Encryption

- A symmetric-key encryption scheme E provides **confidentiality**, e.g., $E = AES$.
- A MAC scheme provides **authentication** (**data origin authentication** and **data integrity**), e.g., $MAC = HMAC$.
- What if confidentiality and authentication are **both** required?

5.1 Encrypt-and-MAC

Algorithm 19: Encrypt-and-MAC

1 Alice sends $(c, t) = (E_{k_1}(m), MAC_{k_2}(m))$ to Bob, where m is the plaintext and k_1, k_2 are secret keys she shares with Bob.
2 Bob decrypts c to obtain $m = E_{k_1}(c)$ and then verifies that $t = MAC_{k_2}(m)$.

However, this generic method might have some security vulnerabilities, since MAC might leak information about the plaintext.

5.2 Encrypt-then-MAC

Algorithm 20: Encrypt-then-MAC

1 Alice sends $(c, t) = (E_{k_1}(m), MAC_{k_2}(E_{k_1}(m)))$ to Bob, where m is the plaintext and k_1, k_2 are secret keys she shares with Bob.
2 Bob first verifies that $t = MAC_{k_2}(c)$ and then decrypts c to obtain $m = E_{k_1}^{-1}(c)$.

This method has been deemed to be secure, provided of course that the encryption scheme E and the MAC scheme employed are secure.

5.2.1 Special-Purpose AE Schemes

Many specialized authenticated encryption schemes have been developed, the most popular of these being **Galois/Counter Mode (GCM)**.

These modes can be faster than generic Encrypt-then-MAC, and also allow for the authentication (but not encryption) of "header" data.

Example 5.1. AES-GCM:

- Authenticated encryption scheme proposed by David McGrew and John Viega in 2004.
- Adopted as a NIST standard in 2007.
- Uses the **CTR** mode of encryption and a custom-designed MAC scheme.

5.2.2 CTR: CounTeR Mode of Encryption

Let $k \in_R \{0,1\}^{128}$ be the secret key. Let $M = (M_1, M_2, ..., M_u)$ be a plaintext message, where each M_i is a 128-bit block, $u \leq 2^{32} - 2$.

Algorithm 21: CTR Encryption

To encrypt M, Alice does the following:
Select $IV \in_R \{0,1\}^{96}$
Let $J_0 = IV \| 0^{31} \| 1$
for $i = 1$ *to* u **do**
    $J_i \leftarrow J_{i-1} + 1$ [increment the counter]
    Compute $C_i = AES_k(J_i) \oplus M_i$
Send $(IV, C_1, ..., C_u)$ to Bob.

Algorithm 22: CTR Decryption

To decrypt, Bob does the following:
Let $J_0 = IV \| 0^{31} \| 1$
for $i = 1$ *to* u **do**
    $J_i \leftarrow J_{i-1} + 1$ [increment the counter]
    Compute $M_i = AES_k(J_i) \oplus C_i$

Note:

- CTR mode of encryption can be viewed as a stream cipher.
- As was the case with CBC encryption, identical plaintexts with different IVs result in different ciphertexts.
- It is critical that the IV **should not be repeated**; this can be difficult to achieve in practice.
- Unlike CBC encryption, CTR encryption is **parallelizable** (work with different counters).
- Note that AES^{-1} is not used.

5.2.3 Multiplying Blocks

- Let $a = a_0 a_1 a_2 ... a_{127}$ be a 128-bit block.
- We 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 define $c = a \bullet b$ to be the block corresponding to the polynomial

$$c(x) = a(x) \cdot b(x) \mod f(x) \text{ in } \mathbb{Z}_2[x]$$

- That is, $c(x)$ is the remainder upon dividing $a(x) \cdot b(x)$ by $f(x)$, where coefficient arithmetic is performed modulo 2.
- This is multiplication in the **Galois Field** $GF(2^{128})$.

5.3 AES-GCM

Definition 5.2. Input of **AES-GCM**:

- Data to be authenticated (but not encrypted) $A = (A_1, A_2, ..., A_v)$
- Data to be encrypted and authenticated: $M = (M_1, M_2, ..., M_u)$
- Secret key $k \in_R \{0,1\}^{128}$

Output of **AES-GCM**: (IV, A, C, t), where:

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

A secret key should be used to encrypt at most 2^{32} messages.

Algorithm 23: AES-GCM Encryption/Authentication

Let $L = L_A \| L_M$, where L_A, L_M are the bitlengths of A, M expressed as 64-bit integers. (L is the length block.)
Select $IV \in_R \{0,1\}^{96}$ and let $J_0 = IV \| 0^{31} \| 1$
// Encryption
for $i = 1$ *to* u **do**
    Compute $J_i = J_{i-1} + 1$
    Compute $C_i = AES_k(J_i) \oplus M_i$.
// Authentication
Let $T = 0^{128}$
Compute $H = AES_k(0^{128})$
for $i = 1$ *to* v **do**
    $T \leftarrow (T \oplus A_i) \bullet H$
// $T = ((((0 + A_1)H + A_2)H + A_3)H + \cdots + A_v)H = A_1H^v + A_2H^{v-1} + \cdots + A_{v-1}H^2 + A_vH$
for $i = 1$ *to* u **do**
    $T \leftarrow (T \oplus C_i) \bullet H$
$T \leftarrow (T \oplus L) \bullet H$
// $T = A_1H^{u+v+1} + A_2H^{u+v} + \cdots + A_vH^{u+2} + C_1H^{u+1} + \cdots + C_uH^2 + LH$
// $T = f_{A,M}(H)$, $f_{A,M}(x) = A_1x^{u+v+1} + \cdots + A_vx^{u+2} + C_1x^{u+1} + \cdots + C_ux^2 + Lx \in GF(2^{128})[x]$
Compute $t = AES_k(J_0) \oplus T$
// Hence, $t = AES_k(J_0) \oplus f_{A,M}(H)$
Output: (IV, A, C, t)

Here is an example of AES-GCM encryption and authentication, where authentication data A is one-block long ($v = 1$), and plaintext data M is two-blocks long ($u = 2$).

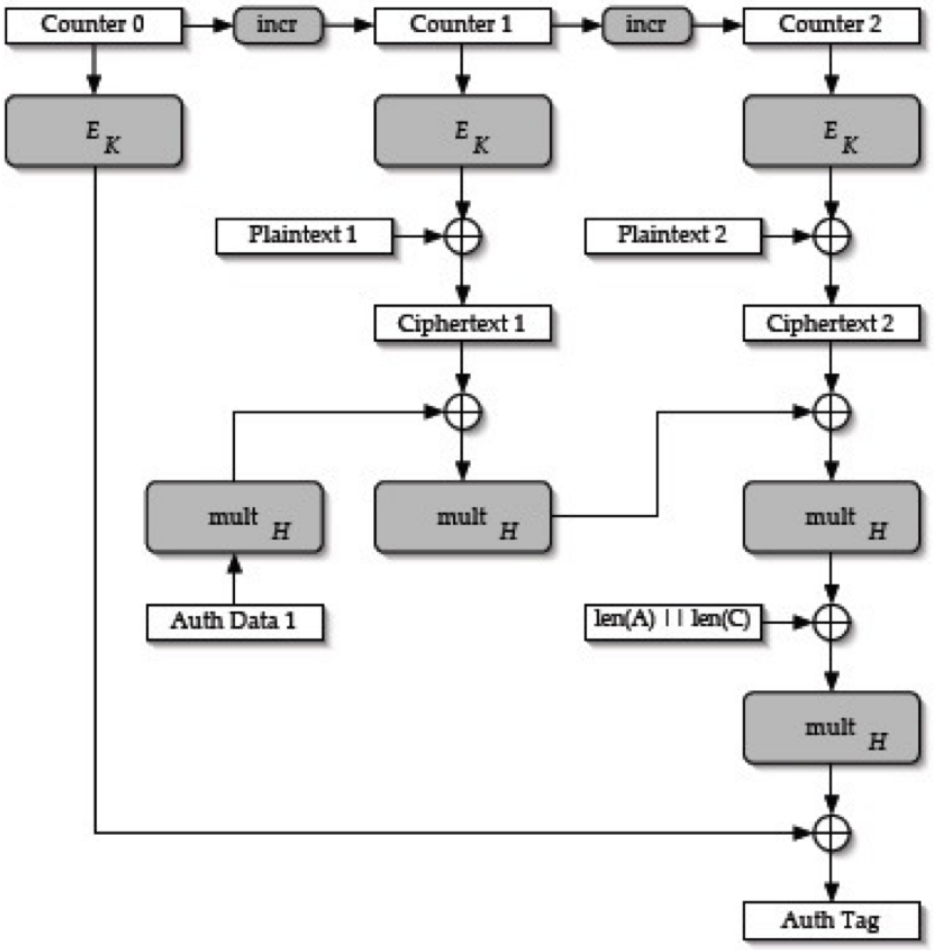


Figure 35: AES-GCM Encryption/Authentication

Algorithm 24: AES-GCM Decryption/Authentication

Upon receiving (IV, A, C, t), Bob does the following:
Let L $= L_A \| L_C$, where L_A, L_C are the bitlengths of A, C expressed as 64-bit integers.
// Authentication
Let $T = 0^{128}$
Compute $H = AES_k(0^{128})$
for $i = 1$ *to* v **do**
    $T \leftarrow (T \oplus A_i) \bullet H$
for $i = 1$ *to* u **do**
    $T \leftarrow (T \oplus C_i) \bullet H$
$T \leftarrow (T \oplus L) \bullet H$
Compute $t' = AES_k(J_0) \oplus T$
if $t' = t$ **then**
    proceed to decryption
else
    reject
// Decryption
Let $J_0 = IV \| 0^{31} \| 1$
for $i = 1$ *to* u **do**
    Compute $J_i = J_{i-1} + 1$
    Compute $M_i = AES_k(J_i) \oplus C_i$
Accept and output (A, M)

For AES-GCM, authentication only:

- Consider AES_GCM with no M, so $u = 0$, $L_M = 0$
- The message to be authenticated is $A = (A_1, A_2, ..., A_v)$, where $v \leq \ell$. The tag is (IV, t), where $IV \in_R \{0,1\}^{96}$ and $t = AES_k(J_0) + f_A(H)$, and $J_0 = IV \| 0^{31} \| 1$, $H = AES_k(0)$
- The attack goal is: Eve has message-tag pairs (for messages of her choosing): (A^j, IV^j, t^j), $1 \leq j \leq r$. Her goal is to produce a message-tag forgery (A^*, IV^*, t^*) where $A^* \notin \{A^1, A^2, ..., A^r\}$
- We can assume that no two IV's in this list are the same
- We can also assume that Eve does not know k or H

Proposition 5.3. AES-GCM is secure if we consider authentication only.

Proof 5.4. Suppose that Eve outputs a forgery (A^*, IV^*, t^*):

- If $IV^* \notin \{IV^1, IV^2, ..., IV^t\}$, then $J_0^* \notin \{J_0^1, ..., J_0^r\}$, and so Eve doesn't know

$AES_k(J_0^*)$, which serves as a one-time pad for $f_{A^*}(H)$. Thus, the probability that Eve can output a valid tag (i.e. $t^* = AES_k(J_0^*) + f_{A^*}(H)$) is only $1/2^{128}$

- Suppose that $IV^* = IV^j$ for some $1 \leq j \leq r$, so $J_0^* = J_0^j$. Then,

$$\begin{aligned} t^* - t^j &= AES_k(J_0^*) + f_{A^*}(H) - AES_k(J_0^j) + f_{A^j}(H) \\ &= f_{A^*}(H) - f_{A^j}(H) \end{aligned}$$

So, Eve has produced A^*, A^j, and α such that $\alpha = f_{A^*}(H) - f_{A^j}(H)$, without knowledge of H. But this can only be done with negligible probability, as the following lemma shows

□

Lemma 5.5. For all distinct $A, B \in (\{0,1\}^{128})^{\leq \ell}$ and $\alpha \in \{0,1\}^{128}$, $Pr[f_A(H) - f_B(H) = \alpha] \leq (\ell+1)/2^{128}$ (which is negligible), where the probability is assessed over random choices of $H \in \{0,1\}^{128}$

Proof 5.6. Let $A, B \in (\{0,1\}^{128})^{\leq \ell}$ with $A \neq B$, and let $\alpha \in \{0,1\}^{128}$.

Suppose $A \in \{0,1\}^{128v}$, and $B \in \{0,1\}^{128w}$. Then,

$$\begin{aligned} f_A(H) - f_B(H) - \alpha = &(A_1 H^{v+1} + A_2 H^v + \cdots + A_v H^2 + L_A H) \\ &- (B_1 H^{w+1} + B_2 H^w + \cdots + B_v H^2 + L_B H) - \alpha \end{aligned}$$

which is a polynomial in H of degree $\leq \max(v, w) + 1 \leq \ell + 1$.

Since a non-zero polynomial of degree $\ell + 1$ can have at most $\ell + 1$ roots, there are at most $\ell + 1$ $H \in \{0,1\}^{128}$ satisfying $f_A(H) - f_B(H) - \alpha = 0$ □

Here are some Features of AES-GCM:

- Performs authentication and encryption.
- Supports authentication only (by using empty M).
- Very fast implementations on Intel and AMD processors because of special **AES-NI** and **PCLMULQDQ** instructions for the AES and $\bullet$ operations.
- Encryption and decryption can be **parallelized**
- AES-GCM can be used in **streaming mode**
- Security is justified by a security proof: Original McGrew-Viega security proof (2004) was wrong. The proof was fixed in 2012 by Iwata-Ohashi-Minematsu.
- AES-GCM is widely used today.

5.4 Encryption at Google

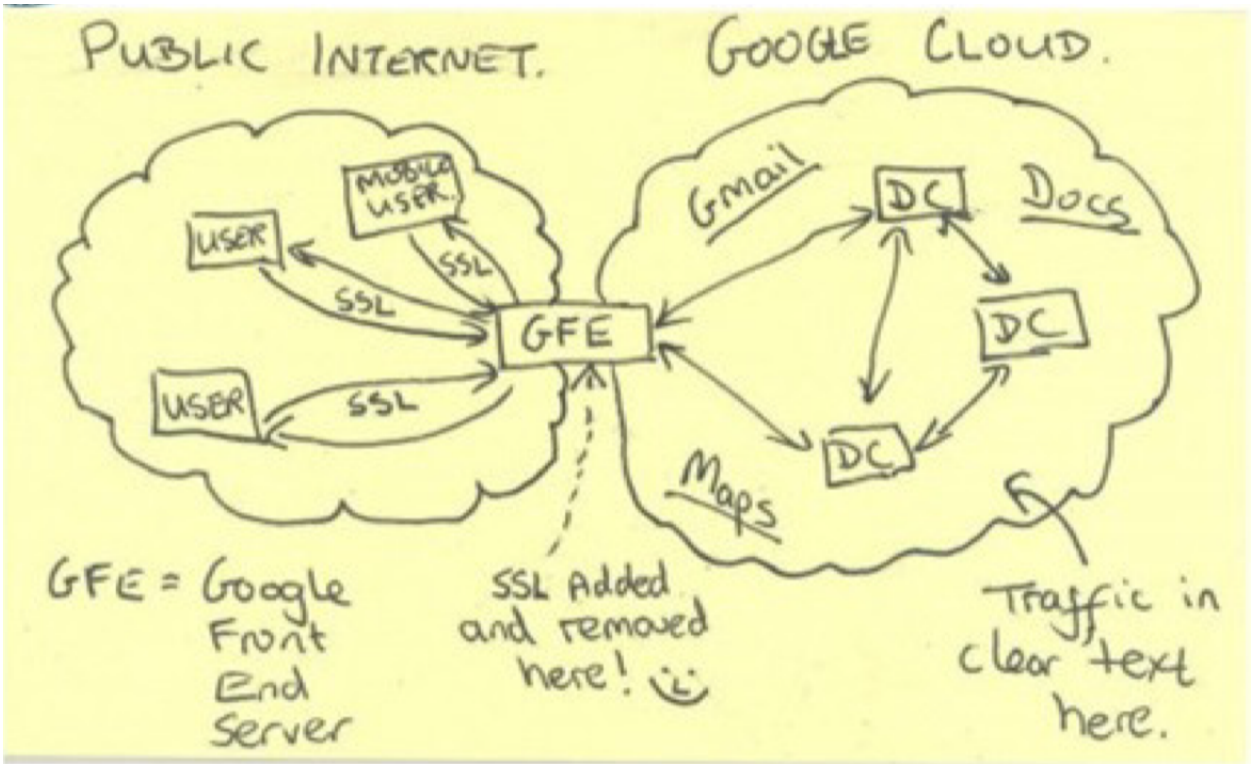


Figure 36: Google's Encryption Flow

Google has 21 data centres around the world:

- A data centre (DC) contains tens of thousands of servers.
- Lots of physical security (cameras, biometric identification, metal detectors, vehicle barriers, etc.)
- Communication between these servers and the outside world is all done via Google Front End (GFE) servers.
- Servers within a data centre communicate via a LAN (Local Area Network).
- Servers in different data centres communicate via a WLAN (Wide Local Area Network).

Google Data Security (Google wants to protect):

1. Data **communicated** between individual users (browsers) and Google (GFEs):
    - Data is encrypted and authenticated using TLS.
    - Symmetric-key enc. (e.g., AES), symmetric-key authentication (e.g., HMAC); authenticated enc. (e.g., AES-GCM); key establishment (e.g., RSA public-key enc., ECDH); public key certificates (RSA signatures).

- TLS is used by **all** web servers and browsers (not Google specific).

2. Data **communicated** between Google servers (perhaps in different data centres):
   - Data is secured using Google's version of TLS (Application Layer Transport Security, **ALTS**)
   - ALTS handles roughly 10^{10} remote procedure calls (RPCs) per second.
3. Data **stored** at data centres.

5.4.1 Key Management Service (KMS)

- All data stored within Google data centres is encrypted with **AES256-GCM** (GCM = Galois Counter Mode). AES128-CTR + HMAC-SHA256 (Encrypt-then-MAC) is used in some legacy applications.
- The KMS manages the many AES secret keys that are used to encrypt/decrypt data by the many storage services within data centres.
- Some of the KMS requirements:
  - **Availability**: $> 99.9995\%$ requests are served.
  - **Latency**: 99% of requests are served in < 10 ms.
  - **Scalability**: Handle $\approx 10^7$ requests/second.
  - **Security**: Effortless and foolproof **key rotation**
  - **Efficiency**: To minimize number of machines needed.

Google's Key Hierarchy:

1. **Storage systems** (millions of processes). Encrypts data with **DEKs** (Data Encryption Keys).
2. **KMS** (tens of thousands). Encrypts DEKs with **KEKs** (Key Encryption Keys).
3. **Root KMS** (hundreds). Encrypts KEKs with **KMS Master Keys**.
4. **Root KMS Master Key Distributor** (hundreds). Encrypts KMS Master Keys with the Root KMS Master Key.
5. **Physical safes** (two). The Root KMS Master Key is backed up on hardware devices.

5.4.2 Google's storage systems

Suppose that a storage system wishes to encrypt some data item m. åThe storage system does the following:

1. Break up m into **chunks**, $m_1, ..., m_\ell$. Each chunk can be up to several Gigabytes in size

2. Generate ℓ **Data Encryption Keys** (DEKs), $k_1, ..., k_\ell$. Multiple sources of entropy are sampled (e.g., Intel's RDRAND instruction; inter-packet arrival times; measurement of disk seeks). The samples are then combined and hashed using a key derivation function (KDF) to produce a 256-bit secret key.
3. Encrypt with **AES256-GCM**: $c_1 = AES_{k_1}(m_1), ..., c_\ell = AES_{k_\ell}(m_\ell)$.
4. Send the DEKs $k_1, ..., k_\ell$ to a KMS.
5. Receive the wrapped (encrypted) keys $w_1, ..., w_\ell$ from the KMS. The KMS encrypts the DEKs with its **Key Encryption Keys** (KEKs).
6. Store $(c_1, w_1), ..., (c_\ell, w_\ell)$ These encrypted chunks are replicated and distributed across Google's storage systems.

To decrypt a chunk (c_j, w_j):

1. The storage system sends the wrapped key w_j to the KMS.
2. The KMS decrypts w_j using the appropriate KEK, and sends the DEK k_j to the storage system.
3. The storage system decrypts c_j using k_j.

Notes:

- Each data chunk has a **unique identifier**.
- The KMS maintains an **Access Control List (ACL)** to ensure that a data chunk can only be decrypted by the authorized storage system.
- Note that each chunk of data is encrypted using a different DEK. This ensures that if a DEK is compromised, then only one chunk of data is potentially compromised.
- The storage system does not store the DEKs $k_1, ..., k_\ell$
- If a chunk of data m_j is updated, it is re-encrypted with a new DEK k'_j rather than using the old DEK k_j.

5.4.3 Google's Key Management Services

- Key Management Services (KMSs) generate the AES256-GCM **Key Encryption Keys** (KEKs), and maintain the Access Control Lists (one ACL list for each KEK)
- The KMS encrypts/decrypts DEKs using the KEKs, in accordance with the ACL.
- The KEKs never leave the KMS.
- The KMS also maintains an audit trail of when a KEK was used
- KEKs are **rotated** (i.e., changed). The standard rotation frequency is once every 90 days.

5.4.4 Google's Root KMS

- The **Root KMS** wraps KEKs with AES256-GCM **KMS Master Keys**. There are about a dozen KMS Master Keys. The Root KMSs are run on dedicated secured machines in Google's data centres.
- The KMS Master Keys are wrapped with the AES256-GCM **Root KMS Master Key**. The Root KMS Master Key is stored in RAM on the Root KMS machines. **The Root KMS Master Key Distributor** ensures that all Root KMSs always have the same version of the Root KMS Master Key.

5.4.5 Google's Physical Safes

- The Root KMS Master Key is backed up on two hardware devices stored in physical safes in highly secured areas in two physically separated Google locations.
- Fewer than 20 Google employees have access to these safes.
- The backups will be used if Google ever has to do a complete reboot.

Chapter 6 Introduction to Public-Key Cryptography

6.1 Drawbacks with Symmetric-Key Cryptography

6.1.1 Key Establishment Problem

The shared secret keys can then be used to achieve confidentiality (e.g., using AES), or authentication (e.g., using HMAC), or both (e.g., using AES-GCM). However, How do Alice and Bob establish the secret key k?

Method 1: **Point-to-point key distribution**. (Alice selects the key and sends it to Bob over a secure channel)

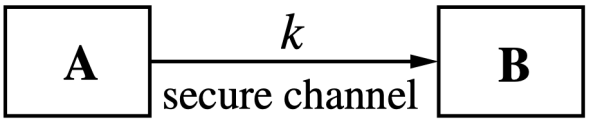


Figure 37: Point to Point Key Distribution

The secure channel could 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.

Method 2: Use 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.
- T serves as a **key distribution centre** (KDC):
  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}.

Drawbacks of using a KDC:

1. The TTP must be unconditionally trusted.
2. The TTP is an attractive target.

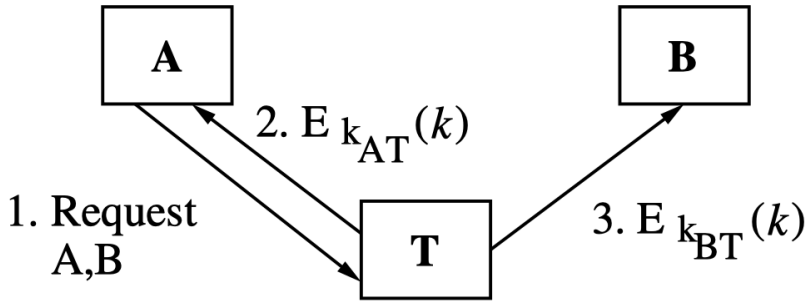


Figure 38: Relationship

3. The TTP must be on-line.
   - Potential bottleneck.
   - Critical reliability point.

6.1.2 Key Management Problem

In a network of n users, each user has to share a different key with every other user. Each user thus has to store $n-1$ different secret keys. The total number of secret keys is $\binom{n}{2} \approx n^2/2$

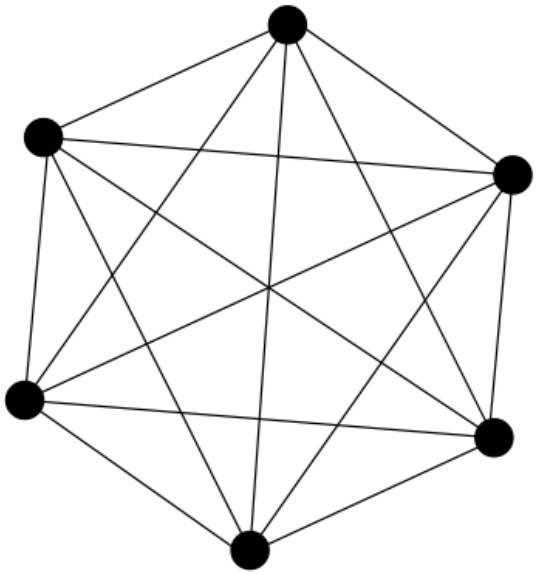


Figure 39: Key Management Problem

6.1.3 Non-Repudiation is Impractical

Strictly speaking, symmetric-key techniques cannot be used to achieve non-repudiation. This is because both Alice and Bob knows the secret keys. You will never know if Bob leaks the secret key to others or not.

6.2 Public-Key Cryptography

The idea of Public-Key Cryptography is communicating parties a priori share some authenticated (but non-secret) information.

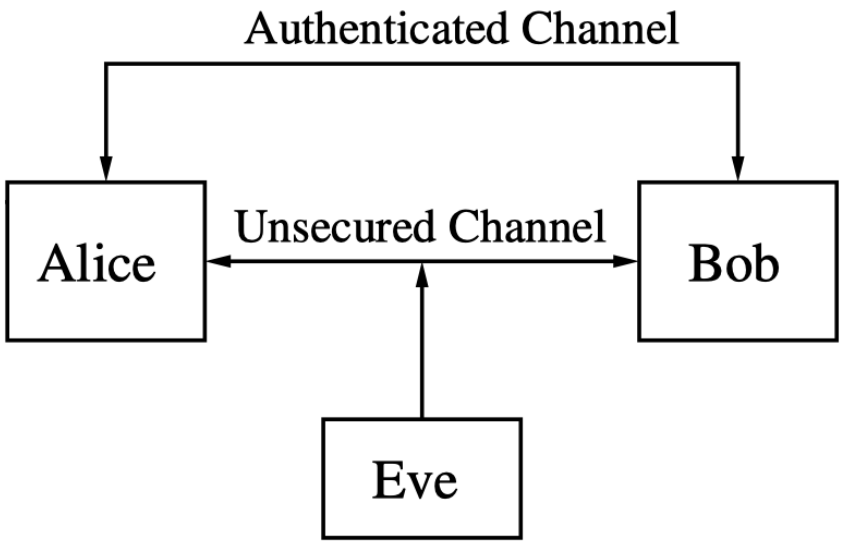


Figure 40: Public Key Cryptography

6.2.1 Merkle Puzzles

The goal is that Alice and Bob establish a secret session key by communicating over an authenticated (but non-secret) channel. Here is what to do:

1. Alice creates N **puzzles** P_i , $1 \leq i \leq N$ (e.g., $N = 10^9$). Each puzzle takes t hours to solve (e.g., $t = 5$). The solution to P_i reveals a 128-bit **session key** sk_i and a randomly-selected 128-bit **serial number** n_i (which Alice selected and stored).
2. Alice sends $P_1, P_2, ..., P_N$ to Bob.
3. Bob selects j at random from $[1, N]$ and solves puzzle P_j to obtain sk_j and n_j.
4. Bob sends n_j to Alice.
5. The secret session key is sk_j.

An eavesdropper has to solve $500,000,000$ puzzles on average to determine the puzzle index j (and thus sk_j).

Example 6.1. We could make $P_i = AES - CBC_{k_i}(sk_i, n_i, n_i)$, where $ki = (r_i \| 0^{88})$ and r_i is a randomly selected 40-bit string.

P_i can be solved in 2^{40} steps by exhaustive key search. But an eavesdropper has to do $\sim 500,000,000$ of this exhaustive key search, which is obviously infeasible.

Algorithm 25: Key Pair Generation for Public-Key Cryptography

1 Each entity A does the following:
2 Generate a key pair (P_A, S_A).
3 S_A is A's **secret key**.
4 P_A is A's **public key**.

- **Security requirement**: It should be infeasible for an adversary to recover S_A from P_A.
- For example, we can take $S_A = (p, q)$ where p are q are randomly-selected prime numbers; $P_A = p \cdot q$.

Algorithm 26: Public-Key Encryption

1 To encrypt a secret message m for Bob, Alice does:
2 Obtain an **authentic** copy of Bob's public key P_B .
3 Compute $c = E(P_B, m)$; E is the encryption function.
4 Send c to Bob.

Algorithm 27: Public-Key Decryption

1 To decrypt c, Bob does:
2 Compute $m = D(S_B, c)$; D is the decryption function.

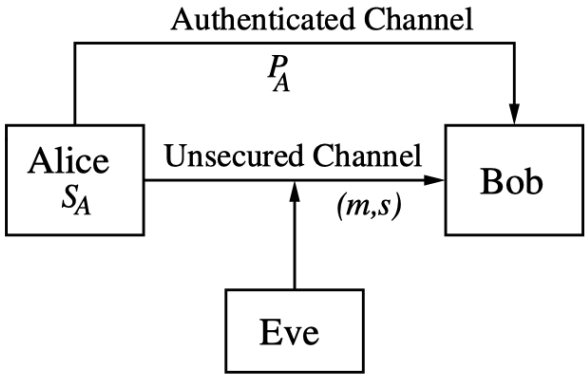


Figure 41: Digital Signatures

Algorithm 28: Public-Key Digital Signatures

To **sign** a message m, Alice does:
Compute $s = Sign(S_A, m)$.
Send m and s to Bob.

Algorithm 29: Public-Key Digital Verification

To **verify** Alice's signature s on m, Bob does:
Obtain an **authentic** copy of Alice's public key P_A.
Accept if $Verify(P_A, m, s)$ = "Accept".

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 property cannot be achieved with a symmetric-key MAC scheme. Hence, digital signatures are widely used to **sign software updates** which are then broadcast to computers around the world.

6.2.2 Public-Key Versus Symmetric-Key

Advantages of public-key cryptography:

- No requirement for a secured channel.
- Each user has only 1 key pair, which simplifies key management.
- A signed message can be verified by anyone.
- Facilitates the provision of non-repudiation services (with digital signatures).

Disadvantages of public-key cryptography:

- Public-key schemes are slower than their symmetric-key counterparts.

6.2.3 Hybrid Schemes

Algorithm 30: Hybrid Scheme Encryption

To **encrypt a secret signed message** m, Alice does:
Compute $s = Sign(S_A, m)$.
Select a secret key k for a symmetric-key encryption scheme such as AES.
Obtain an authentic copy of Bob's public key P_B.
Send $c_1 = E(P_B, k)$ and $c_2 = AES_k(m, s)$.

Algorithm 31: Hybrid Scheme Decryption

To recover m and verify its authenticity, Bob does:
Decrypt $c_1 : k = D(S_B, c_1)$.
Decrypt c_2 using k to obtain (m, s).
Obtain an authentic copy of Alice's public key P_A.
Check that $Verify(P_A, m, s) =$ "Accept".

6.3 Algorithmic Number Theory

6.3.1 Fundamental Theorem of Arithmetic

Theorem 6.2. Fundamental Theorem of Arithmetic: Every integer $n \geq 2$ has a unique prime factorization (up to ordering of factors).

Interesting questions:

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

6.3.2 Basic Concepts from Complexity Theory

Definition 6.3. 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 an integer factorization algorithm, the **input** is a positive integer n, and the **output** is the prime factorization of n.

Definition 6.4. The **efficiency** of an algorithm is measured by the scarce resources it consumes (e.g. time, space, number of processors, number of chosen plaintext-ciphertext pairs).

Definition 6.5. The **input size** is the number of bits required to write down the input using a reasonable encoding.

The size of a positive integer n is $\lfloor \log_2 n \rfloor + 1$ bits. Note that $k = \log_2 n \Leftrightarrow n = 2^k$

Definition 6.6. The **running time** of an algorithm is an upper bound as a function of the input size, of the worst case number of basic steps the algorithm takes over all inputs of a fixed size.

Definition 6.7. An algorithm is a **polynomial-time** (efficient) algorithm if its (expected) running time is $O(k^c)$, where c is a fixed positive integer, and k is the **input size**.

Recall that if $f(n)$ and $g(n)$ 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 $n \geq n_0$.

6.3.3 Basic Integer Operations

Input: Two k-bit positive integers a and b.

Input size: $O(k)$ bits.

Operation	Running time of naive algorithm (in bit operations)
Addition: $a + b$	$O(k)$
Subtraction: $a - b$	$O(k)$
Multiplication: $a \cdot b$	$O(k^2)$
Division: $a = qb + r$	$O(k^2)$
GCD: $\gcd(a, b)$	$O(k^2)$

Note that GCD can be efficiently computed using the Euclidean Algorithm.

6.3.4 Basic Modular Operations

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

Input size: $O(k)$ bits.

Operation	Running time of naive algorithm (in bit operations)
Addition: $a + b \mod n$	$O(k)$
Subtraction: $a - b \mod n$	$O(k)$
Multiplication: $a \cdot b \mod n$	$O(k^2)$
Inversion: $a^{-1} \mod n$	$O(k^2)$
Exponentiation: $a^m \mod n$	$O(k^3)$

Note that $x = a^{-1} \mod n$ means $ax \equiv 1 \pmod{n}$ and $1 \leq x \leq n-1$. $a^{-1} \mod n$ exists if and only if $\gcd(a, n) = 1$. $a^{-1} \mod n$ can be efficiently computed using the Extended Euclidean Algorithm.

6.3.5 Modular Exponentiation

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

Output: $a^m \mod n$.

Algorithm 32: Modular Exponentiation Naive Algorithm 1

```
1 Compute d = a^m.
2 return d mod n
```

The bitlength of d is (approximately) $\log_2 d = \log_2 a^m = m \cdot \log_2 a = O(2^k k)$ since $m \approx 2^k$. Hence the algorithm is not polytime.

Algorithm 33: Modular Exponentiation Naive Algorithm 2

```
1 A ← a
2 for i = 2 to m do
3     A ← A × a mod n.
4 return A
```

This algorithm is also not polytime.

To explore an efficient algorithm for modular exponentiation, first, we need to note that if we write down the binary representation of m, i.e. let $m = \sum_{i=0}^{k-1} m_i 2^i$, then

$$a^m \equiv a^{\sum_{i=0}^{k-1} m_i 2^i} \equiv \prod_{i=0}^{k-1} a^{m_i 2^i} \equiv \prod_{0 \leq i \leq k-1, m_i = 1} a^{2^i} \pmod{n}$$

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

Algorithm 34: Modular Exponentiation: Repeated Square-and-Multiply Algorithm

```
Write m in binary: m = sum_{i=0}^{k-1} m_i 2^i
if m = 1 then
    B ← a
else
    B ← 1
A ← a
for i = 1 to k − 1 do
    A ← A^2 mod n
    if m_i = 1 then
        B ← B × A mod n
return B
```

At most k modular squarings and k modular mults, so worst-case running time is $O(k^3)$ bit operations (polytime).

Chapter 7 RSA

7.1 Basic RSA Public Key Encryption

7.1.1 RSA Key Generation

Algorithm 35: RSA Key Generation

1 Each entity Alice does the following:
2 Randomly select two large distinct primes p, q of the same bit length. (This can be done efficiently)
3 Compute $n = pq$ and $\phi = \phi(n) = (p-1)(q-1)$
4 Select an arbitrary integer e such that $1 < e < \phi$ and $\gcd(\phi, e) = 1$
5 Compute the integer d, $1 < d < \phi$, with $ed \equiv 1 \pmod{\phi}$
6 Alice's public key is (n, e); her private key is d

Note:

- n is called the RSA modulus
- e is called the encryption exponent
- e is called the decryption exponent

7.1.2 Basic RSA Encryption Scheme

Algorithm 36: RSA Encryption

1 To encrypt a message for Alice, Bob does:
2 Obtain an authentic copy of Alice's public key (n, e)
3 Represent the message as an integer $m \in [0, n-1]$
4 Compute the ciphertext $c = m^e \mod n$
5 Send c to Alice

Algorithm 37: RSA Decryption

1 To decrypt c, Alice does:
2 Compute $m \equiv c^d \mod n$

Example 7.1. Here is a toy example of RSA encryption:

To encrypt a plaintext $m = 13$ for Alice, Bob does:

1. Obtains Alice's public key $(n = 851, e = 631)$.
2. Computes $c = 13^{631} \mod 851$ using the repeated-square-and-multiply algorithm:

(a) Write $e = 631$ in binary:

$$e = 2^9 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0$$

(b) Compute successive squarings of $m = 13$ modulo n:

$$\begin{aligned} 13 &\equiv 13 \pmod{851} & 13^2 &\equiv 169 \pmod{851} \\ 13^{2^2} &\equiv 478 \pmod{851} & 13^{2^3} &\equiv 416 \pmod{851} \\ 13^{2^4} &\equiv 303 \pmod{851} & 13^{2^5} &\equiv 752 \pmod{851} \\ 13^{2^6} &\equiv 440 \pmod{851} & 13^{2^7} &\equiv 423 \pmod{851} \\ 13^{2^8} &\equiv 219 \pmod{851} & 13^{2^9} &\equiv 305 \pmod{851} \end{aligned}$$

(c) Multiply together the squares 13^{2^i} for which the ith bit (where $0 \leq i \leq 9$) of the binary representation of 631 is 1:

$$\begin{aligned} 13^{631} &= 13^{2^9+2^6+2^5+2^4+2^2+2^1+2^0} \\ &= 13^{2^9} \cdot 13^{2^6} \cdot 13^{2^5} \cdot 13^{2^4} \cdot 13^{2^2} \cdot 13^{2^1} \cdot 13^{2^0} \\ &\equiv 305 \cdot 440 \cdot 752 \cdot 303 \cdot 478 \cdot 169 \cdot 13 \pmod{851} \\ &\equiv 616 \pmod{851} \end{aligned}$$

3. Bob sends the ciphertext $c = 616$ to Alice.

To decrypt $c = 616$, Alice uses her private key $d = 487$ as follows:

1. Compute $m = 616^{487} \mod 851$ to get $m = 13$.

Theorem 7.2. RSA works: For all integers m, $m^{ed} \equiv m \pmod{n}$

Proof 7.3. Since $ed \equiv 1 \pmod{\phi}$, we can write $ed = 1 + k(p-1)(q-1)$ for some $k \in \mathbb{Z}$. Since $ed > 1$, and $p, q \geq 2$, we have $k \geq 1$. We will prove that $m^{ed} \equiv m \pmod{p}$:

- Suppose first that $p \mid m$, then $m \equiv 0 \pmod{p}$, so $m^{ed} \equiv 0 \pmod{p}$. Hence, $m^{ed} \equiv m \pmod{p}$
- Suppose now that $p \nmid m$. Then $\gcd(p, m) = 1$, so $m^{p-1} \equiv 1 \pmod{p}$ by Fermat's Little Theorem. Raising both sides to the power $k(q-1)$ and then multiplying by m gives $m^{1+k(p-1)(q-1)} \equiv m \pmod{p}$. Hence, $m^{ed} \equiv m \pmod{p}$

Similarly, $m^{ed} \equiv m \pmod{q}$. Since p, q are distinct primes, $\gcd(p, q) = 1$, and hence $m^{ed} \equiv m \pmod{pq}$. □

7.1.3 Basic RSA Signature Scheme

Key Generation: Same as for RSA encryption

Algorithm 38: RSA Signature Generation

To sign $m \in \{0,1\}^*$, Alice does:
Compute $M = H(m)$, where H is a hash function, e.g. SHA-256
Compute the **signature** $S = M^d \mod n$
Alice's signed message is (m, S)

Algorithm 39: RSA Signature Verification

To verify (m, S), Bob does:
Obtain an authentic copy of Alice's public key (n, e)
Compute $M = H(m)$
Compute $M' = S^e \mod n$
Accept (m, S) if and only if $M = M'$

7.2 QQ Browser Encryption

7.2.1 Background

The Snowden documents suggested that the NSA (and collaborators) were exploiting vulnerabilities in the **UC Browser** (a browser for mobile devices that is popular in China) to track users.

This prompted Knockel, Senft and Deibert (UToronto) in **2016** to study the security of browsers used in China.

They studied **QQ Browser**, a free web browser for Android, Windows, Mac, iOS, developed by **Tencent**.

QQ Browser is used by hundreds of millions of cell phone users in China.

7.2.2 Version 1

When a user launches the QQ Browser (the **client**) on Android, the browser makes a series of **WUP requests** to QQ Browser's server (the **server**). Via a WUP request, the browser sends personal user data to the server. This data includes:

- QQ username. (Personal information)
- WiFi MAC address of client. (Location information)
- MAC addresses of all nearby WiFi access points.
- URL of each page visited by the broswer. (User habits)

(**UC Browser** and **Baidu Browser** also collect similar data.)

Of course, the personal user data needs to be protected as it is transmitted over the internet. Hybrid encryption is used to protect these data.

To encrypt a **WUP request** m, the client does the following:

1. Randomly generate a **128-bit AES session key** k:

   ```
   int i = 10000000 + new Random().nextInt(89999999);
   int j = 10000000 + new Random().nextInt(89999999);
   return (String.valueOf(i) + String.valueOf(j)).getBytes();
   ```

   ($k = i \| j$, where i, j are 8-byte ASCII strings.)

2. Encrypt k with the server's RSA public key (n, e):

   $$c_1 = k^e \mod n, \text{ where } e = 65537, n = 24540641757374088471004774586996502346 3$$

3. Encrypt m: $c_2 = AES - ECB_k(m)$.

4. Send (c_1, c_2) to the server.

The server does the following:

1. Decrypt c_1 using its **RSA private key** d: $k = c_1^d \mod n$.

2. Decrypt c_2 : $m = AES - ECB_k^{-1}(c_2)$

3. Encrypt the **WUP response** m': $c' = TEA{-}CBC_{k'}(m')$ where $k' =$ `sDf434ol * 123+-KD` (in ASCII). Send c'

The client decrypts $m' = TEA - CBC_{k'}^{-1}(c')$ using the hard-coded 128-bit key k'.

However there are several serious problems (The goal is to create a security level of 128 bits):

1. i and j are each randomly selected integers in the interval $[10000000, 99999998]$, so the total number of keys k is $\approx 2^{26.4} \times 2^{26.4} \approx 2^{52.8}$, not 2^{128}.

2. The server's RSA public key is easily factored (since it is only 128 bit long):

   $$n = 14119218591450688427 \times 17381019776996486069$$

3. The response uses the fixed key k' that is hard-coded in all QQ browsers!

4. Don't use ECB mode!

Summary: The user data is protected using **extremely weak crypto**, and thus is very vulnerable to **passive eavesdropping**.

7.2.3 Version 2

Tencent then upgrade the QQ browser as the following.

To encrypt a **WUP request** m, the client does the following:

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

The server does the following:

1. Decrypt c_1 using its RSA private key d: $K = c_1^d \mod n$, and let k be the 128 least significant bits of K.
2. Decrypt c_2 : $m = AES - ECB_k^{-1}(c_2)$.
3. If m is a properly formatted WUP request, then encrypt the WUP response m' : $c' = AES - ECB_k(m')$ and send c' . If m is not properly formatted, then don't respond.

The client decrypts $m' = AES - ECB_k^{-1}(c')$.

However, this is still insecure under **restricted Chosen-Ciphertext Attack**:

- In the first step for server, 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 software did not check that the remaining $1024 - 128 = 896$ bits of K are all 0 (as they should be).
- This flaw can be exploited using a **restricted chosen-ciphertext attack**:
    1. The adversary intercepts a ciphertext $c = (c_1, c_2)$.
    2. She then sends to the QQ server a carefully-chosen modification $\hat{c} = (\hat{c_1}, \hat{c_2})$ of c
    3. **Depending on whether the server responds or not**, the adversary learns one bit of the secret key k.
    4. Steps 2 and 3 are repeated to obtain all the 128 bits of k.
- The attack requires 128 interactions with the QQ server and very little computation. (Note that the RSA private key d is not computed.)

7.3 Security of RSA encryption

Security of RSA Key Generation: If an adversary can factor n, then 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.

Definition 7.4. RSA Problem (RSAP): Given an RSA public key (n, e), and $c = m^e \mod n$ where $(m \in [0, n-1])$, compute m

The only effective method known for solving RSAP is to factor n (and then compute d and m). Henceforth, we shall assue that $RSAP$ is intractible.

7.3.1 Attacks for RSA Public Key Encryption

We can have two different attacks for RSA Public Key Encryption:

1. Dictionary Attack. Suppose that the plaintext is chosen from a relatively small (and known) set $\mathcal{M}$ of messages. Then, given a target ciphertext c, the adversary can encrypt each message in $\mathcal{M}$ until c is obtained.

   Counter Measure: we can append a randomly selected 128-bit string (called a **salt**) to m prior to encryption.

salt	m

   Note that m is now encrypted to one of 2^{128} possible ciphertexts.

2. Chosen-Ciphertext Attack: Suppose the adversary E has a target ciphertext c intended for Alice. Suppose also that E can induce Alice to decrypt **any** ciphertext for her, **except for** c **itself** (we say that Eve has a **dycryption oracle**). Then E can decrypt c as follows:

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

   (b) Compute $\hat{c} = c \cdot x^e \mod n$, where (n, e) is Alice's public key. Note that $\hat{c} \neq c$, unless $\gcd(c, n) \neq 1$. We can assume that, otherwise we can break RSA easily by just factoring out n

   (c) Obtain the decryption $\hat{m}$ of $\hat{c}$ from the decryption oracle. Note that $\hat{m} \equiv \hat{c}^d \equiv (cx^e)^d \equiv c^d x^{ed} \equiv m \cdot x \pmod{n}$

   (d) Compute $m = \hat{m} \cdot x^{-1} \mod n$

   Counter Measure: Add some **prescribed formatting** to m prior to encyption. After decrypting the ciphertext c, if the plaintext is not properly formatted, then c is rejected (so the decryption oracle does not return a plaintext).

So, RSA encryption should incorporate **salting** and **formatting**.

7.3.2 Security Definitions

Definition 7.5. Security Definition for Public-Key Encryption Scheme: A public-key encryption scheme is **secure** if it is semantically secure against chosen-ciphertext attack by a computationally bounded adversary.

To **break** a public-key encryption scheme, E should do:

1. E is given a challenge ciphertext c (and the public key (n, e))
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)

7.3.3 RSA Optimal Asymmetric Encryption Padding (OAEP)

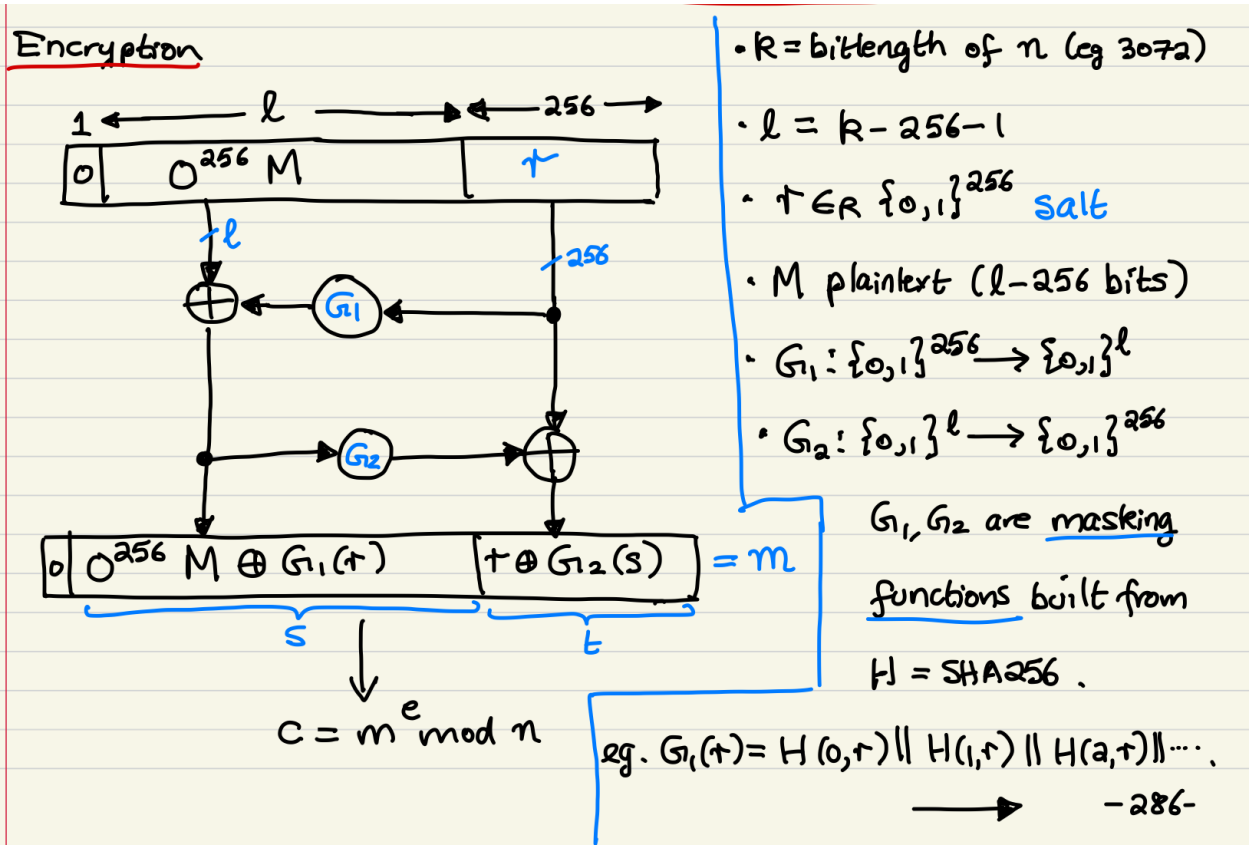


Figure 42: OAEP Encryption

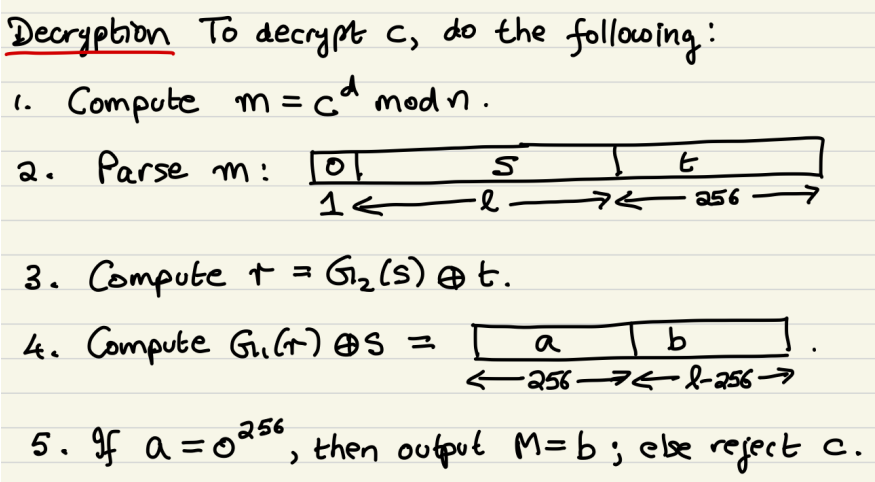


Figure 43: OAEP Decryption

Theorem 7.6. Suppose that RSAP is intractible, and G_1, G_2 are random functions. Then $RSA-OAEP$ is a secure public-key encryption scheme.

7.4 Status of Integer Factorization

7.4.1 Big-O and Little-o Notation

Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive real numbers.

Definition 7.7. Big-O notation: 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 $n \geq n_0$.

Definition 7.8. Little-o notation: We write $f(n) = o(g(n))$ if

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

7.4.2 Measures of Running Time

Definition 7.9. Polynomial-time algorithm One whose worst-case running time function is of the form $O(n^c)$, where n is the **input size** and c is a constant.

Definition 7.10. Exponential-time algorithm: One whose worst-case running time function is not of the form $O(n^c)$.

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

Definition 7.11. Subexponential-time algorithm 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, e.g. $O(2^{\sqrt{n}})$.

Roughly speaking, "polynomial-time = efficient", "fully exponential-time = terribly inefficient", "subexponential-time = inefficient, but not terribly so".

Example 7.12. Here is an example of trial division for factoring RSA-moduli n:

We trial divide n by the primes $2, 3, 5, 7, ..., \lfloor\sqrt{n}\rfloor$. 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 $O(\sqrt{n})$

Is this a polynomial-time algorithm for factoring RSA moduli?

No. Since the input size is $k = O(\log n)$ and the running time is $O(\sqrt{n}) = O(n^{\frac{1}{2}}) = O(2^{\frac{k}{2}})$ since $n = 2^k$. This is a fully exponential runtime.

Example 7.13. Here is an example of subexponential time.

Let A be an algorithm whose inputs are elements of the integers modulo n, $\mathbb{Z}_n$, or an integer n (so the input size is $O(\log n)$).

If the expected running time of A is of the form

$$L_n[\alpha, c] = O(\exp((c + o(1)(\log_e n)^\alpha)(\log_e \log_e n)^{1-\alpha}))$$

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

Note that:

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

Here are some **Special-Purpose Factoring Algorithms**: Trial division, Pollard's $p - 1$ algorithm, Pollard's ρ algorithm, elliptic curve factoring algorithm, special number field sieve.

These are only efficient if the number n being factored has a **special form** (e.g., n has a prime factor p such that $p-1$ has only small factors; or n has a prime factor p that is relatively small).

To maximize resistance to these factoring attacks on RSA moduli, one should select the RSA primes p and q **at random** and **of the same bitlength**.

General-Purpose Factoring Algorithms: These are factoring algorithms whose running times do not depend of any properties of the number being factored.

There have been two major developments in the history of factoring:

1. (1982) **Quadratic sieve factoring algorithm (QS)**: Running time: $L_n[\frac{1}{2}, 1]$.
2. (1990) **Number field sieve factoring algorithm (NFS)**: Running time: $L_n[\frac{1}{3}, 1.923]$.

Recall that

$$L_n[\alpha, c] = O(\exp((c + o(1)(\log_e n)^\alpha)(\log_e \log_e n)^{1-\alpha}))$$

7.4.3 History of Factoring

Year	Number	Bits	Method	Notes
1903	$2^{67}-1$	67	Naive	F. Cole (3 years of Sundays). In 2020: 0.02 secs in Maple
1988	$\approx 10^{100}$	332	QS	Distributed computation by 100's of computers
1994	RSA-129	425	QS	1600 computers around the world; 8 months
1999	RSA-155	512	NFS	300 workstations + Cray; 5 months
2003	RSA-174	576	NFS	
2005	RSA-200	663	NFS	(55 years on a single workstation)
2009	RSA-768	768	NFS	2000 core years
2019	RSA-240	795	NFS	900 core years
2020	RSA-250	829	NFS	2700 core years

The largest "hard" number factored to date is RSA-250 (250 decimal digits, 829 bits), which was factored on Feb 28 2020.

```
2140324650240744961264423072839333563008614715144755017797754920881418023447140136643345519095804679
6109928518724709145876873962619215573630474547705208051190564931066876915900197594056934574522305893
2597669747168173806936489469987157849497593749793
=
6413528947707158027879019017057738908482501474294344720811685963202453234463023862359875266834770873
766192558569463979885336733720275949781565562260106053551142279407603447675546667845209870238417 29
```

```
2100370802574486732968818 7
7565718986258036932062711
```

RSA-1024 factoring challenge (1024 bits, 309 decimal digits):

```
135066410865995223349603216278805969938881475605667027524485143851526510604859533833940287150571909441798207282164471551373680419703964191743046496589274256239341020864383202110372958725762358509643110564073501508187510676594629205563685529475213500852879416377328533906109750544334999811150056977236890927563
```

Equivalent Security Levels

Security in bits	Block cipher	Hash Function	RSA $\log_2 n$
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

Recall that 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

Summary:

- Factoring is **believed** to be a hard problem. However, we have no **proof** or **theoretical evidence** that factoring is indeed hard.
- However, factoring is **known** to be **easy** on a quantum computer (**Shor's algorithm**). The largest number factored with Shor's algorithm is the number **21**. The big open question is whether large-scale quantum computers can ever be built.
- 512-bit RSA is considered insecure today
- 1024-bit RSA is considered risky today (but still deployed)
- Applications are **moving** to **2048-bit** and **3072-bit** RSA.

7.5 RSA Signature Scheme

7.5.1 Basic RSA Signature Scheme

Algorithm 40: Key Generation of RSA Signature Scheme

Each entity Alice does the following:
Randomly select two large distinct primes p, q of the same bit length. (This can be done efficiently)
Compute $n = pq$ and $\phi = \phi(n) = (p-1)(q-1)$
Select an arbitrary integer e such that $1 < e < \phi$ and $\gcd(\phi, e) = 1$
Compute the integer d, $1 < d < \phi$, with $ed \equiv 1 \pmod{\phi}$
Alice's public key is (n, e); her private key is d

Algorithm 41: Signature Generation of RSA Signature Scheme

To sign a message $m \in \{0,1\}^*$, A does the following:
Compute $M = H(m)$, where H is a hash function
Compute $s = M^d \mod n$ (so $s^e \equiv M^{ed} \equiv M \mod n$)
A's signature on m is s (the signed message (m, s))

Algorithm 42: Signature Verification of RSA Signature Scheme

To verify A's signature s on m, B does the following:
Obtain an authentic copy of A's public key (n, e)
Compute $M = H(m)$
Compute $M' = s^e \mod n$
Accept (m, s) if and only if $M = M'$

7.5.2 Security of RSA Signature Scheme

Hardness of RSAP:

We require that RSAP be intractible, since 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
4. Then s is A's signature on m

Security Properties of the Hash Function:

Preimage Resistance: If H is not preimage resistant, and the range of H is $[0, n-1]$, E can forge signatures as follows:

1. Select $s \in_R [0, n-1]$
2. Compute $M = s^e \mod n$
3. Find m such that $H(m) = M$

4. Then s is A's signature on m

2nd Preimage Resistance: If H is not 2nd preimage resistant, E can forge signatures as follows:

1. Suppose (m, s) is a valid signed message
2. Find an m', $m \neq m'$ such that $H(m) = H(m')$
3. Then (m', s) is a valid signed message

Collision Resistance: If H is not collision resistant, E can forge signatures as follows:

1. Select m_1, m_2 such that $H(m_1) = H(m_2)$ where m_1 and m_2 are two distinct messages
2. Induce A to sign m_1: $s = H(m_1)^d \mod n$
3. Then s is also A's signature on m_2

Objectives of the Adversary:

1. **Total Break**: E recovers A's private key, or a method for systematically forging A's signatures (i.e. E can compute A's signature for arbitrary messages)
2. **Existential Forgery**: E forges A's signature for a single message; E may not have any control over the content or structure of this message.

Attack Model: Types of attacks E can launch:

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

Security Definition:

Definition 7.14. A signature scheme is said to be **secure** if it is existentially unforgeable by a computationally bounded adversary who launches a chosen-message attack

Note: the adversary has access to a signing oracle. Its goal is to compute a single valid message/signature pair for any message that was not previously given to the signing oracle.

Is the basic RSA signature scheme secure?

- No if H is SHA-256
- Yes if H is a "full domain" hash function.

7.5.3 Full Domain Hash RSA (RSA-FDH)

It's the same as the basic RSA signature scheme, except that the hash function is

$$H : \{0,1\}^* \to [0, n-1]$$

In practice, one could use:

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

Theorem 7.15. Bellare & Rogaway, 1996: If RSAP is intractable and H is a random function, then RSA-FDH is a secure signature scheme.

7.5.4 RSA PKCS #1 v1.5 Signatures

PKCS stands for public key cryptographic standards.

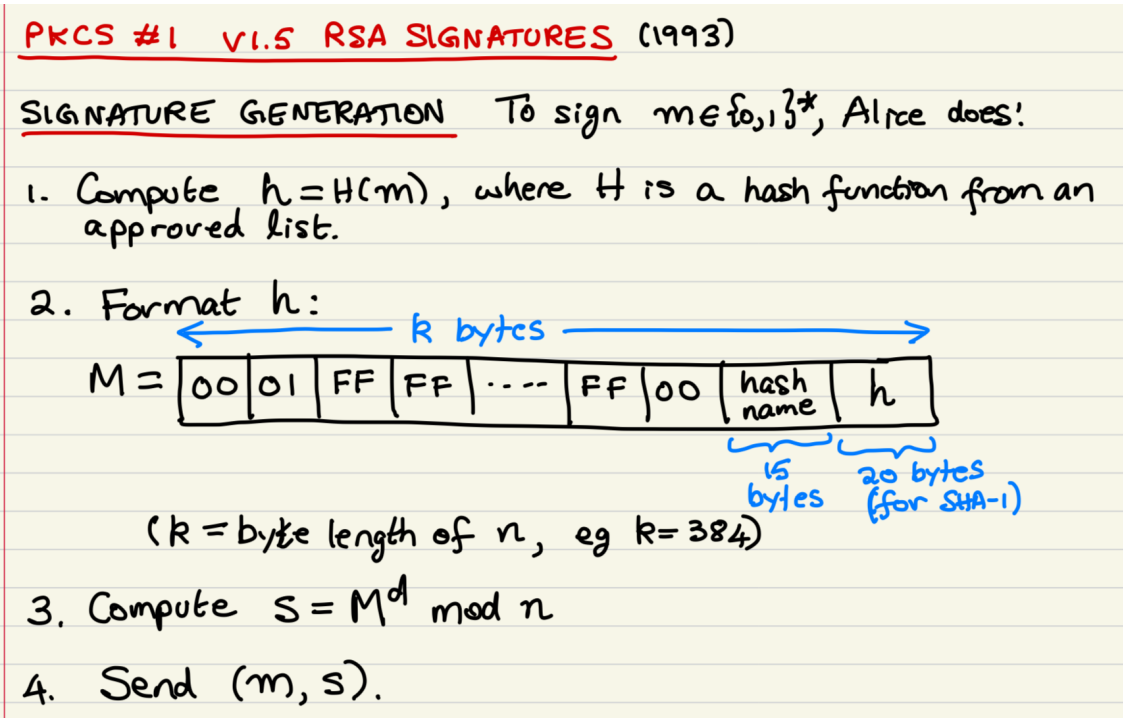


Figure 44: PKCS Signature Generation

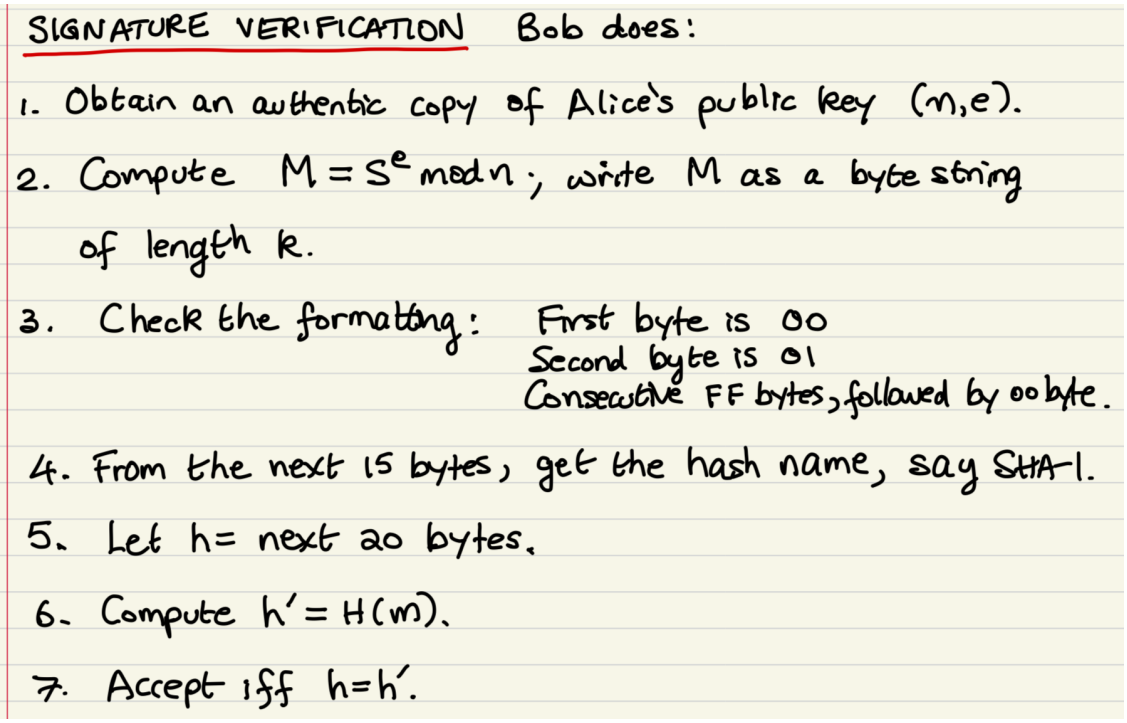


Figure 45: PKCS Signature Verification

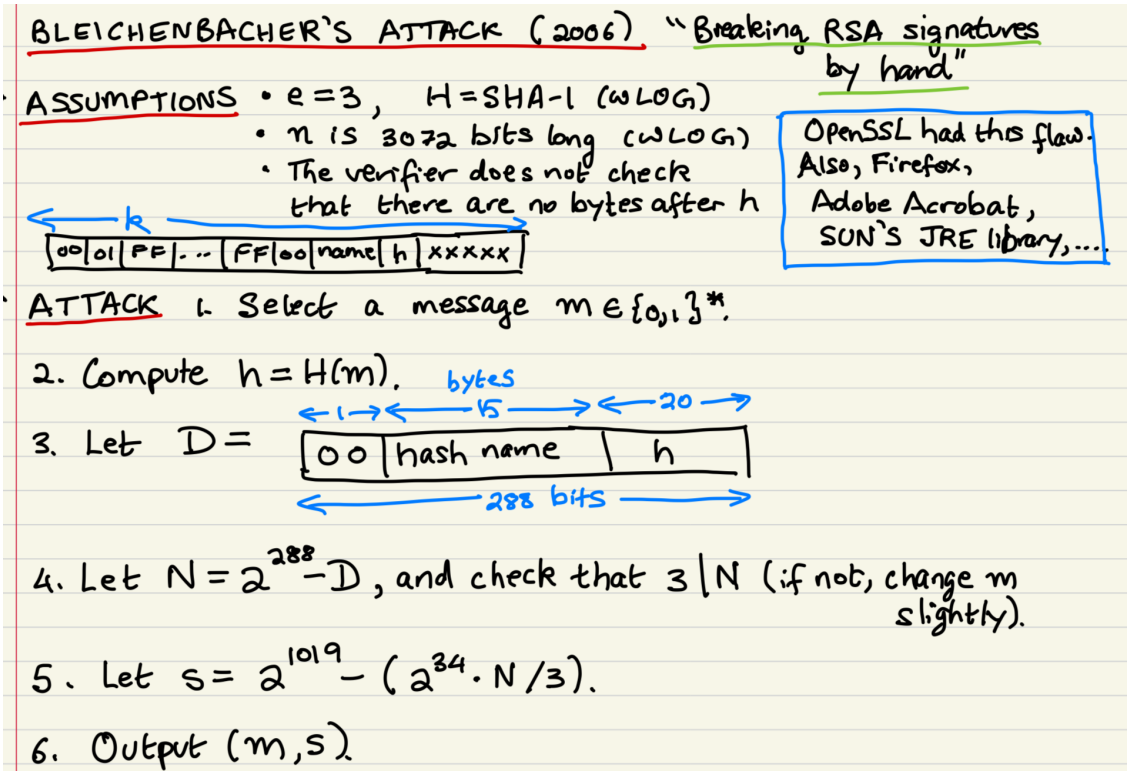


Figure 46: PKCS Bleichenbacher's Attack

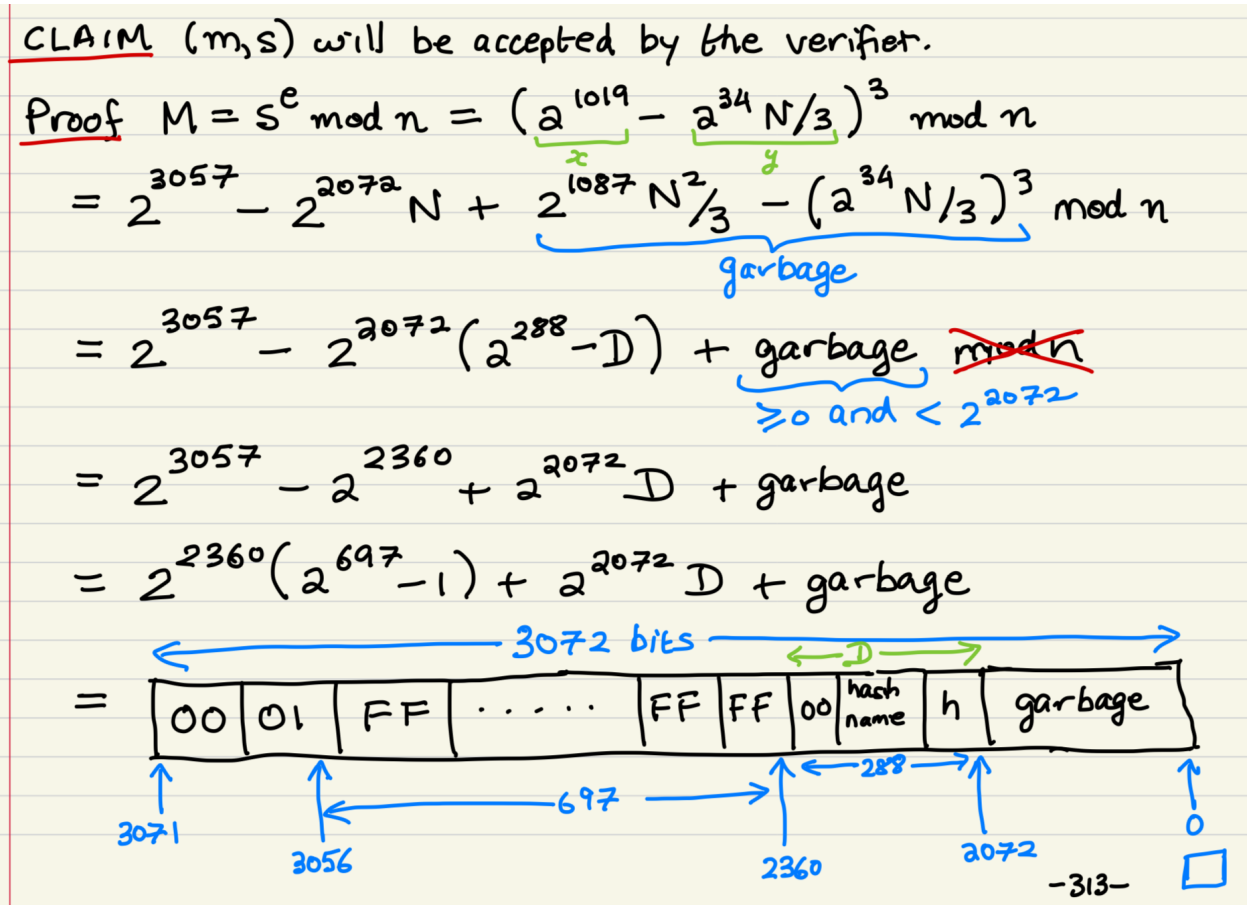


Figure 47: PKCS Bleichenbacher's Attack Correctness

Chapter 8 Elliptic Curve Cryptography

8.1 Elliptic Curves

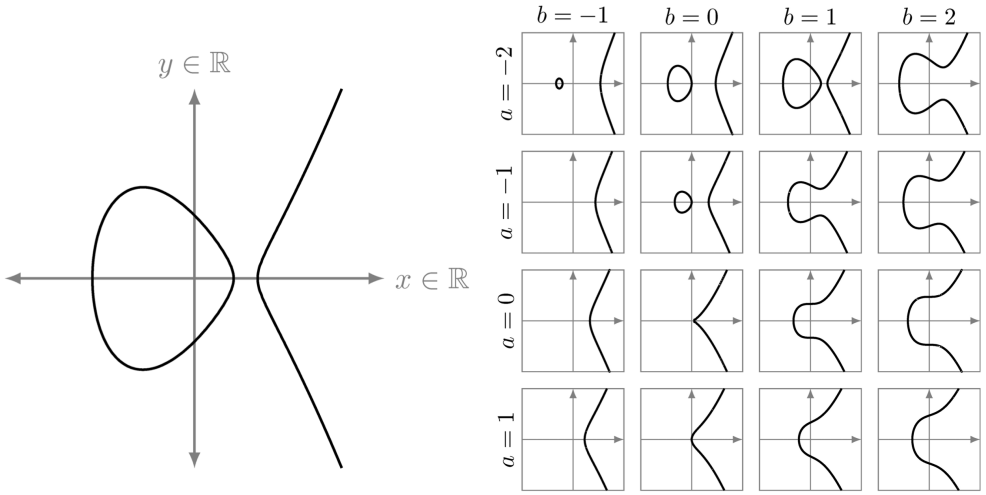
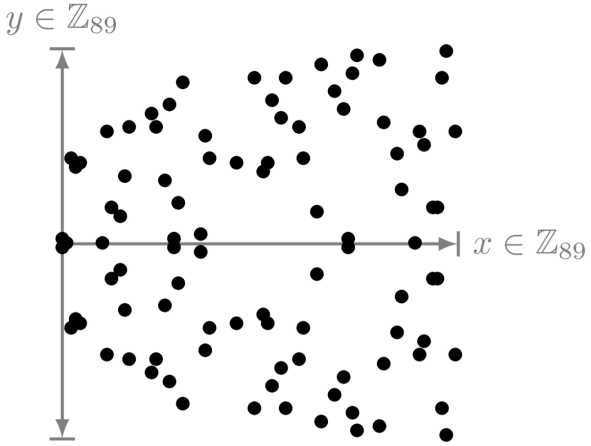


Figure 48: Elliptic Curves



$$E/\mathbb{Z}_{89} : Y^2 = X^3 - 2X + 1,\ \#E(\mathbb{Z}_{89}) = 96$$

Figure 49: Elliptic Curves over Finite Fields

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

Definition 8.1. 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 $4a^3 + 27b^2 \neq 0$

Definition 8.2. 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 called **the point at infinity**.

Here are some examples of elliptic curves.

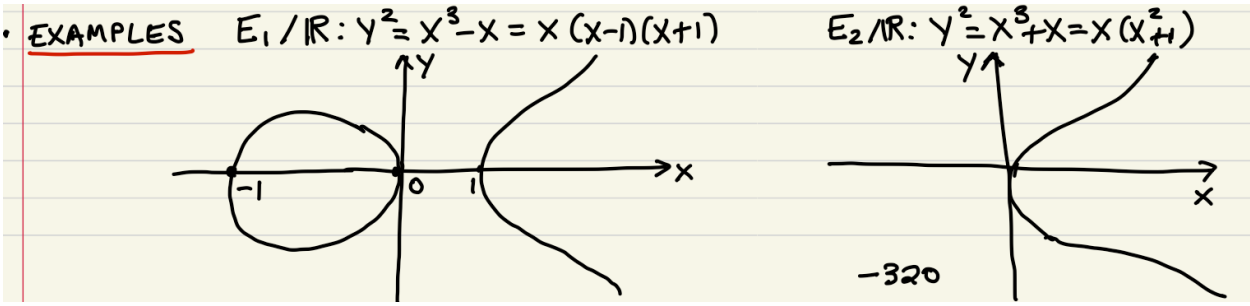


Figure 50: Elliptic Curves Examples

Example 8.3. Now, consider $E/\mathbb{Z}_{11} : Y^2 = X^3 + X + 6$. Then

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

So $\#E(\mathbb{Z}_{11} = 13)$. Note that $\#S$ is the cardinality of S

Note that $\#E(\mathbb{Z}_p)$ is finite. It is easy to see that $1 \leq \#E(\mathbb{Z}_p) \leq 2p+1$. In fact, we have the following theorem:

Theorem 8.4. 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, $\#E(\mathbb{Z}_p) \approx p$

Point Addition: There is a "natural" way to add two points in $E(F)$ to get a third point in $E(F)$

Geometric Description of the Addition Rule: Let E be an elliptic curve over $\mathbb{R}$. Think of ∞ as an imaginary point through which every vertical line passes (in either direction)

The geometric rule is: Let $P, Q \in E(F)$. Let $T \in E(F)$ be the third point of intersection of the line ℓ through P and Q with the elliptic curve. Then $P + Q$ is the reflection of T in the X-axis.

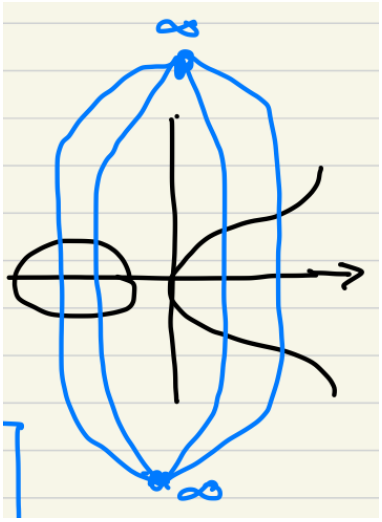
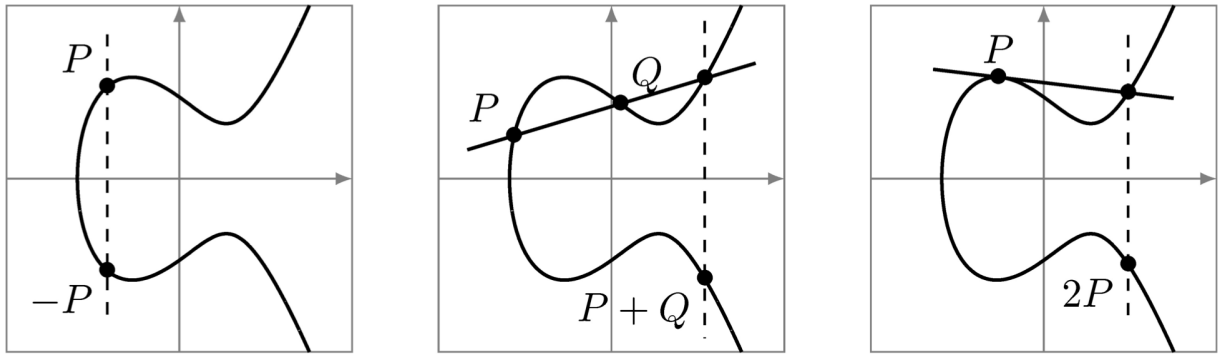


Figure 51: Elliptic Curves Geometric Rule Description



Inverse element $-P$ | Addition $P + Q$ "Chord rule" | Doubling $P + P$ "Tangent rule"

Figure 52: Elliptic Curves Addition Rules

Proposition 8.5. The geometric rules of elliptic curves are defined as follows:

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 $-\infty = \infty$. Also, $P + (-P) = (-P) + P = \infty$, 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

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

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

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

Proof 8.6. Derivation of the addition formula in A3 above:

Let $P = (x_1, y_1), Q = (x_2, y_2) \in E(F)$ with $P \neq \pm Q$

The equation of the line ℓ through P and Q is $\ell : Y = y_1 + \lambda(X - x_1)$, denote it as $(*)$, where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ is the slope of ℓ

To find the third point of intersection of ℓ with E, we substitute $(*)$ into the equation for $E : Y^2 = X^3 + aX + b$ to get:

$$[y_1 + \lambda(X - x_1)]^2 = X^3 + aX + b$$

So

$$X^3 + aX + b - [y_1 + \lambda(X - x_1)]^2 = 0$$

This equation has two solutions in F, namely x_1 and x_2. Thus, it must have a third solution in F, say x_3. Thus, we have

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

Equating coefficients of X^2 of both sides gives

$$-\lambda^2 = -(x_1 + x_2 + x_3)$$

So

$$x_3 = \lambda^2 - x_1 - x_2$$

The coordinates of T are (x_3, y_3'), where $y_3' = y_1 + \lambda(x_3 - x_1)$. Hence, the coordinates of $P + Q$ are $(x_3, y_3) = (x_3, -y_3')$ so

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

For derivation of the addition formula in A4, the process is similar. The slope of the tangent line ℓ at $P = (x_1, y_1)$ is $\lambda = \frac{3x_1^2+a}{2y_1}$.

Also, $E : Y^2 = X^3 + aX + b$, we can compute derivatives: $2Y \cdot \frac{dY}{dX} = 3X^2 + a$, so $\frac{dY}{dX} = \frac{3X^2+a}{2Y}$ □

Example 8.7. 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

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

$$(2,4) + (2,7) = \infty$$
$$(2,4) + (8,3) = (5,2)$$
$$(2,4) + (2,4) = (5,9)$$

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

P1: $P + \infty = \infty + P = P$ for all $P \in E(F)$

P2: For each $P \in E(F)$, there exists $Q \in E(F)$ such that $P + Q = Q + P = \infty$

P3: $P + Q = Q + P$ for all $P, Q \in E(F)$

P4: $(P + Q) + R = P + (Q + R)$ for all $P, Q, R \in E(F)$

8.2 Elliptic Curves Discrete Logarithm Problem(ECDLP)

Let $E : Y^2 + X^3 + aX + b$ be an elliptic curve defined over $F = \mathbb{Z}_p$. Let $n = \#(\mathbb{Z}_p)$ and suppose that n is a prime. Note: $n \approx p$ by Hasse's Theorem.

Definition 8.8. 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 operation is called **point multiplication**.

Theorem 8.9. Let $P \in E(\mathbb{Z}_p)$, $P \neq \infty$. Then:

1. $nP = \infty$
2. The points $\infty, P, 2P, \cdots, (n-1)P$ are distinct, and so $E(\mathbb{Z}_p) = \{\infty, P, 2P, \cdots, (n-1)P\}$. P is called a generator of $E(\mathbb{Z}_p)$

Definition 8.10. The **elliptic curve discrete logarithm problem(ECDLP)** is the following:

Given $E, p, n, P \in E(\mathbb{Z}_p)$ (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 $\ell = \log_p Q$

We have several approaches to this problem: **Brute Force**: The ECDLP can be solved by computing $P, 2P, 3P, ...$ until Q is encountered. This attack takes time $O(n)$ point additions, or $O(p)$ additions (recall that $n \approx p$).

This is **fully exponential** since an ECDLP instance has size $O(\log p)$ bits

Shank's algorithm (for solving ECDLP): Idea. Let $m = \lfloor \sqrt{n} \rfloor$ By the division algorithm, there exists unique integers q, r with $\ell = qm + r$, where $0 \leq r < m$ and $0 \leq q < m$. Hence, $\ell - qm = r$. Multiplying both sides by P gives

$$\ell P - qmP = rP$$

so $Q - q(mP) = rP$

This equation suggests the following algorithm for finding $\ell = \log_p Q$:

1. For each $e \in [0, m-1]$, compute rP and store (rP, r) in a sorted table
2. Compute $M = mP$
3. For each $q \in [0, m-1]$, compute $R = Q - qM$ and look it up in the table. If $R = rP$, then output "$\ell = qm + r$" and STOP

Runtime: $O(m) = O(\sqrt{n}) = O(\sqrt{p})$ point additions.

Storage: $O(\sqrt{p})$ points

Pollard's algorithm: The ECDLP can be solved in time $O(\sqrt{p})$ point additions and negligible storage. Moreover, VW collision research can be used to perfectly parallelize Pollard's algorithm.

Note that Pollard's algorithm is the fastest method known for solving the ECDLP (except for some very special elliptic curves that can easily be avoided in practice).

Shor's algorithm: The ECDLP can be solved in polynomial time on a quantum computer

8.3 Elliptic Curves Cryptography

It is invented by Neal Koblitz and Victor Miller. Why do we want Elliptic Curves Cryptography?

RSA: Security is based on intractability of **integer factorization**. Fastest known attacks take **subexponential time**.

ECC: Security is based on intractability of **ECDLP**. Fastest known attacks take **fully exponential time**.

Hence, elliptic curve cryptographic systems can use **smaller parameters** than their RSA counterparts, while achieving the same security level. Smaller parameter leads to faster implementation and smaller public keys and signature sizes.

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

Deployment of ECC generally use an elliptic curve that has been standardized (by NIST). Examples include `P-256`, `Curve25519`, `P-384`.

8.3.1 P-256 Elliptic Curve

- P-256 is an elliptic curve chosen by the National Security Agency in 1998 for U.S. government use.
- P-256 should be used for applications that require the **128-bit security level**.
- $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ is prime.
- P-256 is the elliptic curve $E : Y^2 = X^3 - 3X + b$ over $\mathbb{Z}_p$, where $b =$

  `41058363725152142129326129780047268409114441015993725554835256314039467401291`
- We have $n = \#E(\mathbb{Z}_p)$ is prime, where $n =$

  `1157920892103562487626974469494075735299969552241357603424222590610685120443 69`

8.3.2 Curve25519

- Selected by **Dan Bernstein** in 2005, and developed with **Tanja Lange** and others.
- Curve25519 should be used for applications that require the **128-bit security level**.
- $p = 2^{255} - 19$ is prime.

- Curve25519 is the elliptic curve $E : Y^2 = X^3 + 486662X^2 + X$ over $\mathbb{Z}_p$. The curve is in "Montgomery form".
- We have $n = \#E(\mathbb{Z}_p) = 8n$, where n is the following 253-bit prime

$$2^{252} + 27742317777372353535851937790883648493$$

8.3.3 P-384 Elliptic Curve

- P-384 is an elliptic curve chosen by the National Security Agency in 1998 for U.S. government use.
- P-384 should be used for applications that require the **192-bit security level**.
- $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$ is prime.
- P-384 is the elliptic curve $E : Y^2 = X^3 - 3X + b$ over $\mathbb{Z}_p$, where $b =$

```
27580193559959705877849011840389048093056905856361568521428707301988689241309860865136260764883745107765439761230575
```

- We have $n = \#E(\mathbb{Z}_p)$ is prime, where $n =$

```
39402006196394479212279040100143613805079739270465446667946905279627659399113263569398956308152294913554433653942643
```

8.3.4 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 and efficiently implemented on a **32-bit machine** without doing long division.

To illustrate this technique, let's consider reduction modulo the prime $p = 2^{192} - 2^{64} - 1$ on a **64-bit machine**. By a **64-bit machine**, we mean a computer with built-in instructions for addition, subtraction, multiplication of 64-bit integers.

So,

- let $p = 2^{192} - 2^{64} - 1$. Suppose that $a, b \in [0, p-1]$. We wish to compute $c = a \cdot b \mod p$.
- We have that $a = a_2 2^{128} + a_1 2^{64} + a_0$ and $b = b_2 2^{128} + b_1 2^{64} + b_0$ where $a_0, a_1, a_2, b_0, b_1, b_2$ are 64-bit integers.
- We first compute $\alpha = a \cdot b$ using ordinary 2^{64}-base integer arithmetic.
- Let $\alpha = \alpha_5 2^{320} + \alpha_4 2^{256} + \alpha_3 2^{192} + \alpha_2 2^{128} + \alpha_1 2^{64} + \alpha_0$ where each α_i is a 64-bit integer.
- We now need to reduce α modulo p.

- We have

$$2^{192} \equiv 2^{64} + 1 \pmod{p}$$
$$2^{256} \equiv 2^{128} + 2^{64} \pmod{p}$$
$$2^{320} \equiv 2^{192} + 2^{128} \equiv 2^{128} + 2^{64} + 1 \pmod{p}$$

  Hence,

$$\begin{aligned} c &= d \pmod{p} \\ &= d_5 2^{320} + d_4 2^{256} + d_3 2^{192} + d_2 2^{128} + d_1 2^{64} + d_0 \pmod{p} \\ &= d_5(2^{128} + 2^{64} + 1) + d_4(2^{128} + 2^{64}) + d_3(2^{64} + 1) + d_2 2^{128} + d_1 2^{64} + d_0 \pmod{p} \\ &= \boxed{d_5 \mid d_5 \mid d_5} + \boxed{d_4 \mid d_4 \mid 0} + \boxed{0 \mid d_3 \mid d_3} + \boxed{d_2 \mid d_1 \mid d_0} \pmod{p} \end{aligned}$$

- This suggests the following algorithm for computing $c = a \cdot b \pmod{p}$

Algorithm 43: Modular Reduction without Long Division

Input: $a, b \in [0, p-1]$
Output: $c = a \cdot b \pmod{p}$

1 compute the 384-bit number $d = a \cdot b = (d_5, d_4, d_3, d_2, d_1, d_0)$ where each d_i is a 64-bit integer
2 Define the 192-bit integers

$$t_1 = (d_2, d_1, d_0) \quad t_2 = (0, d_3, d_3) \quad t_3 = (d_4, d_4, 0) \quad t_4 = (d_5, d_5, d_5)$$

3 Compute $c = t_1 + t_2 + t_3 + t_4$
4 If $c \geq p$, then repeatedly subtract p from c until $c \in [0, p-1]$
5 **return** c

Note that there is no long division perfomed. In step 3, we have $0 \leq c < 4p$, so at most three subtractions by p are required in step 4.

8.4 Elliptic Curves Diffie-Hellman (ECDH)

ECDH is the elliptic curve analogue of a key agreement protocol first proposed by Diffie and Hellman in 1975.

The objective of ECDH is that: two communicating parties agree upon a shared secret key k. They can then use k in a symmetric-key scheme such as HMAC or AES-GCM.

ECDH Domain Parameters (These parameters are public):

- An elliptic curve $E : Y^2 = X^3 + aX + b$ defined over $\mathbb{Z}_p$, such as P-256
- $n = \#E(\mathbb{Z}_p)$ where n is prime
- A generator $P \in E(\mathbb{Z}_p)$
- A key derivation function KDF (such as a hash function)

8.4.1 Unautenticated ECDH

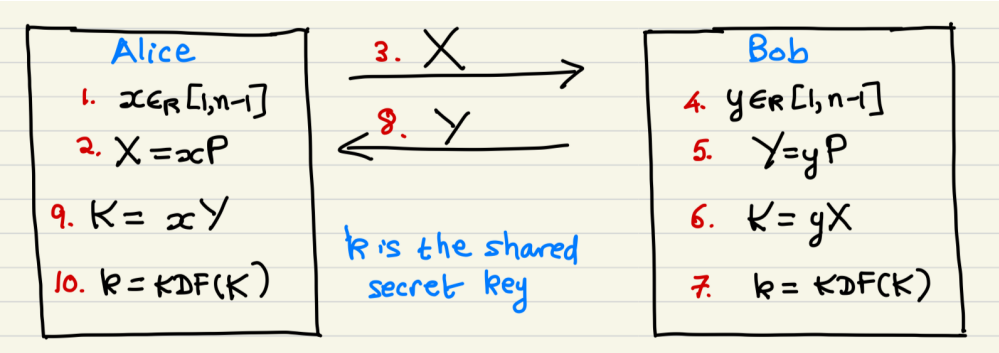


Figure 53: Unautenticated ECDH

Alice and Bob computes the same k since

$$xY = x(yP) = (xy)P = (yx)P = y(xP) = yX$$

An eavesdropper sees X, Y, and the domain parameters. Her goal is to compute $K = xY = yX$; this is called the **ECDH problem**. The fastest way konwn to solve this problem is to first compute x or y, which is precisely ECDLP.

8.4.2 Malicious-Intruder-In-the-Middle-Attack (MITM)

Since X and Y are not authenticaed, unautenticated UCDH is vulnerable to a MITM attack. The attack is as follows:

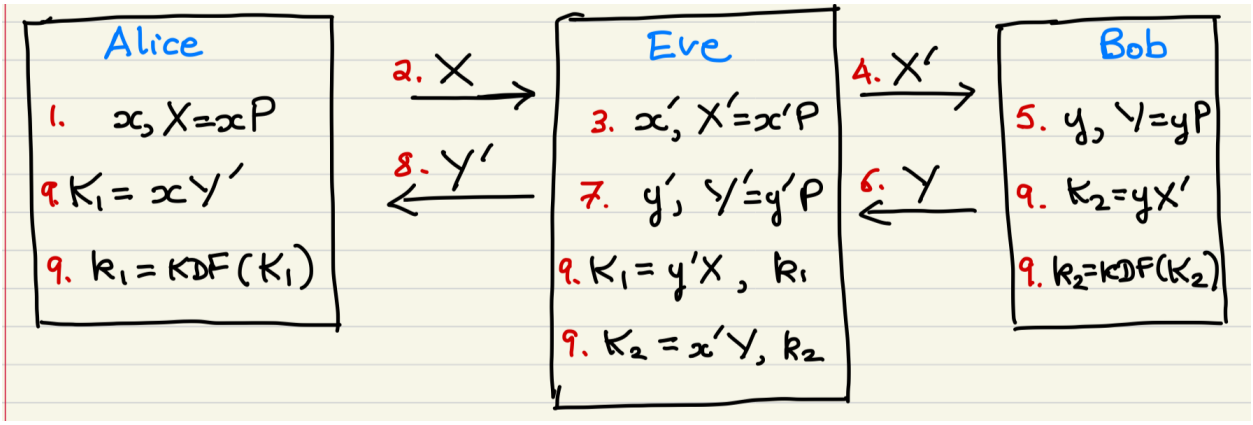


Figure 54: Malicious Intruder In the Middle Attack

Suppose now that Alice sends $c = E_{k_1}(m)$ to Bob, where $E = AES - GCM$. Eve intercepts c, computes $m = E_{k_1}^{-1}(c), c' = E_{k_2}(m)$, and sends c' to Bob. Bob computes $m = E_{k_2}^{-1}(c')$. Alice and Bob won't know Eve has computed m as well. Confidentiality is compromised.

8.4.3 Authenticated ECDH

- Alice sends Bob X, her RSA/ECDSA signature on X, and a certificate for her RSA/ECDSA public key
- Bob verifies the certificate, and then uses Alice's public key to verify the signature on X.
- Similarly, Bob signs Y.

Note that MITM attack is thwarted.

8.5 Google and ECC

In November 2011, Google started using ECDH as its default key establishment mechanism in TLS-secured applications including **Gmail** (`https://gmail.com`) and encrypted **search** (`https://google.com`).

ECDH was chosen because it provides **forward secrecy**.

The elliptic curve used was **P-256**.

8.5.1 TLS Previous Implementation

Here is a description of the TLS protocol, which was commonly implemented. When a web browser (Alice) visits a secured web page (Bob):

1. Bob sends its **certificate** to Alice.
2. Alice verifies the signature in the certificate using the (intermediate) Certificate Authority's RSA public key. If the certificate verifies, then Alice is assured that she has an authentic copy of Bob's RSA public key.
3. Alice selects a random **session key** k, and encrypts k with Bob's RSA public key. Alice sends the resulting ciphertext c to Bob.
4. Bob decrypts c using his RSA private key and obtains k
5. The session key is used to authenticate (with HMAC) and encrypt (with AES) all data exchanged for the remainder of the session.

This does not provide **forward secrecy**. That is, suppose that an eavesdropper saves a copy of c and the encrypted data. If, at a future point in time, the eavesdropper is able to break into Bob's machine and learn his RSA private key, then the eavesdropper is able to decrypt c and thus recover k and the data that was encrypted with k.

Alternatively, a law enforcement agent could demand that Bob hand over his RSA private key.

Using ECDH to establish a session key k provides forward secrecy.

8.5.2 TLS Google Implementation

So, Google implemented TLS in the following way. When a web browser (Alice) visits a secured web page (Bob):

1. Bob sends its **certificate** to Alice.
2. Alice verifies the signature in the certificate using the (intermediate) Certificate Authority's RSA public key. If the certificate verifies, then Alice is assured that she has an authentic copy of Bob's RSA public key.
3. Alice selects $x \in_E [1, n-1]$ and sends $X = xP$ to Bob. (P is a fixed point on P-256.)
4. Bob selects $y \in_R [1, n-1]$, signs $Y = yP$ with its RSA signing key, and sends Y and the signature to Alice.
5. Alice verifies the signature using Bob's RSA public key, and is thus assured that Y was sent by Bob.
6. Both Alice and Bob compute the shared secret $K = xY = yX = xyP$, and derive a session key k from K.
7. Alice deletes x. Bob deletes y.
8. The session key is used to authenticate and encrypt (with AES-GCM) all data exchanged for the remainder of the session.
9. At the end of the session, Alice and Bob delete k.

Note that forward secrecy is provided.

8.5.3 RSA vs DL vs ECC Usage in TLS

Percentage of https connections that use RSA, DL or ECC as of November 2016:

Key exchange		Signatures	
RSA	39%	RSA	99%
DH	10%	DSA	$> 0\%$
ECDH	51%	ECDSA	1%

8.6 Elliptic Curves Digital Signature Algorithm (ECDSA)

8.6.1 ECDSA Basics

The ECDSA Domain parameters are:

- Elliptic Curve E, defined over $\mathbb{Z}_p$, such as P-256
- $n = \#E(\mathbb{Z}_p)$ where n is a prime
- A generator $P \in E(\mathbb{Z}_p)$
- A collision-resistant hash function H, such as SHA-256

Algorithm 44: ECDSA Key Generation

1 Select $a \in_R [1, n-1]$ and compute $A = aP$
2 Alice's **public key** is A, and her **private key** is a

Note that computing a from A is precisely an instance of the ECDLP. Note that a fresh

Algorithm 45: ECDSA Signature Generation

1 To sign a message $M \in \{0,1\}^*$, Alice does the following:
2 Compute $m = H(M)$, and interpret m as an integer
3 Select a **per-message secret** $k \in_R [1, n-1]$
4 Compute $R = kP$. Let $r = x(R) \mod n$, and check that $r \neq 0$. Note that $x(R)$ is the x-coordinate of the point R
5 Compute $s = k^{-1}(m + ar) \mod n$, and check that $s \neq 0$
6 Alice signature on M is (r, s)

random k should be selected each time Alice signs a message (Otherwise Eve gets a system of two equations with two unknowns, namely k and a. Eve could solve this efficiently and computes the private key a). Also k should be securely destroyed after it is used. The

Algorithm 46: ECDSA Signature Verification

1 To verify Alice's signature (r, s) on M, Bob does:
2 Obtain an authentic copy of Alice's public key A
3 Check that $1 \leq r, s \leq n-1$
4 Compute $m = H(M)$
5 Compute $u_1 = ms^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$
6 Compute $V = u_1 P + u_2 A$ and verify that $V \neq \infty$
7 Compute $v = x(V) \mod n$
8 **Accept** $M, (r, s)$ if and only if $v = r$

correctness of this algorithm relies on the following equivalence relation:

$$\begin{aligned} s = k^{-1}(m + ar) \mod n &\Leftrightarrow k \equiv s^{-1}(m + ar) \mod n \\ &\Leftrightarrow k \equiv u_1 + u_2 a \mod n \\ &\Leftrightarrow kP = u_1 P + u_2 aP \\ &\Leftrightarrow kP = u_1 P + u_2 A \\ &\Leftrightarrow kP = V \\ &\Rightarrow x(kP) = x(V) \\ &\Leftrightarrow r = v \end{aligned}$$

Thus, if Alice generated $M, (r, s)$, Bob will accept.

8.6.2 Security of ECDSA

ECDSA is believed to be **secure**, assuming that the ECDLP is intractable, and H is a "secure" hash function

8.6.3 ECDSA vs RSA

In practice, for **RSA-FDH** we use 3072-bit n, and for **ECDSA** we use 256-bit p:

1. In practice, RSA is used with $e = 3$ or $e = 2^{16} + 1$. So, RSA **signature verification**, which is computing $s^e \mod n$ is generally faster than ECDSA, which is computing $V = u_1P + u_2A$
2. RSA signature generation, which is computing $H(M)^d \mod n$ (where d is a 3072-bit integer) is generally slower than ECDSA, which is computing $R = kP$.
3. RSA signatures are 3072 bits in length; ECDSA signatures have bitlength 512 bits.

8.6.4 Performance

Running benchmarks using openssl:

Technology	# of sig generations per second	# of sig verifications per second
ECDSA with P-256	1730	2295
ECDSA with P-384	807	1207
RSA with 2048-bit n	938	17015
RSA with 4096-bit n	125	4489

Chapter 9 Bluetooth Security

9.1 Brief about Bluetooth

- Industry standard for short-range low-power wireless communication technology.
- Primarily used to establish wireless personal area networks (WPANs).
- Bluetooth v1.0 released in 1999.
- **Bluetooth v5.2** released on **December 31, 2019**.
- In 2017, Bluetooth product shipments were more than 3.6 billion.
- Four variants: **BR** (basic rate), **EDR** (enhanced data rate), **HS** (high speed), **LE** (low energy)
- The Bluetooth specification is about 3,000 pages long, which makes a comprehensive security analysis very difficult.

Five basic security services are specified in the Bluetooth standard:

1. **Confidentiality**: preventing information compromise caused by eavesdropping by ensuring that only authorized devices can access and view transmitted data
2. **Message Integrity**: verifying that a message sent between two Bluetooth devices has not been altered in transit
3. **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 secret keys and the storing of these keys for use in subsequent connections in order to form a trusted device pair.

To establish a long-term secure Bluetooth connection, the two devices have to "pair".

There are three security mechanisms:

1. Legacy
2. Secure Simple Pairing
3. **Secure Connections**

9.2 Secure Connections

Four association models depending on the I/O capabilities of the devices:

1. **Numeric Comparision**: requires both devices to have displays for 6-digit numbers, and one of the two devices to have an "OK/Reject" confirmation button.

2. **PassKey Entry**: requires one device to have input capability (e.g., keyboard), and the other device to have display (but not input) capability.
3. **Just Works**: designed for the situation where at least one of the pairing devices has neither a display nor a keyboard (e.g. headsets)
4. **Out-of-Band** (vendor specific): designed for devices that suppport a common additional wireless (e.g. NFC) or wired technology for the purpose of device discovery and cryptographic value exchange.

Here is notations that we will be using:

- P-256 elliptic curve: E, p, n, P.
- Alice: the initiating device.
- Bob: the responder 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.

9.2.1 Phase 1: Public Key Exchange

The purpose is to establish an (unauthenticated) shared secret.

Algorithm 47: Public Key Exchange in Bluetooth Secure Connections

ECDH is used:
Alice selects $x \in_R [1, n-1]$, computes $X = xP$ and sends X to Bob.
Bob selects $y \in_R [1, n-1]$, computes $Y = yP$ and sends Y to Alice. Bob computes $K =$ the x-coordinate of yX.
Alice computes $K =$ the x-coordinate of xY.
The shared secret is K (256 bits)

Algorithm 48: Authentication Stage 1 in Bluetooth Secure Connections

1 Bob selects **nonce** $N_B \in_R \{0,1\}^{128}$, computes a **commitment** $C_B = \texttt{HMAC-SHA-256}N_B(Y, X)$, 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 $C_B = \texttt{HMAC-SHA-256}N_B(Y, X)$.
5 Alice computes **verification value** $V_A = (\texttt{SHA-256}(X, Y, N_A, N_B) \mod 2^{32}) \mod 10^6$, and displays this value.
6 Bob computes **verification value** $V_B = (\texttt{SHA-256}(X, Y, N_A, N_B) \mod 2^{32}) \mod 10^6$, and displays this value.
7 If $V_A = V_B$, the owner presses the "OK" button.

9.2.2 Phase 2: Authentication Stage 1

The purpose is to provide some protection against active MITM attacks. Notes:

- The use of the commitment C_B means that Alice and Bob each selects their nonces before seeing the other party's nonce.
- Failure in step 4 indicates the presence of an attacker, or other transmission error.
- An active MITM attack will result in the two 6-digit verification values V_A, V_B being different with probability 0.999999.

9.2.3 Phase 3: Authentication Stage 2

The purpose is to confirm that both devices have successfully completed the exchange

Algorithm 49: Phase 3: Authentication Stage 2 in Bluetooth Secure Connections

1 Alice computes $E_A = 128$ least significant bits of $\texttt{HMAC-SHA-256}_K(N_A, N_B, A, B)$ and sends **exchange confirmation value** E_A to Bob.
2 Bob verifies that the 128 least significant bits of $\texttt{HMAC-SHA-256}_K(N_A, N_B, A, B)$ are equal to E_A, computes **exchange confirmation value** $E_B = 128$ most significant bits of $\texttt{HMAC-SHA-256}_K(N_B, N_A, B, A)$, and sends E_B to Alice.
3 Alice verifies that the 128 most significant bits of $\texttt{HMAC-SHA-256}_K(N_B, N_A, B, A)$ are equal to E_B.

9.2.4 Phase 4: Link Key Calculation

The purpose is to compute the long-term **link key** LK.

Algorithm 50: Phase 4: Link Key Calculation in Bluetooth Secure Connections

1 Both parties compute $LK = 128$ most significant bits of $\texttt{HMAC-SHA-256}_K(N_A, N_B, A, B)$.

Notes:

- The link key is the long-term authentication key.
- The link key is used to maintain the pairing.
- The nonces ensure the freshness of the link key even if long-term ECDH values are used by both sides.

9.2.5 Authentication and Encryption

The purpose is to check that the communicating devices hold the same link key (and thus are paired), and then derive a shared secret input to the encryption-key generation procedure.

Algorithm 51: Authentication and Encryption in Bluetooth Secure Connections

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:
4 Device authentication key: $h = 128$ most significant bits of $\texttt{HMAC-SHA-256}_{LK}(A, B)$.
5 Device authentication confirmation values: $t = \texttt{HMAC-SHA-256}_h(R_A, R_B)$, $S_A =$ leftmost 32 bits of t, $S_B =$ next 32 bits of t, $ACO =$ next 64 bits of t.
6 Alice sends S_A to Bob (**signed response**).
7 Bob ends S_B to Alice (**signed response**).
8 Alice and Bob compare the signed responses they received with the values they computed.

- 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 the two communicating devices.
- For 128-bit keys: $K_E = 128$ most significant bits of $\texttt{HMAC-SHA-256}_{LK}(A, B, ACO)$. For smaller ℓ-bit keys, take the ℓ most signficant bits, and set the remaining bits to 0.
- Encryption is performed using either the **E0 stream cipher** in Bluetooth version 4.0 and earlier, or **AES-CCM** (AES-CTR encryption + Cipher Block Chaining MAC) in Bluetooth version 4.1 and later.

9.2.6 KNOB Attack

Discovered by Antonioli-Tippenhauer-Rasmussen in August 2019.

- The encryption key negotiation protocol is used by two Bluetooth devices to agree on the length of encryption keys K_E. This was introduced to cope with international encryption regulations and to facilitate security upgrades.
- **The negotiation is neither authenticated nor encrypted**.
- So, an active adversary can cause the devices to agree to produce an 8-bit key K_E. This key can then be brute forced in real time (and thus confidentiality is lost)
- The attack was implemented, and used to decrypt an encrypted file exchanged over Bluetooth between a Nexus 5 phone and a Motorola G3 phone.
- Most Bluetooth chips (Intel, Broadcom, Apple...) and devices were vulnerable to the attack.
- **Countermeasure**: Set the key length to 128 bits.

Chapter 10 Key Management

10.1 Public Key Management

Definition 10.1. Key Management: A set of techniques and procedures supporting the establishment and maintenance of keying relationships between authorized parties.

We will consider management of public keys that are used for public-key encryption, Diffie-Hellman key agreement, and for verification of digital signatures. Consider the following two scenarios:

1. Suppose that A wishes to use public-key encryption (or hybrid encryption) to encrypt a message for B . To do this, A needs an **authentic** copy of B's public key. For example:
   - A wishes to send B a confidential email.
   - A wishes to send her credit number to B while making an online purchase.
2. Suppose that A receives a message purportedly signed by B . To verify the signature on the message, A needs an **authentic** copy of B's public key. For example:
   - A person A wishes to verify the authenticity of a software patch that was purportedly signed by Microsoft (B).
   - The manager B of a bank branch can authorize financial transactions worth up to $20,000.

Some questions and concerns:

- Where does A get B's public key from?
- How does A know she really has B's public key?
- How can a bank limit use of B's public/private key pair?
- What happens if B's private key is compromised? Who is **liable**?
- How can a bank **revoke** B's public key?
- How can B's public keys be updated?

Techniques for Distributing Public Keys:

1. Point-to-point delivery over a trusted channel.
   - Trusted courier.
   - One-time user registration.
   - Voice.
   - Embedded in a browser or operating system.

2. Direct access to a trusted public file. Such as Digitally signed file.
3. Digitally signed file. Such as Online Certification Status Protocol (OCSP).
4. **Off-line certification authority (CA).** We will focus on this in this course.

10.2 Certification Authorities (CAs)

A CA issues **certificates** that bind an entity's identity A and its public key.

A's **certificate** $Cert_A$ consists of:

1. Data part D_A : A's identity, her public key, and other information such as validity period.
2. Signature part S_T : The CA's signature on the data part.

B obtains an authentic copy of A's public key as follows:

1. Obtain an authentic copy of the CA's public key (e.g., shipped in browsers or in an operating system).
2. Obtain $Cert_A$ (over an unsecured channel).
3. Verify the CA's signature S_T on D_A.

Note that the CA does not have to be trusted with users' private keys. Also, the CA has to be trusted to not create false certificates.

10.3 Public-Key Infrastructures (PKI)

Definition 10.2. Public-Key Infrastructures (PKI): A collection of technologies and processes for managing public keys, their corresponding private keys, and their use by applications.

Some components of a PKI:

- Certificate format.
- The certification process.
- Certificate revocation.
- Trust models.
- Certificate distribution.
- **Certificate policy**: Details of intended use and scope of a particular certificate.
- **Certification practices statement (CPS)**: Practices and policies followed by a CA.

Although conceptually very simple, there are many practical problems that are encountered when deploying PKI on a large scale. Many of these problems arise from **business**, **legal**, and **useability** considerations. The problems include:

- **Interoperability** (alleviated by standards and certificate formats).
- **Certificate revocation**. It is estimated that $\gg 500,000$ TLS private keys were compromised due to the Heartbleed bug (discovered in 2014). Lots of certificates needed to be revoked.
- **Liability**
- **Trust models**

10.4 Case Study: TLS

- **SSL** (Secure Sockets Layer) was designed by Netscape.
- **TLS** is an IETF version of SSL.
- **SSL/TLS** is used by web browsers to protect web transactions.
- There are many versions of SSL/TLS
  - SSL 2.0 (1995)
  - SSL 3.0 (1996) [almost identical to SSL 2.0]
  - TLS 1.0 (1999)
  - TLS 1.1 (2006)
  - **TLS 1.2** (2008)
  - **TLS 1.3** (2018)
- Google removed SSL 3.0 from Chrome only in 2014

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.

10.4.1 TLS Handshake Protocol

TLS Handshake Protocol:

1. **Phase 1**: Establish security capabilities. Negotiate protocol version, cryptographic algorithms, security levels, etc.
2. **Phase 2**: Server authentication and key exchange. 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.

The main key establishment schemes are:

1. **RSA key transport**: The shared secret k is selected by the client and encrypted with the server's RSA public key. (Not allowed in TLS 1.3)
2. Elliptic Curve Diffie-Hellman (ECDH):
   - The server selects a one-time EC Diffie-Hellman public key $X = xP$ and signs it with its RSA or ECDSA signature key.
   - The client selects a one-time EC Diffie-Hellman public key $Y = yP$.
   - The session key is $K = KDF(xyP)$.

10.4.2 TLS 1.2 Record Protocol

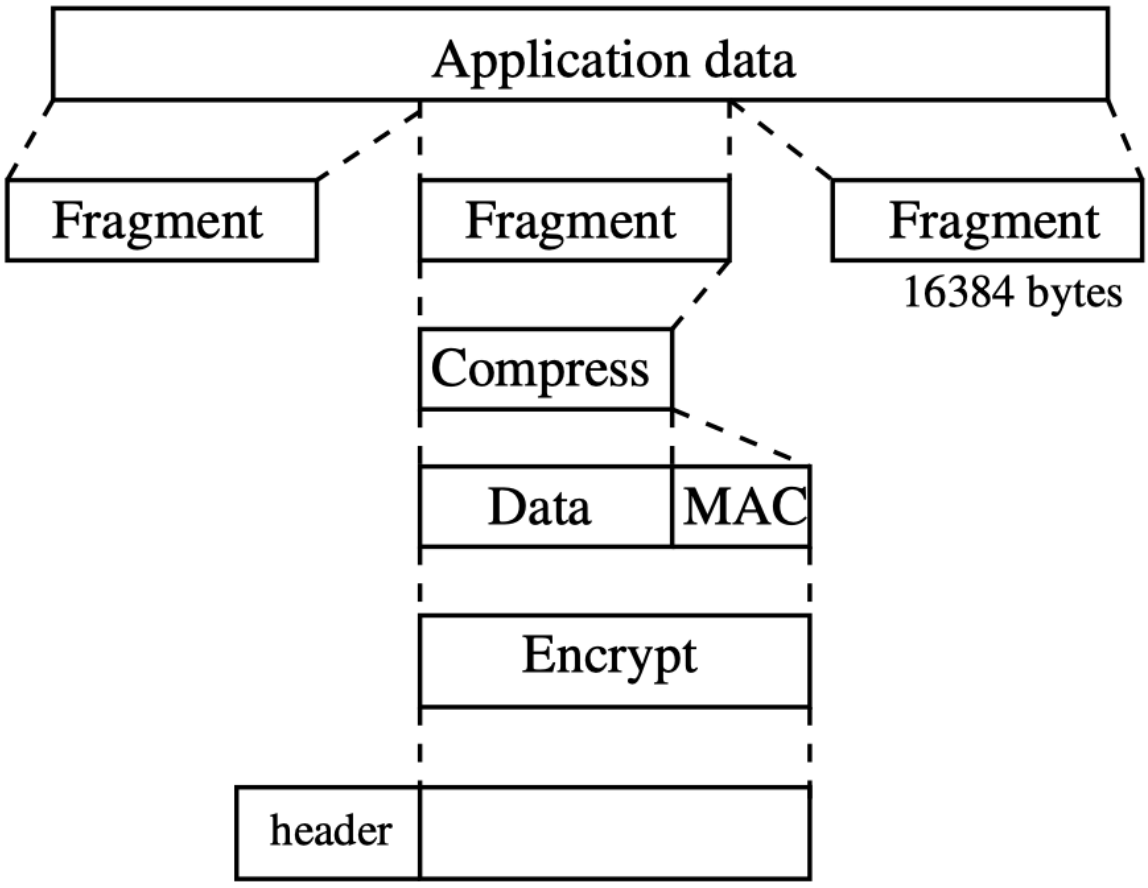


Figure 55: TLS 1.2 Record Protocol

Suppose that client and server share a MAC secret key and a session encryption key.

In summary, TLS 1.2 Record Protocol uses:

- **MAC**: HMAC-SHA-1, HMAC-MD5, HMAC-SHA256, ...
- **Symmetric-key encryption**: RC4, ChaCha20, Triple-DES, AES, ...
- **Authenticated encryption**: MAC-then-encrypt, AES-GCM, ChaCha20 + Poly1305.

10.4.3 TLS 1.3

TLS 1.3, approved in August 2018, was a major overhaul of TLS 1.2. Some of the changes made in TLS 1.3 were:

- Removed RC4, Triple-DES, CBC-mode.
- Removed MAC-then-encrypt.
- Removed MD5 and SHA1.
- Removed RSA key transport.
- Mandates **AES-GCM** (and optionally ChaCha20 + Poly1305).
- All public-key exchanges are based on **ephemeral elliptic curve Diffie-Hellman**.
- Elliptic curves include **P-256**, **Curve25519** and **P-384**.

10.5 Public Key Management in TLS

- **Root CA** keys are pre-installed in browsers.
- Root CAs certify public keys of **Intermediate CAs**.
- **Web servers** get their public keys certified by Intermediate CAs (perhaps for a fee).
  - **DigiCert**'s web server certification business: `https://www.digicert.com/`
  - **Let's Encrypt** `https://letsencrypt.org/` Non-profit CA, that provides free web certificates. Automatic certificate issuance; relies on domain validation.
- **Clients** (users) can obtain their own certificates:
  - However, most users do not have their own certificates.
  - If clients do not have certificates, then authentication is only one-way (the server authenticates itself to the client).

10.6 Example of an X.509 Certificate

Go to `https://www.cibc.com` and click on the padlock.

- Subject name: `Canadian Imperial Bank of Commerce | www.cibc.com`

- Issuer name: DigiCert SHA2 Secure Server CA
- Serial number: `08 E9 7A 69 ... BB`
- Signature algorithm: SHA-256 with RSA Encryption.
- Valid from: Apr 22, 2020
- Valid to: May 13, 2022
- Public Key Info/Algorithm: RSA Encryption
- Subject RSA public key info: 2048-bit n, $e = 65537$
- DigiCert's 2048-bit RSA signature

Let's say **Alice** is the Client (a person using a web browser), and **Bob** is the CIBC's server. Public keys:

1. CIBC's RSA public key: (n_C, e_C), private key: d_C . CIBC's public key is certified by the **intermediate CA DigiCert-S** (full name: "DigiCert SHA2 Secure Server CA").
2. DigiCert-S's RSA public key: (n_S, e_S), private key: d_S . DigiCert-S's public key is certified by the **root CA DigiCert-G** (full name: "DigiCert Global Root CA").
3. DigiCert-G's RSA public key: (n_G, e_G), private key: d_G. DigiCert-G's RSA public key is certified by itself (i.e., it is **self-signed**), and is embedded in all browsers.

When Alice visits `www.cibc.com`, CIBC sends Alice the following:

1. **CIBC's certificate** which contains CIBC's name, RSA public key (n_C, e_C), etc., and DigiCert-S's RSA signature s_1 on this data.
2. **DigiCert-S**'s certificate which contains DigiCert-S's name, RSA public key (n_S, e_S), etc., and DigiCert-G's RSA signature s_2 on this data.
3. **DigiCert-G**'s certificate which contains DigiCert-G's name, RSA public key (n_G, e_G), etc., and DigiCert-G's RSA signature s_3 on this data.
4. A randomly selected point $X(= xP)$ on the elliptic curve P-256, and CIBC's RSA signature s_4 on this point.

Upon receiving the three certificates and (X, s_4), Alice does:

1. **Verify** DigiCert-G's signature s_3 using DigiCert-G's public key (n_G, e_G) (which is embedded in Alice's browser).
2. **Verify** DigiCert-G's signature s_2 using DigiCert-G's public key (n_G, e_G), thereby authenticating DigiCert-S's public key.
3. **Verify** DigiCert-S's signature s_1 using DigiCert-S's public key (n_S, e_S), thereby authenticating CIBC's public key.
4. **Verify** CIBC's signature s_4 using CIBC's public key (n_C, e_C), thereby obtaining an authentic copy of X.

5. Select a random y and send $Y = yP$ to CIBC.
6. Compute the session key $k = KDF(yX)$.

CIBC receives Y and computes the **session key** $k = KDF(xY)$.
Alice and CIBC now share a session key k which is used to encrypt and authenticate (with **AES-GCM**) all communications for the remainder of the session.

10.7 DigiCert

Alice and CIBC are relying on DigiCert to do its job well, which includes:

- Carefully check the identify of the certificate holder (CIBC's name and url), and not issue certificates to imposters (e.g., `https://www.c1bc.com`).
- Carefully guard its private keys d_G and d_S from disclosure.
- Carefully guard misuse of its private keys d_G and d_S by company employees.

To engender confidence and trust in its certification practice, DigiCert publishes its **Certification Practices Statement** (see `http://tinyurl.com/DigiCertCPS2020`) and has its security practices audited by external parties (such as Deloitte or KPMG).

10.8 CA Security

Even though CIBC might have full confidence in DigiCert, there remains the possibility of another Root CA or Intermediate CA issuing fraudulent certificates.

There are several hundred root CA public keys in a browser:

- Mistakenly-issued certificate.
- Maliciously-issued certificate.
- Maliciously-requested certificate.

CAB (CA/Broswer Forum) Voluntary consortium of CAs and browser vendors that prepares and maintains industry guidelines concerning the issuance and management of certificates. `https://cabforum.org/`

Common CA Database (managed by Mozilla) is a repository of information about CAs whose root and intermediate certificates are included in browsers. `https://ccadb.org`

10.8.1 DigiNotar

DigiNotar was a Netherlands-based CA, whose root CA public key was embedded in all browsers.

DigiNotar issued TLS certificates as well as certificates for government agencies in the Netherlands.

In July 2011, DigiNotar issued several hundred fraudulent certificates for domains including `aol.com`, `microsoft.com`, and `*.google.com`.

Shortly after, $\approx 300,000$ gmail users from Iran were redirected to websites that looked like the gmail site.

- The fraudulent web sites would appear to be valid and secured to the gmail users (and the users would establish a secret key with the fraudulent web site using TLS).
- Presumably, the fraudulent web site then captured the gmail userids and passwords of the users.
- See `http://tinyurl.com/SlateDigiNotar`.

10.8.2 Other CA Breaches

It is suspected that the DigiNotar attack was mounted by an individual or organization in Iran, but this has not been proven.

In 2015 Google discovered that the root CA **CNNIC** (China Internet Network Information Center) had issued an intermediate CA certificate to an Egyptian company MCS Holdings, which in turn issued fraudulent certificates for several Google domains. See: **http://tinyurl.com/GoogleCNNIC.**

In 2016, it was discovered that the Chinese root CA **WoSign** had issued fraudulent certificates for several domains including GitHub and Alibaba. See: `http://tinyurl.com/GoogleWoSign`.

In response to the CA breaches, DigiNotar, CNNIC and WoSign's root CA keys were removed from the list of trusted root CA keys in browsers.

Root CA's had to meet certain requirements to qualify for **Extended Validation (EV)** status.

Google's **Certificate Transparency**:

- A certificate logging mechanism to allow anyone to check which certificates a CA has issued.
- Auditors monitor CAs to watch for malicious behaviour.
- Domain name owners monitor the logs to check for certificates issues for their domains.
- `https://www.certificate-transparency.org/`

Chapter 11 Random Bit Generation

Definition 11.1. A **random bit generator (RBG)** is a device that outputs a sequence of independent and unbiased bits.

Some applications of RBGs in cryptography:

- To generate secret cryptographic keys. Examples: secret key k for AES, HMAC and AES-GCM; primes p and q in RSA; secret key in ECDSA.
- To generate per-message secrets k in ECDSA.
- Padding bits for schemes such as RSA-OAEP.

Main security requirement: The random bits should be **unpredictable** to an **active** adversary.

However, how to generate random bits in practice? John von Neumann said that "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

We may consider

- Repeated tossing of a fair coin.
- Elapsed time between emission of particles during radioactive decay.
- Timing between user key strokes or mouse button clicks.
- Samples from I/O buffers.
- Sample bits from a microphone.
- **RDRAND**: instruction for obtaining random numbers from Intel/AMD on-chip hardware random number generators.

The general recommendation is: Sample as many "random" sources as possible, and hash the concatenation of the samples.

`https://www.youtube.com/embed/1cUUfMeOijg`

11.1 Weak Random Bit Generation

Lenstra et al. (2012) `http://eprint.iacr.org/2012/064`. By taking GCDs, 0.27% of publicly available 1024-bit RSA keys n can be factored.

Bernstein et al. (2013) `http://eprint.iacr.org/2013/599`:

- Efficiently factored 184 distinct **1024-bit RSA keys** from Taiwan's national "citizen digital certificate" database.

Figure 56: Cloudflare Random Bit Generation

- These keys were generated by government-issued smart cards that have built-in hardware random number generators.
- 103 of these keys shared primes.
- One prime factor appears 46 different times with different second primes:

  0xC000000000000000000000000000000000000000
  0000000000000000000000000000000000000000
  00000000000000000000000000000000000002F9

  This is the first prime after $2^{511} + 2^{510}$.

- The next common prime, repeated 7 times, is:

  0xC92424922492924992494924492424922492929 9
  2494924492424922492924992494924492424922492
  9249924949244924242249292499249 4924492424E5

- In binary, this prime is:

  1100100100100100001001001001001000100100100
  1001010010010010010011001001001001001010010 0
  1001001000100100100100100001001001001001 00
  0100100100100101001001001001001100100100100

```
1001010010010010010001001001001001000010010
1001001000100100100100101001001001001001100100
1001001001010010010010010001001001001001000010010
1001001001000100100100100101001001001001001100
1001001001001010010010010001001001001001001100
0010010010010010001001001001001010010010010010
1100100100100100101001001001001000100100100100
1000010010011100101
```

Random bit generation can be very slow. So, in practice, a random bit string is used to **seed** a pseudorandom bit generator. HMAC can be used to derive many pseudorandom bit strings $s_1, s_2, s_3, ...$ from a "random" bit string s:

$$s_1 = HMAC_s(1), s_2 = HMAC_s(2), s_3 = HMAC_s(3), ...$$

Chapter 12 FIDO U2F

Definition 12.1. FIDO = **F**ast **ID**entity **O**nline **U2F** = Universal 2nd Factor authentication

- Developed by the FIDO alliance.
- Members include Amazon, Bank of America, Google, Microsoft, Facebook, PayPal, Samsung, VISA, ...
- FIDO mission: Reduce the world's reliance on passwords to better secure the web.
- Some of the many problems with passwords:
  - Selection of weak passwords.
  - Reuse of passwords.
  - Too many passwords to remember.
  - Forgotten passwords.
  - Phishing attacks.
  - Servers have to securely store many passwords.

12.1 U2F Protocol

1. **Authenticator Registration**
   - Alice generates a public-private key pair (for ECDSA with the P-256 elliptic curve).
   - Alice registers her userid, password, and public key with a web service.
2. **User Authentication**
   - Alice visits the web site (via her web browser).
   - She enters her userid and password (which the web server verifies).
   - The web server sends Alice a random challenge r.
   - Alice uses her private key to sign a message m comprising of r, the web server's URL, and the TLS channel ID of the connection (optional); the signature s is transmitted to the sever.
   - The server verifies (m, s) using Alice's public key.

12.2 U2F Notes

- Alice has one public-private key pair for each account.

- All her private keys can be securely stored on a single **security token**.
- The ECDSA signature operations are performed by the security token; the private keys never leave the token.
- Alice can also use different userids for each of her accounts; there is then no linkability between accounts. This makes it easier for Alice to maintain anonymity.
- U2F is phishing-resistant.
- U2F also resists MITM attacks.

12.3 Google's Titan Security Key

- Implements the FIDO U2F standard for two-factor authentication.
- Compatible with popular browsers including Chrome.
- The tokens are built with a tamper-resistant hardware chip that includes firmware engineered by Google. This helps to ensure that the keys haven't been physically tampered with.
- Also built into Pixel 3, 3a, 4, 4a, 5 phones featuring the tamper-resistant Titan M security chip.
- To activate the Titan security key, press a button on a Bluetooth/USB device or tap over NFC.
- This physical "test of user presence" ensures that a signature happens only with the user's consent. It also ensures that malware cannot sign messages when the user is not present.

12.4 FIDO2

- "Passwordless" authentication.
- **F**IDO2 = **CTAP2** + **WebAuthn**
- **CTAP2** = Client-to-Authenticator-Protocol Uses ECDH with P-256.
- **WebAuthn** = W3C's Web Authentication protocol (W3C = World Wide Web Consortium) Uses RSA-PKCS-1-v1.5 and RSA-PSS.
- Passwords (or PINs) are only used in an initialization phase, when a user registers their security device to a browser (rather than to a website).
- **No password use** when a user registers with a website, or authenticates to the website.
- Support in major browsers (Chrome, Firefox, Safari, ...) Native platform support (Windows, Android, iOS, ...)

Chapter 13 The Signal Protocol

13.1 Introduction

The **Signal Protocol** was designed by Moxie Marlinspike and Trevor Perrin. It is free, open source, and is used in:

- Signal (free messaging app).
- Facebook Messenger ("secret conversations" optional feature in the Messenger app on iOS and Android).
- Skype ("private conversations" optional feature).
- WhatsApp.

13.2 WhatsApp

- WhatsApp is owned by Facebook.
- Has over 2 billion users (India, Brazil, Mexico, France, UK, ...), and handles 10's of billions of messages everyday.
- Is banned, or has been temporarily blocked, in several countries.
- Has very low revenues (it's free, works over WiFi, no advertisements, and no user data to mine except metadata).
- In Jan 2019, Facebook announced plans to tightly integrate WhatsApp, Facebook Messenger, and Instagram into a single "privacy-focused platform": `http://tinyurl.com/NYTWhatsApp`
- A new privacy policy will be enforced on May 15, 2021.

13.3 Signal Objectives

Participants: Alice, Bob, WhatsApp, ThirdParty (E, Eavesdropper, could be WhatsApp itself though)

1. **Long-lived sessions**. Alice and Bob establish a long-lived secure communications session. The session lasts until events such as app reinstall or device change.
2. **Asynchronous setting**. Alice can send Bob a secure message even if Bob is offline. Messages can be delayed, delivered out of order, or can be lost entirely without problem.
3. **Fresh session keys**. Each message is encrypted/authenticated with a fresh session key. In particular, we use Encrypt-then-MAC: $c = AES - CBC_{k_1}(m)$, $t = HMAC_{h_1}(c)$, where k_1 and h_1 are each 256-bits.
4. **Immediate decryption**. Bob can decrypt a ciphertext as soon as he receives it.
5. **End-to-end encryption**. WhatsApp and E do not possess any of Alice's or Bob's secret keys, nor do they get access to any plaintext. However, WhatsApp (but not E)

does get all the **metadata**, e.g., who sent a message to whom and when, your contacts, your profile name, etc.

6. **Forward secrecy**. If a party's state is leaked, then none of the previous messages should be compromised (assuming they have been deleted from the state).
7. **Post-compromise security**. Parties recover from a state compromise (if the attacker remains passive).

13.4 Cryptographic Ingredients

1. **AES-CBC**: 128-bit IV, 256-bit key.
2. **HMAC**: with SHA-256, and a 256-bit key.
3. **KDF**: A key derivation function (either HMAC or HKDF, but we will not get into the details).
4. **Curve25519**: See notes here.
5. **Elliptic curve key pairs**: (X, x), $x \in_R [1, n-1]$ is a secret key, $X = xP$ is the corresponding public key.
6. **ECDH**.
7. **EdDSA**: (an ECDSA-like signature scheme).

13.5 Signal Protocol

Three stages:

1. **Registration**
2. **Root key establishment**
3. **Message transmission**

Note that:

- All of Alice's and Bob's message are sent via WhatsApp's servers.
- All communication between Alice/Bob and WhatsApp is encrypted/authenticated using a TLS-like protocol.
- Alice (and Bob) always deletes a secret key as soon as she no longer needs it.

13.5.1 Registration

After Alice has downloaded the WhatsApp app, she sends WhatsApp:

- ID_A: her identifier (cell phone number)
- A: her long-term public key

- U: her medium-term public key
- $Sign_A(U)$: her signature on U
- $S_1, S_2, ..., S_\ell$: one-time public keys
- (and Alice securely stores her secret keys $a, u, s_1, s_2, ..., s_\ell$).

Similarly, Bob sends WhatsApp: $ID_B, B, V, Sign_B(V), T_1, T_2, ..., T_\ell$.

13.5.2 Root Key Establishment

Alice (initiator) wishes to connect with Bob (responder).

1. Alice $\rightarrow$ WhatsApp: **request** to create session with Bob.
2. WhatsApp $\rightarrow$ Alice: $B, V, Sign_B(V), T_1$ (and deletes T_1).
3. Alice does the following:
   (a) **Verify** $(V, Sign_B(V))$ using B.
   (b) Select an **ephemeral key pair** (X_1, x_1).
   (c) Compute **root key** $root_0 = KDF(aV, x_1B, x_1V, x_1T_1)$. ($root_0$ has bitlength 256 bits).

Given A and X_1, Bob can also compute

$$root_0 = KDF(vA, bX_1, vX_1, t_1X_1)$$

13.5.3 Verifying Long-Term Public Keys

QR codes and 60-digit numbers encode identifiers and long-term public keys; (Alice, A) and (Bob, B). Alice and Bob should **verify** these prior to sending each other messages.

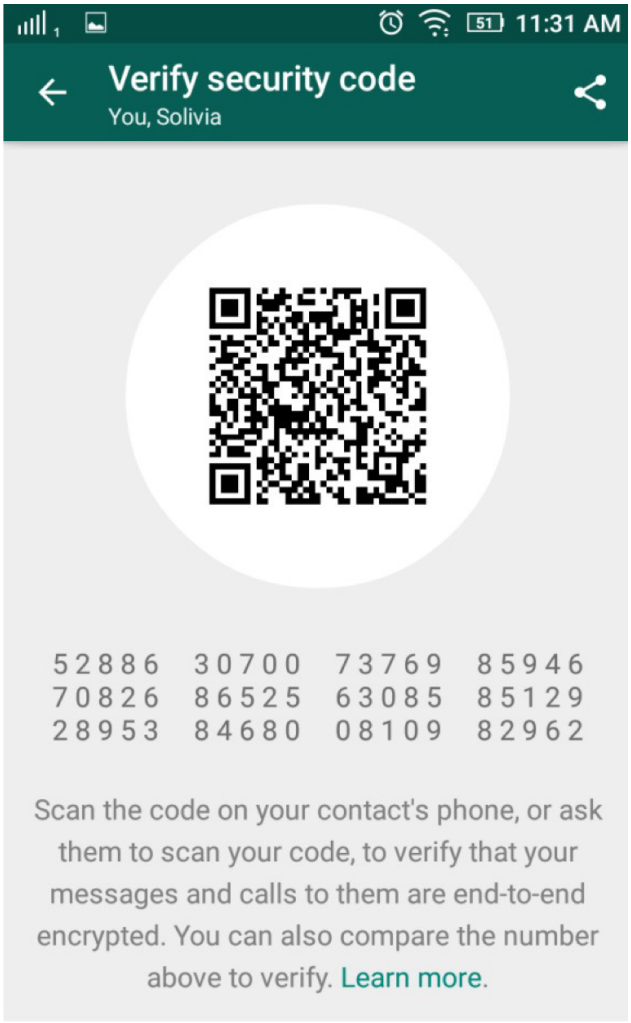


Figure 57: Verifying Long-Term Public Keys using QR code

13.5.4 Forward Secrecy

Suppose that Alice and Bob share a secret key k. They can **ratchet** k and derive message encryption keys $mk_1, mk_2, mk_3, ...$ as follows:

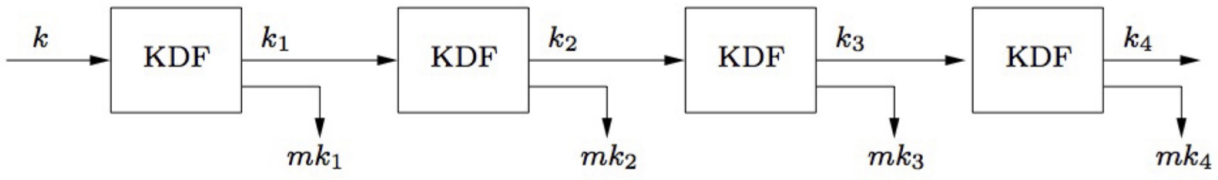


Figure 58: Ratchet in Signal Protocol

- Keys are deleted as soon as they are no longer needed.

- For example, k is deleted as soon as k_1 and mk_1 are computed. Also, mk_1 is deleted as soon as it is used to encrypt (or decrypt) a message
- Suppose that E learns k_2 and mk_2 (by gaining access to Alice's device). Then E can compute $k_3, mk_3, k_4, mk_4, ...$ But E cannot compute mk_1. Thus, ciphertext that was generated using mk_1 cannot be decrypted by E.

13.5.5 Post-Compromise Security

In order to achieve **post-compromise security**, a fresh ECDH shared secret established by Alice and Bob is used each time the KDF is applied.

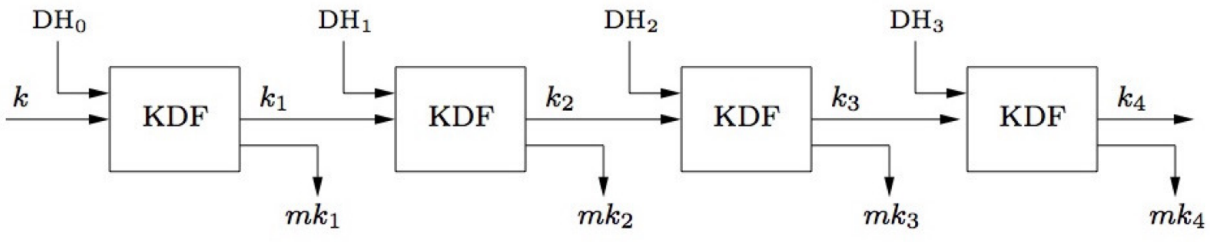


Figure 59: Post Compromise Security in Signal Protocol

Here, $DH_i = ECDH(X_i, Y_i)$, where X_i is contributed by Alice, and Y_i is contributed by Bob. Suppose that E learns k_2 and mk_2. Then E cannot compute k_3, mk_3 unless she also learns x_2 (or y_2).

13.6 Message Transmission

Alice maintains three key chains:

1. A **root key chain** (used to seed the other two chains).
2. A **sending key chain** (to generate message sending keys).
3. A **receiving key chain** (to generate message receiving keys).

Bob also maintains three key chains:

1. A root key chain (the same one as Alice's).
2. A receiving key chain (the same as Alice's sending chain).
3. A sending key chain (the same as Alice's receiving chain).

Consider the following example:

1. Alice $\rightarrow$ Bob: $M_{11}^1, M_{11}^2, M_{11}^3, M_{11}^4$. (Alice's first sending chain of 4 messages)
2. Alice $\leftarrow$ Bob: M_{12}^1, M_{12}^2. (Alice's first receiving chain of 2 messages)

3. Alice → Bob: M_{22}^1. (Alice's second sending chain of 1 messages)
4. Alice ← Bob: $M_{23}^1, M_{23}^2, M_{23}^3$ (Alice's second receiving chain of 3 messages)

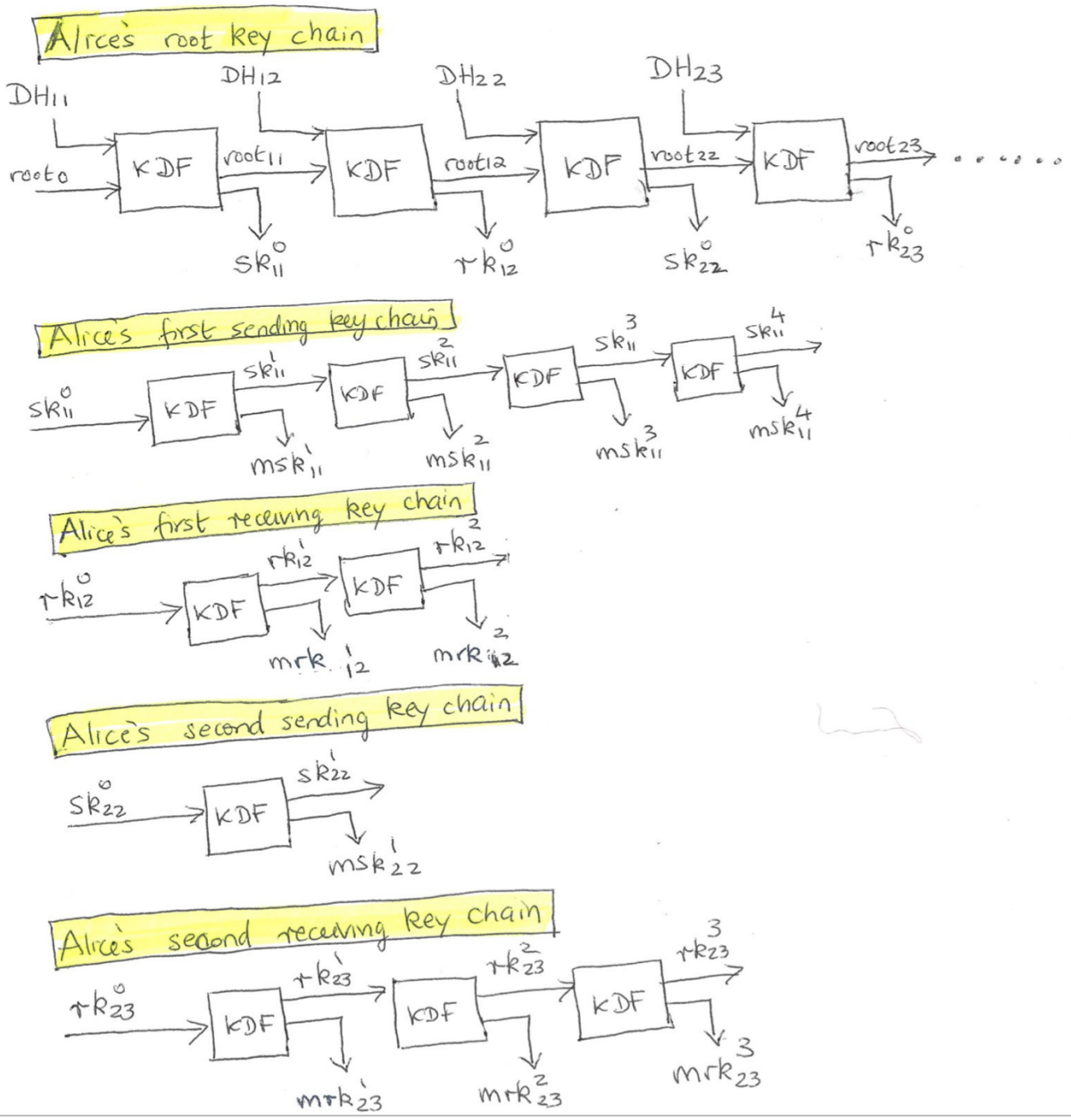


Figure 60: Example of Message Transmission in Signal Protocol

It works as follows:

Alice's root key chain	Alice's sending key chains	Alice's receiving key chains
1.$DH_{11} = ECDH(X_1, Y_1), Y_1 = V$ 2.$KDF(root_0, DH_{11}) \to root_{11}, sk_{11}^0$	3.$KDF(sk_{11}^0) \to sk_{11}^1, msk_{11}^1$ 4.$KDF(sk_{11}^1) \to sk_{11}^2, msk_{11}^2$ 5.$KDF(sk_{11}^2) \to sk_{11}^3, msk_{11}^3$ 6.$KDF(sk_{11}^3) \to sk_{11}^4, msk_{11}^4$	
7.Receive Y_2 8.Compute $DH_{12} = ECDH(X_1, Y_2)$ 9.$KDF(root_{11}, DH_{12}) \to root_{12}, rk_{12}^0$		10.$KDF(rk_{12}^0) \to rk_{12}^1, mrk_{12}^1$ 11.$KDF(rk_{12}^1) \to rk_{12}^2, mrk_{12}^2$
12.Select X_2 13.Compute$DH_{22} = ECDH(X_2, Y_2)$ 14.$KDF(root_{12}, DH_{22}) \to root_{22}, sk_{22}^0$	15.$KDF(sk_{22}^0) \to sk_{22}^1, msk_{22}^1$	
16.Receive Y_3 17.Compute $DH_{23} = ECDH(X_2, Y_3)$ 18.$KDF(root_{22}, DH_{23}) \to root_{23}, rk_{23}^0$		19.$KDF(rk_{23}^0) \to rk_{23}^1, mrk_{23}^1$ 20.$KDF(rk_{23}^1) \to rk_{23}^2, mrk_{23}^2$ 21.$KDF(rk_{23}^2) \to rk_{23}^3, mrk_{23}^3$

Notation:

- sk = chaining key for sending key chain.
- rk = chaining key for receiving key chain.
- msk = message sending key.
- mrk = message receiving key.

Alice sending M_{ii}^j is a process like follows:

$$C_{ii}^j = AE_{msk_{ii}^j}((A, B, X_i, j, L_{i-1}), M_{ii}^j)$$

where L_{i-1} is the length of Alice's $(i-1)$th sending chain.

Here,

$$AE_k(T, M) = AES - CBC_{IV,k_1}(M), HMAC_{k_2}(AES - CBC_{IV,k_1}(M), T), T$$

where $k = (IV, k_1, k_2)$ with $IV \in \{0,1\}^{128}, k_1, k_2 \in \{0,1\}^{256}$

Chapter 14 Post-Quantum Cryptography (PQC)

14.1 Quantum Computers

- Conceived by Yuri Manin (1980) and Richard Feynman (1981), **quantum computers** are devices that use quantum-mechanical phenomena such as **superposition, interference, and entanglement** to perform operations on data.
- A **qubit** is the quantum analogue of a classical bit, and can be in two states at the same time, each with a certain probability.
- An n**-qubit register** can be in 2^n states at the same time, each with a certain probability.
- When a function f is applied to an n-qubit register, it is simultaneously evaluated at all 2^n states.
- However, when the n-qubit register is measured, it reverts to being in one of the 2^n states according to its underlying probability distribution. So, quantum computers are **not** "massively parallel machines".

14.2 The Threat of Quantum Computers

14.2.1 The Threat of Quantum Computers - Shor

The public-key systems used in practice are:

- **RSA**: Security is based on the hardness of integer factorization.
- **DL**: Security is based on the hardness of the discrete logarithm problem.
- **ECC**: Security is based on the hardness of the elliptic curve discrete logarithm problem.

Shor's Algorithm: In 1994, Peter Shor discovered a very efficient (polytime) quantum algorithm for solving these three problems.

So, all RSA, DL and ECC implementations can be **totally broken** by quantum computers.

14.2.2 The Threat of Quantum Computers - Grover

Let $F : \{0,1\}^n \to \{0,1\}$ be a function such that:

1. F is efficiently computable; and
2. $F(x) = 1$ for exactly p inputs $x \in \{0,1\}^n$.

Definition 14.1. Grover's Algorithm (1996) is a quantum algorithm for finding an $x \in \{0,1\}^n$ with $F(x) = 1$ in $\dfrac{2^{n/2}}{p^{1/2}}$ evaluations of F.

Exhaustive key search: Consider AES with an ℓ-bit key. Suppose we have t known plaintext-ciphertext pairs (m_i, c_i), where t is such that the expected number of false keys is very close to 0.

Define $F : \{0,1\}^\ell \to \{0,1\}$ by

$$F(k) = \begin{cases} 1 & \text{if } AES_k(m_i) = c_i \text{ for all } 1 \le i \le t \\ 0 & \text{otherwise} \end{cases}$$

Then Gover's algorithm (with $p = 1$) can find the secret key in $2^{\ell/2}$ operations. Thus, 256-bit AES keys should be used in order to achieve a 128-bit security against quantum attacks.

14.2.3 The Threat of Shor and Grover

What does this mean for internet security?

- Automatic software updates
- TLS

Should we care?

- The NSA and other organizations are capturing and storing large amounts of internet traffic right now.

What, if anything, should we do to mitigate the threat?

When should we take action?

- Now?
- In 5 years?
- In 10 years?
- In 15 years?

14.3 When will Quantum Computers be Built?

- 1998: (Jones and Mosca) 2-qubit quantum computer
- 2000: 7 qubits
- 2011: 14 qubits
- 2017: 50 qubits (IBM) `http://tinyurl.com/IBMqc50`
- 2019: 53 qubits (Google)

The largest number factored using Shor's algorithm on a quantum computer is $21 = 3 \times 7$.

Note: Larger numbers have been factored on **adiabatic quantum computers** (e.g., DWAVE computers), but these implementations do not scale.

14.4 Fault-Tolerant Quantum Computers?

- A quantum computer that can factor a 2048-bit RSA modulus using Shor's algorithm will need (at least) **2048-qubit** registers.
- These qubits will have to be **fault tolerant**, i.e., be error resistant.
- **The physical qubits that have been built so far are not fault tolerant**.
- So, the plan is to use **quantum error correction** to combine many (imperfect) physical qubits into one (almost perfect) **logical qubit**.
- Optimistic estimates are that thousands of physical qubits will be needed to build one logical qubit.
- So, factoring 2048-bit RSA moduli might need **millions of physical qubits**.

In Oct 2019, Google announced that they had achieved **quantum supremacy**:

- `http://tinyurl.com/GoogleQC53`
- Perform **some** task (not necessarily useful, but not too contrived) on a quantum computer much faster than is possible on classical computers.
- Google's task was to apply a random quantum circuit to the all-0 53-qubit initial state; it did this in **3 minutes**.
- Google estimated that a classical supercomputer would **take 10,000 years** to perform the same task.
- IBM researchers lowered that estimate to **2.5 days** using a different classical technique.
- It is important to note that Google's 53-qubit quantum computer is **not fault tolerant**.
- So, while it is major scientific achievement, **it does not in any way threaten the security of presently-deployed cryptosystems**.
- The next major milestone is to build a single **logical qubit**.
- It's still too early to be able to predict when scalable quantum computers will be built. "It is highly unexpected that a quantum computer that can factor 2048-bit RSA numbers will be built within the next decade."
- On the other hand, there is no fundamental reason why a large, fault-tolerant quntum computer cannot be built.

14.5 Long-Term Security

According to the **National Security Agency**: "Algorithms often require **20 years** to be fully deployed on National Security Systems (NSS). NSS equipment is often used for **30 years or more**. National security information intelligence value is often **30 years (sometimes more)**, although it may vary depending on classification, sensitivity, and subject."

Recommendation: Information security systems should be **crypto-agile**. In other words, they should be designed so that new cryptographic primitives can be employed without significant change to system infrastructure.

In 2015, NSA announced that: **IAD will initiate a transition to quantum resistant algorithms in the not too distant future.** Based on experience in deploying Suite B, we have determined to start planning and communicating early about the upcoming transition to quantum resistant algorithms. **Our ultimate goal is to provide cost effective security against a potential quantum computer.** We are working with partners across the USG, vendors, and standards bodies to ensure there is a clear plan for getting a new suite of algorithms that are developed in an open and transparent manner that **will form the foundation of our next Suite of cryptographic algorithms**.

Until this new suite is developed and products are available implementing the quantum resistant suite, we will rely on current algorithms. For those partners and vendors that have not yet made the transition to Suite B elliptic curve algorithms, we recommend not making a significant expenditure to do so at this point but instead to **prepare for the upcoming quantum resistant algorithm transition**.

Source of the above: "A riddle wrapped in an enigma; `https://eprint.iacr.org/2015/1018.pdf`

14.6 PQC Standardization

- U.S. National Institute of Standards and Technology (**NIST**) `http://tinyurl.com/pqc-nist`
- Solicited proposals for quantum-resistant signature and encryption (key encapsulation) algorithms.
- Nov 30, 2017: **69 submissions** in Round 1.
- Jan 30, 2019: **26 submissions** selected for Round 2.
- Jul 22, 2020: **7+8 submissions** selected for Round 3.
- Evaluation and standardization are expected to take 2+ years

14.7 Quantum-Safe Candidates

- (Classical) symmetric-key cryptography
- Hash-based signatures
- Code-based public-key encryption. Error-correcting codes
- Lattice-based public-key encryption and signatures
- Multivariate polynomials signatures
- Isogeny-based key agreement. SIKE: elliptic curve Supersingular Isogeny-based Key Encapsulation
- Quantum key distribution

14.8 Large-scale experiments

1. **Google's ongoing experiment** with quantum-safe cryptography in its Chrome browser: ECDH + Lattice-based key agreement. See `www.imperialviolet.org/2018/12/12/cecpq2.html`
2. **Cloudflare's ongoing experiment** with quantum-safe cryptography in TLS 1.3: ECDH + Lattice-based key agreement and ECDH + SIKE. See `tinyurl.com/CloudflareTLS`
3. **Amazon's AWS Key Management Service** now supports hybrid post-quantum TLS: See `tinyurl.com/AmazonPQC`

14.9 Commercialization

- ISARA (Waterloo): `https://www.isara.com`
- evolutionQ (Waterloo): `https://www.evolutionq.com`
- Crypto4A (Ottawa): `https://www.crypto4a.com`
- Infosec Global (Toronto): `https://www.infosecglobal.com`
- ID Quantique (Switzerland): `https://www.idquantique.com`
- Post-Quantum (UK): `https://post-quantum.com`
- PQShield (UK): `https://pqshield.com`
- CryptoNext (France): `https://cryptonext-security.com`

Chapter 15 BitCoin

15.1 Paper Cash

- **Coins** (including **paper bills**) are issued by the Bank of Canada in accordance with an economic policy.
- Suppose that Alice wishes to give a coin to Bob (in return for some goods or services).
- Bob can examine the coin to ensure that it is **valid** (i.e., not a **forgery**).
- **Double spending** is not a concern because Alice cannot give the same (valid) coin to two different parties.
- **Payer anonymity** (during payment) and **payment untraceability** (after payment) are facilitated.

Features of Paper Cash

- **Recognizable** (as legal tender)
- **Portable** (easily carried)
- **Transferable** (without involvement of the financial network)
- **Divisible** (has the ability to make change)
- **Unforgeable** (difficult to duplicate)
- **Anonymous** (no record of who spent the money)
- **Untraceable** (difficult to keep a record of where money is spent)

Note: Many of these features are not available with credit card payments.

15.2 Bitcoin Basics

- An electronic cash scheme invented by **Satoshi Nakamoto** (a pseudonym) in 2008.
- Bitcoin is **decentralized**, i.e., no central authority such as a "Bank" or "Government" is required.
  - Q: Who creates coins? Anyone can create it.
  - Q: How can the creation of coins be regulated? We will see.
  - Q: How does the recipient of a coin ensure it has not been previously spent?
- Anyone can use Bitcoin:
  - Download a **wallet** from `http://bitcoin.org`.
  - Obtain bitcoins by "**mining**" or from an exchange such as **Coinbase** or **kraken**.
- Note: Payer anonymity and payment untraceability are **not** primary goals of Bitcoin.

- The first bitcoins were generated by Satoshi Nakamoto on **Jan 3 2009**.
- The basic unit of bitcoin currency is 1 **BTC**.
- Each BTC can be divided into 100 million pieces, the smallest of which, i.e., **0.00000001 BTC**, is a **satoshi**.
- Bitcoins can be generated (i.e., **mined**) by anyone.
- They are generated at the rate of R BTC every 10 minutes (on average).
- Initially, $R = 50$. On Nov 28 2012, R was lowered to 25. On Jul 9 2016, R was lowered to 12.5. On May 11 2020, R was lowered to 6.25.
- R will be halved over time, until the year **2140**, when a total of **21 million BTC** will have been generated.
- As of March 25 2021, around **18.5 million BTC** have been generated.

15.2.1 Value of a Bitcoin

The US dollar value of 1 BTC has fluctuated wildly:

May 22 2010	\$0.0025	Jan 1 2014	\$747.56
Jul 17 2010	\$0.08	Jan 3 2015	\$289.86
Jan 1 2011	\$0.30	Jan 2 2016	\$433.23
Feb 9 2011	\$1.00	Mar 29 2017	\$1183.65
Jun 8 2011	\$31.91	Dec 17 2017	\$19205.11
Jan 1 2013	\$13.30	Apr 2 2018	\$7083.80
Apr 9 2013	\$223.10	Mar 20 2019	\$4087.11
Jul 6 2013	\$69.31	Apr 7 2020	\$7366.26
Nov 30 2013	\$1128.82	Mar 25 2021	\$51955

15.2.2 Why Use Bitcoin?

1. It's **decentralized**.
   - Not under the control of any government.
   - Not under the control of any bank, credit card company, or other financial institution.
   - Anyone can use it (even if you don't have a credit history).
   - It's (relatively) easy to use.
2. Bitcoin's monetary policy is **fixed** and **public**.
3. Transactions are **irreversible**.
4. Transaction **fees are low** (even across borders).
5. Transactions can be **anonymous** (with a bit of care).

Of course, there are many reasons **not** to use Bitcoin. We will not dwell on those for now....

15.2.3 Distributed Ledger

The most important contribution of Bitcoin was the implementation of a **public decentralized storage** system (also known as a **distributed ledger**) in the form of a **blockchain**:

A sequence of data organized in blocks with the following properties:

- **public**: readable by everyone.
- **consensus**: all participants agree on the data.
- **liveliness**: writeable by everyone.
- **persistence**: unchangeable by anyone.

15.3 Elements of Bitcoin

1. **Transaction**: The transferring of a coin from one user to another. All transactions are public and are broadcast to all users.
2. **Peer-to-peer network**: The users of Bitcoin are organized in a peer-to-peer network.
3. **Blocks**: Every 10 minutes or so, the latest transactions are verified and collected in a block. This block is hashed and (cryptographically) linked with previous blocks. The block is broadcast to the peer-to-peer network.
4. **Blockchain**: The list of blocks is called the blockchain. It contains a record of **all** past transactions.
5. **Mining**: The process of verifying transactions and compiling a block is called mining. A successful miner receives a **reward** (new bitcoins).
6. **Proof-of-work**: To successfully compile a block and receive a reward, the miner has to solve a cryptographic challenge.

Main Cryptographic Ingredients:

1. **SHA-256** hash function.
2. **ECDSA** with the **secp256k1** elliptic curve:

$$E : Y^2 = X^3 + 7 \text{over} \mathbb{Z}_p$$

where

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$

$n = \#E(\mathbb{Z}_p)$ is a 256-bit prime, $P \neq \infty$ is a fixed point in $E(\mathbb{Z}_p)$.

Key Pairs (for ECDSA):

- Each user selects $a \in_R [1, n-1]$ and computes the elliptic curve point $A = aP$.

- The user's ECDSA **private** key is a; the user's ECDSA public key is A.
- We will denote Alice's key pair by (a, A), Bob's key pair by (b, B), Chris's key pair by (c, C), etc.
- In Bitcoin, a user's public key A serves as a **pseudonym** for the user Alice.
- More generally, a user can select a **different key pair** for each transaction.
- See `http://bitaddress.org` for a JavaScript client-side key pair generator.

The first bitcoins were generated by Satoshi Nakamoto on Jan 3 2009.Here, S is Satoshi Nakamoto's public key.

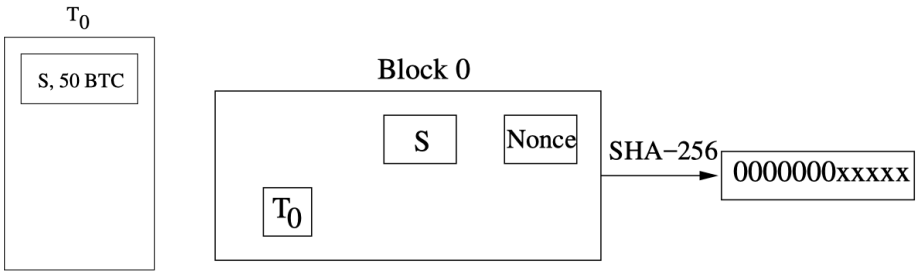


Figure 61: The First BitCoins and The Genesis Block

The genesis block is created by Satoshi Nakamoto (S) on Jan 3 2009. Block 0 is embedded in the Bitcoin software.

15.4 Transactions

- A **transaction** is the transfer of a coin from one user to another user.
- Suppose that Alice has a coin, say of value 1 BTC.
- The transaction in which Alice obtained this bitcoin is represented by the string T_{XA}.
- Suppose Alice wishes to give this coin to Bob. The transaction is represented as follows:

$$T_{AB} = \{\widetilde{T_{XA}}, A, B, 1BTC\}_A$$

  where $\widetilde{m}$ denotes the hash of m, and $\{M\}_A$ denotes a message M and its ECDSA signature with respect to the public key A.
- This transaction is **broadcast** to the entire network.
- Bob (and anyone else) can verify the authenticity of the ECDSA signature using the public key A

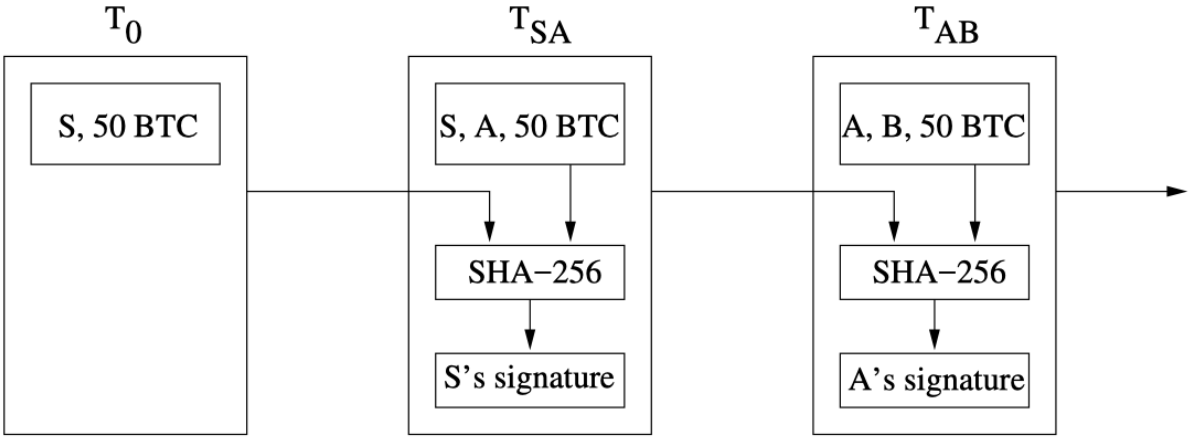


Figure 62: Transaction Chain for the First Bitcoins

15.4.1 Transaction Chain for the First Bitcoins

Note: The transaction chain contains Alice's and Bob's public keys, but **not** their names.

Questions: How can the recipient verify that a coin has not been **double-spent**? (without using a trusted central authority.) How are the other bitcoins generated?

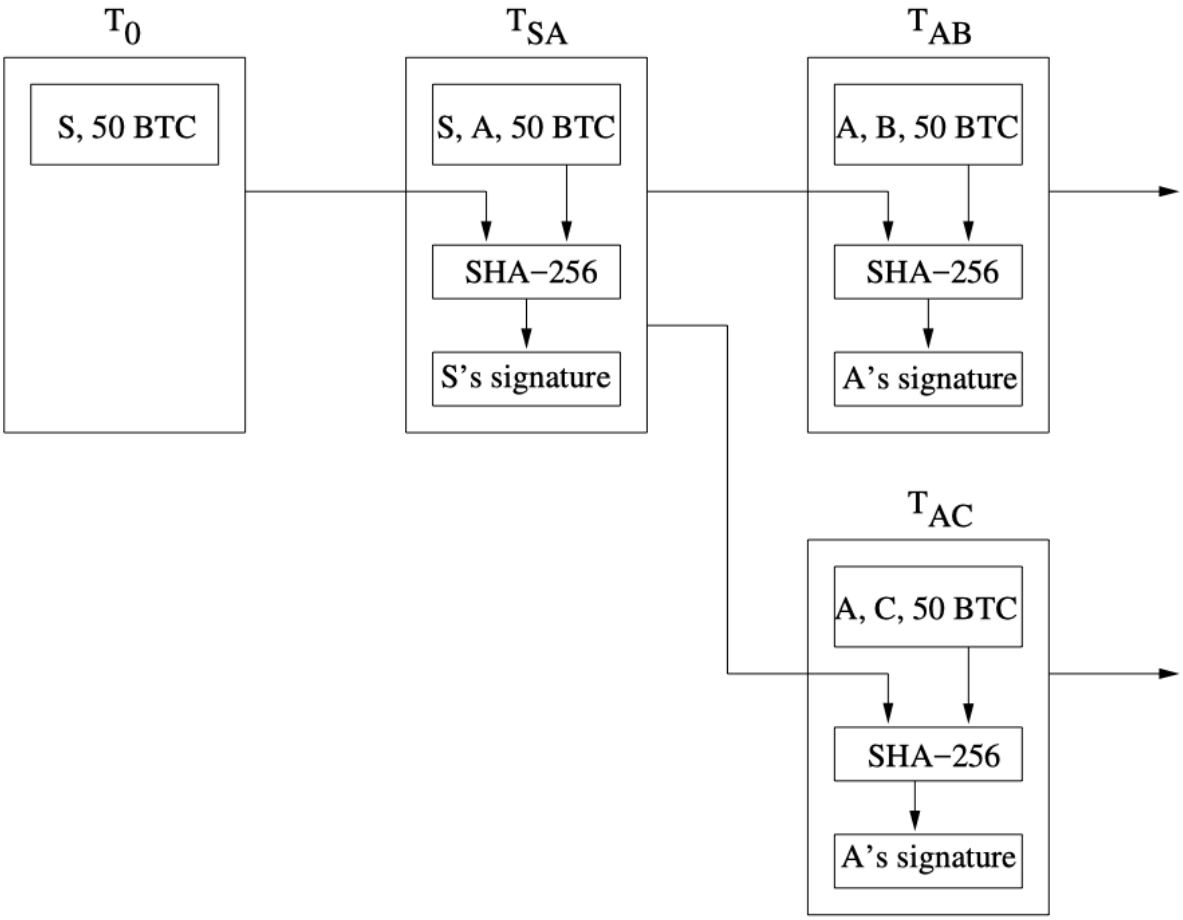


Figure 63: Transaction Chain for the First Bitcoins Continued

Only one of the transactions T_{AB}, T_{AC} should be accepted as valid; the other transaction should be rejected.

15.4.2 Proof-of-Work

- Recall that all transactions are broadcast to all users.
- Any user (called a **miner**) with public key W can volunteer to collect all transactions $T_1, T_2, ..., T_n$that it received in an interval of time, say the previous 10 minutes.
- The user W verifies that these transactions are valid and that the corresponding coins have not been previously spent.
- The user forms a **block** consisting of the hash of the previous block, the user's public key W, a **nonce**, and $T_1, ..., T_n$.
- The nonce is incremented until a hash value that begins with t zeros is obtained.

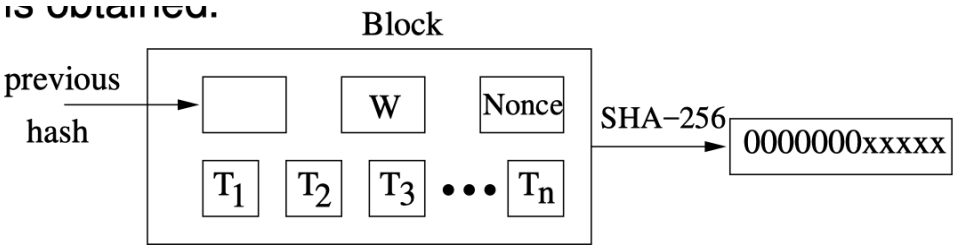


Figure 64: BitCoin Proof of Work

- The block is broadcast to the network.

15.5 The Blockchain

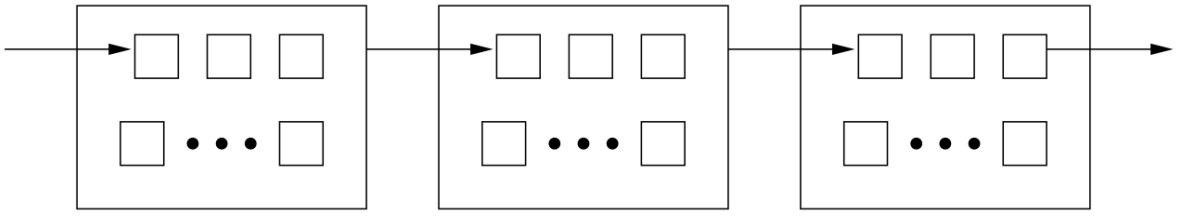


Figure 65: The Blockchain

- Users will accept a block if all the transactions in it are valid, if the coins have not been previously spent, and if the hash value begins with t zeros.
- Users show their acceptance of the block by using its hash as the "previous hash" for the next block, thereby growing the blockchain.
- The blockchain serves as a **public decentralized ledger** that records all transactions.

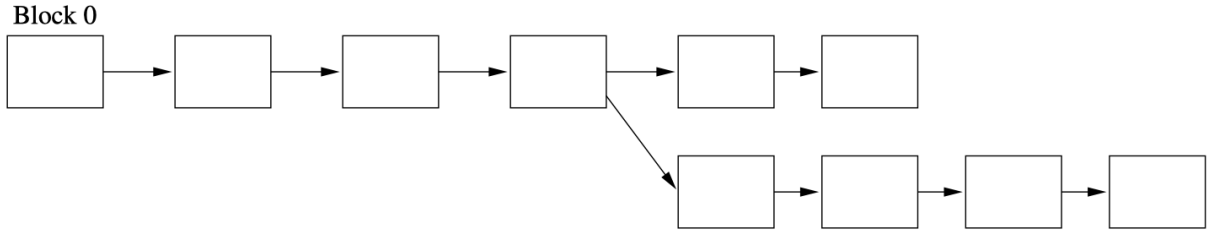


Figure 66: Forks in the Blockchain

- There is a possibility that two blocks are created around the same time by two different users.
- This causes a **fork** in the blockchain.
- To remedy the fork, **users will trust the longest chain** and continue to grow that chain. More precisely, users will trust the chain that that was most difficult to generate.
- The blocks that are not part of the longest chain are dropped and the valid transactions in them are returned to the miners' memory pool of unverified transactions.

15.5.1 Mining

- **Incentive**: The block creator (W) is awarded R BTC (currently, $R = 6.25$) [**mining**].
- **Work factor**: The target t is updated every 2016 blocks (2 weeks) to ensure that the average time it takes to generate a block is 10 minutes.
- Currently, the bitcoin network is generating hashes at the rate of approximately $2^{67.1}$ per second. The hash difficulty is approximately $t = 77$.
- A PC can do approximately 2^{22} hashes per second. So, one PC will take about 700,000,000 years to generate one block.
- **Mining pools**: Users form mining pools and share an award.

15.5.2 Security Notes

- If A gives B a coin, then B should complete the transaction with A only after the transaction T_{AB} appears in the blockchain, perhaps followed by several more blocks (e.g., six blocks):
  - Transactions are **not** instantaneous.
  - If the transaction is accepted instantaneously, B has to accept the risk that A might double spend the coin.
- Since all transactions are public, **payer anonymity** and **payment untraceability** are not guaranteed.
- Bitcoin is "secure" as long as honest users collectively control more CPU power than any cooperating group of users.

15.5.3 Transactions with Multiple Inputs/Outputs

Bitcoins can be combined and split:

- Suppose that Alice (A):
  - received 25 BTC from Bob (B) in Transaction T_{BA}
  - received 20 BTC from Chris (C) in Transaction T_{CA}

- Suppose that Alice wishes to:
  - give 30 BTC to David (D)
  - leave 14 BTC to herself as change
  - give 1 BTC as a **transaction fee**.
- Here is the corresponding transaction

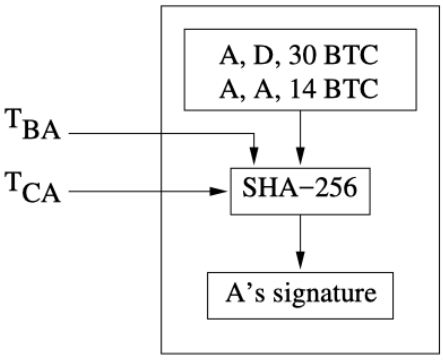


Figure 67: Transactions with Multiple Inputs/Outputs

- The transaction fee is claimed by the miner who validates this transaction.

Now, suppose that Bob owns two public keys, B and H. Suppose that Bob received 9 BTC in transactions T_1 and T_2:

- T_1: 1 BTC from A to B, 3 BTC from D to E, 2 BTC from C to H
- T_2: 5 BTC from C to F, 6 BTC from G to B

Suppose Bob wishes to give 2.5 BTC to F, 3 BTC to I, 1.5 BTC as change to B, 1.75 BTC as change to H, and offer a transaction fee of 0.25 BTC. He forms the transaction T_3:

- Inputs are: $(\widetilde{T}_1, 1)$, $(\widetilde{T}_1, 3)$, $(\widetilde{T}_2, 2)$
- Outputs are: $(F, 2.5), (I, 3.0), (B, 1.5), (H, 1.75)$.
- T_3 has 3 signatures, with public keys B, H, B.

15.5.4 Protecting Your Bitcoins

- If an attacker obtains a copy of Alice's wallet (and thus her private key a), then the attacker can spend the coins associated to A. If Alice deletes (or loses) her private key a, then all her coins corresponding to A are lost forever.
- Alice could store her bitcoins at an **exchange**.

  - **Mt. Gox**, a Bitcoin exchange based in Tokyo, "lost" 850,000 BTC and declared bankruptcy in February 2014. Later, it "found" 200,000 BTC.
  - In early December 2018, the founder of Canadian cryptocurrency exchange **QuadrigaCX** died. Apparently he was the only person who knew the password to access $190 million worth of QuadrigaCX's customer cryptocurrencies (including bitcoin). See `http//tinyurl.com/WPQuadrigaCX`

- Alice could use a **hardware wallet** (e.g. Ledger) or a **cosigner** service (e.g. BitGo).

15.5.5 Miscellaneous Notes

- **Scalability**: The maximum size of a block limits the number of transaction to about 7 per second. In contrast, VISA processes about 2,000 transactions per second.
- **Mining costs**: Mining requires **hardware** and **electricity**. Energy costs per transaction: $\approx$ Cdn $25.
- **Proof-of-stake**
  - Miners put up collateral (stake) and are chosen based on the size of their stake.
  - Misbehaving miners are penalized (e.g. by slashing their stake).
  - Examples: Algorand, Cardano, Tendermint, EOS, Casper.
- Several technical details have been omitted including:
  - A public key is identified by its 160-bit hash value.
  - **Merkle trees** are used to minimize the size of a block.
  - Simplified payment verification (**SPV**).

15.5.6 The Future of Bitcoin

Today, **Bitcoin is rarely used for (legal) transactional purposes**. It is mostly used:

- illegal transactions (moving funds out of a country, illegal purchases on the internet, ransomware, etc.);
- as a mechanism for storing value in countries that are experiencing hyperinflation (easier to store and use than gold or diamonds);
- as a (highly) speculative investment.

The future of Bitcoin is uncertain, and many challenges remain: technological, economic, government regulations.

Nonetheless, Bitcoin has been a remarkably successful proof-of-concept, and there is now a very high rate of innovation in the cryptocurrency/blockchain space.

15.5.7 Exploring Bitcoin

- Bitcoin magazine: `http://bitcoinmagazine.com`
- Download a wallet: `http://bitcoin.org`
- Live blockchain: `http://blockchain.info`
- Bitcoin Blocks: `http://explorer.btc.com`
- Genesis Block: `http://tinyurl.com/BTCBlock0`
- Block 1: `http://tinyurl.com/BTCBlock1`
- Block 100,000: `http://tinyurl.com/BTCBlock100000`

15.6 Ethereum

- Invented by **Vitalik Buterin** in 2013. `http//tinyurl.com/NYTButerin`
- **Blockchain-based decentralized computing platform**
- Underlying cryptocurrency is called **ether**.
- Supports a Turing-complete programming language.
- Permits full **smart contract** functionality.
- Potential **Decentralized Applications (DAPPs)** include:
  - Timestamping and notarization of documents.
  - Record asset ownership (domain names, stocks, student transcripts, home ownership, etc.).
  - Contract signing (without a "trusted" lawyer).
  - Crowdfunding (a.k.a. Kickstarter).
  - Democratic autonomous organizations (DAOs).

15.6.1 Enterprise Ethereum Alliance (EEA)

- EEA members include: Microsoft, Intel, Cisco, J.P.Morgan, Credit Suisse, PriceWaterhouseCoopers, Ernst & Young.
- Feb 28, 2017: "Together, we will learn from and build upon the only smart contract supporting blockchain currently running in real-world production - Ethereum - to define enterprise-grade software capable of handling the most complex, highly demanding applications at the speed of business."
- `https://entethalliance.org/`

15.6.2 Selected Topics for Further Study

- **Zcash**: Anonymous cryptocurrency.
- **Algorand**: Proof-of-stake blockchain platform.
- **Diem**: Facebook's proposed cryptocurrency.
- **Filecoin**: Cryptocurrency-incentivized decentralized and verifiable file storage network.

Chapter 16 Wrap up

This course has been about **BORING CRYPTO**: Cryptography that is well studied, widely standardized, and widely deployed.

MAIN PROBLEM: Alice and Bob wish to communicate securely.

- They encrypt/authenticate data using: **AES-GCM** or ChaCha20/Poly1305.
- They establish a shared secret key using: ECDH (or RSA key transport).
- The public keys need to be authenticated:
  - **Certificates** issued by a **CA**. (e.g. TLS).
  - Preinstallation of a public key (e.g. in an operating system).
  - Visual inspecting each other's public keys (e.g. Bluetooth, WhatsApp).

So is Cryptography a Solved Problem? No! A lot of challenges remain:

- The possibility of cryptanalytic advances
- The threat of quantum computers
- Efficient implementation
- Secure implementation
- Key management
- The insecurity of the Internet of Things (IoT)
- Balancing the privacy rights of individuals and the needs of law enforcement

16.1 Cool Cryptography

Cool Crypto includes:

- Lightweight cryptography
- **Quantum-safe cryptography**
  - Lattice-based cryptography
  - Isogeny-based cryptography
- Cryptocurrencies
- **Differential privacy** (e.g. Google's RAPPOR)
- Multi-party computation
- **Computing with encrypted data**
  - Fully homomorphic encryption

  - Encrypted databases
  - Privacy-preserving machine learning

16.2 Further Study

Other undergraduate courses in crypto/security at UW:

- **CO 485**: Mathematics of Public-Key Cryptography (Fall)
- **CS 458**: Computer Security and Privacy (Fall, Winter, Spring)
- **CO 481/CS 467**: Intro to Quantum Computing (Winter)
- **ECE 409**: Cryptography and System Security (Winter)
- **ECE 458**: Computer Security (Spring)

16.3 Final Exam

- April 22 (4:00 pm) to April 23 (4:00 pm)
- Two components:
  - Quiz (on LEARN)
  - Written answers (on Crowdmark)
- Exam preparation:
  - Review the slides
  - Redo the assignment questions
  - Try some of the practice problems
  - Come to office hours