CO 487 with Douglas Stebila

Eason Li

2025 F

Contents

1 Introduction

Lecture 1 - Wednesday, September 03

Cryptography is about securing communications in the presence of malicious adversaries. The fundamental goals of Cryptography are:

- Confidentiality: Keeping data secret from all but those authorized to see it.
- Integrity: Ensuring data has not been altered by unauthorized means.
- Authentication: Corroborating the source of data or identity of an entity.
- Non-repudiation: Preventing an entity from denying previous commitments or actions.

There are three states information can be in:

- Data at rest
- Data at transit
- Data while processing

2 Symmetric Encryption

Lecture 2 - Friday, September 05

2.1 Overview and Historical Ciphers

Caesar cipher works as following:

```
for i = 1, . . . , |m|
    x ← Encode(mi)
    y ← x + 23 mod 26
    c_i ← Decode(y)
return c
```

Encrypt(m) where $m \in \{A, ..., Z\}^*$

```
for i = 1, . . . , |c|
    x ← Encode(ci)
    y ← x - 23 mod 26
    m_i ← Decode(y)
return m
```

Decrypt(c) where $c \in \{A, ..., Z\}^*$

Comment 2.1

Notice that there is an essential difference between encrypt/ decrypt and encode/ decode: encode and decode map letters to numbers without trying to add security, while encrypt and decrypt try to add security.

A natural question to ask about a encryption method is whether it is secure or not. It is intuitive to consider the Caesar Cipher as insecure. However, what does it really mean to be secure?

2.1.1 Principle 1. Avoid Security by Obscurity

Definition 2.1: Kirckhoff's Principle

The Kirckhoff's Principle states that we shall assume the adversary knows everything about the system–including how the algorithm works – except secret keys.

As an example, we modify Caesar cipher to obey Kirckhoff's principle by introducing a secret key, which would be the amount of shift we perform. This cipher method is known as the Shift cipher. However, this is yet still not secure. There are two main methods to retrieve the plaintext from the ciphertext without knowing the key:

- Try all 26 possible secret key values $k = 0, \ldots, 25$. ("Brute-force attack" or "Exhaustive key search")
- Frequency analysis.

2.1.2 Brute-force Attack a.k.a. Exhaustive Key Search

Algorithm 2.1

Given sufficient amounts of ciphertext c, decrypt c using each possible key k until c decrypts to a plaintext message which "makes sense".

Comment 2.2

In order to know whether we have found anything that "makes sense", we will have to make some assumption on the plaintext space, such as it's a passage of English text or it's a PDF file.

2.1.3 Frequency Analysis

Algorithm 2.2

Compare the distribution of letters in the ciphertext with the distribution of letters in the underlying plaintext space. In particular,

1. Sort the characters in the ciphertext by frequency.
2. Guess that the i^{th} most frequent ciphertext letter represents the i^{th} most frequent English letter.
3. Adjust the guess as needed until the message makes sense.

Comment 2.3

We also assume that the plaintext is a piece of English language text.

Check out `interactive notebook/ Historical ciphers.ipynb`.

2.1.4 Defining Symmetric-key Encryption

Definition 2.2: Symmetric-key encryption scheme

A **symmetric-key encryption scheme** consists of:

- $\mathcal{M}$ – the plaintext space,
- $\mathcal{C}$ – the ciphertext space,
- $\mathcal{K}$ – the key space,
- an encryption algorithm, $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$,
- a decryption algorithm, $D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$.

such that $D(k, E(k, m)) = m$ for all $m \in \mathcal{M}$, $k \in \mathcal{K}$.

2.1.5 Substitution Cipher

Idea: instead of just shifting all letters by the same amount, have a table randomly mapping each letter to another fixed letter in a reversible way

Key space: $\mathcal{K}$ = all permutations of $\{A, \ldots, Z\}$; Message and ciphertext space: $\mathcal{M} = \mathcal{C} = \{A, \ldots, Z\}^*$.

```
1    for i = 1, ..., |m|
2        c_i ← apply permutation k to m_i
3    return c
```

Encrypt(k, m)

```
1    for i = 1, . . . , |c|
2        m_i ← apply inverse permutation
             k^{-1} to c_i
3    return m
```

Decrypt(k, c)

Question 2.1. Can you recover the plaintext?

To break the Substitution Cipher, exhaustive search would be quite expensive as the number of keys to try is $26! \approx 4 \times 1026 \approx 2^{88}$, but frequency analysis would work.

2.1.6 Transposition Cipher

Idea: randomly rearrange the order of the letters.

We first fix a block length $B \in \mathbb{N}$. Key space $\mathcal{K}$ = all permutations of $\{1, \ldots, B\}$. Message and ciphertext space $\mathcal{M} = \mathcal{C} = \bigcup_{i \geq 0} \{A, \ldots, Z\}^{iB}$.

```
1  for i = 1, ..., |m|/B
2      x ← m[(i − 1)B ... iB − 1]
3      y ← apply k to positions of x
4      i^th block of c ← y
5  return c
```

Encrypt(k, m)

```
1  for i = 1, ..., |c|/B
2      y ← i^th block of c
3      x ← apply k^{-1} to positions of y
4      i^th block of m ← x
5  return m
```

Decrypt(k, c)

Question 2.2. Can you recover the plaintext?

Exhaustive key search's cost is $B!$. When B is large, this can be infeasible. For frequency analysis: Single-letter frequency distribution stays the same since all we did is rearrange the letters, but bigram frequency analysis can be useful.

Suppose we add another attack assumption:

We are given the ability to have a plaintext of our choice encrypted under the secret key ("chosen plaintext attack").

Now can you recover the plaintext? Yes! Just ask for the plaintext "abcdefghijklmnopqrstuvwxyz" to be encrypted. Given the corresponding ciphertext, we can determine the permutation and thus the secret key.

Lecture 3 - Monday, September 08

2.1.7 Defining Security

Three key questions:

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

Theorem 2.1

We always make the basic assumption of Kirckhoff's principle: The adversary knows everything about the scheme, except the particular key k chosen by Alice and Bob. (Avoid security by obscurity!!)

A security model or security definition contains three elements:

1. the adversary's goal,
2. how the adversary interacts with the communicating parties, and
3. the computational powers of the adversary.

Defining security: 1. Adversary's Goal Some possible goals for the adversary in breaking symmetric key encryption:

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

Definition 2.3: Totally Insecure (Totally Broken)

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

Definition 2.4: Semantically Secure

If the adversary cannot learn any partial information about the plaintext from the ciphertext (except possibly its length), the scheme is said to be **semantically secure**.

Defining security: 2. Adversary's Interaction Some possible ways the adversary might be able to interact with communicating parties are

1. Passive attacks:
   - `Ciphertext-only attack`: The adversary only sees some encrypted ciphertexts.
   - `Known-plaintext attack`: The adversary also knows some plaintext and the corresponding ciphertext.
2. Active attacks:
   - `Chosen-plaintext attack`: The adversary can also choose some plaintext(s) and obtain the corresponding ciphertext(s).
   - `Chosen-ciphertext attack`: The adversary can also choose some ciphertext(s) and obtain the corresponding plaintext(s). Includes the powers of chosen-plaintext attack.
3. Other attacks:
   - `Side-channel attacks`: monitor the encryption and decryption equipment (timing attacks, power analysis attacks, electromagnetic-radiation analysis, etc.)
   - `Physical attacks`: bribery, blackmail, rubber hose, etc.

Defining security: 3. Computational Power of the Adversary Some possible limits on the computational powers of the adversary:

- `Information-theoretic security`: The adversary has infinite computational resources.
- `Complexity-theoretic security`: The adversary is a "polynomial-time Turing machine".
- `Computational security`: The adversary has X number of real computers/workstations/supercomputers. ("computationally bounded"). Equivalently, the adversary can do X basic operations, e.g., CPU cycles.

We also have to consider:

- `classical adversary`: the adversary does not have access to a quantum computer
- `quantum adversary`: the adversary has access to a quantum computer

Definition 2.5: Easy vs Feasible

In this course:

- 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.

code 2.1: Landauer limit

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).

Definition 2.6: Security Level

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

Comment 2.4

As of 2025, a security level of 128 bits is desirable in practice.

To break semantic security under chosen plaintext attack, the adversary has to win the following game:

1 `A secret key k is chosen at random.`
2 `Chosen-plaintext attack: The adversary can select plaintexts` m `and obtains the corresponding ciphertext` $c = E_k(m)$.
3 `The adversary picks two messages` m_0 `and` m_1 `of the same length.`

4 The challenger picks random $b \leftarrow \$\{0,1\}$ and gives the adversary $c^* \leftarrow E_k(m_b)$.
5 The adversary can again use their chosen-plaintext attack oracle.
6 After a bounded amount of computation, the adversary outputs its guess b' for the hidden bit

Security game: indistinguishability under chosen plaintext attack (IND-CPA)

Here is the IND-CPA security game defined in pseudocode, which is how you will normally see in textbooks or papers. Let $\mathcal{A}$ be computationally bounded classical adversary.

code 2.2: Security game: indistinguishability under chosen plaintext attack (IND-CPA)

1 $k \leftarrow \$\ \mathcal{K}$ // $\leftarrow \$$ means ''sample uniformly at random from the given set''
2 $(m_0, m_1) \leftarrow \mathcal{A}^{E_k(\cdot)}()$ // The superscript $E_k(\cdot)$ means the adversary can query an encryption oracle
3 $b \leftarrow \$\{0,1\}$ // and get the result, modelling a chosen-plaintext attack
4 $c^* \leftarrow E_k(m_b)$
5 $b' \leftarrow \mathcal{A}^{E_k(\cdot)}(c^*)$
6 if $b' = b$ then return 1 else return 0

Security game: indistinguishability under **chosen plaintext attack** (IND-CPA)

Definition 2.7: (Adversary's) Advantage

We measure the probability the experiment outputs 1, minus 1/2, which is called the adversary's advantage:

$$Adv^{\text{INDCPA}}(\mathcal{A}) = \left| Prob[\text{IND-CPA experiment outputs 1}] - \frac{1}{2} \right|$$

code 2.3: Security game: indistinguishability under chosen ciphertext attack (IND-CCA)

Still let $\mathcal{A}$ be computationally bounded classical adversary.

1 $k \leftarrow \$\ \mathcal{K}$ // $\leftarrow \$$ means ''sample uniformly at random from the given set''
2 $(m_0, m_1) \leftarrow \mathcal{A}^{E_k(\cdot), D_k(\cdot)}()$ // The superscript $E_k(\cdot)$ means the adversary can query an encryption oracle
3 $b \leftarrow \$\{0,1\}$ // and get the result, modelling a chosen-plaintext attack
4 $c^* \leftarrow E_k(m_b)$
5 $b' \leftarrow \mathcal{A}^{E_k(\cdot), D_k(\cdot \neq c^*)}(c^*)$
6 if $b' = b$ then return 1 else return 0

Security game: indistinguishability under **chosen ciphertext attack** (IND-CCA)

where similarly, we have

$$Adv^{\text{INDCCA}}(\mathcal{A}) = \left| Prob[\text{IND-CCA experiment outputs 1}] - \frac{1}{2} \right|$$

We will generally want symmetric key encryption schemes to satisfy one of the following:

Definition 2.8: IND-CPA Security

Semantic security (equivalently, indistinguishability) under chosen-plaintext attack by a computationally bounded classical adversary.

Definition 2.9: IND-CCA Security

Semantic security (equivalently, indistinguishability) under chosen-ciphertext attack by a computationally bounded classical adversary.

Comment 2.5

Elements of the definition:

1. Goal: semantic security (or equivalently, indistinguishability)
2. Adversary interaction: chosen-plaintext or chosen-ciphertext attack
3. Computational power: computationally bounded classical computer (or quantum if we can)

2.1.8 Vigenère cipher

Idea: Use different shift ciphers for different parts of the message to reduce effect of frequency analysis ("Polyalphabetic cipher")

Fix a block length $B \in \mathbb{N}$. Key space $\mathcal{K} = \{A, \ldots, Z\}^B$. Message and ciphertext space $\mathcal{M} = \mathcal{C} = \bigcup_{i \geq 0} \{A, \ldots, Z\}^{iB}$.

```
1  for i = 1, ..., |m|
2      c_i ← m_i + k[i mod B] mod 26
3  return c
```

Encrypt(k, m)

```
1  for i = 1, ..., |m|
2      m_i ← c_i − k[i mod B] mod 26
3  return m
```

Decrypt(k, c)

Question 2.3. Is the Vigenère Cipher secure?

It is totally insecure against a known-plaintext attack. However, a ciphertext-only attack could

1. Determine the block length B (if not already known) using frequency analysis.
2. Determine the key using B separate frequency analyses.

2.1.9 One-time pad a.k.a. Vernam cipher

Idea: A modification of the Caesar and Vigenère ciphers where the key is as long as the message.

Fix a message length $\ell \in \mathbb{N}$. Key space $\mathcal{K} = \{A, \ldots, Z\}^\ell$. Message and ciphertexts space $\mathcal{M} = \mathcal{C} = \{A, \ldots, Z\}^\ell$.

```
1   for i = 1, ..., ℓ
2       c_i ← m_i + k_i  mod 26
3   return c
```

Encrypt(k, m)

```
1   for i = 1, ..., ℓ
2       m_i ← c_i − k_i  mod 26
3   return m
```

Decrypt(k, c)

2.2 Stream ciphers

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

Definition 2.10: Stream Cipher

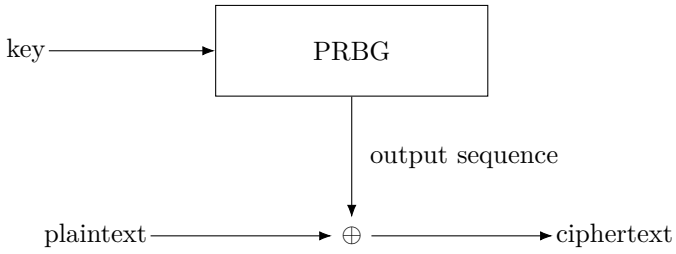
A **stream cipher** is a symmetric-key encryption scheme in which a pseudorandom sequence of the required length is generated to encrypt each successive character of plaintext.

2.2.1 Overview of Stream Ciphers

Definition 2.11: Pseudorandom Big Generator

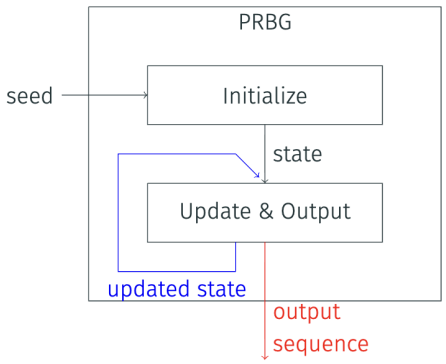
A **pseudorandom bit generator** (PRBG) is a deterministic algorithm that takes as input a short random seed, and outputs a longer pseudorandom sequence, also known as a keystream.

We can use a pseudorandom bit generator to construct a stream cipher. The *seed* is the secret key shared by Alice and Bob.



Compared to the one-time pad, a stream cipher will not have perfect secrecy. Instead, security depends on the quality of the PRBG. Security Requirements for the PRBG are:

1. `Indistinguishability`: The output sequence should be indistinguishable from a random sequence.
2. `Unpredictability`: given portions of the output sequence, it should be infeasible to learn any information about the rest of the output sequence.



Definition 2.12: Pseudorandom Big Generator

A pseudorandom bit generator consists of:

1. `Initialize(seed)`→`state`: A deterministic algorithm that takes as input a seed and outputs a state
2. `Update&Output(state)`→`(state', out)` A deterministic algorithm that takes as input a state and outputs an updated state and a output sequence a.k.a.keystream.

Original stream cipher designs used a pseudorandom generator with just one input: the key. The drawback comes along with it is that RBG is deterministic so each key produces a single output sequence, and as a result, we cannot use the same key to encrypt multiple messages.

Modern stream cipher designs use pseudorandom generators with two inputs: the key and an initialisation vector. While the key is kept private, the IV can be made public, so we can generate multiple (different) output sequences from the same key but different IV's.

Result 2.1

However, this means that we need to transmit the IV with the ciphertext, which is a constant overhead.

2.2.2 Linear Feedback Shift Registers

The idea is to use linear recurrence relation for update & output. An infinite periodic sequence can be defined via a recurrence relation:

$$s_{t+n} = c_{n-1}s_{t+n-1} \oplus c_{n-2}s_{t+n-2} \oplus \cdots \oplus c_1 s_{t+1} \oplus c_0 s_t$$

where c_i are binary constants.

Algorithm 2.3: linear feedback shift registers (LFSR)

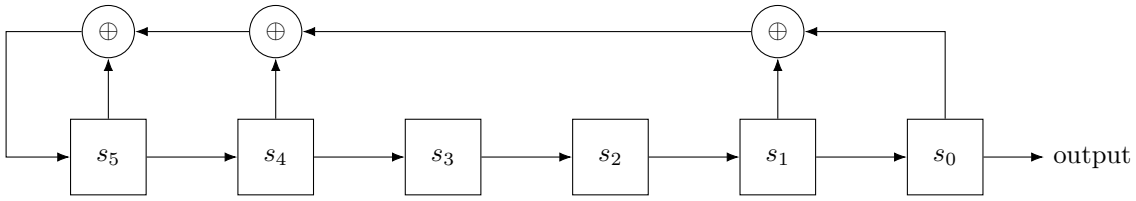
1. Load the n-bit initial state into values $(s_0, s_1, \ldots, s_{n-1})$.
2. Update operation: generate the next element of the sequence; only need to store the most recent n elements.
3. Output operation: output the next element generated by the sequence: $s_n, s_{n+1}, \ldots$.
   This is called a linear feedback shift registers (LFSR).

Comment 2.6

Fast to implement in software and hardware.

Example 2.1

LFSR diagram for the recurrence relation $s_{t+6} = s_{t+5} \oplus s_{t+4} \oplus s_{t+1} \oplus s_t$ is shown below:



This represents the following algorithm:

```
1        output ← s_t
2        s'_{t+5} ← s_{t+5} ⊕ s_{t+4} ⊕ s_{t+1} ⊕ s_{t+0}
3        for i = 0 ... 4 : s'_{t+i} ← s_{t+i+1}
4        return(state' = s⃗', output)
5    return m
```

Update&Output(state $= \vec{s} = (s_{t+5}, \ldots, s_t)$)

Is an LFSR a good pseudorandom bit generator?

Question 2.4. Is an LFSR a good pseudorandom bit generator?

- If it has a short period (meaning, the sequence repeats after a small number of positions), then we will effectively have used the same keystream to encrypt multiple parts of a message. Just like attacking reused keys in the one-time pad.
- If we have a partial known plaintext attack, where we know n consecutive bits of plaintext within a ciphertext, then we can recover the entire state at that point in the LFSR, and then generate all subsequent output. Violates unpredictability of PRBGs.
- Hiding the constants c_i doesn't help much: in a $2n$-bit known plaintext attack, we can construct a system of n linear equations to recover the n unknown constants c_i.

A single LFSR on its own is not a good pseudorandom generator: knowing the output leads to knowing the intermediate state so you can predict all future output. It also does not attain very high linear complexity: can be modelled perfectly by a relatively small linear equation.

Three ways to build a better pseudorandom generator from LFSRs:

1. Select multiple outputs from a single sequence and combine them through a non-linear function (called **non-linear filter**).
2. Use multiple LFSRs and combine their output using a non-linear function (a **non-linear combiner**).
3. Use multiple LFSRs but only advance some of them at each step using an irregular clock. (Example to follow: A5/1.)

2.2.3 A5/1

The A5/1 algorithm uses three LFSRs whose output is combined. The three LFSRs are irregularly clocked which means that the overall output is non-linear. Each LFSR has a clocking bit. Each LFSR is advanced "clocked" (i.e., updated using the Update function) if its own clocking bit equals the majority of all LFSR's clocking bits.

2.2.4 RC4

- Pros: Extremely simple; extremely fast; variable key length.
- Cons: Design criteria are proprietary; not much public scrutiny until the year 2001. Has small biases exploitable with ≈millions of ciphertexts. Not recommend for use today.

```
1   // K[i], K̄[i] and S[i] are 8-bit
        integers (bytes)
2   for i = 0, ..., 255 :
3       S[i] ← i
4       K̄[i] ← K[i  mod d]
5   j ← 0
6   for i = 0, ..., 255 :
7       j ← (K̄[i] + S[i] + j)  mod 256
8       Swap(S[i], S[j])
9   return S // a 256-byte array S[0], S
        [1], ..., S[255]
```

Initialize($K = K[0], K[1], \ldots, K[d-1]$)

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

Comment 2.7

Do not memorize this algorithm or look into the details too much. Some basic takeaways:

- Simple description
- Easy to implement in software
- But not secure any more

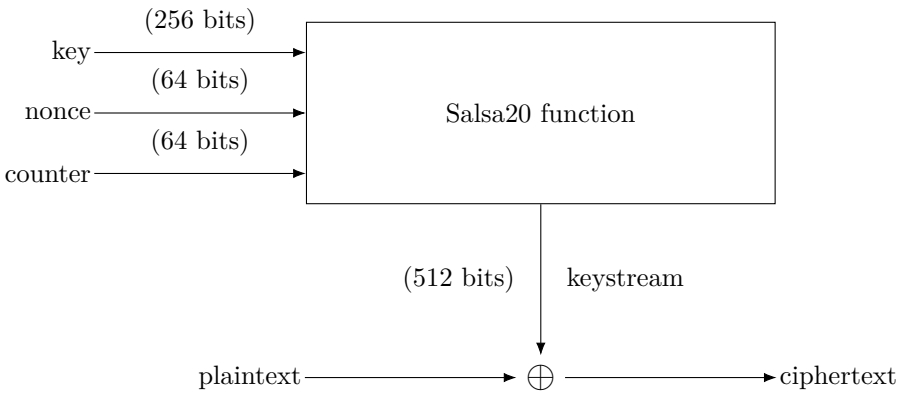
```
1   i ← 0; j ← 0
2   while more keystream bytes are required
3       i ← (i + 1) mod 256
4       j ← (S[i] + j) mod 256
5       Swap(S[i], S[j])
6       t ← (S[i] + S[j]) mod 256
7       \textbf{return} S[t]
```

Update&Output($S = S[0], S[1], \ldots, S[d-1]$)

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

2.2.5 Salsa and ChaCha

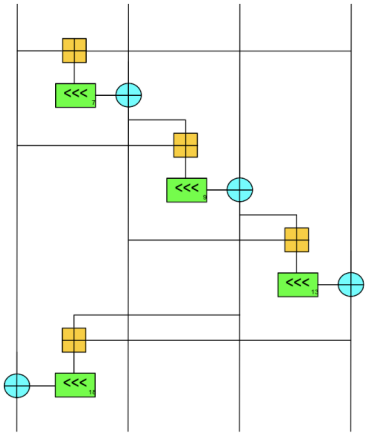
- Salsa20: Modern stream cipher, designed by Daniel J. Bernstein in 2005.
- ChaCha20: Variant of Salsa20, 2008. Faster on some architectures.



Salsa20 initial state: 4×4 array of 32-bit words:

constant_1	key	key	key
key	constant_2	nonce	nonce
counter	counter	constant_3	key
key	key	key	constant_4

Salsa20 quarter-round function (image from Wikipedia):



- Yellow square plus: integer addition modulo 2^{32}
- Blue circle plus: exclusive OR
- Green left arrows: cyclic shift by 7 / 9 / 13 / 18 positions

Algorithm 2.4

- `Salsa20 Initialize`: Load the key, nonce, and counter into the state;
- Salsa20 Update&Output

1. Make a copy of the initial state
2. Do the following 10 times:
   (a) Apply the quarter-round function to each column of the state (4 columns)
   (b) Apply the quarter-round function to each row of the state (4 rows)
3. XOR the current state with the copy of the initial state
4. Output that as the keystream

Salsa20 intuition : Do some simple operations many times. XORing in the initial state acts a little like the Vigenère cipher. The cyclic shifts and applying the quarter-round function alternately to columns and rows are a little like the transposition cipher. Together these ensures that the effect of key bits gets spread out across the entire state ("diffusion") in a way that's hard to reverse

2.2.6 Case Study: Wireless (IEEE 802.11) Security

IEEE 802.11 standard for wireless LAN communications includes a protocol called **Wired Equivalent Privacy** (WEP), whose goal is (only) to protect link-level data during wireless transmission between mobile stations and access points.

Description of WEP Protocal Mobile station shares a secret key k with access point, and messages are divided into packets of some fixed length. 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.

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

Algorithm 2.5

1. Select a 24-bit IV v;
2. Compute a 32-bit checksum: $S = CRC(m)$;
3. Compute $c = (m\|S) \oplus RC4(v\|k)$ ($\|$ also denotes concatenation);
4. Send (v, c) over the wireless channel.

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

Algorithm 2.6

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

Question 2.5. Are confidentiality data integrity and access control achieved?

NO! [Borisov, Goldberg & Wagner; 2001]

Problem 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. (Some contents of network traffic are predictable.)

Problem 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:

1. Suppose (v, c) is an encrypted packet.
2. Let $c = RC4(v\|k) \oplus (m\|S)$, where k, m, S are unknown.
3. 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.)
4. Let $c' = c \oplus (\Delta\|CRC(\Delta))$.
5. Then (v, c') is a valid encrypted packet for m'.

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

2.3 Block ciphers

Definition 2.13: Block Cipher

A **block cipher** is a symmetric-key encryption scheme in which a fixed-length block of plaintext is mapped to an equal-sized block of ciphertext.

Design principles described by Claude Shannon in 1949:

- Security:
  1. `Diffusion`: each ciphertext bit should depend on all plaintext and all key bits.
  2. `Confusion`: the relationship between key bits, plaintext bits, and ciphertext bits should be complicated.
  3. `Cascade or avalanche effect`: changing one bit of plaintext or key should change about half the bits of ciphertext
  4. `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; and suitability for hardware or software.

2.3.1 The Advanced Encryption Standard (AES)

Definition 2.14: Substitution-Permutation Network

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

Comment 2.8

During each round, a round key is XORed into the state. The round keys ki are derived from the main key k using a key schedule function.

AES is an SPN where the "permutation" operation consists of two linear transformations (one of which is a permutation). All operations are byte oriented. The block size of AES is 128 bits and each round key is 128 bits.

AES accepts three different key lengths. The number of rounds depends on the key length:

key length	**number of rounds** h
128	10
192	12
256	14

Theorem 2.2: Design principles

Design principles:

- The substitution operation (S-box) is the only non-linear component of the cipher.
- The permutation operations (permutation and linear transformation) spread out the non-linearities in each round.

AES Round Operations : Each round updates a variable called State which consists of a 4×4 array of bytes (note: $4 \cdot 4 \cdot 8 = 128$, the block size). State is initialized with the 128-bit plaintext:

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

$\longleftarrow$ *plaintext*

Algorithm 2.7

The AES round function uses four operations:

- AddRoundKey (key mixing)
- SubBytes (S-box) (substitution)
- ShiftRows (permutation)
- MixColumns (matrix multiplication / linear transformation)

AES Encryption : From the key k, derive $h+1$ round keys $k_0, k_1, \ldots, k_h$ via the key schedule.

```
1 State ← plaintext
2     for i = 1 ... h − 1 do
3         State ← State ⊕ k_{i−1}
4         State ← SubBytes(State)
5         State ← ShiftRows(State)
6         State ← MixColumns(State)
7     State ← State ⊕ k_{h−1}
```

```
    State ← SubBytes(State)
    State ← ShiftRows(State)
    State ← State ⊕ k_h
ciphertext ← State
```

The encryption function

Comment 2.9

Note that in the final round, `MixColumns` is not applied.

AES Decryption : From the key k, derive $h+1$ round keys $k_0, k_1, \ldots, k_h$ via the key schedule.

```
State ← ciphertext
State ← State ⊕ k_h
for i = h − 1 ... 1 do
    State ← InvShiftRows(State)
    State ← InvSubBytes(State)
    State ← State ⊕ k_i
    State ← InvMixColumns(State)
State ← InvShiftRows(State)
State ← InvSubBytes(State)
State ← State ⊕ k_0
```

The decryption function

AES key schedule (for 128-bit keys) : For 128-bit keys, AES has ten rounds, so we need eleven subkeys. Each k_i is a 32-bit word (viewed as a 4-byte array). Each group of four k_i's forms a 128-bit subkey. The first round subkey (k_0, k_1, k_2, k_3) equals the actual AES key.

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

- Left-shift the input cyclically by 8 bits.
- Apply the AES S-box to each byte.
- Bitwise XOR the left-most byte with a constant which varies by round according to the following table.

Round	constant	Round	constant
1	**0x01**	6	**0x20**
2	**0x02**	7	**0x40**
3	**0x04**	8	**0x80**
4	**0x08**	9	**0x1B**
5	**0x10**	10	**0x36**

- Output the result.

2.3.2 Data Encryption Standard (DES)

Block cipher with 64-bit blocks, 56-bit key, and 16 rounds of operation.

Each round of the Feistel network applies a component function, which is effectively a substitution-permutation network

- XOR 48-bit round key
- Apply eight S-boxes each mapping $\{0,1\}^6 \to \{0,1\}^4$
- Fixed permutation on 32 bits

DES Problem 1: Small Key Size Exhaustive search on key space takes 256 steps and can be easily parallelized.

DES Problem 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 ≈ 232 (by the birthday paradox). Hence the ciphertext reveals some information about the plaintext.

Small block length is also damaging to some authentication applications (more on this later).

Sophisticated Attacks on DES

1. Differential cryptanalysis [Biham & Shamir 1989]:
   - Recovers key given 247 chosen plaintext/ciphertext pairs.
   - DES was designed to resist this attack.
   - Differential cryptanalysis has been more effective on some other block ciphers.
2. Linear cryptanalysis [Matsui 1993]:
   - Recovers key given 243 known plaintext/ciphertext pairs.
   - Storing these pairs takes 131,000 Gbytes.
   - Implemented in 1993: 10 days on 12 machines.

2.3.3 Trying to save DES: Multiple Encryption

Definition 2.15: Multiple Encryption

Multiple encryption: Re-encrypt the ciphertext one or more times using independent keys, and hope that this operation increases the effective key length.

Comment 2.10

Multiple encryption does not always increase security.

Example 2.2

If E_π denotes the simple substitution cipher with key π, then is $E_{\pi 1} \circ E_{\pi 2}$ any more secure than E_π?

Attack on Double DES `Main idea`: If $c = E_{k_2}(E_{k_1}(m))$, then $E_{k_2}^{-1}(c) = E_{k_1}(m)$. (Meet-in-the-middle)

1. Given: Known plaintext pairs (m_i, c_i), $i = 1, 2, 3, \ldots$
2. For each $h_2 \in \{0,1\}^{56}$:
   2.1 Compute $E_{h_2}^{-1}(c_1)$, and store $[E_{h_2}^{-1}(c_1), h_2]$ in a table.
3. For each $h_1 \in \{0,1\}^{56}$ do the following:
   3.1 Compute $E_{h_1}(m_1)$.
   3.2 Search for $E_{h_1}(m_1)$ in the table.
   3.3 If $E_{h_1}(m_1) = E_{h_2}^{-1}(c_1)$:
   - Check if $E_{h_1}(m_2) = E_{h_2}^{-1}(c_2)$
   - Check if $E_{h_1}(m_3) = E_{h_2}^{-1}(c_3)$
   - $\vdots$

   If all checks pass, then output (h_1, h_2) and STOP.

Result 2.2

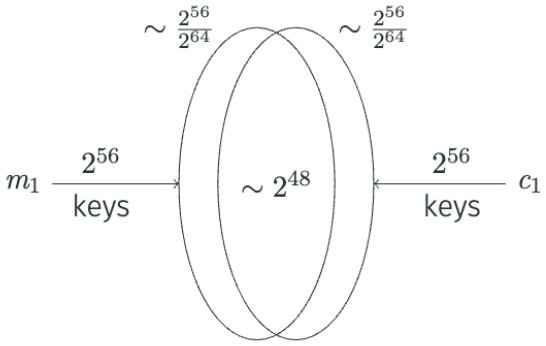
Complexity of the attack is $\approx 2^{57}$.

Now we analyze the meet-in-middle attack. Number of known plaintext/ciphertext pairs required to avoid false keys: 2 suffice, with high probability. Number of DES operations is

$$\approx 2^{56} + 2^{56} + 2 \cdot 2^{48} \approx 257$$

(We are not counting the time to do the sorting and searching). Space requirements:

$$2^{56}(64 + 56)bits \approx 1,080,863 \text{ Tbytes}$$



Approximately 2^{48} keys h_1 encrypting m_1 yield intermediate ciphertext equal to a decryption of c_1 under some key h_2. But with high probability $E_{h_1}(m_2) \neq D_{h_2}(c_2)$, so we have to do $\sim 2^{48}$ DES operations for checking next pair. In total

$$2^{56} + 2^{56} + \sim 2^{48}$$

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

$$\text{Time: } 2^{56+s} \text{ steps; memory: } 2^{56-s} \text{ units}, \quad 1 \leq s \leq 55$$

Result 2.3

In conclusion,

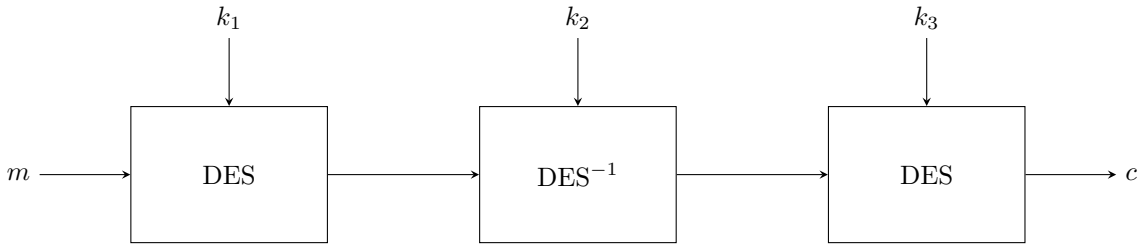
- Double-DES has the same effective key length as DES.
- Double-DES is not much more secure than DES.

Comment 2.11

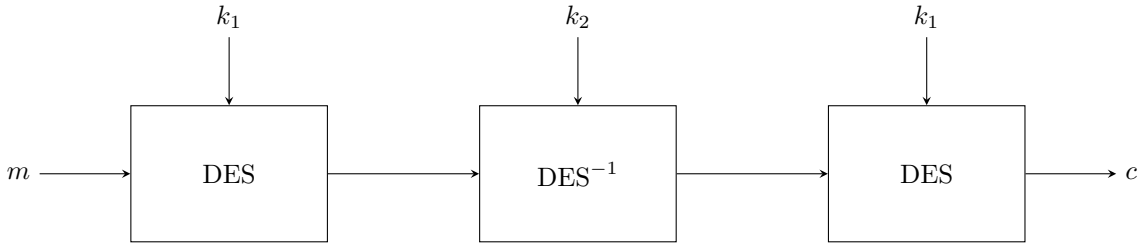
There is also three-key triple encryption, whose meet-in-the-middle attack takes $\approx 2^{112}$ steps. No proof that Triple-DES is more secure than DES. Triple-DES is widely deployed.

2.3.4 Some Variants

`EDE Triple-DES`: for backward compatibility with DES



`Two-key Triple-DES`: 112-bit key but effective security of only 56 bits



2.4 Cryptanalysis of Block Ciphers

2.4.1 The Heys Cipher – a Substitution-Permutation Networks

Definition 2.16: Substitution-Permutation Network

A **substitution-permutation network** (SPN) is a type of iterated block cipher where a round consists of:

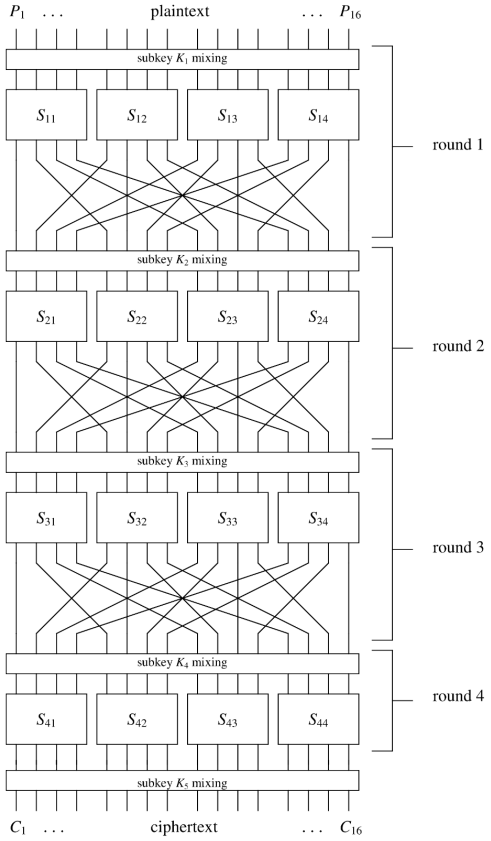
- Incorporation of the round key, followed by
- A substitution operation, followed by
- A permutation operation.

Example 2.3: Examples of Substitution-Permutation Network designs

DES component function, AES, and Heys cipher, etc.

The Heys cipher contains following components:

- 4-round SPN
- 16-bit block size
- All S-boxes are identical
- All permutations are identical
- No key schedule algorithm
- 80-bit key



S-box:

In	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Out	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7

Permutation:

In	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Out	1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16

S-box is identical to the first line of the DES S1 S-box. Permutation is the "butterfly" permutation. The K5 subkey prevents an adversary from reversing the final round of substitution and permutation. In general this technique is **called key whitening**.

Example 2.4

Example encryption: Take the plaintext $P = 0110100111011011$, and suppose the key consists of all 1's ($K_i = 1111111111111111$ for $i = 1, 2, 3, 4, 5$).

P	0110 1001 1101 1011
U_1	1001 0110 0010 0100
V_1	1010 1011 1101 0010
U_2	0001 1101 0010 1001
V_2	0100 1001 1101 1010
U_3	1000 0101 1110 1001
V_3	0011 1111 0000 1010
U_4	1010 1011 0010 0011
V_4	0110 1100 1101 0001
C	1001 0011 0010 1110

2.4.2 Linear Cyptanalysis

Goal of linear cryptanalysis: find linear equations that hold often (but not necessarily always). With enough known plaintext/ciphertext pairs, maybe we can still solve for the key bits.

Comment 2.12

The following is an example.

- Look for linear (boolean) relations among the plaintext/ciphertext/key bits which hold with for a significant fraction of inputs, then use a known plaintext attack to figure out the mostly key.
- For example:

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \ \oplus \ P_5 \oplus P_7 \oplus P_8 \ \simeq \ 0$$

- In the above equation, $U_{4,6}, U_{4,8}, U_{4,14}, U_{4,16}$ depend only on the ciphertext C and the eight key bits $K_{5,5}, K_{5,6}, K_{5,7}, K_{5,8}, K_{5,13}, K_{5,14}, K_{5,15}, K_{5,16}$.
- Given enough known-plaintext pairs (P, C), guess the appropriate partial subkey bits until a guess is found for which the relation holds for a fraction of inputs much different from 50%, over all the known (P, C)-pairs.

Algorithm 2.8

Given many plaintext-ciphertext pairs and a linear relation among the plaintext/ciphertext/key bits which holds for a fraction of inputs much different from 50%

- For the "active" partial subkey bits, go through all possible values: For each plaintext-ciphertext pair, record whether the relation holds for these key bits and this plaintext-ciphertext pair.
- The choice of key bits which yields the largest magnitude of bias is probably correct.

Repeat with many different linear relations to recover as much of the key as possible, then use a brute force search on the remaining key bits.

Our goal is to find linear relations which hold for an abnormally large or abnormally small fraction of inputs. Start with the S-box!

Linear Relations for S-boxes Recall that we have the following S-box relations:

X	X_1	X_2	X_3	X_4	Y	Y_1	Y_2	Y_3	Y_4
0	0	0	0	0	14	1	1	1	0
1	0	0	0	1	4	0	1	0	0
2	0	0	1	0	13	1	1	0	1
3	0	0	1	1	1	0	0	0	1
4	0	1	0	0	2	0	0	1	0
5	0	1	0	1	15	1	1	1	1
6	0	1	1	0	11	1	0	1	1
7	0	1	1	1	8	1	0	0	0
8	1	0	0	0	3	0	0	1	1
9	1	0	0	1	10	1	0	1	0
10	1	0	1	0	6	0	1	1	0
11	1	0	1	1	12	1	1	0	0
12	1	1	0	0	5	0	1	0	1
13	1	1	0	1	9	1	0	0	1
14	1	1	1	0	0	0	0	0	0
15	1	1	1	1	7	0	1	1	1

So we have the table of all biases of all Heys S-box linear approximations:

		Output Sum															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Input Sum	0	+8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	−2	−2	0	0	−2	+6	+2	+2	0	0	+2	+2	0	0
	2	0	0	−2	−2	0	0	−2	−2	0	0	+2	+2	0	0	−6	+2
	3	0	0	0	0	0	0	0	0	+2	−6	−2	−2	+2	+2	−2	−2
	4	0	+2	0	−2	−2	−4	−2	0	0	−2	0	+2	+2	−4	+2	0
	5	0	−2	−2	0	−2	0	+4	+2	−2	0	−4	+2	0	−2	−2	0
	6	0	+2	−2	+4	+2	0	0	+2	0	−2	+2	+4	−2	0	0	−2
	7	0	−2	0	+2	+2	−4	+2	0	−2	0	+2	0	+4	+2	0	+2
	8	0	0	0	0	0	0	0	0	0	0	+2	+2	+2	−2	−2	−6
	9	0	0	−2	−2	0	0	−2	−2	−4	0	−2	+2	0	+4	+2	−2
	A	0	+4	−2	+2	−4	0	+2	−2	+2	+2	0	0	+2	+2	0	0
	B	0	+4	0	−4	+4	0	+4	0	0	0	0	0	0	0	0	0
	C	0	−2	+4	−2	−2	0	+2	0	+2	0	+2	+4	0	+2	0	−2
	D	0	+2	+2	0	−2	+4	0	+2	−4	−2	+2	0	+2	0	0	+2
	E	0	+2	+2	0	−2	0	+2	−2	0	0	−2	−4	+2	−2	0	0
	F	0	−2	−4	−2	−2	0	+2	0	0	−2	+4	−2	−2	0	+2	0

How to read this table:

- Linear relation is
$$a_1X_1 \oplus a_2X_2 \oplus a_3X_3 \oplus a_4X_4 = b_1Y_1 \oplus b_2Y_2 \oplus b_3Y_3 \oplus b_4Y_4$$
where $a_i \in \{0,1\}$, $b_i \in \{0,1\}$ for $i = 1,2,3,4$
- "Input sum" is the value of $a_1a_2a_3a_4$ in binary.
- "Output sum" is the value of $b_1b_2b_3b_4$ in binary.
- The bias of this linear relation is x/16 where x is the value of the table cell.

Building a linear approximation from top to bottom We now know how to find linear approximations of individual S-boxes.

Need to chain together one or more S-box approximation from each round to construct an approximation all the way from top to bottom.

Theorem 2.3: Piling-up Lemma (Matsui, 1993)

Let $X_1, X_2, \ldots, X_n$ be independent binary random variables with bias $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ respectively. Then

$$\text{Prob}\big(X_1 \oplus X_2 \oplus \cdots \oplus X_n = 0\big) = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \varepsilon_i$$

Note 2.1

Our inputs are not necessarily independent, but are usually close enough (for practical block ciphers) that the result is close enough.

We will see how combining the following 4 S-box approximations:

$$S_{12}: \quad X_1 \oplus X_3 \oplus X_4 \approx Y_2 \qquad \text{(bias } +1/4)$$

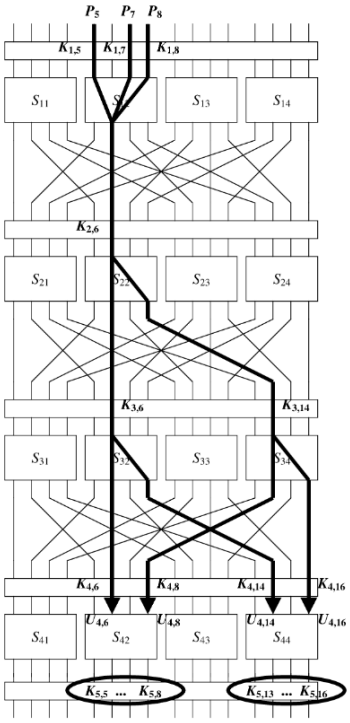
$$S_{22}: \quad X_2 \approx Y_2 \oplus Y_4 \qquad \text{(bias } -1/4)$$

$$S_{32}: \quad X_2 \approx Y_2 \oplus Y_4 \qquad \text{(bias } -1/4)$$

$$S_{34}: \quad X_2 \approx Y_2 \oplus Y_4 \qquad \text{(bias } -1/4)$$

will allow us to construct a linear approximation from the top to (nearly the) bottom that holds with bias $-1/32$:

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 \approx 0$$



Round 1 : For the approximation:

$$S_{12}: X_1 \oplus X_3 \oplus X_4 \approx Y_2 \qquad \text{(bias } +1/4)$$

$$U_1 = P \oplus K_1$$

$$X_1 = U_{1,5} = P_5 \oplus K_{1,5}$$

$$X_3 = U_{1,7} = P_7 \oplus K_{1,7}$$

$$X_4 = U_{1,8} = P_8 \oplus K_{1,8}$$

$$Y_2 = V_{1,6}$$

$$V_{1,6} \approx U_{1,5} \oplus U_{1,7} \oplus U_{1,8} \qquad \text{(bias } +1/4)$$

$$= (P_5 \oplus K_{1,5}) \oplus (P_7 \oplus K_{1,7}) \oplus (P_8 \oplus K_{1,8})$$

Round 2 : Our Round 1 approximation was for $V_{1,6}$. When we follow that wire through the permutation, we see that it leads to the 2nd input bit to $S_{2,2}$. Look at S-box approximations that rely on 2nd input bit X_2.

$$S_{22}: X_2 \approx Y_2 \oplus Y_4 \qquad \text{(bias } -1/4)$$

For the approximation:

$$S_{22} : X_2 \approx Y_2 \oplus Y_4 \qquad \text{(bias } -1/4\text{)}$$

$$U_{2,6} \approx V_{2,6} \oplus V_{2,8} \qquad \text{(bias } -1/4\text{)}$$

$$U_{2,6} = V_{1,6} \oplus K_{2,6}$$

$$V_{1,6} \oplus K_{2,6} \approx V_{2,6} \oplus V_{2,8} \qquad \text{(bias } -1/4\text{)}$$

Round 1–2 : Combining the approximations for Rounds 1 and 2:

$$V_{1,6} \oplus K_{2,6} \approx V_{2,6} \oplus V_{2,8} \qquad \text{(bias } -1/4\text{)}$$
$$V_{1,6} \approx P_5 \oplus P_7 \oplus P_8 \oplus K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \qquad \text{(bias } +1/4\text{)}$$

yields:

$$V_{2,6} \oplus V_{2,8} \oplus P_5 \oplus P_7 \oplus P_8 \oplus K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \approx 0.$$

By the Piling-up Lemma, this formula holds with bias:

$$2 \cdot \left(-\frac{1}{4}\right) \cdot \left(+\frac{1}{4}\right) = -\frac{1}{8}$$

Round 3 : Our Round 2 approximation was for $V_{2,6} \oplus V_{2,8}$. When we follow those wires through the permutation, we see that they lead to the 2nd input bits for $S_{3,2}$ and $S_{3,4}$. Look at S-box approximations that rely on 2nd input bit X_2.

$$S_{32} : X_2 \approx Y_2 \oplus Y_4 \quad \text{(bias } -1/4\text{)}$$

$$S_{34} : X_2 \approx Y_2 \oplus Y_4 \quad \text{(bias } -1/4\text{)}$$

For the approximations:

$$S_{32} : X_2 \approx Y_2 \oplus Y_4 \quad \text{(bias } -1/4\text{)}$$

$$S_{34} : X_2 \approx Y_2 \oplus Y_4 \quad \text{(bias } -1/4\text{)}$$

$$U_{3,6} \approx V_{3,6} \oplus V_{3,8} \qquad \text{(bias } -1/4\text{)}$$

$$U_{3,14} \approx V_{3,14} \oplus V_{3,16} \quad \text{(bias } -1/4\text{)}$$

$$U_{3,6} = V_{2,6} \oplus K_{3,6}$$

$$U_{3,14} = V_{2,8} \oplus K_{3,14}$$

$$V_{2,6} \oplus K_{3,6} \approx V_{3,6} \oplus V_{3,8} \qquad \text{(bias } -1/4\text{)}$$

$$V_{2,8} \oplus K_{3,14} \approx V_{3,14} \oplus V_{3,16} \quad \text{(bias } -1/4\text{)}$$

Combining the two approximations for Round 3:

$$V_{2,6} \oplus K_{3,6} \approx V_{3,6} \oplus V_{3,8} \qquad \text{(bias } -1/4\text{)}$$

$$V_{2,8} \oplus K_{3,14} \approx V_{3,14} \oplus V_{3,16} \qquad \text{(bias } -1/4\text{)}$$

we obtain

$$V_{3,6} \oplus V_{3,8} \oplus V_{3,14} \oplus V_{3,16} \oplus V_{2,6} \oplus K_{3,6} \oplus V_{2,8} \oplus K_{3,14} \approx 0$$

with bias

$$2 \cdot \left(-\frac{1}{4}\right) \cdot \left(-\frac{1}{4}\right) = +\frac{1}{8}.$$

Round 1-3 : We have

$$V_{3,6} \oplus V_{3,8} \oplus V_{3,14} \oplus V_{3,16} \oplus V_{2,6} \oplus K_{3,6} \oplus V_{2,8} \oplus K_{3,14} \approx 0 \qquad \text{(Round 3)}$$

$$V_{2,6} \oplus V_{2,8} \oplus P_5 \oplus P_7 \oplus P_8 \oplus K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \approx 0 \qquad \text{(Rounds 1–2)}$$

with biases $-1/8$ and $+1/8$ respectively. Hence an approximation for Rounds 1–3 is

$$V_{3,6} \oplus V_{3,8} \oplus V_{3,14} \oplus V_{3,16} \oplus K_{3,6} \oplus K_{3,14} \oplus P_5 \oplus P_7 \oplus P_8 \oplus K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \approx 0$$

with bias

$$2 \cdot \left(-\frac{1}{8}\right) \cdot \left(+\frac{1}{8}\right) = -\frac{1}{32}.$$

Round 1-4 : Our Round 1–3 approximation was for

$$V_{3,6} \oplus V_{3,8} \oplus V_{3,14} \oplus V_{3,16}.$$

When we follow those wires through the permutation and the round key, we obtain formulas for intermediate state U_4. Combine

$$V_{3,6} \oplus V_{3,8} \oplus V_{3,14} \oplus V_{3,16} \oplus K_{3,6} \oplus K_{3,14} \oplus P_5 \oplus P_7 \oplus P_8 \oplus K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \approx 0$$

with:

$$U_{4,6} = V_{3,6} \oplus K_{4,6} \qquad U_{4,8} = V_{3,14} \oplus K_{4,8}$$

$$U_{4,14} = V_{3,8} \oplus K_{4,14} \qquad U_{4,16} = V_{3,16} \oplus K_{4,16}$$

Final Approximation : From the previous computations, we obtained the approximation up to the U_4 in Round 4:

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 \oplus K_{4,6} \oplus K_{4,8} \oplus$$
$$K_{4,14} \oplus K_{4,16} \oplus K_{3,6} \oplus K_{3,14} \oplus K_{1,5} \oplus K_{1,7} \oplus K_{1,8} \oplus K_{2,6} \approx 0 \qquad \text{(bias } -1/32\text{)}$$

We don't know the various intermediate key bits $K_{1,5}, K_{1,7}, \ldots, K_{4,16}$. But whatever they are, they are fixed across all our known plaintext/ciphertext pairs. So this formula is effectively:

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 \oplus c \approx 0 \qquad \text{(bias } -1/32\text{)}$$

Since c is fixed (though unknown to us), it's either always 0 or always 1, so it just always leaves the result unchanged or always flips. Thus we can eliminate it and incorporate the effect into the bias:

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 \approx 0 \qquad \text{(bias } \pm 1/32\text{)}$$

Our approximation up to the U_4 in Round 4 is:

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 \approx 0 \qquad \text{(bias } \pm 1/32\text{)}$$

But we don't have the U_4 values. We only know plaintext/ciphertext pairs: P and C.

Comment 2.13

Idea: compute U_4 by taking C and guessing what the relevant ("active") round key bits K_5 are.

$$(U_{4,5}, U_{4,6}, U_{4,7}, U_{4,8}) = S_{4,2}^{-1}\Big((C_5, C_6, C_7, C_8) \oplus (K_{5,5}, K_{5,6}, K_{5,7}, K_{5,8})\Big)$$

2.4.3 A known-plaintext attack

Given many plaintext–ciphertext pairs:

1. For every possible choice of active subkey bits $K_{5,5}, K_{5,6}, K_{5,7}, K_{5,8}$ and $K_{5,13}, K_{5,14}, K_{5,15}, K_{5,16}$:
   - 1.1. For each plaintext/ciphertext pair P/C:
     - 1.1.1. Compute $U_{4,6}, U_{4,8}, U_{4,14}, U_{4,16}$ from C and the guessed active subkey bits (using approach on previous slide).
     - 1.1.2. Record whether the linear approximation holds for this plaintext/ciphertext pair and this subkey guess:
       $$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 = 0$$
2. The choice of subkey bits which most closely matches the predicted bias for the linear approximation is probably correct.
   - (With high probability this condition is equivalent to picking the subkey bits with the largest magnitude of bias.)

Example 2.5

Suppose $P = 0110\ 1001\ 1101\ 1011$ and $C = 1100\ 0110\ 1000\ 1100$. We guess $K_{5,5} \cdots K_{5,8} = 0111$ and $K_{5,13} \cdots K_{5,16} = 0110$.

Work backwards from C and K_5 to compute U_4:

$C_5 \cdots C_8$	0110	$C_{13} \cdots C_{16}$	1100
$K_{5,5} \cdots K_{5,8}$	0111	$K_{5,13} \cdots K_{5,16}$	0110
$V_{4,5} \cdots V_{4,8}$	0001	$V_{4,13} \cdots V_{4,16}$	1010
$U_{4,5} \cdots U_{4,8}$	0011	$U_{4,13} \cdots U_{4,16}$	1001

Does the linear approximation hold for these values?

$$U_{4,6} \oplus U_{4,8} \oplus U_{4,14} \oplus U_{4,16} \oplus P_5 \oplus P_7 \oplus P_8 = 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 = 0$$

Yes! Record answer, then repeat with all plaintext/ciphertext pairs and all subkey bits.

A known-plaintext attack: How to recover the whole key

- One top-to-bottom linear approximation will give you some of the K_5 subkey bits.
- Once you've found those, do more top-to-bottom linear approximations to get the rest of the K_5 subkey bits.
- Then construct linear approximations from the top to the U_3 level, and you can use the now known K_5 subkey to work backwards up one more level.
- Repeat until you have the entire key.
- If at any stage you can't come up with a linear approximation for some subkey bits, you could guess them (do a brute force search), but your brute force search will involve doing every subsequent linear approximation application for each possible guess of the ones you're brute force searching over.

Suppose ε is the bias from 1/2 that the linear expression holds for the complete cipher. Number of known plaintext-ciphertext pairs required is $\approx 1/\varepsilon^2$. Implications of this approach:

- The fewer S-boxes involved in a linear approximation, the higher the bias;
- Thus, can reduce effectiveness of linear cryptanalysis by designing ciphers that have a high number of "active" S-boxes

2.4.4 Differential cryptanalysis

The idea is to look for flaws in the cipher where related plaintexts get encrypted to related ciphertexts. Build up from individual S-boxes.

In a truly random cipher, if two plaintext inputs P and P' are related by some fixed difference $\Delta P = P \oplus P'$, the corresponding ciphertexts C and C' should look completely unrelated. But sometimes we see that plaintexts related by some ΔP result in ciphertexts related by some difference $\Delta C = C \oplus C'$. A plaintext difference ΔP leading to a ciphertext difference ΔC with better-than-random chance is called a **differential characteristic**.

Therefore, differential cryptanalysis works as follows:

1. Find a differential characteristic from plaintext pairs ΔP to intermediate pairs ΔU_4 that occurs with high probability and involves only a few bits of U_4. (Do this by chaining together differential characteristics that hold for S-boxes at each round).
2. Choose a bunch of plaintext pairs with difference ΔP and obtain the corresponding ciphertexts.

3. For every possible value of the partial subkey bits that would affect the bits of U_4 appearing in the characteristic: Check how many chosen plaintext pairs satisfy the characteristic.

4. The partial subkey bits that satisfy the characteristic most frequently is probably correct.

2.5 Block cipher modes of operation

2.5.1 Block Cipher Modes of Operation

Supposewe need to encrypt large quantities of data

- With a stream cipher, just encrypt each character.
- With a block cipher, there are some complications if:
  - the input is larger than one block, or
  - the input does not fill an integer number of blocks.

  To deal with these problems for block ciphers, we use a mode of operation.

Definition 2.17: Mode of Operation

Mode of operation is a specification for how to encrypt multiple and/or partial data blocks using a block cipher.

2.5.2 Electronic Codebook (ECB) mode

The obvious approach is to encrypt each ℓ bits independently, where ℓ is the block size. ECB is the most basic mode of a block cipher.

1 $C_i = E(K, P_i)$
2 `Plaintext block` P_i `is encrypted with the key` K `to produce ciphertext block` C_i`.`

Encryption

1 $P_i = D(K, C_i)$
2 `Ciphertext block` C_i `is decrypted with the key` K `to produce plaintext block` P_i`.`

Decryption

Discovery 2.1

This is INSECURE. ECB mode is equivalent to a giant substitution cipher where each ℓ-bit block is a "character". Semantic security is thus immediately violated: One can tell by inspection whether or not two blocks of ciphertext correspond to identical plaintext blocks (violates "no partial information").

Randomised	✗
Padding	Required
Error propagation	Errors propagate within blocks
IV	None
Parallel encryption?	✓
Parallel decryption?	✓
IND-CPA secure?	✗

Comment 2.14

Because it is deterministic, ECB mode is not normally used for bulk encryption

Note 2.2

We could use a new (non-secret) initialization vector for each block of plaintext, but we will have to send an IV for each block of ciphertext, adding linear overhead.

2.5.3 Cipher Block Chaining (CBC) mode

Idea: Use a single (non-secret) initialization vector for the first block of plaintext, and "chain" the (pseudorandom) ciphertext blocks as the next blocks' initialization vectors. The IV is included as part of the ciphertext.

CBC "chains" the blocks together. A random initialisation vector IV is chosen and sent together with the ciphertext blocks.

1 $C_i = E(K, P_i \oplus C_{i-1})$ where $C_0 = IV$
2 P_i is XOR'd with the previous ciphertext block C_{i-1}, and encrypted with key K to produce ciphertext block C_i. For the first plaintext block IV is used for the value C_0.

Encryption

1 $P_i = D(K, C_i) \oplus C_{i-1}$ where $C_0 = IV$
2 C_i is decrypted with the key K, and XOR'd with the previous ciphertext block C_{i-1} to produce plaintext block P_i. As in encryption, IV is used in place of C_0.

Decryption

Randomised	✓
Padding	Required
Error propagation	Errors propagate within blocks and into specific bits of next block
Dropped blocks	Can't decrypt next block, can decrypt subsequent blocks
IV	Must be random
Parallel encryption?	✗
Parallel decryption?	✓
IND-CPA secure?	✓

Comment 2.15

CBC mode is commonly used for bulk encryption and is supported in most libraries and protocols.

2.5.4 Cipher Feedback (CFB) mode

CFB "feeds" the ciphertext block back into the enciphering/deciphering process, thus "chaining" the blocks together.

$$C_i = E(K, C_{i-1} \oplus P_i) \quad \texttt{where} \quad C_0 = IV$$

Encryption

$$P_i = E(K, C_{i-1}) \oplus C_i \quad \texttt{where} \quad C_0 = IV$$

Decryption

Note 2.3

Propagation of channel errors: a one-bit change in C_i produces a one-bit change in P_i, and complete corruption of P_{i+1}.

Randomised	✓
Padding	Not required
Error propagation	Errors occur in specific bits of current block and propagate into next block
IV	Must be unique
Parallel encryption?	✗
Parallel decryption?	✓
IND-CPA secure?	✓

2.5.5 Make a Stream Cipher from a Block Cipher

Output Feedback (OFB) Mode OFB "feeds" the output block back into enciphering/deciphering process. OFB is, in effect, a stream cipher. The keystream is:

$$O_i = E(K, O_{i-1})$$

where $O_0 = IV$ is unique or chosen at random.

1 $C_i = O_i \oplus P_i$

Encryption

1 $P_i = O_i \oplus C_i$

Decryption

The underlying block cipher is only used in **encryption** mode.

Randomised	✓
Padding	Not required
Error propagation	Errors occur in specific bits of current block
IV	Must be unique
Parallel encryption?	✗ (but keystream can be computed in advance)
Parallel decryption?	✗ (but keystream can be computed in advance)
IND-CPA secure?	✓

Note 2.4

OFB mode is a synchronous stream cipher: the sender and receiver must stay aligned.

Counter (CTR) Mode CTR is a synchronous stream cipher. The keystream is generated by encrypting successive values of a "counter", initialised using a nonce (randomly chosen value) N:

$$O_i = E(K, T_i)$$

where $T_i = N \| i$ is the concatenation of the nonce and block number i.

1 $C_i = O_i \oplus P_i$

Encryption

1 $P_i = O_i \oplus C_i$

Decryption

Comment 2.16

Propagation of channel errors: a one-bit change in the ciphertext produces a one-bit change in the plaintext at the same location.

The underlying block cipher is only used in **encryption** mode.

Randomised	✓
Padding	Not required
Error propagation	Errors occur in specific bits of current block
IV	Nonce must be unique
Parallel encryption?	✓
Parallel decryption?	✓
IND-CPA secure?	✓

Note 2.5

- CTR mode is a synchronous stream cipher mode.
- CTR mode is also good for access to specific plaintext blocks without decrypting the whole stream.

2.5.6 Summary of Mode Properties

	ECB	CBC	CFB	OFB	CTR
IND-CPA secure?	✗	✓	✓	✓	✓
Randomised	✗	✓	✓	✓	✓
Padding required?	✓	✓	✗	✗	✗
Error propagation within block? next? subsequent?	✓✗✗	✓✓✗	✓✓✗	✓✗✗	✓✗✗
Dropped block decrypt next? subsequent?	✓✓	✗✓	✗✓	✓*✓*	✓*✓*
IV	none	random	unique	unique	unique
Parallel encryption?	✓	✗	✗	✗	✓
Parallel decryption?	✓	✓	✓	✗	✓
Pre-compute encryption?	✗	✗	✗	✓	✓
Pre-compute decryption?	✗	✗	✗	✗	✓

* if the receiver detects a dropped block and advances the keystream appropriately.

3 Integrity and More Symmetric Primitives

3.1 Hash functions

Hashing functions are very difficult to design because of very stringent security and performance requirements. The main options available today are: SHA-224, SHA-256, SHA-384, SHA-512, and SHA-3.

3.1.1 Overview of Hash Functions

Comment 3.1

A hash function is a checksum designed to be safe from malicious tampering.

Definition 3.1: Hashing Function

A **hash function** is a mapping H such that:

1. H maps inputs of arbitrary lengths to outputs of length n, where n is fixed. (More generally, H maps elements of a set S to a set T where $|S| > |T|$.)
2. $H(x)$ can be efficiently computed for all $x \in \{0,1\}^*$.

Comment 3.2

H is called an n-bit hash function. $H(x)$ is called the hash value, hash, or message digest of x.

Note 3.1

The description of a hash function is public. There are no secret keys.

Example 3.1: $SHA256 : \{0,1\}^* \to \{0,1\}^{256}$

We have

SHA256("Hello there") = `0x4e47826698bb4630fb4451010062fadbf85d61427cbdfaed7ad0f23f239bed89`

SHA256("Hello There") = `0xabf5dacd019d2229174f1daa9e62852554ab1b955fe6ae6bbbb214bab611f6f5`

SHA256("Welcome to CO487″) = `0x819ba4b1e568e516738b48d15b568952d4a35ea73f801c873907d3ae1f5546fb`

SHA256("Welcome to CO687″) = `0x404fb0ee527b8f9f01c337e915e8beb6e03983cfd9544296b8cf0e09c9d8753d`

Typical Cryptographic Requirements (informally)

- Preimage resistance: Hard to invert given just an output.
- 2nd preimage resistance: Hard to find a second input that has the same hash value as a given first input.
- Collision resistance: Hard to find two different inputs that have the same hash values.

Definition 3.2: Preimage Resistance

Let m be a positive integer. We say that H is **preimage resistant** for messages of length m if, given $y = H(x)$ for x picked uniformly at random from $\{0,1\}^m$, it is computationally infeasible to find (with non-negligible success probability) any input z such that $H(z) = y$.

Definition 3.3: Second Preimage Resistance

Let m be a positive integer. We say that H is **second preimage resistant** for messages of length m if, given an input $x \in_R \{0,1\}^m$, it is computationally infeasible to find (with non-negligible probability of success) a second input $x' \neq x$ such that $H(x) = H(x')$.

Comment 3.3

$x \in_R \{0,1\}^m$ means x chosen uniformly at random from $\{0,1\}^m$.

Definition 3.4: Collision Resistance

Collision resistance means that it is computationally infeasible to find (with non-negligible probability of success) two distinct inputs x, x' such that $H(x) = H(x')$.

Note 3.2

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

Discovery 3.1

How are second preimage resistance and collision resistance different?

- Collision resistance: the attacker has freedom to pick both x and x'
- Second preimage resistance: the attacker given some x and has to find an x'

Definition 3.5: One-way Hash Function

A hash function that is preimage resistant is sometimes called a **one-way hash function** (OWHF).

Definition 3.6: Cryptographic Hash Function

A hash function that is preimage-, second preimage-, and collision-resistant is called a **cryptographic hash function**.

Some Applications of Hash Function

1. Password protection on a multi-user computer system;

2. Modification Detection Codes (MDCs);
3. Message digests for digital signature schemes;
4. Message Authentication Codes (MACs);
5. Pseudorandom bit generation;
6. Key derivation function (KDF).

Note 3.3

Collision resistance is not always necessary. Depending on the application, other properties may be needed, for example 'near-collision resistance', 'partial preimage resistance', etc.

Relationships between Properties In cryptography we often want to show theorems like the following:

Theorem 3.1

If scheme S satisfies security property X, then scheme T (built using S) satisfies security property Y.

Another way of phrasing this is:

Theorem 3.2

If there is no efficient adversary that breaks security property X of scheme S, then there is no efficient adversary that breaks security property Y of scheme T.

Comment 3.4

It is often more convenient to prove the (equivalent) contrapositive statement:

Theorem 3.3

Suppose there exists an efficient adversary B that breaks security property Y of scheme T. Then there exists an efficient adversary A (that uses B) that breaks security property X of scheme S.

To prove the contrapositive statement, our task is come up with the algorithm A, assuming that algorithm B exists with the desired property.

Collision resistance implies 2nd preimage resistance

Theorem 3.4

Collision resistance implies 2nd preimage resistance.

Written slightly differently:

Theorem 3.5

If H is collision resistant, then H is 2nd preimage resistant.

Written in the contrapositive form:

Theorem 3.6

If there exists an efficient algorithm B that breaks the 2nd preimage resistance of H, then there exists an efficient algorithm A that breaks the collision resistance of H.

Proof. Suppose B is an efficient algorithm that breaks 2nd preimage resistance of H for messages of length m. Algorithm A proceeds as follows:

1 `Select` $x \in_R \{0,1\}^m$.
2 `Now, since` H `is not 2nd preimage resistant for messages of length` m`, we can use` B `to efficiently find` $x' \in \{0,1\}^*$ `with` $x' \neq x$ `and` $H(x') = H(x)$.
3 `Thus, we have efficiently found a collision` (x, x') `for` H.

Hence H is not collision resistant. □

Theorem 3.7

2nd preimage resistance does not guarantee collision resistance.

Proof. Let $H : \{0,1\}^* \to \{0,1\}^n$ be a 2nd preimage resistant hash function for messages of length m. Consider $\overline{H} : \{0,1\}^* \to \{0,1\}^n$ defined by

$$\overline{H}(x) = \begin{cases} H(0^m), & \text{if } x = 1^m, \\ H(x), & \text{if } x \neq 1^m. \end{cases}$$

- $\overline{H}$ is still 2nd preimage resistant: Suppose the random 2nd preimage challenge is $x \in \{0,1\}^m$. With high probability $x \neq 0^m$. So if a given adversary finds x' such that $\overline{H}(x) = \overline{H}(x')$, then x' is also a 2nd preimage of x in H.
- $\overline{H}$ is not collision resistant: $\overline{H}(0^m) = \overline{H}(1^m)$. □

Theorem 3.8

Collision resistance does not guarantee preimage resistance.

Proof. Let $H : \{0,1\}^* \to \{0,1\}^n$ be a collision-resistant hash function.
Consider $\overline{H} : \{0,1\}^* \to \{0,1\}^{n+1}$ defined by

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

- $\overline{H}$ is collision-resistant. *(Sketch: No collisions on outputs starting with* 1*. Collisions on outputs starting with* 0 *imply collisions in* H*.)*
- $\overline{H}$ is not preimage-resistant for messages of length n. *(Easy to invert outputs starting with* 1*.)* □

Corollary 3.1

Collision resistance and randomness (i.e., all hash values have roughly the same number of preimages) implies preimage resistance.

Proof. Suppose that H is not preimage resistant. Select random $x \in \{0,1\}^m$ for some $m \geq 2n$ and compute $y = H(x)$. Find a preimage x' of y; this is successful with some non-negligible probability. Then, if H is uniform, we expect that $x' \neq x$ with very high probability. Thus, we have efficiently found a collision (x, x') for H, hence H is not collision resistant. □

Generic Attacks on Hash Functions A **generic attack** on hash functions $H : \{0,1\}^* \to \{0,1\}^n$ does not exploit any properties a 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 $x \in \{0,1\}^*$, the value $y = H(x)$ was chosen by selecting y uniformly at random from $\{0,1\}^n$ (written $y \in_R \{0,1\}^n$).

Definition 3.7: Generic Attack for Finding Preimages

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

Note 3.4

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

Definition 3.8: Generic Attack for Finding Collisions

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

Note 3.5

- Expected number of steps: $\sqrt{\pi 2^n/2} \approx \sqrt{2n}$ (by birthday paradox). (Here, a step is a hash function evaluation.)
- It has been proven that this generic attack for finding collisions is optimal in terms of the number of hash function evaluations.
- Expected space required: $\sqrt{\pi 2^n/2} \approx \sqrt{2n}$.
- This attack is infeasible if $n \geq 160$.

Theorem 3.9: The birthday paradox

Suppose q values are picked uniformly at random, with replacement, from a set of size N. Then the probability of no collisions among the q selected values is

$$\Pr(\text{no collision}) = \prod_{i=1}^{t-1} \left(1 - \frac{i}{N}\right)$$

The probability of a collision among the q selected values is approximately

$$\Pr(\text{collision}) \approx \frac{q^2}{N}.$$

3.1.2 Pollard's rho algorithm for collision-finding

1 `Set` $x_{i+1} = H(x_i)$
2 `Search for` $x_i = x_j$
3 `Then` $H(x_{i-1}) = H(x_{j-1})$

Naive implementation needs to store all the x_i values to know when it has looped.

3.1.3 VW Parallel Collision Search

Define a distinguished point to be a bitstring whose first k bits are zeros.

1 `Compute` $x_i = H(x_{i-1})$
2 `Store` x_i `if and only if` x_i `is a distinguished point.`
3 `If` x_i `is a distinguished point, compare it to all previously stored distinguised points, to see if there is a match.`
4 `If there is a match, ''backtrack'' to find the collision.`

Discovery 3.2

Compared to brute-force search, VW Parallel Collision search:

- Reduces storage by a factor of 2^k
- Increases computational time by an additive 2^k
- Parallelizes trivially to multiple machines or processors

3.1.4 Iterated Hash Function

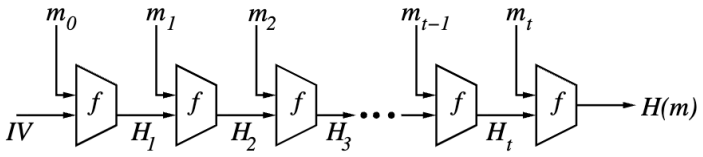
Idea: Build a hash function from a block cipher

Algorithm 3.1

Let E_k be an m-bit block cipher with n-bit key k. Let IV be a fixed m-bit initializing value. To compute $H(x)$, do:

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

The Merkle–Damgard construction



Components:

- Fixed initialization vector $IV \in \{0,1\}^n$.
- Compression function $f : \{0,1\}^{n+r} \to \{0,1\}^n$ (efficiently computable).

Algorithm 3.2

To compute $H(m)$ where m has bitlength $b < 2^r$ do:

1. Break up m into r-bit blocks:
$$\overline{m} = (m_0, m_1, m_2, \ldots, m_{t-1})$$
padding the last block with 0 bits if necessary.
2. Define m_t, the *length-block*, to hold the right-justified binary representation of b.
3. Define $H_0 = IV$.
4. Compute $H_{i+1} = f(H_i, m_i)$ for $i = 0, 1, 2, \ldots, t$.
5. The H_i's are called *chaining variables*.
6. Return $H(m) = H_{t+1}$.

Theorem 3.10: Merkle

If the compression function f is collision resistant, then the hash function H is also collision resistant.

Proof Sketch. Suppose $H(m) = H(m')$ but $m \neq m'$. Suppose for simplicity that $|m| = |m'|$. Let the blocks of m and m' be

$$m = m_0\, m_1 \cdots m_t \qquad \text{and} \qquad m' = m'_0\, m'_1 \cdots m'_t$$

Since $m \neq m'$, there exists an index $i \in \{0, \ldots, t\}$ such that $m_i \neq m'_i$. Since $H(m) = H(m')$, we have

$$f(m_t, H_t) = f(m'_t, H'_t)$$

If $(m_t, H_t) \neq (m'_t, H'_t)$, then we found a collision in $f \Rightarrow$ done. If $(m_t, H_t) = (m'_t, H'_t)$, then recurse on H_t, H'_t. In particular, if $H_t = H'_t$, then

$$f(m_{t-1}, H_{t-1}) = f(m'_{t-1}, H'_{t-1}).$$

If $(m_{t-1}, H_{t-1}) \neq (m'_{t-1}, H'_{t-1})$, then we found a collision in $f \Rightarrow$ done. We can construct an inductive argument that, if $H_j = H'_j$, then either

$$(m_{j-1}, H_{j-1}) \neq (m'_{j-1}, H'_{j-1})$$

which is a collision in f, or $H_{j-1} = H'_{j-1}$. By assumption, $\exists\, i$ such that $m_i \neq m'_i$. Thus we will eventually find a collision. □

MDx, SHA-1, SHA-2 Families

Definition 3.9: MDx

MDx is a family of iterated hash functions using the Merkle–Damgard construction.

Definition 3.10: SHA

Secure Hash Algorithm (SHA) was designed by NSA and published by NIST in 1993

High-Level Description of SHA-1:

- Iterated hash function (Merkle–Damgrard construction).
- Chaining value size $n = 160$, message block size $r = 512$.
- IV is a public constant $\in \{0,1\}^{160}$.
- Compression function $f : \{0,1\}^{160+512} \to \{0,1\}^{160}$.
- *Input*: bitstring x of arbitrary bitlength $b \geq 0$.
- *Padding*: pad x with a single 1 followed by 0s so that its bitlength is 64 less than a multiple of 512; then append a 64-bit representation of the plaintext bitlength b.
- *Output*: 160-bit hash value $H(x)$ of x.

SHA-1 compression function – high level idea:

1. Load chaining value $(H_1, H_2, H_3, H_4, H_5) \in \{0,1\}^{5\times 32}$ into five 32-bit state variables.

2. Expand the 512-bit message block into 80 32-bit variables X_i using the message schedule.
   - Small number of XORs and shifts, introducing some diffusion.
3. Do 80 rounds of state updates, combining the state with the message schedule. In each round: (a) do some cyclic shifts, (b) apply one of four different nonlinear functions to the state, (c) mix in a portion of the message state, and (d) do some 32-bit integer additions.
   - Cyclic shifts help diffuse data across the state to achieve the avalanche effect.
   - Using both bitwise XORs and 32-bit integer additions breaks linearity.
4. Combine the final state with the input chaining value to produce the output chaining value.

SHA-1 Notation:

A, B, C, D, E	32-bit quantities.
$+$	addition modulo 2^{32}.
$A \hookleftarrow s$	cyclic left rotation by s bit-positions.
$\neg A$	bitwise complement.
AB	bitwise AND.
$A \vee B$	bitwise inclusive-OR.
$A \oplus B$	bitwise exclusive-OR.
$F(A, B, C)$	$AB \vee (\neg A)C$ (select B or C using A).
$G(A, B, C)$	$AB \vee AC \vee BC$ (majority rule).
$H(A, B, C)$	$A \oplus B \oplus C$ (bitwise add A, B, C).

- $IV = (h_1, h_2, h_3, h_4, h_5)$ where $h_1 =$ `0x67452301`, $h_2 =$ `0xefcdab89`, $h_3 =$ `0x98badcfe`, $h_4 =$ `0x10325476`, $h_5 =$ `0xc3d2e1f0`.
- More constants: $k_1 =$ `0x5a827999`, $k_2 =$ `0x6ed9eba1`, $k_3 =$ `0x8f1bbcdc`, $k_4 =$ `0xca62c1d6` (digits of $\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{10}$).

SHA-1 compression function details:

$$f : \{0,1\}^{160} \times \{0,1\}^{512} \to \{0,1\}^{160}$$

is the compression function.

1. Let $(H_1, H_2, H_3, H_4, H_5) \in \{0,1\}^{5\times 32=160}$ be the left input (chaining value or IV). Set $(A, B, C, D, E) \leftarrow (H_1, H_2, H_3, H_4, H_5)$.
2. Let $M \in \{0,1\}^{512}$ be the message block input (sixteen 32-bit words).
   Message schedule:

$$\begin{aligned} &\text{For } j = 0 \text{ to } 15: \quad X_j \leftarrow M_j, \\ &\text{For } j = 16 \text{ to } 79: \quad X_j \leftarrow \left(X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}\right) \overset{1}{\hookleftarrow}. \end{aligned}$$

Note 3.6

The rotate operation $\hookleftarrow 1$ is not present in SHA. This is the only difference between SHA and SHA-1.

3. **Rounds 1–20** For j from 0 to 19 do:

$$t \leftarrow \big((A \hookleftarrow 5) + F(B,C,D) + E + X_j + k_1\big), \qquad (A,B,C,D,E) \leftarrow \big(t,\ A,\ B \hookleftarrow 30,\ C,\ D\big)$$

Rounds 21–40 For j from 20 to 39 do:

$$t \leftarrow \big((A \hookleftarrow 5) + H(B,C,D) + E + X_j + k_2\big), \qquad (A,B,C,D,E) \leftarrow \big(t,\ A,\ B \hookleftarrow 30,\ C,\ D\big)$$

Rounds 41–60 For j from 40 to 59 do:

$$t \leftarrow \big((A \hookleftarrow 5) + G(B,C,D) + E + X_j + k_3\big), \qquad (A,B,C,D,E) \leftarrow \big(t,\ A,\ B \hookleftarrow 30,\ C,\ D\big)$$

Rounds 61–80 For j from 60 to 79 do:

$$t \leftarrow \big((A \hookleftarrow 5) + H(B,C,D) + E + X_j + k_4\big), \qquad (A,B,C,D,E) \leftarrow \big(t,\ A,\ B \hookleftarrow 30,\ C,\ D\big)$$

4. Update H values:

$$(H_1,H_2,H_3,H_4,H_5) \leftarrow (H_1 + A,\ H_2 + B,\ H_3 + C,\ H_4 + D,\ H_5 + E)$$

Attacks on MDx, SHA-1, SHA-2 Generic Attacks: These attacks are *generic* because they work on any hash function.

- To find a preimage for an ℓ-bit hash:
  - Try random inputs until the desired hash is found.
  - Requires $O(2^\ell)$ operations on average.
- To find a collision for an ℓ-bit hash:
  - Try random inputs until two matching hashes are found.
  - Requires $O(2^{\ell/2})$ operations on average (birthday paradox).
- Generic attacks against MD5:
  - 2^{64} operations (collision).
  - 2^{128} operations (preimage).
- Generic attacks against SHA-1:
  - 2^{80} operations (collision).
  - 2^{160} operations (preimage).

Non-generic Attacks:

- MD4 (RSA Laboratories):
  - Collisions found in 2^{15} steps (Dobbertin, 1996)
  - ... found in 2^{3} steps (Wang et al., 2005)
  - ... found in 2^{0} steps (Sasaki et al., 2009)
  - Preimages found in 2^{102} steps (Leurent, 2008)
- MD5 (RSA Laboratories):
  - Collisions found in MD5 compression function (Dobbertin, 1996)
  - ... found in 2^{39} steps (Wang and Yu, 2004)
  - ... found in 31 seconds on a notebook computer (Klima, 2006)
  - Preimages (theoretical) in $2^{123.4}$ steps (Sasaki and Aoki, 2009)
- MD5 should not be used if collision resistance is required, but is not horrible as a one-way hash function.
- MD5 remains widely used in legacy software
- MD5 shows up about 850 times in Windows source code,

3.2 Message authentication codes and authenticated encryption

3.2.1 Message authentication codes

Definition 3.11: Message authentication code

A **message authentication code** (MAC) scheme is an efficiently computable function

$$M : \{0,1\}^{\ell} \times \{0,1\}^{*} \to \{0,1\}^{n}$$

written $M(k, m) = t$ where k is the key, m is the message, and t is the tag.

The application is to provide data integrity and data origin authentication:

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

Comment 3.5

To avoid replay, add a timestamp, or sequence number.

Comment 3.6

Widely used in communication protocols. No confidentiality or non-repudiation.

Question 3.1.

What does security mean?

Let k be the secret key shared by Alice and Bob. The adversary knows everything about the MAC scheme except the value of k.

Definition 3.12: MAC Security

A MAC scheme is secure if:

- Given some number of MAC tags M(k, mi) for messages mi chosen adaptively by the adversary (interaction),
- it is computationally infeasible (computational resources)
- to compute (with non-negligible probability of success) the value of $M(k, m)$ for any message $m \neq m_i$ (goal).

In other words, a MAC scheme is secure if it is existentially unforgeable against chosen-message attack.

Generic attacks `Guessing the MAC of a message` m:

1. Select $y \in \{0,1\}^n$ and guess that $M(k, m) = y$.
2. Assuming that $M(k, \cdot)$ is a random function, the probability of success is $1/2^n$.
3. **Note 3.7**

   Guesses cannot be directly checked without interaction.

4. Depending on the application where the MAC algorithm is employed, one could choose n as small as 32 (say). In general, $n \geq 128$ is preferred.

`Exhaustive search on the key space`:

1. Given r known message-MAC pairs: $(m_1, t_1), \ldots, (m_r, t_r)$, one can check whether a guess k of the key is correct by verifying that $M(k, m_i) = t_i$, for all i.
2. Assuming that the $M(k, \cdot)$'s are random functions, the expected number of keys for which the tags verify is $K = 2^\ell / 2^{nr}$.
3. Example: If $\ell = 56$, $n = 64$, $r = 2$, then $K \approx 1/2^{72}$.
4. Requires $\approx 2^\ell$ computations. Exhaustive search is infeasible if $\ell \geq 128$.

Constructing MACs

Algorithm 3.3: CBC-MAC

Let E be an n-bit block cipher with key space $\{0,1\}^\ell$. To compute $M(k, m)$:

```
1  Divide (padded) m into n-bit blocks m_1, m_2, ..., m_r
2  Compute H_1 = E(k, m_1).
3  For 2 ≤ i ≤ r, compute H_i = E(k, H_{i-1} ⊕ m_i).
4  Then M(k, m) = H_r.
```

CBC-MAC

Theorem 3.11: Bellare, Kilian & Rogaway 1994

(Informal statement): 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 algorithm.

Discovery 3.3

CBC-MAC (as described here without additional measures) is not secure if variable length messages are allowed. Here is a chosen-message attack on CBC-MAC:

1 Let m_1 be an n-bit block.
2 Let (m_1, t_1) be a known message-MAC pair.
3 Request the MAC t_2 of the message t_1.
4 Then t_2 is also the MAC of the 2-block message $(m_1, 0)$. (Since $t_2 = E_k(E_k(m_1))$.)

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

Question 3.2.

Question: How to use them to construct secure MACs?

MACs based on hash functions: Secret prefix method.

Definition 3.13: Secret Prefix Method

$M(k, m) = H(K \| m)$.

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

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

MACs based on hash functions: Secret suffix method.

Definition 3.14: Secret Suffix Method

$M(k, m) = H(m \| K)$.

The attack on the secret prefix method does not work here. Here is a different attack which works if H is not collision resistant:

1 Suppose that a collision (m_1, m_2) can be found for H (i.e., $H(m_1) = H(m_2)$). Assume that m_1 and m_2 both have bitlengths that are multiples of r.

2 Then $H(m_1\|K) = H(m_2\|K)$, and so $M(k, m_1) = M(k, m_2)$.
3 The MAC for m_1 can be requested, giving the MAC for m_2.
4 Hence if H is not collision resistant, then the secret suffix method MAC is insecure.

MACs based on hash functions: Envelope method a.k.a. Sandwich method.

Definition 3.15: Envelope Method

$M(k, m) = H(K\|m\|K)$.

Theorem 3.12: Koblitz and Menezes, 2013

Suppose that the compression function used in H is a secure MAC with fixed length messages and a secret IV as the key. Then Envelope MAC with m padded to a multiple of the block length of H is a secure MAC algorithm.

HMAC: Define 2 r-bit strings (in hexadecimal notation): ipad = `0x36`, opad = `0x5C`; each repeated $r/8$ times.

Definition 3.16: HMAC

$$M(k, m) = H\Big((K \oplus \text{opad})\|H((K \oplus \text{ipad})\|m)\Big)$$

Theorem 3.13: Bellare, Canetti & Krawczyk, 1996

Suppose that the compression function 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.

3.2.2 Authenticated encryption

In many situations, one wishes to achieve both data confidentiality and data authentication, so we combine encryption and authentication.

Comment 3.7

Some obvious remarks:

- Don't use the same key for both sending and receiving.
- Don't use the same key for authentication and encryption.

Definition 3.17: Security Goals

`Confidentiality`: semantic security under chosen plaintext or chosen ciphertext attack
`Integrity`: existential unforgeability under chosen message attack

- INT-PTXT (integrity of plaintexts): inability to generate a ciphertext that successfully decrypts to a new plaintext (with chosen message attack powers)

- INT-CTXT (integrity of ciphertexts): inability to generate a new ciphertext that successfully decrypts (with chosen message attack powers)

Example 3.2

- MAC-then-encrypt: SSL/TLS up to v1.2
- encrypt-then-MAC: IPsec
- encrypt-and-MAC: SSH

Encrypt and MAC (E&M) Consider encrypt-and-MAC:

$$\text{compute } c = \text{Enc}(m) \text{ and } t = \text{MAC}(m), \text{ transmit } c\|t$$

When decrypting, the recipient checks that the MAC is correct.

Note 3.8

Problem: MACs are not required to ensure confidentiality. For example, the MAC might leak one plaintext bit, and still be "secure" as a MAC. Violates semantic security!

MAC then encrypt (MtE) Consider MAC-then-encrypt:

$$\text{compute } t = \text{MAC}(m) \text{ and } c = \text{Enc}(m\|t), \text{ transmit } c$$

When decrypting, the recipient checks that the MAC is correct.

Note 3.9

Problem: SKESs are not required to ensure integrity. For example, changing the ciphertext might not change the plaintext for certain values of plaintext. Violates integrity (of ciphertexts).

One can often then also learn information about the plaintext.

Comment 3.8

Even if the SKES is secure and the MAC is secure, the MAC-then-encrypt combination might be insecure.

Encrypt then MAC (EtM) EtM is safe!

Theorem 3.14: Canetti & Krawczyk, 2001

Suppose the SKES is semantically secure under chosen plaintext attack, and the MAC is existentially unforgeable under chosen message attack. Then encrypt-then-MAC is semantically secure and provides integrity (specifically "ciphertext unforgeability under chosen plaintext attack").

An extra complication is padding. Often times, when using block ciphers, a message needs to be padded before encrypting, to align it with block boundaries. Now comes a question to us:

Question 3.3.

Should the padding be included in the MAC input?

Suppose we are designing SSL/TLS using MAC-then-encrypt. We now have two options:

`Option 1.` Apply MAC, then pad, then encrypt:

$$\text{compute } t = \text{MAC}(m) \text{ and } c = \text{Encrypt}(m\|p\|t), \text{ transmit } c$$

`Option 2.` Apply padding, then MAC, then encrypt:

$$\text{compute } t = \text{MAC}(m\|p) \text{ and } c = \text{Encrypt}(m\|p\|t), \text{ transmit } c$$

SSL/TLS uses Option 1. Unfortunately, this is the wrong choice.

Theorem 3.15: The Cryptographic Doom Principle by Moxie Marlinspike, 2011

If you have to perform any cryptographic operation before verifying the MAC on a message you've received, it will somehow inevitably lead to doom.

The fastest way to achieve authenticated encryption is to use a block cipher mode of operation with authentication built-in:

- CCM mode (Counter with CBC-MAC)
- EAX mode
- GCM mode (Galois/Counter Mode)
- OCB mode (Offset Codebook Mode) [patented!]

These modes of operation require a block cipher, but not a separate MAC (authentication functionality is built-in).

Example: Galois Counter Mode GCM ciphertexts (ignoring the authentication tag) are identical to counter (CTR) mode ciphertexts.

Comment 3.9

In particular, the last ciphertext block is truncated if the plaintext length is not an integral number of blocks.

Authentication tags are computed in

$$GF(2^{128}) = \mathbb{F}_2[x]/(x^{128} + x^7 + x^2 + x + 1)$$

Hence, GCM requires a 128-bit block size (e.g. AES). "Auth Data 1" is a 128-bit block of authenticated unencrypted data, viewed as an element of $GF(2^{128})$. More than one such block is supported, but only one

is shown. H is defined as

$$H = E_k(0^{128}) = E_k(0) \in GF(2^{128})$$

Computing H requires knowledge of the key. Computing authentication tags can be parallelized using field arithmetic.

3.3 Key Derive Functions and Pseudorandom Functions

Definition 3.18: Pseudorandom generator

A **pseudorandom generator** is a deterministic function that takes as input a uniform random seed $k \in \{0,1\}^\lambda$ and outputs a random-looking binary string of length ℓ.

$$PRG : \{0,1\}^\lambda \to \{0,1\}^\ell$$

Definition 3.19: Pseudorandom function

A **pseudorandom function** is a deterministic function that takes as input a uniform random seed $k \in \{0,1\}^\lambda$ and a (non-secret) label in $\{0,1\}^*$ and outputs a random-looking binary string of length ℓ.

$$PRF : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\ell$$

They should achieve indistinguishability: Assuming the seed is uniformly random on $\{0,1\}^\lambda$, it should be computationally infeasible for an adversary to distinguish the output of a PRG/PRF from a uniformly random string.

- For PRGs, the adversary gets either the real output of the PRG under an unknown seed, or a random output, and must decide which.
- For PRFs, the adversary can make many calls to an oracle where the adversary can supply a label, and either always gets the real output of the PRF using the same unknown seed applied to the label, or always gets a randomly chosen output (for distinct labels), and must decide which.

Comment 3.10

PRGs and PRFs assume that the random seed is a truly random (uniform) secret.

Definition 3.20: Key derivation function

A key derivation function is a deterministic function that takes as input a (not-necessarily uniform) random seed $k \in \{0,1\}^\lambda$ and a (non-secret) label in $\{0,1\}^*$ and outputs a random-looking binary string of length ℓ.

$$KDF : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\ell$$

Note 3.10

Difference between KDFs and PRFs: KDF output should be indistinguishable from random even if the key k is non-uniform but sufficiently high entropy.

3.3.1 Uses of PRGs, PRFs, KDFs

- PRGs: expanding a strong uniform short key into a long pseudorandom key (e.g., stream cipher)
- PRFs: deriving many pseudorandom keys from a single short uniform key
  - If Alice and Bob share one key k, but they need two keys for their application (e.g., one for encryption, one for MAC), they can compute $k' = \mathsf{PRF}(k, \text{“label for encryption”}), k'' = \mathsf{PRF}(k, \text{“label for MAC”})$.
- KDFs: turning longer non-uniform keys into shorter uniform-ish keys

3.3.2 HMAC as a PRG/PRF/KDF

- PRG: compute $\mathsf{HMAC}_k(1) \| \mathsf{HMAC}_k(2) \| \mathsf{HMAC}_k(3) \| \ldots$
- PRF: compute $\mathsf{HMAC}_k(label)$
- KDF: compute $\mathsf{HMAC}_{label}(k)$

Note 3.11

Better way: special construction HKDF based on HMAC which first “extracts” high entropy secret from the keying material (similar to a KDF) then “expands” it to arbitrary length output with a label (like a PRF).

3.3.3 Application: Key stretching

Our goal is to convert a (possibly) short and weak key into a long(er) and (more) secure key.

Useful for:

- Password storage in databases (harder to brute-force)
- Fixed-length keys (user types in a passphrase which is converted into a 256-bit key)

Idea: “slow down” hashing for both honest users and attackers.

Password-Based Key Derivation Function 2 (PBKDF2) We have

$$k = \mathsf{PBKDF2}(F, p, s, c, \ell)$$

where

$$
\begin{aligned}
F &= \text{keyed hash function} \\
p &= \text{passphrase} \\
s &= \text{salt} \\
c &= \text{number of iterations} \\
\ell &= \text{output length}
\end{aligned}
$$

This function is supposed to be slow. Larger iteration counts yield more security (and slower performance). Recall $c =$ number of iterations, and $\ell =$ output length. So

$$\mathsf{PBKDF2}(F, p, s, c, \ell) = T_1 \| T_2 \| \cdots \| T_\ell$$

where

$$
\begin{aligned}
T_i &= U_{i,1} \oplus U_{i,2} \oplus \cdots \oplus U_{i,c} \\
U_{i,1} &= F(p,\, s \| i) \\
U_{i,2} &= F(p,\, U_{i,1}) \\
&\vdots \\
U_{i,c} &= F(p,\, U_{i,c-1})
\end{aligned}
$$

Each T_i requires c calls to F. Slow down computation by choosing large c.

Example 3.3

WPA2 (Wi-Fi Protected Access) uses the following function to derive a 256-bit key (truncated from 320 bits) from a passphrase.

$$k = \mathsf{PBKDF2}(\text{HMAC-SHA1}, \text{passphrase}, \text{ssid}, 4096, 2)$$

Example 3.4

iOS 4 used PBKDF2 with $c = 10000$. LastPass used $c \geq 100000$.

3.4 Password Hashing

Definition 3.21: User Authentication

Authenticators can be categorised as:

- Knowledge-Based (Something you know)
- Object-Based (Something you have)

- ID-Based (Something you are)
- Location-based (Somewhere you are)

Definition 3.22: Multifactor Authentication

Multi-factor authentication uses combinations from multiple different categories of authenticators

Definition 3.23: Entropy

Entropy measures the uncertainty in values generated from a random process

Comment 3.11

Entropy is a (somewhat okay) measure of password strength.

Suppose a process X generates n values $x_1, \ldots, x_n$ with probabilities $p_1, \ldots, p_n$. Formula for entropy of process X:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i).$$

Or alternatively:

$$H(X) = -p_1 \log_2(p_1) - p_2 \log_2(p_2) - \cdots - p_n \log_2(p_n).$$

Note 3.12

Simple way of thinking about it: If a password is chosen uniformly at random from a set of size 2^n, then its entropy is n bits, and we require around 2^{n-1} guesses on average to find it.

Example 3.5: Calculating Entropy

Suppose we have a dictionary of 16 words.

Scenario 1: Passwords generated uniformly at random from the dictionary — i.e., each password is equally likely

Scenario 2: Passwords were *not* generated uniformly at random from the dictionary — i.e., some passwords are more likely than others

If uniform, we have

$$\begin{aligned} H(X) &= -\sum_{i=1}^{16} p_i \log_2(p_i) \\ &= -\sum_{i=1}^{16} \frac{1}{16} \log_2\left(\frac{1}{16}\right) = 4. \end{aligned}$$

If you are trying to guess the password, you need to make about $2^{4-1} = 8$ guesses on average.

3.4.1 Strong Passwords on Servers

There are two steps about storing passwords on servers (take 1):

(Registration) Store username and password in database

(Login)
(a) User supplies username and purported password
(b) Look up username and real password in database
(c) Check if purported password = real password

We demand to store passwords securely. Security requirements for system files storing passwords:

- C: Can non-administrators read the password database? What useful information is in there?
- I: Can the password file be modified? Can unauthorised modification be detected?
- A: Need to be available when required for verification

Note 3.13

No non-repudiation if password is known to system (or to others outside the system).

Discovery 3.4

Confidentiality of passwords

- Storage (on authentication server)
- Transmission (between client and server over network)
- Use (display on screen when being entered?)

Login and registration, take 2,

(Registration) Store username and an encrypted version of the password in database

(Login)
(a) User supplies username and purported password
(b) Look up username and encrypted password in database
(c) Decrypt the stored password to recover the real password
(d) Check if purported password = real password

Result 3.1

If someone learns the key, they can decrypt the database and recover all the passwords.

Login and registration, take 3,

(Registration) Store username and an irreversible transformation (hash) of the password in database

(Login)
(a) User supplies username and purported password
(b) Look up username and hash in database
(c) Apply same irreversible transformation to the purported password
(d) Check if hash of purported password = hash of real password

Benefits and Drawbacks of Password Hashing Benefite:

1. Compromise of the database doesn't reveal the user's password
2. Almost no overhead for storage and login

Drawbacks:

1. Can't recover passwords for users who forget
2. Attackers could create a table of password hashes to compare against database
3. Can learn if two users use the same password (even if you don't know what it is)

3.4.2 Password hash cracking

Suppose you learn that the hash of Alice's password is `3e2e95f5ad970eadfa7e17eaf73da97024aa5359`.

Our goal is to find Alice's password.

We can of course use brute force. We could also attack using hash tables:

Definition 3.24: Hashing Table

Hash table: A table containing hashes of many/all possible passwords.

Example 3.6

Hash tables allow for instant cracking of a password hash, but require a massive amount of storage. For intance, if password set: 8 character passwords, $26 + 26 + 10 = 62$ characters ($62^8 = 2^{47.6}$ passwords). SHA-1 hash table would take 160 bits = 20 bytes per password, so approx. $2^{52.4}$ bytes = 6 petabytes.

Question 3.4.

Can we find a time-memory trade-off where we can store less, but not increase time too much?

We can attack using rainbow table.

Definition 3.25: Rainbow Table

Rainbow tables are an example of a time-space tradeoff using hash chains.

Example 3.7

Ophcrack and RainbowCrack are examples of software that can crack passwords using rainbow tables.

Constructing a rainbow table

Algorithm 3.4

1. Pick a random password
2. Construct a hash chain (hash with H, map hash back to the password space with "reduction function" R)
3. Store the start and end of the chain
4. Repeat many times

Comment 3.12

The reduction function R doesn't have to be a cryptographic function – just something that maps from the hash space back to the password space with not too many collisions.

Login and registration, take 4,

(Registration)
(a) Pick a random $\geq$ 80-bit salt
(b) Store username, salt, and $H(\text{password}, \text{salt})$ in database where H is a cryptographic hash function

(Login)
(a) User supplies username and purported password'
(b) Look up username, salt, and hash in database
(c) Check if $H(\text{password'}, \text{salt}) = $ stored hash

Result 3.2

Salting protects against precomputed hash tables / rainbow tables since you would need a different table for each salt.

- Salting doesn't make it harder to do a brute force search against a single hash
- Salting does make brute-force attacks against many hashes harder because you can't reuse the work from one attack on another attack.

Definition 3.26: Password hardening

You can slow down brute-force attacks even more by **hashing the password multiple times**.

Login and registration, take 5,

(Registration)
(a) Pick a random $\geq$ 80-bit salt
(b) Store username, salt, and $H(\text{password}, \text{salt})$ in database where H is a password hardening function

(Login)
(a) User supplies username and purported password'
(b) Look up username, salt, and hash in database
(c) Check if $H(\text{password'}, \text{salt}) = $ stored hash

4 Public key cryptography

4.1 Overview

Key Establishment Problem: 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 (confidential + authentic) channel). The secure channel could be:

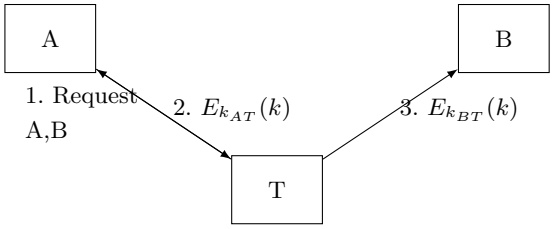
- A trusted courier.
- A face-to-face meeting.
- Installation of an authentication key in a SIM card.

Comment 4.1

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 `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}

Comment 4.2

Drawbacks: (1) The TTP must be unconditionally trusted. (2) The TTP it an attractive target. (3) The TTP must be on-line.

□

4.1.1 Key Management Problem

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

Discovery 4.1

Strictly speaking, symmetric-key techniques cannot be used to achieve non-repudiation.

However, symmetric-key techniques can be used to achieve some degree of non-repudiation, but typically requires the services of an on-line TTP (e.g., use a MAC algorithm where each user shares a secret key with the TTP).

Definition 4.1: Public key cryptography

Communicating parties a priori share some authenticated (but non-secret) information.

4.1.2 Merkle Puzzles

Goal: Alice and Bob establish a secret session key by communicating over an authenticated (but non-secret) channel.

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 $P_1, P_2, \ldots, P_N$ 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.

Example 4.1

$P_i = \text{AES-CBC}_{k_i}(sk_i,\, n_i,\, n_i)$, where $k_i = (\, r_i \,\|\, 0^{88}\,)$ and r_i is a randomly selected 40-bit string.

P_i can be solved in 2^{40} steps by exhaustive key search.

Algorithm 4.1: Key Pair Generation for Public-Key Crypto

Each entity A does the following:

1. Generate a key pair (P_A, S_A).
2. S_A is A's secret key.
3. P_A is A's public key.

Comment 4.3

Security requirement: It should be infeasible for an adversary to recover S_A from P_A.

Public Key Encryption/ Decryption To encrypt a secret message m for Bob, Alice does:

1 `Obtain an authentic copy of Bob's public key` P_B.
2 `Compute` $c = E(P_B, m)$; E `is the encryption function.`
3 `Send` c `to Bob.`

To decrypt c, Bob does:

1 `Compute` $m = D(S_B, c)$; D `is the decryption function.`

Digital Signatures To sign a message m, Alice does:

1 `Compute` $s = Sign(S_A, m)$.
2 `Send` m `and` s `to Bob.`

To verify Alice's signature s on m, Bob does:

1 `Obtain an authentic copy of Alice's public key` P_A.
2 `Accept if` $Verify(P_A, m, s)$ `= ''Accept''.`

Discovery 4.2

Suppose that Alice generates a signed message (m, s). Then anyone who has an authentic copy of Alice's public key PA can verify the authenticity of the signed message. (This authentication property cannot be achieved with a symmetric-key MAC scheme). Digital signatures are widely used to sign software updates which are then broadcast to computers around the world.

Result 4.1: 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).

Result 4.2: Disadvantages of public-key cryptography:

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

Hybrid Schemes: combining public-key and symmetric-key In practice, symmetric-key and public-key schemes are used together. Here is an example: To send a message m with confidentiality and authenticity, Alice does:

1 Select a secret key k for a symmetric-key encryption scheme such as AES.
2 Obtain an authentic copy of Bob's public key P_B.
3 Send $c_1 \leftarrow \mathsf{PKE.Enc}(P_B, k)$, $c_2 \leftarrow \mathsf{AES.Enc}_k(m)$, and $s \leftarrow \mathsf{Sign}(S_A, H(c_1 \| c_2))$

To recover m and verify its authenticity, Bob does:

1 Obtain an authentic copy of Alice's public key P_A.
2 Check that $\mathsf{Verify}(P_A, H(c_1 \| c_2), s)$ = Accept.
3 Decrypt $c_1 : k \leftarrow \mathsf{PKE.Dec}(S_B, c_1)$.
4 Decrypt $c_2 : m \leftarrow \mathsf{AES.Dec}_k(c_2)$.

4.2 RSA encryption

Here is the steps for key-generation:

1 Choose random primes p and q with $\log_2 p \approx \log_2 q \approx \ell/2$.
2 Compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$.
3 Choose an integer e with $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
4 Compute $d = e^{-1} \pmod{\varphi(n)}$. The public key is (n, e) and the private key is (n, d).

The message space is:

$$M = C = Z_n^* = \{m \in \mathbb{Z} : 0 \leq m < n \text{ and } \gcd(m, n) = 1\}.$$

1 Encryption: $\mathcal{E}((n, e), m) = m^e \pmod{n}$
1 Decryption: $\mathcal{D}((n, d), c) = c^d \pmod{n}$

Theorem 4.1

Let $a, b \in \mathbb{Z}_n$. Then a/b is defined in $\mathbb{Z}_n$ if and only if $\gcd(b, n) = 1$. Furthermore, when a/b is defined, there is an efficient algorithm to compute its value.

Proof. Extended Euclidean Algorithm. □

Correctness of RSA

Theorem 4.2

Let (n, e) be an RSA public key with private key (n, d). Then

$$\mathcal{D}((n, d), \mathcal{E}((n, e), m)) = m$$

for all $m \in \mathbb{Z}_n$ such that $\gcd(m, n) = 1$.

Note 4.1

The theorem can be generalized to the case with no restrictions on $\gcd(m, n)$.

Some elementary number theory.

Theorem 4.3: Fermat's Little Theorem

Let p be a prime. For all integers a, it holds that $a^p \equiv a \mod p$. Moreover, if a is coprime to p, then $a^{p-1} \equiv 1 \mod p$.

Definition 4.2: Euler's phi function a.k.a. Euler's totient function

Let $\varphi(n)$ denote the number of integers $k \in [1, n]$ that are coprime with n.

Theorem 4.4: Formula for Euler's phi function

We have

- $\varphi(p) = p - 1$, if p is prime
- $\varphi(pq) = (p-1)(q-1)$, if p and q are both prime
- $\varphi(n) = p_1^{e_1-1}(p_1 - 1) \cdots p_r^{e_r-1}(p_r - 1)$, if n has prime factorization $p_1^{e_1} \cdots p_r^{e_r}$

Theorem 4.5: Euler's Theorem

Let n be a non-negative integer. For all integers a coprime to n, it holds that $a^{\varphi(n)} \equiv 1 \mod n$.

Proof of Theorem 4.2. Assume $\gcd(m, n) = 1$.

1. By definition of $\mathcal{E}$ and $\mathcal{D}$:

$$\mathcal{D}\big((n,d),\, \mathcal{E}((n,e),m)\big) = \mathcal{D}\big((n,d),\, m^e \bmod n\big) = \big(m^e \bmod n\big)^d \bmod n = m^{ed} \bmod n.$$

2. By definition of e and d, we have $ed \equiv 1 \pmod{\varphi(n)}$. In other words, $ed = 1 + k \cdot \varphi(n)$ for some $k \in \mathbb{Z}$.

3. Since m is coprime to n, Euler's Theorem implies:

$$m^{ed} = m^{1+k\varphi(n)} = m^1 \cdot \big(m^{\varphi(n)}\big)^k \equiv m^1 \cdot 1^k = m \pmod n.$$

□

4.2.1 Implementation issues

Non-trivial algorithms involved in implementing RSA:

- How to obtain random numbers?
- How to generate random large primes?
- How to compute $\gcd(e, \varphi(n))$?
- How to compute $e^{-1} \mod \varphi(n)$?
- How to compute $m^e \mod n$ for (potentially) large e?
- How to compute $c^d \mod n$ for (potentially) large d?

Comment 4.4

The difficult part is performing these operations *efficiently*.

Basic integer operations Input: Two ℓ-bit positive integers a and b.
Input size: $O(\ell)$ bits.

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

Greatest common divisor

```
1  Input: a, b ∈ ℕ
2  if b = 0 then output a
3  else: output gcd(b, a mod b)
```

Find $\gcd(a, b)$

Basic Modular Operations Input: An ℓ-bit integer n, and integers $a, b \in [0, n-1]$.
Input size: $O(\ell)$ bits.

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

Extended Euclidean algorithm

```
Input: a, b ∈ ℕ
if b = 0 then output (1, 0)
else:
    q ← ⌊a/b⌋
    r ← a mod b
    (s, t) ← extended\_gcd(b, r)
    output (t, s − qt)
```

Computing extended_gcd(a, b)

Computing modular inverses

```
Input: a, n ∈ ℕ, a < n, gcd(a, n) = 1
(x, y) ← extended\_gcd(a, n)
output x mod n
```

Computing $a^{-1} \mod n$

Modular exponentiation: square-and-multiply algorithm, sequential version

```
Write b = b_{t−1} b_{t−2} ... b_1 b_0 in binary.
y_0 ← a
for i = 1, ..., t − 1: y_i ← y_{i−1}^2
z ← 1
for i = 0, ..., t − 1:
    if b_i = 1 then z ← z · y_i mod n
return z
```

Sequential algorithm for computing $a^b \mod n$.

Comment 4.5

Each for loop has $\ell = \log_2 n$ iterations, and each iteration does at most one modular multiplication ($O(\ell^2)$ runtime) so the total runtime is *polynomial* in the input size: $O(\ell^3)$.

4.3 Diffie-Hellman key exchange

Definition 4.3: Order

The order of an element $x \in \mathbb{Z}_n^*$ is defined to be the smallest positive integer t such that $x^t = 1$ in $\mathbb{Z}_n^*$.

Definition 4.4: Generator

An element g of $\mathbb{Z}_n^*$ is defined to be a generator of $\mathbb{Z}_n^*$ if every y in $\mathbb{Z}_n^*$ can be written as $y = g^x$ for some integer x.

code 4.1

For any $n \in \mathbb{N}$. Every element has order that divides $\varphi(n)$. If n is composite, then $\mathbb{Z}_n^*$ is not cyclic and does not have a generator. If n is a prime, then $\mathbb{Z}_n^*$ is cyclic and every generator has order $\varphi(n) = n-1$.

Diffie–Hellman key exchange allows two people to establish a shared secret, without transmitting any secret information. First, pick a prime p and an element $g \in \mathbb{Z}_p^*$ of large prime order q.

Alice: Pick $a \in \mathbb{Z}_q$ at random. Send $g^a \mod p$ to Bob.

Bob: Pick $b \in \mathbb{Z}_q$ at random. Send $g^b \mod p$ to Alice.

so both Alice and Bob can compute $g^{ab} \equiv (g^a)^b \equiv (g^b)^a \mod p$.

4.3.1 Diffie–Hellman assumption

Definition 4.5: Computational Diffie–Hellman assumption (CDH)

Let $a, b \in_R \mathbb{Z}_q$. Given g, g^a, g^b, it is computationally infeasible to determine g^{ab}.

Definition 4.6: Decisional Diffie–Hellman assumption (DDH)

Let $a, b \in_R \mathbb{Z}_q$. Given g, g^a, g^b, and either g^{ab} or g^c for a random $c \in_R \mathbb{Z}_q$, it is computationally infeasible to determine whether you were given g^{ab} or g^c.

Note 4.2

If we could compute a from g and g^a, then we could (trivially) break Diffie–Hellman.

Definition 4.7: Discrete logarithm assumption (DLOG)

Let $a \in_R \mathbb{Z}_q$. Given g and g^a, it is computationally infeasible to determine a.

Theorem 4.6: U. Maurer & S. Wolf, 1999

For almost all values of p, the CDH assumption and the DLOG assumption are equivalent.

4.3.2 Elgamal public key encryption

- **Setup:**
  - Choose a single, globally public prime p.
  - Choose a single, globally public element $g \in \mathbb{Z}_p^*$ of large prime order q.
- **Key generation:**
  - Choose $x \in_R \mathbb{Z}_q$.
  - Set $pk = g^x \bmod p$ and $sk = x$.
- **Encryption:** Given $m \in \mathbb{Z}_p^*$,
  - Choose $r \in_R \mathbb{Z}_q$.
  - Set $E(m) = \left(g^r,\, m \cdot (g^x)^r\right) \bmod p$.
- **Decryption:** Given a ciphertext $(c_1, c_2) \in (\mathbb{Z}_p^*)^2$, compute

$$D(c_1, c_2) = c_2 \cdot (c_1^{-1})^x \bmod p.$$

4.4 Security of public key encryption

Definition 4.8: Public Key Encryption Scheme

A **public-key encryption scheme** consists of:

- M – the plaintext space; C – the ciphertext space,
- K_{pubkey} – the space of public keys; K_{privkey} – the space of private keys,
- A randomized algorithm $G \to K_{\text{pubkey}} \times K_{\text{privkey}}$, called a key-generation algorithm,
- A (possibly randomized) encryption algorithm $\mathcal{E} : K_{\text{pubkey}} \times M \to C$,
- A decryption algorithm $\mathcal{D} : K_{\text{privkey}} \times C \to M$.

Definition 4.9: Correctness of Public Key Encryption

For any key pair $(k_{pubkey}, k_{privkey})$ produced by G, and all $m \in M$,

$$\mathcal{D}(k_{privkey}, \mathcal{E}(k_{pubkey}, m)) = m$$

Comment 4.6

The same principles also apply to public-key cryptography.

> Basic assumption (Kirckhoffs's principle, Shannon's maxim): The adversary knows everything about the algorithm, except the secret key k. (Avoid security by obscurity!!)

Adversary's Interaction Possible methods of attacks against a public-key encryption scheme:

- Passive attacks:
  - Key-only attack: The adversary is given the public key(s). (Already implied by Kirckhoff's principle.)
  - Chosen-plaintext attack: The adversary can choose some plaintext(s) and obtain the corresponding ciphertext(s). Equivalent to a key-only attack in the PKE setting.
- Active attacks:
  - Non-adaptive chosen-ciphertext attack: The adversary can choose some ciphertext(s) and obtain the corresponding plaintext(s).
  - (Adaptive) chosen-ciphertext attack: Same as above, except the adversary can also iteratively choose which ciphertexts to decrypt, based on the results of previous queries.

Adversary's goal Possible goals when attacking a public-key encryption scheme:

- Total break: Determine the private key, or determine information equivalent to the private key.
- Decrypt a given ciphertext: Adversary is given a ciphertext c and decrypts it (without querying for the decryption of c).
- Learn some partial information about a message: Adversary is given/chooses a ciphertext c and learns some partial information about the decryption of c (without querying for the decryption of c).

System Designer's Goal The designer's goals are the opposite of the adversary's goals. A public-key encryption scheme is defined to be:

- **Totally insecure** if the adversary can obtain the private key.
- **One-way** if the adversary cannot decrypt a given ciphertext.
- **Semantically secure** if the adversary cannot learn any partial information about a message.

As with symmetric encryption, it is often easier to work with the *indistinguishability* security notion which is equivalent to semantic security:

- an adversary who picks two messages m_0, m_1, and then is given the encryption of one of them $c = \mathcal{E}(p_k, m_b)$, cannot decide which it was given.

Example 4.2

A full security definition consists of a security goal, a method of attack, and a statement of the computational power of the adversary. For example, here are three different security definitions:

1. One-way secure under a chosen-plaintext attack by a computationally bounded adversary (OW-CPA).
2. Semantically secure / indistinguishable under a chosen-plaintext attack by a computationally bounded adversary (IND-CPA).
3. Semantically secure / indistinguishable under an adaptive chosen-ciphertext attack by a computationally bounded adversary (IND-CCA2).

4.4.1 Security of RSA Encryption

Theorem 4.7

If integer factorization is easy, then RSA encryption is totally insecure.

Proof. Given (n, e). Factor n into p and q. Compute $d = e^{-1} \mod (p-1)(q-1)$. Return secret key (n, d). □

Theorem 4.8

If RSA is totally insecure, then integer factorization is easy.

Proof. Suppose that for any RSA public key (n, e) we can also compute the corresponding private exponent d (i.e., we can obtain (n, d)). We can then factor n using the following procedure.

Algorithm 1: Factoring n given an oracle that returns d for (n, e)

Input: n (RSA modulus)

Output: Prime factors p, q of n (with high probability)

for $i \leftarrow 1$ **to** 20 **do**
  Choose a large prime exponent $e_i < n$ that is coprime to n
  Query the oracle to obtain d_i such that $e_i d_i \equiv 1 \pmod{\varphi(n)}$
  Set $x_i \leftarrow e_i d_i - 1$
end
$x \leftarrow \gcd(x_1, x_2, \ldots, x_{20})$ `// Very likely` $x = \varphi(n) = (p-1)(q-1)$
Compute

$$p, q = \frac{1 + n - x \pm \sqrt{(1+n-x)^2 - 4n}}{2}.$$

return p, q

With high probability, p and q will be the prime factors of n. □

Definition 4.10: The RSA Problem

The **RSA Problem** is as follows: given

- An RSA public key (n, e), and
- An element $c \in \mathbb{Z}_n$ such that $\gcd(c, n) = 1$,

find an element $m \in \mathbb{Z}_n$ such that

$$c = m^e \mod n.$$

Theorem 4.9

If the RSA problem is computationally infeasible, then the RSA public-key encryption scheme is one-way secure under a chosen plaintext attack (OW-CPA).

Proof. Follows immediately from the definitions of OW-CPA and the RSA problem. □

Theorem 4.10

RSA is not semantically secure under a ciphertext-only attack.

Proof. Let $c = m^e \mod n$ be an encryption of a message m under the RSA public key (n, e). The algorithm below determines some partial information about m:

1. Does $c = 1$? If yes, then $m = 1$, else $m \neq 1$.

Therefore RSA is not semantically secure. □

Discovery 4.3

A deterministic encryption algorithm (such as RSA) cannot yield semantic security.

Proof. Given a ciphertext c and a public key, choose m at random and compute $c' = \mathcal{E}_{pubkey}(m)$. If $c = c'$ then we know the plaintext was m. If $c \neq c'$ then we know the plaintext was not m. Either way, we have learned information about the plaintext. □

Definition 4.11: Semantic Security of Public Key Encryption

An encryption scheme is **semantically secure** if an adversary cannot learn any information about a plaintext from the corresponding ciphertext (except possibly its length).

Comment 4.7

As a deterministic encryption algorithm (such as RSA) cannot yield semantic security, a randomized encryption algorithm avoids this problem.

Randomized encryption is a necessary but not sufficient condition for semantic security.

Note 4.3

SA does not use a randomized encryption algorithm, but one can modify the encryption algorithm to make it randomized.

Elgamal uses a randomized encryption algorithm: $E(m) = (g^r, m \cdot (g^x)^r)$ where r is random. (Tsiounis and Yung, 1998) showed that Elgamal is semantically secure under the Decisional Diffie-Hellman assumption if the message space M is the group $\langle g \rangle$ generated by the element g.

Definition 4.12: Decisional Diffie-Hellman assumption (DDH)

Let $x, y, z \in_R \mathbb{Z}_q$. Given g, g^x, g^y, and either g^{xy} or g^z, it is computationally infeasible to determine whether you were given the real g^{xy} or the random g^z.

Comment 4.8

The DDH assumption does not always hold!

4.4.2 Difficulty of Factoring

Definition 4.13: 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 $\ell = O(\log n)$). If the expected running time of A is of the form

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

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

When $\alpha = 0$: $L_n[0, c] = O\big((\log n)^{c+o(1)}\big) = O\big(\ell^{c+o(1)}\big)$ (polytime).

When $\alpha = 1$: $L_n[1, c] = O\big(n^{c+o(1)}\big) = (2^{\ell})^{c+o(1)}$ (fully exponential time).

There are special-purpose factoring algorithm such as 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).

Note 4.4

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.

Result 4.3

These are factoring algorithms whose running times do not depend of any properties of the number being factored. Two major developments in the history of factoring:

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

Result 4.4

In 1994, Peter Shor published an efficient algorithm for factoring integers using a quantum computer.

- Theoretical resources: approximately $O((\log n)^2)$ quantum gates to factor n
- Estimated resources: 2^{24} quantum bits and running for 2^{25} seconds to factor a 2048-bit modulus (see at https://arxiv.org/pdf/1902.02332.pdf)

Question 4.1.

When will we have a cryptographically-relevant quantum computer?

Answer. $\sim 65\%$ of experts surveyed think there's $\geq 50\%$ chance of a cryptographically relevant quantum computer by 2039. □

Summary of security of RSA and factoring actoring is believed to be a hard problem for classical computers. However, we have no proof or theoretical evidence that factoring is indeed hard.

Note however that factoring is known to be easy on a quantum computer. Open question: when will we have a cryptographically-relevant quantum computer?

- 512-bit RSA is considered insecure today.
- 1024-bit RSA should be avoided today.
- Applications have moved to 2048-bit RSA or elliptic curve cryptography
- Applications should start moving to post-quantum algorithms.

Comment 4.9

Flawed implementations may be breakable without needing to solve a hard mathematical problem

1. `Side-channel attacks`, which leak information about the secret key during computation
2. `Bad random number generators`, which generate secret keys
3. `Implementation bugs`, which may leak memory contents or do some computations incorrectly

4.4.3 Security of Diffie–Hellman key exchange and Elgamal encryption

Security of Diffie–Hellman key exchange and Elgamal encryption rests on the following two (approximately equivalent) assumptions:

Computational Diffie-Hellman assumption (CDH): Given g, g^a, g^b, it is computationally infeasible to determine g^{ab}.

Discrete logarithm assumption (DLOG): Given g and g^a, it is computationally infeasible to determine a.

Computing discrete logarithms: The random powers method Suppose we want to find x such that $g^x = 2$.

1. For $i = 1, \ldots, t+1$, choose an integer x_i at random until $g^{x_i} \bmod p$ factors completely using only the first t primes.
$$\begin{aligned} g^{x_1} \bmod p &= 2^{e_{1,1}} 3^{e_{1,2}} \cdots p_t^{e_{1,t}}, \\ g^{x_2} \bmod p &= 2^{e_{2,1}} 3^{e_{2,2}} \cdots p_t^{e_{2,t}}, \\ &\vdots \\ g^{x_{t+1}} \bmod p &= 2^{e_{t+1,1}} 3^{e_{t+1,2}} \cdots p_t^{e_{t+1,t}}. \end{aligned}$$
2. Take $\log_g$ of both sides:
$$\begin{aligned} x_1 &\equiv e_{1,1} \log_g 2 + e_{1,2} \log_g 3 + \cdots + e_{1,t} \log_g p_t \pmod{p-1}, \\ x_2 &\equiv e_{2,1} \log_g 2 + e_{2,2} \log_g 3 + \cdots + e_{2,t} \log_g p_t \pmod{p-1}, \\ &\vdots \end{aligned}$$
3. The values x_i and $e_{i,j}$ are known. The values $\log_g p_i$ are unknown. There are $t+1$ equations in t unknowns. Solve!

In this way we obtain $\log_g 2,\ \log_g 3,\ \log_g 5, \ldots,\ \log_g p_t$.

- What if we want $\log_g h$ for $h \neq 2, 3, 5, \ldots$?
- **Solution:** choose an integer r at random until $h \cdot g^r \bmod p$ factors completely using only the first t primes:
$$h \cdot g^r \bmod p \ = \ 2^{e_1} 3^{e_2} \cdots p_t^{e_t}.$$
- Take $\log_g$ of both sides:
$$\log_g(h \cdot g^r) = \log_g h + r \ = \ e_1 \log_g 2 + e_2 \log_g 3 + \cdots + e_t \log_g p_t \pmod{p-1}.$$
- Solve for $\log_g h$.

Result 4.5

The *index calculus* algorithm can solve discrete logarithms over $\mathbb{Z}_p^*$ in running time
$$L_p\left(\tfrac{1}{3},\ \sqrt[3]{\tfrac{64}{9}}\right) = O\left(\exp\left(\left(\sqrt[3]{\tfrac{64}{9}} + o(1)\right)(\log p)^{1/3}(\log\log p)^{2/3}\right)\right).$$
Discrete logarithm in prime fields and elliptic curve groups is believed to be hard problem for classical computers. However, we have no proof or theoretical evidence that they are indeed hard.

4.5 Hybrid public key encryption

Basic idea:

- Use public-key encryption to establish a shared secret key
- Use symmetric-key encryption with the shared secret key to encrypt data

Advantages:

- Key management in hybrid encryption is identical to key management in public-key cryptography (no shared secrets).
- Performance is close to symmetric-key.
- Can sometimes build strongly secure (IND-CCA) hybrid encryption from weaker building blocks (IND-CPA) if done right.

Disadvantages: Attack surface increases—if either the public-key or symmetric-key cryptosystem is totally broken, the hybrid encryption will be broken.

Algorithm 2: Hybrid (KEM–DEM) Encryption

Input: Public key $\mathsf{k}_{\mathsf{pubkey}}$ for PKE; message m; symmetric scheme $(\textsc{Sym.Enc}, \textsc{Sym.Dec})$ with ℓ-bit keys

Output: Ciphertext pair (c_1, c_2)

Pick $k \xleftarrow{\$} \{0,1\}^\ell$ `// ephemeral symmetric key`

$c_1 \leftarrow \text{PKE.}\textsc{Enc}(\mathsf{k}_{\mathsf{pubkey}}, k)$

$c_2 \leftarrow \textsc{Sym.Enc}(k, m)$

return (c_1, c_2)

Algorithm 3: Hybrid Decryption

Input: Private key $\mathsf{k}_{\mathsf{privkey}}$ for PKE; ciphertext (c_1, c_2)

Output: Message m

$k \leftarrow \text{PKE.}\textsc{Dec}(\mathsf{k}_{\mathsf{privkey}}, c_1)$

$m \leftarrow \textsc{Sym.Dec}(k, c_2)$

return m

Comment 4.10

Would like semantic security under adaptive chosen ciphertext attack (IND-CCA2). It is easy to show: if public-key cryptosystem and symmetric-key are IND-CCA2-secure, then basic hybrid encryption is IND-CCA2-secure.

Question 4.2.

Can we make IND-CCA2-secure hybrid encryption using weaker building blocks?

Answer. Yes! See below. □

4.6 Improvements to basic hybrid encryption

Idea #1: Hash the key k before using it.
Encryption:

$$(c_1, c_2) = \big(\text{PKE.Enc}(k_{\text{pubkey}},\, k),\ \text{Sym.Enc}(H(k),\, m)\big).$$

Decryption:

$$m = \text{Sym.Dec}(H(\text{PKE.Dec}(k_{\text{privkey}},\, c_1))\,,\, c_2)\,.$$

Theorem 4.11: Kurosawa, Matsuo, ACISP 2004

Hashed Elgamal hybrid encryption is IND-CCA2-secure, assuming:

- the symmetric-key encryption scheme is IND-CCA2-secure,
- the hash function is a random oracle,
- the "Strong DH" problem is intractable.

Idea #2: Add a MAC.

Encryption: To encrypt m, choose r at random, and compute

$$\begin{aligned}(k_1, k_2) &= H\big((g^\alpha)^r\big),\\ c &= \text{Sym.Enc}(k_1, m),\\ t &= \text{MAC}(k_2, c).\end{aligned}$$

The ciphertext is $(g^r,\, c,\, t)$.

Decryption: Given a ciphertext (c_1, c_2, c_3), compute

$$\begin{aligned}(\hat{k}_1, \hat{k}_2) &= H(c_1^\alpha)\,,\\ \hat{m} &= \text{Sym.Dec}(\hat{k}_1, c_2),\\ \hat{t} &= \text{MAC}(\hat{k}_2, c_2).\end{aligned}$$

If $\hat{t} = c_3$, output $\hat{m}$, otherwise output NULL.

Theorem 4.12

DHIES is semantically secure under adaptive chosen ciphertext attack (IND-CCA2), assuming:

- The symmetric-key encryption scheme is IND-CPA-secure,
- The MAC is secure (EUF-CMA),
- The hash function is a random oracle, and
- The Diffie-Hellman problem is intractable.

Idea #3: Instead of a MAC, a simple hash check is enough. Key generation: Use PKE.KeyGen to generate public/private key pairs.

Encryption: To encrypt $m \in \{0,1\}^*$, compute

$$c_1 = \text{PKE.Enc}(k_{\text{pubkey}}, k),$$
$$c_2 = \text{Sym.Enc}(H_1(k), m),$$
$$c_3 = H_2(m, k),$$

for k chosen at random.

Decryption: To decrypt a ciphertext of the form (c_1, c_2, c_3):

$$\hat{k} = \text{PKE.Dec}(k_{\text{privkey}}, c_1), \qquad \hat{m} = \text{Sym.Dec}(H_1(\hat{k}), c_2).$$

Output

$$\begin{cases} \hat{m}, & \text{if } c_3 = H_2(\hat{m}, \hat{k}), \\ \text{NULL}, & \text{otherwise.} \end{cases}$$

Theorem 4.13

The Fujisaki–Okamoto public-key cryptosystem is IND-CCA2-secure if we assume:

- The public key encryption scheme is one-way secure under chosen plaintext attack (OW-CPA),
- The symmetric-key encryption scheme is IND-CPA-secure,
- H1 and H2 are random oracles.

Shoup's KEM/DEM approach

Idea: Don't need a full public key encryption scheme to transfer a user-selected message, only a "key encapsulation mechanism" to establish a random shared secret.

Definition 4.14: Key encapsulation mechanism (KEM)

A **key encapsulation mechanism** consists of:

- K_{pubkey} – the space of public keys; K_{privkey} – the space of private keys,
- C – the ciphertext space; K_{shared} – the space of shared secrets,
- A randomized key generation algorithm KeyGen : $K_{\text{pubkey}} \times K_{\text{privkey}}$,
- A randomized encapsulation algorithm Encap : $K_{\text{pubkey}} \to C \times K_{\text{shared}}$,
- A decapsulation algorithm Decap : $K_{\text{privkey}} \times C \to K_{\text{shared}}$.

Definition 4.15: Correctness of a KEM

For any key pair $(k_{\text{pubkey}}, k_{\text{privkey}})$ produced by KeyGen, if $\text{Encap}(k_{\text{pubkey}}) = (c, m)$, then $\text{Decap}(k_{\text{privkey}}, c) = m$.

Public key encryption	**Key encapsulation mechanism**
Can select and transmit an arbitrary message m in the PKE's message space	No control over the shared secret K_{shared} that ends up being conveyed

Comment 4.11

KEMs are okay for hybrid encryption because we don't need control over the symmetric key selected, we just need a random one.

Hybrid public key encryption using KEM/DEM approach

Note 4.5

DEM = Data Encapsulation Mechanism; synonym for symmetric key encryption scheme

- Hyb.Enc(K_{pubkey}, m):
  1. Compute $(c_1, K_{\text{shared}}) \leftarrow \text{KEM.Encap}(K_{\text{pubkey}})$
  2. Compute $c_2 \leftarrow \text{Sym.Enc}(K_{\text{shared}}, m)$
  3. Transmit combined ciphertext (c_1, c_2)
- Hyb.Dec(K_{privkey}, (c_1, c_2)):
  1. $m \leftarrow \text{Sym.Dec}(\text{KEM.Decap}(K_{\text{privkey}}, c_1), c_2)$

Example 4.3: Example KEMs

RSA-based KEM

- **KeyGen:**
  1. $K_{\text{privkey}} = (n, d), \quad K_{\text{pubkey}} = (n, e)$
- **Encap**($K_{\text{pubkey}} = (n, e)$):
  1. Select random $r \in_R \mathbb{Z}_n^*$
  2. $c_1 \leftarrow r^e \bmod n$
  3. $K_{\text{shared}} \leftarrow H(r, c_1)$

Diffie–Hellman–based KEM

- **KeyGen:**
  1. $K_{\text{privkey}} = x, \quad K_{\text{pubkey}} = g^x$
- **Encap**($K_{\text{pubkey}} = g^x$):
  1. Select random $y \in_R \mathbb{Z}_q$
  2. $c_1 \leftarrow g^y$
  3. $K_{\text{shared}} \leftarrow H\big(g^{xy}, g^y \,\|\, g^x\big)$

4.7 Elliptic Curve Cryptography

Diffie–Hellman key exchange works in every group: Pick a group G and $g \in G$ of large prime order q

Alice: Pick $x \in_R \mathbb{Z}_q$. Send $g^x \in G$ to Bob.

Bob: Pick $y \in_R \mathbb{Z}_q$ at random. Send $g^y \in G$ to Alice.

so both Alice and Bob can compute $g^{xy} \equiv (g^x)^y \equiv (g^y)^x \in G$.

Note 4.6

However, some groups are unsuitable for use in Diffie–Hellman:

- Infinite groups are unwieldy on a computer.
- Some finite groups are insecure for Diffie–Hellman

Elliptic curve cryptography idea: Use the points on an elliptic curve (of the form $y^2 = x^3 + ax + b$) to create a group, then do Diffie–Hellman.

Definition 4.16: Elliptic Curve

An **elliptic curve** is a curve of the form $y^2 = x^3 + ax + b$. A **point** on a curve is a point $P = (x_P, y_P)$ such that $y_P^2 = x_P^3 + ax_P + b$

Comment 4.12

Consider an elliptic curve $E : y^2 = x^3 + ax + b$. Over an infinite number system (real numbers, complex numbers, etc.), E is computationally unwieldy. To avoid infinite sets, we work with integers modulo p: $x, y, a, b \in \mathbb{Z}_p$. We require p to be a prime (so that we can perform division).

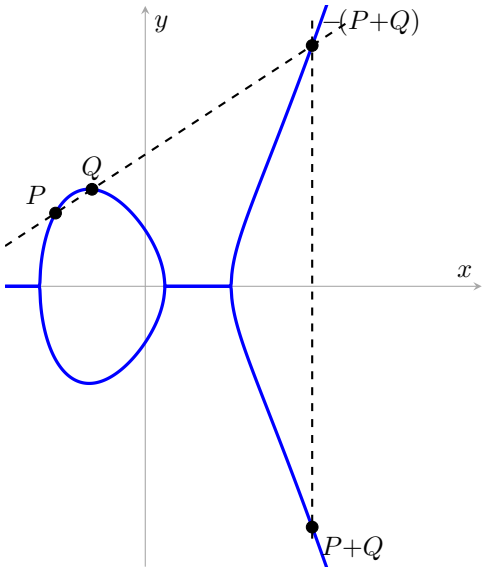
Example 4.4

$E : y^2 = x^3 + x + 6$, with $p = 11$, and $GF(11) = \mathbb{F}_{11} = \mathbb{Z}_{11} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Points on E are: $\{(2, 4), (2, 7), (3, 5), (3, 6), (5, 2), (5, 9), (7, 2), (7, 9), (8, 3), (8, 8), (10, 2), (10, 9)\}$.

Adding points on elliptic curves

Let E be an elliptic curve. Suppose $P, Q \in E$, and we want to add P and Q.

1. Draw a line through P and Q.
2. Find where this line crosses E.
3. Reflect around the x-axis.
4. The reflected point is $P + Q$.



Doubling points on elliptic curves To compute $P + Q$ when $P = Q$:

1. Draw the tangent line through P.
2. Find where this line crosses E.
3. Reflect around the x-axis.
4. The reflected point is $P + P$.

Note 4.7

Another way of thinking about it: drawing a line through three points always sums to 0.

Theorem 4.14

For any elliptic curve $E : y^2 = x^3 + ax + b$, the set

$$\{(x, y) : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$$

forms a group under the operation of point addition, with identity element $\mathcal{O}$.

Point addition formulae Let P and Q be elements of an elliptic curve group (say on $E : y^2 = x^3+ax+b$).

- If $Q = \mathcal{O}$, then set $P + Q = P$.
- If $P = \mathcal{O}$, then set $P + Q = Q$.

Otherwise, write $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$.

- If $x_P = x_Q$ and $y_P = -y_Q$, then set $P + Q = \mathcal{O}$.

- Otherwise, set

$$m = \begin{cases} \dfrac{y_Q - y_P}{x_Q - x_P}, & \text{if } P \neq Q, \\ \dfrac{3x_P^2 + a}{2y_P}, & \text{if } P = Q, \end{cases}$$

$$x_{P+Q} = m^2 - x_P - x_Q, \qquad y_{P+Q} = -(\, m(x_{P+Q} - x_P) + y_P \,),$$

and set $P + Q = (x_{P+Q}, y_{P+Q})$.

4.7.1 Elliptic curve Diffie–Hellman key exchange

Pick a curve E and an element $P \in E$ of large prime order q

Alice: Pick $x \in_R \mathbb{Z}_q$. Send $xP \in E$ to Bob.

Bob: Pick $y \in_R \mathbb{Z}_q$ at random. Send $yP \in G$ to Alice.

so both Alice and Bob can compute $xyP \equiv x(yP) \equiv y(xP) \in E$.

In analogy with square-and-multiply, the value of kP can be computed efficiently using the double-and-add algorithm.

```
1 Given: P ∈ E, k ∈ ℕ.
2     Write k = k_{t-1}k_{t-2}...k_1k_0 in binary.
3     Q_0 ← P
4     for i = 1,...,t-1: Q_i ← 2Q_{i-1}
5     R ← 𝒪
6     for i = 0,...,t-1: if k_i = 1 then R ← R + Q_i
7     return R
```

Sequential algorithm for computing kP.

Here's an equivalent formulation of the double and add algorithm as a loop.

```
1 Given: P ∈ E, k ∈ ℕ.
2     Write k = k_{t-1}k_{t-2}...k_1k_0 in binary.
3     Q ← 𝒪
4     for i from t-1 down to 0 do
5         Q ← 2Q
6         if k_i = 1 then
7             Q ← Q + P
8     return Q
```

Iterative algorithm for computing kP.

4.8 Learning with errors and post-quantum cryptography

4.8.1 Lindner–Peikert public key encryption

Let n, q be integers, and χ be a distribution. All arithmetic modulo q.

- **KeyGen():** Select $s \leftarrow_R \chi(\mathbb{Z}^n)$, $A \leftarrow_R \mathbb{Z}_q^{n\times n}$, $e \leftarrow_R \chi(\mathbb{Z}^n)$. Compute $b \leftarrow As + e$. Return $pk \leftarrow (A, b)$ and $sk \leftarrow s$.
- **Enc($pk, x \in \{0,1\}$):** Select $s' \leftarrow_R \chi(\mathbb{Z}^n)$, $e' \leftarrow_R \chi(\mathbb{Z}^n)$, $e'' \leftarrow_R \chi(\mathbb{Z})$. Compute $b' \leftarrow s'A + e'$, $v' \leftarrow \langle s', b\rangle + e''$, and $c \leftarrow \frac{q}{2}x + v'$. Return $ctxt \leftarrow (b', c)$.
- **Dec($sk, (b', c)$):** Compute $v \leftarrow \langle b', s\rangle$. Return

$$\begin{cases} 0, & \text{if } c - v \in \{0, \dots, \frac{q}{4} - 1\} \cup \{\frac{3q}{4}, \dots, q-1\}, \\ 1, & \text{if } c - v \in \{\frac{q}{4}, \dots, \frac{3q}{4} - 1\}. \end{cases}$$

Theorem 4.15

If the decision learning with errors problem is hard, then Lindner–Peikert encryption is semantically secure against chosen plaintext attacks.

Definition 4.17: Lattice

Let $\mathbf{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_n\} \subseteq \mathbb{Z}_q^{n\times n}$ be a set of linearly independent basis vectors for $\mathbb{Z}_q^n$. Define the corresponding **lattice**

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) = \left\{ \sum_{i=1}^{n} z_i \mathbf{b}_i : z_i \in \mathbb{Z} \right\}.$$

(In other words, a lattice is a set of *integer* linear combinations.)

Theorem 4.16: Regev 2005

For appropriate modulus q and error distribution, solving the decision LWE problem is at least as hard as solving a variant of the shortest vector problem on an n-dimensional lattice.

4.8.2 Post-quantum cryptography

In qunatum computing, we represent and process information using quantum mechanics. Processing information in superposition can dramatically speed some computations (But not necessarily all).

Theorem 4.17: Shor 1984

There exists a polynomial-time quantum algorithm that can factor and compute discrete logarithms.

Comment 4.13

Cryptography based on computational assumptions believed to be resistant to attacks by quantum computers.

4.9 Digital Signatures

Recall 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.

We want a public key primitive that achieves data integrity, data origin authentication, and non-repudiation.

4.9.1 RSA Signature Scheme

Key generation: Same as in RSA encryption. $pk = (n, e),\ sk = (n, d)$

Signature generation: To sign a message m:

1. Compute $s = m^d \bmod n$.
2. The signature on m is s.

Signature verification: To verify a signature s on a message m:

1. Obtain an authentic copy of the public key (n, e).
2. Compute $s^e \bmod n$.
3. Accept (m, s) if and only if $s^e \bmod n = m$.

4.9.2 Defining Signature Schemes

Definition 4.18: Digital signature scheme

A **digital signature scheme** consists of:

- M – the message space,
- S – the signature space,
- K_{pubkey} – the space of public keys,
- K_{privkey} – the space of private keys,
- A *randomized* key generation algorithm $\mathcal{G} \to K_{\text{pubkey}} \times K_{\text{privkey}}$,
- A (usually probabilistic) signing algorithm $\mathcal{S} : K_{\text{privkey}} \times M \to S$,
- A verification algorithm $\mathcal{V} : K_{\text{pubkey}} \times M \times S \to \{\textbf{true}, \textbf{false}\}$.

A valid signature is one which verifies. An invalid signature is one which does not verify. Correctness requirement: For a given key pair $(k_{\text{pubkey}}, k_{\text{privkey}})$ produced by $\mathcal{G}$,

$$\mathcal{V}\big(k_{\text{pubkey}},\, m,\, \mathcal{S}(k_{\text{privkey}}, m)\big) = \textbf{true}$$

for all $m \in M$.

Goals of a digital signature scheme, from the designer's perspective:

- *Authenticate* the origin of a message.
- Guarantee the *integrity* of a message.
- Basic security requirements:
  - It should be infeasible to deduce the private key from the public key.
  - It should be infeasible to generate valid signatures without the private key.

Goals of an adversary:

1. Total break: Recover the private key.
2. Selective forgery: Given a message or a subset of messages, forge a signature for those messages.
3. Existential forgery: Forge a signature for some message (possibly out of your control).

Types of interactions allowed:

1. Key-only attack: The public key is known.
2. Known-message attack: Some messages and their valid signatures are known.
3. Chosen-message attack: The adversary may choose some messages and obtain their signatures.

Definition 4.19: (Signature Scheme) Secure

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

4.9.3 RSA Signature

Recall the statement of the RSA problem: Given an RSA public key (n, e) and an element $c \in \mathbb{Z}_n$ such that $\gcd(c, n) = 1$, find an element $m \in \mathbb{Z}_n$ such that

$$c = m^e \bmod n.$$

Theorem 4.18

A necessary condition for RSA signatures to be secure is that the RSA problem must be intractable.

Proof. If the RSA problem is easy, one can forge signatures as follows:

1. Let m be any message.
2. Find s such that $s^e \equiv m \pmod{n}$.
3. Then s is a valid signature for m.

□

Theorem 4.19

A necessary condition for RSA signatures to be secure is that the RSA problem must be intractable.

Even if the RSA problem is intractable, the basic RSA scheme is still insecure. Here is an existential forgery under a key-only attack:

1. Select $s \in \mathbb{Z}_n$ with $\gcd(s, n) = 1$.
2. Compute $s^e \bmod n$.
3. Set $m = s^e \bmod n$.
4. Then s is a valid signature for m.

Here is a selective forgery under a chosen message attack: Given $m \in \mathbb{Z}_n$ with $\gcd(m, n) = 1$:

1. Compute $m' = 2^e \cdot m \bmod n$
2. Request the signature s' of m'
3. Compute $s = s'/2 \bmod n$.
4. Then s is a valid signature for m.

This takes advantage of the malleability property of the basic RSA function: given $c = m^e \bmod n$ for an unknown m, for any $x \in \mathbb{Z}_n^*$, we can construct c' encrypting mx by computing

$$c' = (x^e \cdot c) \bmod n = (xm)^e \bmod n$$

Full Domain Hash RSA (RSA-FDH) Let $H : \{0,1\}^* \to \mathbb{Z}_n$ be a hash function.
Key generation: Same as in RSA. $pk = (n, e)$, $sk = (n, d)$
Signature generation: To sign a message $m \in \{0,1\}^*$:

1. Compute $s = H(m)^d \bmod n$.
2. The signature on m is s.

Signature verification: To verify a signature s on a message m:

1. Obtain an authentic copy of the public key (n, e).
2. Compute $s^e \bmod n$
3. Accept (m, s) if and only if $s^e \bmod n = H(m)$.

Theorem 4.20: Bellare & Rogaway, 199

If the RSA problem is intractable and H is a random function, then RSA-FDH is a secure signature scheme.

Note 4.8

This theorem does NOT always hold if H is not a random function!

Comment 4.14

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

1. Select $s \in [0, n-1]$.
2. Compute $s^e \bmod n$.
3. Find m such that $H(m) = s^e \bmod n$.
4. Then s is A's signature on m.

Comment 4.15

If H is not 2nd preimage resistant, E could forge signatures as follows:

1. Suppose that (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.

Comment 4.16

If H is not collision resistant, E could forge signatures as follows:

1. Select m_1, m_2 such that $H(m_1) = H(m_2)$, where m_1 is an "innocent" message, and m_2 is a "harmful" message.
2. Induce A to sign m_1: $s = H(m_1)^d \bmod n$.
3. Then s is also A's signature on m_2.

4.9.4 Diffie-Hellman Based Signature Schemes

Digital Signature Algorithm (DSA) The Digital Signature Algorithm (NIST FIPS 186-3) is a digital signature scheme based on Diffie–Hellman/ElGamal.

- Setup: A prime p, a prime q dividing $p-1$, an element $g \in \mathbb{Z}_p^*$ of order q, a hash function $H : \{0,1\}^* \to \mathbb{Z}_q$.

- Key generation: Choose $\alpha \in_R \mathbb{Z}_q^*$ at random. Return $(k_{\text{pubkey}}, k_{\text{privkey}}) = (g^\alpha \bmod p, \alpha)$
- Signing: To sign a message $m \in \{0,1\}^*$,
  - Choose $k \in_R \mathbb{Z}_q^*$ at random
  - Calculate $r = (g^k \bmod p) \bmod q$ and $s = \dfrac{H(m) + \alpha r}{k} \bmod q$.
  - Repeat if k, r, or s are zero. Otherwise, return signature $\sigma = (r, s)$.
- Verification: Given $k_{\text{pubkey}} = g^\alpha$, m, and $\sigma = (r, s)$,
  - Check $0 < r < q$ and $0 < s < q$ and

$$\left(g^{H(m)/s}\, g^{\alpha r/s} \bmod p\right) \bmod q = r.$$

Elliptic Curve Digital Signature Algorithm (ECDSA)

- Setup: A prime p, a prime q, an elliptic curve E over $\mathbb{Z}_p$ of cardinality $|E| = q$, a generator $P \in E$ of order q, and a hash function $H : \{0,1\}^* \to \mathbb{Z}_q$.
- Key generation:
  - Choose $\alpha \in_R \mathbb{Z}_q^*$ at random.
  - $(k_{\text{pubkey}}, k_{\text{privkey}}) = (\alpha P, \alpha)$
- Signing: To sign a message $m \in \{0,1\}^*$,
  - Choose $k \in_R \mathbb{Z}_q^*$ at random
  - Calculate $r = x_{kP} \bmod q$, and $s = \dfrac{H(m) + \alpha r}{k} \bmod q$.
  - Repeat if k, r, or s are zero.
  - The signature is $\sigma = (r, s)$.
- Verification: Given αP, m, and (r, s),
  - Check $0 < r < q$ and $0 < s < q$.
  - Check that the x-coordinate of

$$\frac{H(m)}{s} P + \frac{r}{s}(\alpha P)$$

    is congruent to r modulo q.

4.10 Zero knowledge

Zero-knowledge proof: Peggy wants to prove to Victor that a certain statement is true, without disclosing anything other than the truth of the statement.
Zero-knowledge proof of knowledge: Peggy wants to prove to Victor that she knows something, without disclosing any information about the thing she knows.

Desired Property:

Completeness: If the statement is true, then an honest prover Peggy (who follows the protocol) can convince an honest verifier Victor that the statement is true.

Soundness: If the statement is false, then no honest or cheating prover Peggy can convince an honest verifier Victor that it is true, except with small probability.

- It is okay if the verifier Victor rejects with at least constant probability, not probability 1.
- Sometimes hard to achieve high probability with one execution.
- But can repeat the protocol many times to amplify soundness probability: with high probability, Victor will reject at least once in many repetitions.

Zero knowledge: If the statement is true, then no verifier learns anything other than the fact that the statement is true.

Example 4.5: Graph Isomorphisms

Peggy wants to prove to Victor that she knows an isomorphism ϕ between two public graphs G_0, G_1, without disclosing any information about ϕ to Victor. Proof of knowledge using commitment–challenge–response:

1. Peggy generates a commitment and sends it Victor
2. Victor generates a challenge and sends it to Peggy
3. Peggy generates a response and sends it to Victor
4. Victor verifies the response

Zero-knowledge proof of knowledge of graph isomorphism:
Commitment: Peggy generates a random secret permutation ψ and computes $H = \psi(G_1)$. Commitment is H.
Challenge: Victor picks a random bit $b \in_R \{0, 1\}$; Challenge is b
Response: Peggy's response will be the isomorphism between G_b and H:

- If $b = 0$: Response is $\chi = \psi \circ \phi$
- If $b = 1$: Response is $\chi = \psi$

Verification: Victor checks that $\chi(G_b) = H$.

Completeness: Three isomorphisms:

- $\phi : G_0 \to G_1$
- $\psi : G_1 \to H$

- $\psi \circ \phi : G_0 \to H$

So Peggy's response

$$\chi = \begin{cases} \psi \circ \phi, & \text{if } b = 0, \\ \psi, & \text{if } b = 1 \end{cases}$$

will indeed map G_b to H.

Soundness:

- If Peggy does not know $\phi : G_0 \to G_1$, she could guess what b Victor will use then generate a random χ and compute $H = \chi(G_b)$.
- But she will only guess b right 50% of the time.
- The other half of the time she'll be wrong and verification will fail.
- Soundness probability: $\frac{1}{2}$.

Zero knowledge:

- If the statement is true, then no verifier learns anything other than the fact that the statement is true.
- How to formalize "learns nothing"?
- Victor learns nothing if he could have generated all of the values he receives on his own.
- In other words, if there exists a simulator that outputs transcripts that are indistinguishable from real transcripts
- But Victor may have to generate them in a different order:
  - Honest execution: 1) generate commitment, 2) receive challenge, 3) compute response
  - Simulation: 1) pick challenge, 2) generate response, 3) retroactively compute commitment

Index