Lecture Notes

APPLIED CRYPTOGRAPHY AND DATA SECURITY

(version 2.5 — January 2005)

Prof. Christof Paar

Chair for Communication Security

Department of Electrical Engineering and Information Sciences

Ruhr-Universität Bochum

Germany

www.crypto.rub.de

Table of Contents

****

Chapter 1

Introduction to Cryptography and Data Security

1.1 Literature Recommendations

1. W. Stallings [Sta02], Cryptography and Network Security. Prentice Hall, 2002. Very accessible book on cryptography, well suited for self-study or the undergraduate level. Covers cryptographic algorithms as well as an introduction to important practical protocols such as IPsec and SSL.

2. D.R. Stinson [Sti02], Cryptography: Theory and Practice. CRC Press, 2002. A real textbook. Much more mathematical than Stallings', but very systematic treatment of the material. The Lecture Notes by Christof Paar loosely follow the material presented in this book.

3. A.Menezes, P. van Oorschot, S. Vanstone [MvOV97], Handbook of Applied Cryptography. CRC Press, October 1996. Great compilation of theoretical and implementational aspects of many crypto schemes. Unique since it includes many theoretical topics that are hard to find otherwise. Highly recommended.

4. B. Schneier [Sch96], Applied Cryptography. 2nd ed., Wiley, 1995. Very accessible treatment of protocols and algorithms. Gives also a very nice introduction to cryptography as a discipline. Is becoming a bit dated.

5. D. Kahn [Kah67], The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet. 2nd edition, Scribner, 1996. Extremely interesting book on the history of cryptography, with a focus on the time up to World War II. Great leisure time reading material, highly recommended!

1.2 Overview on the Field of Cryptology

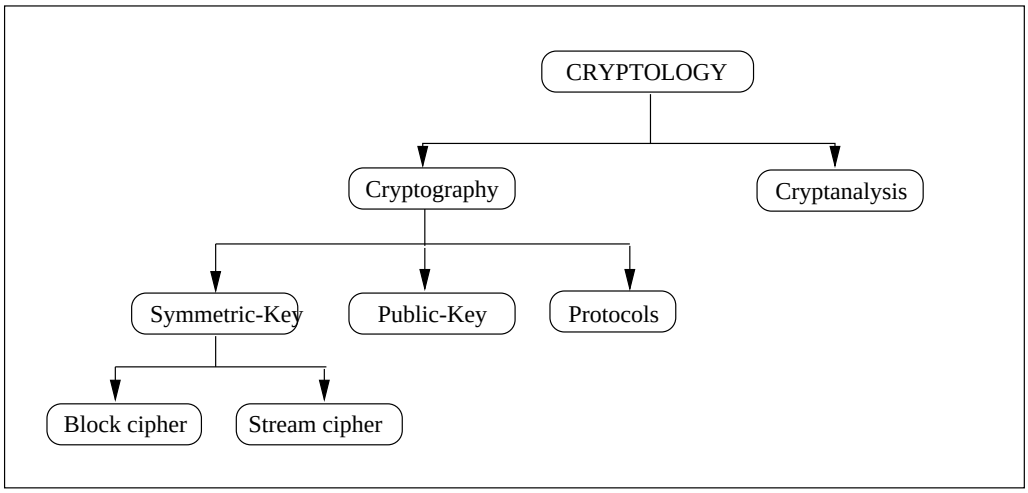


Figure 1.1: Overview on the field of cryptology

Extremely Brief History of Cryptography

Symmetric-Key All encryption and decryption schemes dating from BC to 1976.

Public-Key In 1976 the first public-key scheme was introduced by Diffie-Hellman key exchange protocol.

Hybrid Approach In today's practical systems, very often hybrid schemes are applied which use symmetric algorithms together with public-key algorithms (since both types of algorithms have advantages and disadvantages.)

1.3 Symmetric Cryptosystems

1.3.1 Basics

Sometimes these schemes are also referred to as *symmetric-key*, *single-key*, and *secret-key* approaches.

Problem Statement: Alice and Bob want to communication over an un-secure channel (e.g., the Internet, a LAN or a cell phone link.) They want to prevent Oscar (the bad guy) from listening.

Solution: Use of symmetric-key cryptosystems (these have been around for thousands of years) such that if Oscar reads the encrypted version y of the message x over the un-secure channel, he will not be able to understand its content because x is what really was sent.

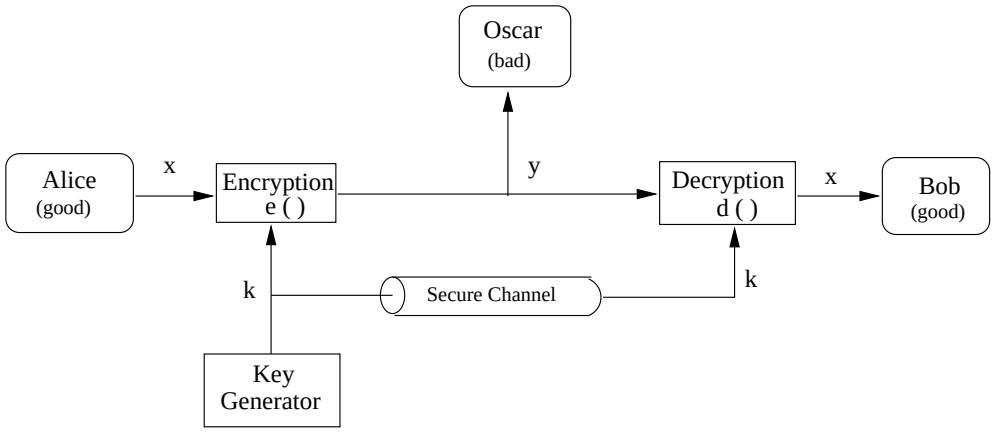


Figure 1.2: Symmetric-key cryptosystem

Remark: In this scenario we only consider the problem of confidentiality, that is, of hiding the contents of the message from an eavesdropper. We will see later in these lecture notes that there are many other things we can do with cryptography, such as preventing Oscar to make changes to the message.

Some important definitions:

1a)	x is called the "plaintext"
1b)	$\mathcal{P}$= $\{x_1, x_2, \ldots, x_p\}$ is the (finite) "plaintext space"
2a)	y is called the "ciphertext"
2b)	$\mathcal{C}$= $\{y_1, y_2, \ldots, y_c\}$ is the (finite) "ciphertext space"
3a)	k is called the "key"
3b)	$\mathcal{K}$= $\{k_1, k_2, \ldots, k_l\}$ is the finite "key space"
4a)	There are l encryption functions e_{k_i} : $\mathcal{P} \rightarrow \mathcal{C}$ (or: $e_{k_i}(x) = y$)
4b)	There are l decryption functions d_{k_i} : $\mathcal{C} \rightarrow \mathcal{P}$ (or: $d_{k_i}(y) = x$)
4c)	e_{k_1} and d_{k_2} are inverse functions if $k_1 = k_2 : d_{k_i}(y) = d_{k_i}(e_{k_i}(x)) = x$ for all $k_i \in \mathcal{K}$

Example: Data Encryption Standard (DES)

- $\mathcal{P} = \mathcal{C}$= $\{0, 1, 2, \ldots, 2^{64} - 1\}$ (each x_i has 64 bits: $x_i = 010 \ldots 0110$)
- $\mathcal{K}$= $\{0, 1, 2, \ldots, 2^{56} - 1\}$ (each k_i has 56 bits)
- encryption (e_k) and decryption (d_k) will be described in Chapter 4

1.3.2 A Motivating Example: The Substitution Cipher

Goal: Encryption of text (as opposed to bits)

Idea: Substitute each letter by another one. The substitution rule forms the key.

Ex.:

$$\begin{aligned} A &\rightarrow K \\ B &\rightarrow D \\ C &\rightarrow W \\ &\dots \end{aligned}$$

Attacks:

Q: Is brute-force attack (i.e., trying of all possible keys) possible?

A:

$$\#\text{keys} = 26 \cdot 25 \cdots 3 \cdot 2 \cdot 1 = 26! \approx 2^{88}$$

A search through such a key space is technically not feasible with today's computer technology.

Q: Other attacks?

A: But: Letter frequency analysis works!

The major weakness of the method is that each plaintext symbol always maps to the same ciphertext symbol. That means that the statistical properties of the plaintext are preserved in the ciphertext. For practical attacks, the following properties of language can be exploited:

1. Determine the frequencies of every ciphertext letter. The frequency distribution (even of relatively short pieces of encrypted text) will be close to that of the given language in general. In particular, the most frequent letters (for instance, in English: "e" is the most frequent one with about 13%, "t" is the second most frequent one with about 9%, "a" is the third most frequent one with about 8%, ...) can often easily be spotted in ciphertext.

2. The method above can be generalized by looking at pairs (or triples, or quadruples, or ...) of ciphertext symbols. For instance, in English (and German and other European languages), the letter "q" is almost always followed by a "u". This behavior can be exploited for detecting the substitution of the letter "q" and the letter "u".

3. If we assume that word separators (blanks) have been found (which is often an easy task), one can often detect frequent short words such as "the", "and", ... , which leaks all the letters in the words involved in those words

In practice the three techniques listed above are often combined to break substitution ciphers.
Lesson learned: Good ciphers should hide the statistical properties of the encrypted plaintext. The ciphertext symbols should appear to be random. Also, a large key space alone is not sufficient for a strong encryption function.

1.3.3 How Many Key Bits Are Enough?

The following table gives a rough indication of the security of symmetric ciphers *with respect to brute force attacks*. As described in Subsection 1.3.2, a large key space is only a necessary but not a sufficient condition for a secure symmetric cipher. The cipher must also be strong against analytical attacks.

key length	**security estimation**
56–64 bits	short term (a few hours or days)
112–128 bits	long term (several decades in the absence of quantum computers)
256 bits	long term (several decades, even with quantum computers (QC) which run the currently known brute force QC algorithms)

Table 1.1: Estimated brute force resistance of symmetric algorithms

1.4 Cryptanalysis

1.4.1 Rules of the Game

What is cryptanalysis? The science of recovering the plaintext x from the ciphertext y. Often cryptanalysis is understood as the science of recovering the plaintext through mathematical analysis. However, there are other methods too such as:

- Side-channel analysis can be used for finding a secret key, for instance by measuring the electrical power consumption of a smart card.
- Social engineering (bribing, black mailing, tricking) or classical espionage can be used for obtaining a secret key by involving humans.

Solid cryptosystems should adhere to *Kerckhoffs' Principle*, postulated by Auguste Kerckhoffs in 1883:

> A cryptosystem should be secure even if the attacker (Oscar) knows all details about the system, with the exception of the secret key. In particular, the system should be secure when the attacker knows the encryption and decryption algorithm.

Important Remark Kerckhoffs' Principle is counterintuitive! It is extremely tempting to design a system which appears to be more secure because we keep the details hidden. This is called "security by obscurity". However, experience has shown time and again that such systems are almost always weak, and they can very often been broken easily as soon as the "secret" design has been reversed engineered or leaked out through other means. An example is the Content Scrambling System (CSS) for DVD contents protection which was broken easily once it was reversed engineered.

1.4.2 Attacks against Crypto Algorithms

If we consider mathematical cryptanalysis we can distinguish four cases, depending on the knowledge that the attacker has about the plaintext and the ciphertext.

1. Ciphertext-only attack

Oscar's knowledge: some $y_1 = e_k(x_1)$, $y_2 = e_k(x_2)$, ...

Oscar's goal : obtain $x_1, x_2, \ldots$ or the key k.

2. Known plaintext attack

Oscar's knowledge: some pairs $(x_1, y_1 = e_k(x_1)), (x_2, y_2 = e_k(x_2)) \ldots$

Oscar's goal : obtain the key k.

3. Chosen plaintext attack

Oscar's knowledge: some pairs $(x_1, y_1 = e_k(x_1)), (x_2, y_2 = e_k(x_2)) \ldots$ of which he can choose $x_1, x_2, \ldots$

Oscar's goal : obtain the key k.

4. Chosen ciphertext attack

Oscar's knowledge: some pairs $(x_1, y_1 = e_k(x_1)), (x_2, y_2 = e_k(x_2)) \ldots$ of which he can choose $y_1, y_2, \ldots$

Oscar's goal : obtain the key k.

1.5 Some Number Theory

Goal Find a *finite* set in which we can perform (most of) the standard arithmetic operations.

Example of a finite set in every day live: The hours on a clock. If you keep adding 1 hour you get:

$$1h, 2h, 3h, \ldots, 11h, 12h, 1h, 2h, 3h, \ldots$$

Even though we keep adding one hour, we never leave the set. Let's look at a general way of dealing with arithmetic in such finite sets. We consider the set of the 9 numbers:

$$\{0, 1, 2, 3, 4, 5, 9, 7, 8\}$$

We can do regular arithmetic as long as the results are smaller than 9. For instance:

$$\begin{aligned} 2 \times 3 &= 6 \\ 4 + 4 &= 8 \end{aligned}$$

But what about $8 + 4$? We try now the following rule: Perform regular integer arithmetic and divide the result by 9. We then consider **only the remainder** rather than the original result. Since $8 + 4 = 12$, and $12/9$ has a remainder of 3, we write:

$$8 + 4 \equiv 3 \bmod 9$$

We now introduce an exact definition of the modulo operation:

Definition 1.5.1 Modulo Operation

Let $a, r, m \in Z$ (where Z is a set of all integers) and $m > 0$. We write $a \equiv r \bmod m$ if m divides $r - a$.

"m" is called the modulus.

"r" is called the remainder.

Some remarks on the modulo operation:

The reminder m is not unique

Ex.:

- $12 \equiv 3 \bmod 9$, 3 is a valid reminder since $9|(12-3)$
- $12 \equiv 21 \bmod 9$, 21 is a valid reminder since $9|(21-3)$
- $12 \equiv -6 \bmod 9$, -6 is a valid reminder since $9|(-6-3)$

Which reminder do we choose?

By agreement, we usually choose:

$$0 \leq r \leq m-1$$

How is the remainder computed?

It is always possible to write $a \in Z$, such that

$a = q \cdot m + r$; $0 \leq r < m$

Now since $a - r = q \cdot m$ (m divides $a - r$) we can write: $a \equiv r \bmod m$.

Note that $r \in \{0, 1, 2, \ldots, m-1\}$.

Ex.:

$a = 42$; $m = 9$

$42 = 4 \cdot 9 + 6$ therefore $42 \equiv 6 \bmod 9$.

The modulo operation can be applied to intermediate results

$(a + b) \bmod m = [(a \bmod m) + (b \bmod m)] \bmod m$.

$(a \times b) \bmod m = [(a \bmod m) \times (b \bmod m)] \bmod m$.

Example: $3^8 \bmod 7 = ?$

(i)$3^8 = 6561 \equiv 2 \bmod 7$, since $6561 = 937 \cdot 7 + 2$ (dumb)

(ii) $3^8 = 3^4 \cdot 3^4 = (81 \bmod 7) \cdot (81 \bmod 7) \equiv 4 \cdot 4 = 16 \equiv 2 \bmod 7$ (smart)

Note: It is almost always of computational advantage to apply the modulo reduction as soon as we can.

Modulo operation in the C programming language

C programming command : "%" (C can return a negative value)

`r = 42 % 9` returns r = 6

but `r = -42 % 9` returns r = -6 → if remainder is negative, add modulus m:

$-6 + 9 = 3 \equiv -42 \bmod 9$

Let's now look at the mathematical structure we obtain if we consider the set of integers from zero to m together with the operations addition and multiplication:

Definition 1.5.2 *The "ring Z_m" consists of:*

1. *The set $Z_m = \{0, 1, 2, \ldots, m-1\}$*
2. *Two operations "$+$" and "$\times$" for all $a, b \in Z_m$ such that:*
   - $a + b \equiv c \bmod m$ $(c \in Z_m)$
   - $a \times b \equiv d \bmod m$ $(d \in Z_m)$

Example: $m = 9$

$Z_9 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

$6 + 8 = 14 \equiv 5 \bmod 9$

$6 \times 8 = 48 \equiv 3 \bmod 9$

All rings (not only the ring Z_m we consider here) have a set of properties which are listed in the following:

Definition 1.5.3 Some important properties of the ring $Z_m = \{0, 1, 2, \ldots, m-1\}$

1. *The* additive identity *is the element zero "0": $a + 0 = a \bmod m$, for any $a \in Z_m$.*
2. *The* additive inverse *"$-a$" of "a" is such that $a+(-a) \equiv 0 \bmod m$: $-a = m-a$, for any $a \in Z_m$.*
3. *Addition is* closed*: i.e., for any $a, b \in Z_m$, $a + b \in Z_m$.*
4. *Addition is* commutative*: i.e., for any $a, b \in Z_m$, $a + b = b + a$.*
5. *Addition is* associative*: i.e., for any $a, b \in Z_m$, $(a + b) + c = a + (b + c)$.*
6. *The* multiplicative identity *is the element one "1": $a \times 1 \equiv a \bmod m$, for any $a \in Z_m$.*
7. *The* multiplicative inverse *"a^{-1}" of "a" is such that $a \times a^{-1} = 1 \bmod m$: An element a has a multiplicative inverse "a^{-1}" if and only if $\gcd(a, m) = 1$.*
8. *Multiplication is* closed*: i.e., for any $a, b \in Z_m$, $ab \in Z_m$.*
9. *Multiplication is* commutative*: i.e., for any $a, b \in Z_m$, $ab = ba$.*
10. *Multiplication is* associative*: i.e., for any $a, b \in Z_m$, $(ab)c = a(bc)$.*

Some remarks on the ring Z_m:

- Roughly speaking, a ring is a structure in which we can add, subtract, multiply, and sometimes divide.

- **Definition 1.5.4** *If* $\gcd(a, m) = 1$*, then* a *and* m *are "*relatively prime*" and the multiplicative inverse of* a *exists.*

  Example:

  i) **Question**: does multiplicative inverse exist with 15 mod 26?

  Answer: yes — $gcd(15, 26) = 1$

  ii) **Question**: does multiplicative inverse exist with 14 mod 26?

  Answer: no — $gcd(14, 26) \neq 1$

- The ring Z_m, and thus the integer arithmetic with the modulo operation, is of central importance to modern public-key cryptography. In practice, the integers are represented with 150–2048 bits.

1.6 Simple Blockciphers

Recall:

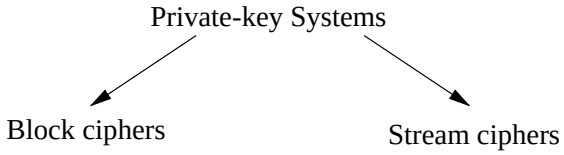


Figure 1.3: Classification of symmetric-key systems

Idea: The message string is divided into blocks (or cells) of equal length that are then encrypted and decrypted.

Input: message string $\bar{X} \rightarrow \bar{X} = x_1, x_2, x_3, \ldots, x_n$, where each x_i is one block.

Cipher: $\bar{Y} = y_1, y_2, y_3, \ldots, y_n$; with $y_i = e_k(x_i)$ where the key k is fixed.

1.6.1 Shift Cipher

One of the most simple ciphers where the letters of the alphabet are assigned a number as depicted in Table 1.2.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Table 1.2: Shift cipher table

Definition 1.6.1 Shift Cipher

Let $\mathcal{P} = \mathcal{C} = \mathcal{K} = Z_{26}$. $x \in \mathcal{P}$, $y \in \mathcal{C}$, $k \in \mathcal{K}$.

Encryption: $e_k(x) = x + k \bmod 26$.

Decryption: $d_k(y) = y - k \bmod 26$.

Remark:

If $k = 3$ the the shift cipher is given a special name — "Caesar Cipher".

Example:

$k = 17$,

plaintext:

$X = x_1, x_2, \ldots, x_6 = ATTACK.$

$X = x_1, x_2, \ldots, x_6 = 0, 19, 19, 0, 2, 10.$

encryption:

$y_1 = x_1 + k \bmod 26 = 0 + 17 = 17 \bmod 26 = R$

$y_2 = y_3 = 19 + 17 = 36 \equiv 10 \bmod 26 = K$

$y_4 = 17 = R$

$y_5 = 2 + 17 = 19 \bmod 26 = T$

$y_6 = 10 + 17 = 27 \equiv 1 \bmod 26 = B$

ciphertext: $Y \doteq y_1, y_2, \ldots, y_6 =$ R K K R T B.

Attacks on Shift Cipher

1. Ciphertext-only: Try all possible keys ($|k| = 26$). This is known as "brute force attack" or "exhaustive search".

   Secure cryptosystems require a sufficiently large key space. Minimum requirement today is $|K| > 2^{80}$, however for long-term security, $|K| \geq 2^{100}$ is recommended.

2. Same cleartext maps to same ciphertext $\Rightarrow$ can also easily be attacked with letter-frequency analysis.

1.6.2 Affine Cipher

This cipher is an extension of the Shift Cipher ($y_i = x_i + k \bmod m$).

Definition 1.6.2 Affine Cipher *Let* $P = C = Z_{26}$.

encryption: $e_k(x) = a \cdot x + b \bmod x$.

key: $k = (a, b)$ *where* $a, b \in Z_{26}$.

decryption: $a \cdot x + b = y \bmod 26$.

$a \cdot x = (y - b) \bmod 26$.

$x = a^{-1} \cdot (y - b) \bmod 26$.

restriction: $gcd(a, 26) = 1$ *in order for the affine cipher to work since* a^{-1} *does not always exist.*

Question: How is a^{-1} obtained?

Answer: $a^{-1} \equiv a^{11} \bmod 26$ (the proof for this is in Chapter 6)

or by trial-and-error for the time being.

1.7 Lessons Learned — Introduction

- Never ever develop your own crypto algorithm unless you have a team of experienced cryptanalysts checking your design.
- Do not use unproven crypto algorithms (i.e., symmetric, asymmetric, hash function) or unproven protocols.
- A large key space by itself is no guarantee for a secure cipher: The cipher might still be vulnerable against analytical attacks.
- Long-term security has two aspects:
  1. The time your crypto implementation will be used (often only a few years.)
  2. The time the encrypted data should stay secure (depending on application: can range from a day to several decades.)
- Key lengths for symmetric algorithms in order to thwart exhaustive key-search attacks:
  1. 64 bits — unsecure except for data with extreme short term value.
  2. 112-128 bits — long-term security of several decades, including attacks by intelligence agencies unless they possess quantum or biological computers. Only realistic attack could come from quantum computers (which do not exist and perhaps never will.)
  3. 256 bits — as above, but also secure against attack by quantum computer.

Chapter 2

Stream Ciphers

Further Reading: [Sim92, Chapter 2]

2.1 Introduction

Remember classification:

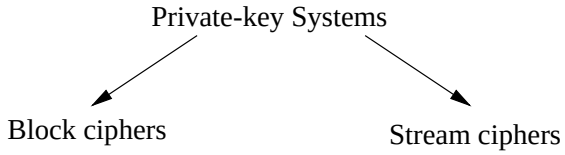


Figure 2.1: Symmetric-key cipher classification

Block Cipher: $y_1, y_2, \ldots, y_n = e_k(x_1), e_k(x_2), \ldots, e_k(x_n)$.
Key features of block ciphers:

- Encrypts blocks of bits at a time. In practice, x_i (and y_i) are 64 or 128 bits long.
- The encryption function $e_k()$ requires complex operation. In practice all block ciphers are iterative algorithms with multiple rounds. Examples: DES (Chapter 3) or AES (Chapter 4).

Stream Cipher: $y_1, y_2, \ldots, y_n = e_{z_1}(x_1), e_{z_2}(x_2), \ldots, e_{z_n}(x_n)$,

where $z_1, z_2, \ldots, z_n$ is the *keystream*.

Key features of stream ciphers:

- Encrypts individual bits at a time, i.e., x_i (and y_i) are single bits.
- The encryption function $e_{z_1}()$ is a simple operation. In practice it is most often a simple XOR.
- The main art of stream cipher design is the generation of the key stream.

Most popular en/decryption function: modulo 2 addition

Assume: $x_i, y_i, z_i \in \{0, 1\}$

$y_i = e_{z_i}(x_i) = x_i + z_i \bmod 2 \rightarrow$ encryption

$x_i = e_{z_i}(y_i) = y_i + z_i \bmod 2 \rightarrow$ decryption

This leads to the following block diagram for a stream cipher encryption/decryption:

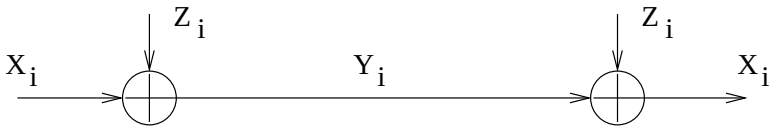


Figure 2.2: Principle of stream ciphers

Remarks:

1. Historical note: A machine realizing the functionality shown above was developed by Vernam for teletypewriters in 1917. Vernam was alumni of Worcester Polytechnic Institute (WPI). Further reading: [Kah67].

   item The modulo 2 operation is equivalent to a 2-input XOR operation.

   Why are encryption and decryption identical operations? Truth table of modulo 2 addition:

a	b	$c = a + b \bmod 2$
0	0	$0 + 0 = 0 \bmod 2$
0	1	$0 + 1 = 1 \bmod 2$
1	0	$1 + 0 = 1 \bmod 2$
1	1	$1 + 1 = 0 \bmod 2$

.

   $\Rightarrow$ modulo 2 addition yields the same truth table as the XOR operation.

2. Encryption and decryption are the same operation, namely modulo 2 addition (or XOR).

   Why? We show that decryption of ciphertext bit y_i yields the corresponding plaintext bit.

   Decryption: $y_i + z_i = \underbrace{(x_i + z_i)}_{\text{encryption}} + z_i = x_i + (z_i + z_i) \equiv x_i \bmod 2.$

   Note that $z_i + z_i \equiv 0 \bmod 2$ for $z_i = 0$ and for $z_i = 1$.

Example: Encryption of the letter 'A' by Alice.

'A' is given in ASCII code as $65_{10} = 1000001_2$.

Let's assume that the first key stream bits are $\rightarrow z_1, \ldots, z_7 = 0101101$

Encryption by Alice:	plaintext x_i:	1000001	= 'A'	(ASCII symbol)
	key stream z_i:	0101101		
	ciphertext y_i:	1101100	= 'l'	(ASCII symbol)
Decryption by Bob:	ciphertext y_i:	1101100	= 'l'	(ASCII symbol)
	key stream z_i:	0101101		
	plaintext x_i:	1000001	= 'A'	(ASCII symbol)

2.2 Some Remarks on Random Number Generators

We distinguish between three types of random number generators (RNG):

True Random Number Generators (TRNG) These are sequences of numbers generated from physical processes. Example: coin flipping, rolling of dices, semiconductor noise, radioactive decay, ...

General Pseudo Random Generators (PRNG) These are sequences which are *computed* from an initial seed value. Often they are computed recursively:

$$\begin{aligned} z_0 &= \text{seed} \\ z_{i+1} &= f(z_i) \end{aligned}$$

Example: linear congruential generator

$$\begin{aligned} z_0 &= \text{seed} \\ z_{i+1} &\equiv a\, z_i + b \bmod m, \end{aligned}$$

where a, b, m are constants.

A common requirement of PRNG is that they posses good statistical properties. There are many mathematical tests (e.g., chi-square test) which verify the statistical behavior of PRNG sequences.

Cryptographically Secure Pseudo Random Generators (CSPRNG) These are PRNG which posses the following additional property: A CSPRNG is *unpredictable*. That is, given the first n output bits of the generator, it is computationally infeasible to compute the bits $n+1, n+2, \ldots$.

It must be stressed that for stream cipher applications it is not sufficient for a pseudo random generator to have merely good statistical properties. In addition, for stream ciphers only cryptographically secure generators are useful. Important: The distinction between PRNG and CSPRN and their relevance for stream ciphers is often not clear to non-cryptographers.

2.3 General Thoughts on Security, One-Time Pad and Practical Stream Ciphers

Definition 2.3.1 *Unconditional Security*

A cryptosystem is unconditionally *secure if it cannot be broken even with infinite computational resources.*

Definition 2.3.2 *One-time Pad (OTP)*

A cryptosystem developed by Mauborgne based on Vernam's stream cipher consisting of:

$|\mathcal{P}| = |\mathcal{C}| = |\mathcal{K}|$,

with $x_i, y_i, k_i \in \{0, 1\}$.

encrypt $\rightarrow e_{k_i}(x_i) = x_i + k_i \bmod 2$.

decrypt $\rightarrow d_{k_i}(y_i) = y_i + k_i \bmod 2$.

Theorem 2.3.1 *The OTP is unconditionally secure if keys are only used once, and if the key consists of true random bits (TRNG.)*

Remarks:

1. The OTP is the only provable secure system:

$$\begin{aligned} y_0 &= x_0 + K_0 \bmod 2 \\ y_1 &= x_1 + K_1 \bmod 2 \\ &\vdots \end{aligned}$$

   each equality is a linear equation with 2 unknowns.

   $\Rightarrow$ for every y_i, $x_i = 0$ and $x_i = 1$ are equally likely.

   $\Rightarrow$ holds only if $K_0, K_1, \ldots$ are not related to each other, i.e., K_i must be generated truly randomly.

2. OTP are impractical for most applications.

Question: In order to build practical stream generators, can we "emulate" a OTP by using a short key?

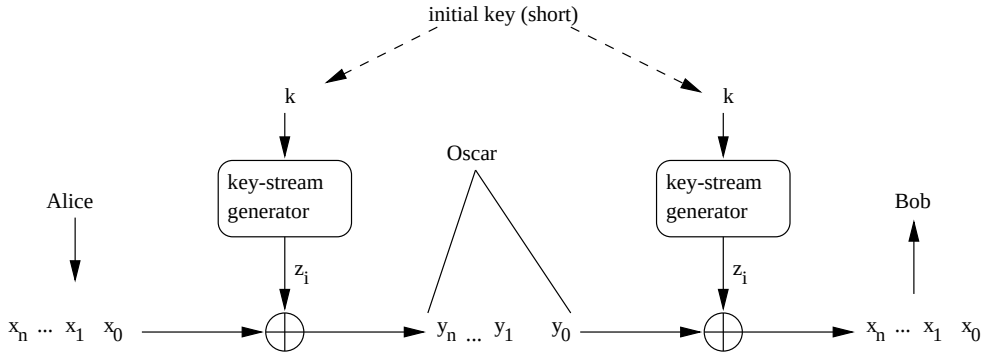


Figure 2.3: Practical stream ciphers

It should be stressed that practical stream ciphers are not unconditionally secure. In fact, **all** known practical crypto algorithms (stream ciphers, block ciphers, public-key algorithms) are at the most *relative secure*, which we define as follows:

Definition 2.3.3 *Computational Security*

A system is "computationally secure" if the ***best possible algorithm*** *for breaking it requires N operations, where N is very large and known.*

Unfortunately, **all** known practical systems are only computational secure for **known algorithms**.

Definition 2.3.4 *Relative Security*

A system is "relative secure" if its security relies on a well studied, very hard problem. However, it is not known which is the best algorithm for computing the problem.

Example

A cryptosystem S is secure as long as factoring of large integers is hard (this is believed for RSA).

Classification of practical key-stream generators:

synchronous stream cipher

$$z_i = f(k, z_{i-i}, \ldots, z_1)$$

asynchronous stream cipher

$$z_i = f(k, y_{i-1}, z_{i-i}, \ldots, z_1)$$

Note that the receiver (Bob) has to match the exact z_i to the correct y_i in order to obtain the correct cleartext. This requires synchronization of Alice's and Bob's key-stream generators.

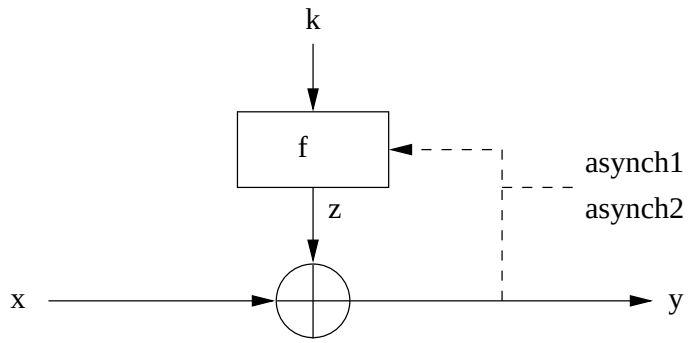


Figure 2.4: Asynchronous stream cipher

2.4 Synchronous Stream Ciphers

The keystream $z_1, z_2, \ldots$ is a pseudo-random sequence which depends only on the key.

2.4.1 Linear Feedback Shift Registers (LFSR)

An LFSR consists of m storage elements (flip-flops) and a feedback network. The feedback network computes the input for the "last" flip-flop as XOR-sum of certain flip-flops in the shift register.

Example: We consider an LFSR of degree $m = 3$ with flip-flops K_2, K_1, K_0, and a feedback path as shown below.

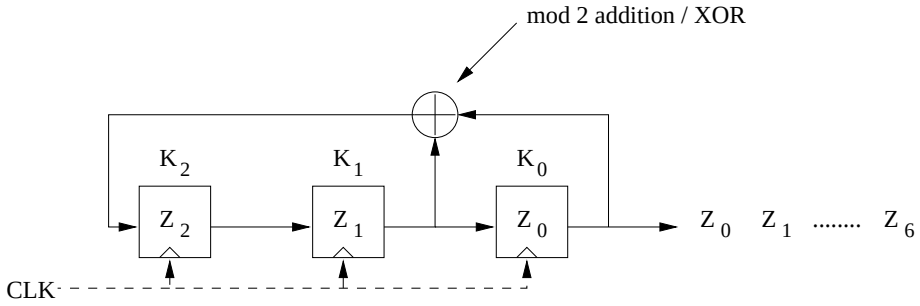


Figure 2.5: Linear feedback shift register

K_2	K_1	K_0
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1
0	1	1
0	0	1
1	0	0

Mathematical description for keystream bits z_i with z_0, z_1, z_2 as initial settings:

$z_3 = z_1 + z_0 \bmod 2$

$z_4 = z_2 + z_1 \bmod 2$

$z_5 = z_3 + z_2 \bmod 2$

$\vdots$

general case: $z_{i+3} = z_{i+1} + z_i \bmod 2;\ i = 0, 1, 2, \ldots$

Expression for the LFSR:

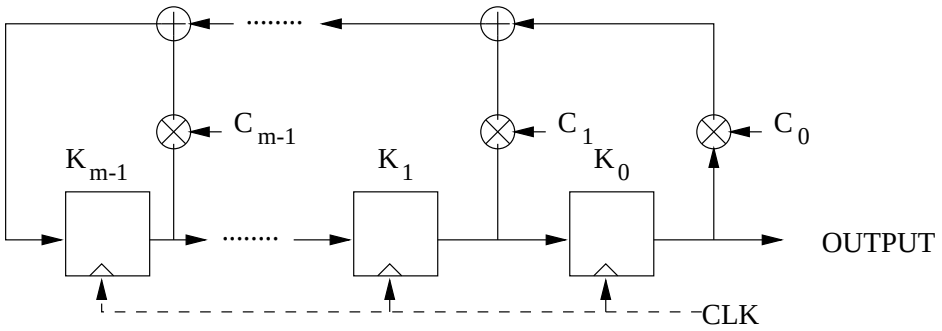


Figure 2.6: LFSR with feedback coefficients

$C_0, C_1, \ldots, C_{m-1}$ are the *feedback coefficients*. $C_i = 0$ denotes an open switch (no connection), $C_i = 1$ denotes a closed switch (connection).

$$z_{i+m} = \sum_{j=0}^{m-1} C_j \cdot z_{i+j} \bmod 2;\ C_j \in \{0, 1\};\ i = 0, 1, 2, \ldots$$

The entire key consists of:

$k = \{(C_0, C_1, \ldots, C_{m-1}), (z_0, z_1, \ldots, z_{m-1}), m\}$

Example:

$k = \{(C_0 = 1, C_1 = 1, C_2 = 0), (z_0 = 0, z_1 = 0, z_2 = 1), 3\}$

Theorem 2.4.1 *The maximum sequence length generated by the LFSR is* $2^m - 1$.

Proof:

There are only 2^m different states $(k_0, \ldots, k_m)$ possible. Since only the current state is known to the LFSR, after 2^m clock cycles a repetition must occur. The all-zero state must be excluded since it repeats itself immediately.

Remarks:

1.) Only certain configurations $(C_0, \ldots, C_{m-1})$ yield maximum length LFSRs.

For example:

if $m = 4$ then $(C_0 = 1, C_1 = 1, C_2 = 0, C_3 = 0)$ has length of $2^m - 1 = 15$

but $(C_0 = 1, C_1 = 1, C_2 = 1, C_3 = 1)$ has length of 5

2.) LFSRs are sometimes specified by polynomials.

such that the $P(x) = x^m + C_{m-1}x^{m-1} + \ldots + C_1 x + C_0$.

Maximum length LFSRs have *"primitive polynomials"*.

These polynomials can be easily obtained from literature (Table 16.2 in [Sch96]).

For example:

$(C_0 = 1, C_1 = 1, C_2 = 0, C_3 = 0) \Longleftrightarrow P(x) = 1 + x + x^4$

2.4.2 Clock Controlled Shift Registers

Example: *Alternating stop-and-go generator.*

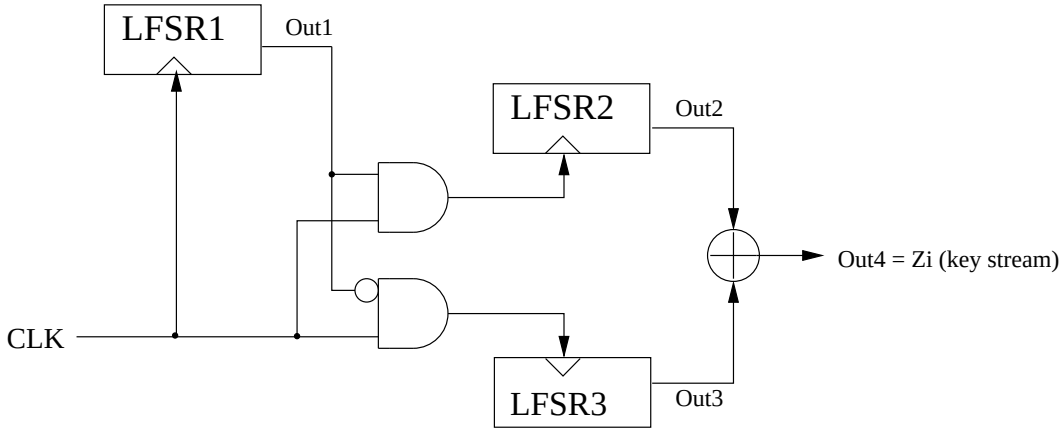


Figure 2.7: Stop-and-go generator example

Basic operation:

When $Out_1 = 1$ then LFSR2 is clocked otherwise LFSR3 is clocked.

Out_4 serves as the keystream and is a bitwise XOR of the results from LFSR2 and LFSR3.

Security of the generator:

- All three LFSRs should have maximum length configuration.
- If the sequence lengths of all LFSRs are relatively prime to each other, then the sequence length of the generator is the product of all three sequence lengths, i.e., $L = L_1 \cdot L_2 \cdot L_3$.
- A secure generator should have LFSRs of roughly equal lengths and the length should be at least 128: $m_1 \approx m_2 \approx m_3 \approx 128$.

2.5 Known Plaintext Attack Against Single LFSRs

Assumption:

For a known plaintext attack, we have to assume that m is known.

Idea:

This attack is based on the knowledge of some plaintext and its corresponding ciphertext.

i) Known plaintext $\rightarrow x_0, x_1, \ldots, x_{2m-1}$.

ii) Observed ciphertext $\rightarrow y_0, y_1, \ldots, y_{2m-1}$.

iii) Construct keystream bits $\rightarrow z_i = x_i + y_i \mod 2; i = 0, 1, \ldots, 2m-1$.

Goal:

To find the feedback coefficients C_i.

Using the LFSR equation to find the C_i coefficients:

$$z_{i+m} = \sum_{j=0}^{m-1} C_j \cdot z_{i+j} \mod 2; C_j \in \{0, 1\}$$

We can rewrite this in a matrix form as follows:

$$\begin{array}{llcll} i=0 & z_m & = & C_0 z_0 + C_1 z_1 + \ldots + C_{m-1} z_{m-1} & \mod 2. \\ i=1 & z_{m+1} & = & C_0 z_1 + C_1 z_2 + \ldots + C_{m-1} z_m & \mod 2. \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ i=m-1 & z_{2m-1} & = & C_0 z_{m-1} + C_1 z_m + \ldots + C_{m-1} z_{2m-2} & \mod 2. \end{array} \tag{2.1}$$

Note:

> We now have m linear equations in m unknowns $C_0, C_1, \ldots, C_{m-1}$. The C_i coefficients are constant making it possible to solve for them when we have $2m$ plaintext-ciphertext pairs.

Rewriting Equation (2.1) in matrix form, we get:

$$\begin{bmatrix} z_0 & \ldots & z_{m-1} \\ \vdots & & \vdots \\ z_{m-1} & \ldots & z_{2m-2} \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} z_m \\ \vdots \\ z_{2m-1} \end{bmatrix} \mod 2 \tag{2.2}$$

Solving the matrix in (2.2) for the C_i coefficients we get:

$$\begin{bmatrix} c_0 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} z_0 & \dots & z_{m-1} \\ \vdots & & \vdots \\ z_{m-1} & \dots & z_{2m-2} \end{bmatrix}^{-1} \cdot \begin{bmatrix} z_m \\ \vdots \\ z_{2m-1} \end{bmatrix} \bmod 2 \tag{2.3}$$

Summary:

> By observing $2m$ output bits of an LFSR of degree m and matching them to the known plaintext bits, the C_i coefficients can exactly be constructed by solving a system of linear equations of degree m.

$\Rightarrow$ **LFSRs by themselves are extremely unsecure!** Even though they are PRNG with good statistical properties, the are not cryptographically secure. However, combinations of them such as the alternating stop-and-go generator **can** be secure.

2.6 Lessons Learned — Stream Ciphers

- Stream ciphers are less popular than block ciphers in most application domains such as Internet security. There are exceptions, for instance the popular stream cipher RC4.
- Stream ciphers are often used in mobile (and presuming military) applications, such as the A5 speech encryption algorithm of the GSM mobile network.
- Stream ciphers generally require fewer resources (e.g., code size or chip area) for an implementation than block ciphers. They tend to encrypt faster than block ciphers.
- The one-time pad is the only provable secure symmetric algorithm.
- The one-time pad is highly impractical in most cases because the key length has to be equal to the message length.
- The requirements for a *cryptographically secure* pseudo-random generator are far more demanding than the requirements for pseudo-random generators in other (engineering) applications such as simulation.
- Many pseudo-random generators with good statistical properties such as LFSRs are not cryptographically secure at all. A stand-alone LFSR makes, thus, a poor stream cipher.

Chapter 3

Data Encryption Standard (DES)

3.1 Confusion and Diffusion

Before we start with DES, it is instructive to look at the primitive operations that can be applied in order to achieve strong encryption. According to Shannon, there are two primitive operations for encryption.

1. **Confusion** — encryption operation where the relationship between cleartext and ciphertext is obscured. Some examples are:
   (a) Shift cipher — main operation is substitution.
   (b) German Enigma (broken by Turing) — main operation is *smart* substitution.
2. **Diffusion** — encryption by spreading out the influence of one cleartext letter over many ciphertext letters. An example is:
   (a) permutations — changing the positioning of the cleartext.

Remarks:

1. Today → changing of one bit of cleartext should result on average in the change of half the output bits.

   $x_1 = 001010 \rightarrow encr. \rightarrow y_1 = 101110.$

   $x_2 = 000010 \rightarrow encr. \rightarrow y_2 = 001011.$

2. Combining confusion with diffusion is a common practice for obtaining a secure scheme. Data Encryption Standard (DES) is a good example of that.

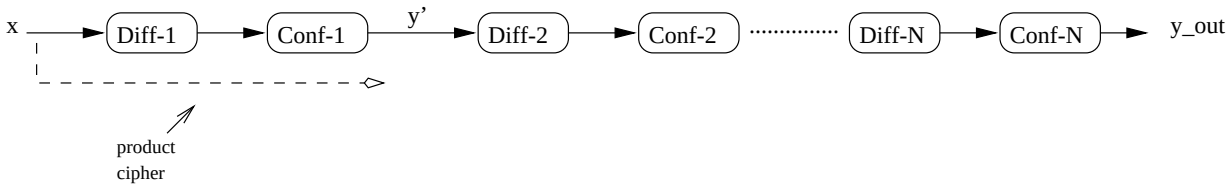


Figure 3.1: Example of combining confusion with diffusion

3.2 Introduction to DES

General Notes:

- DES is by far the most popular symmetric-key algorithm.
- It was published in 1975 and standardized in 1977.
- Expired in 1998.

System Parameters:

→ block cipher.

→ 64 input/output bits.

→ 56 bits of key.

Principle: 16 rounds of encryption.

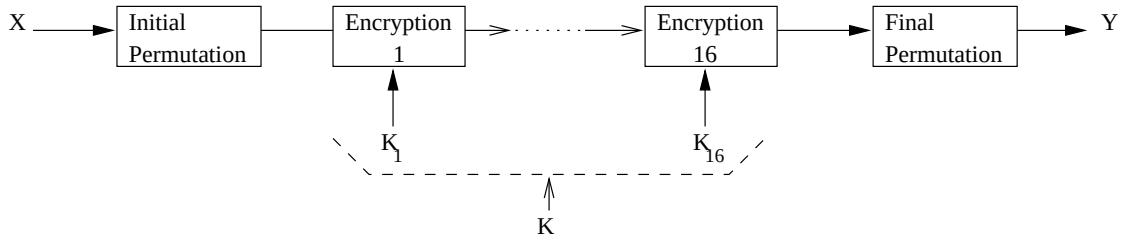


Figure 3.2: General Model of DES

3.2.1 Overview

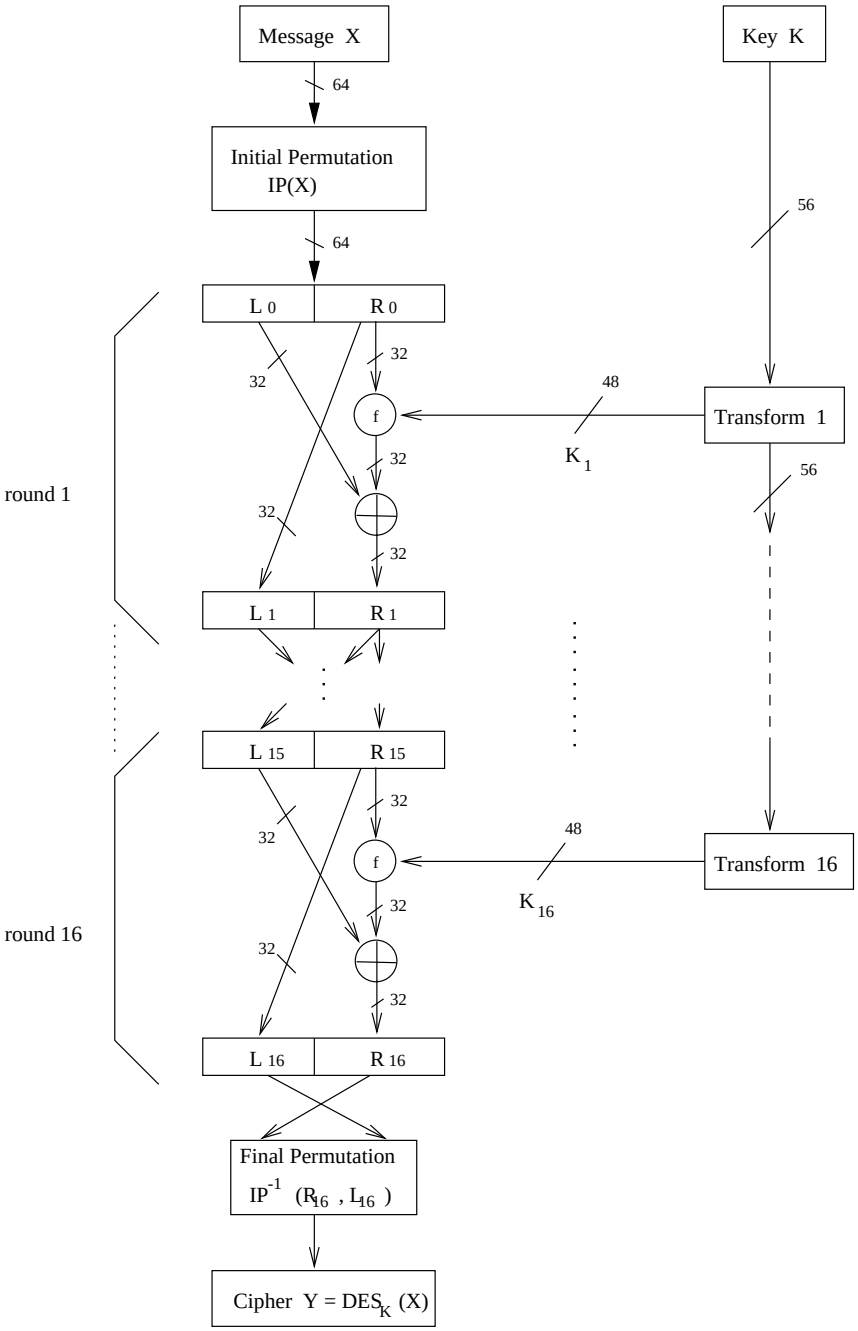


Figure 3.3: The Feistel Network

3.2.2 Permutations

a) Initial Permutation IP.

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

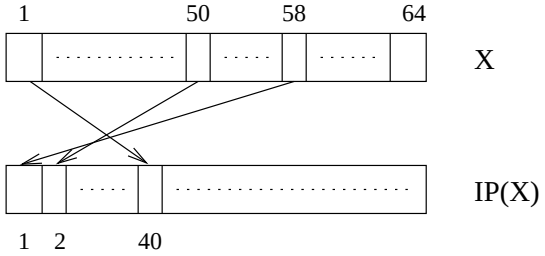


Figure 3.4: Initial permutation

b) Inverse Initial Permutation IP^{-1} (final permutation).

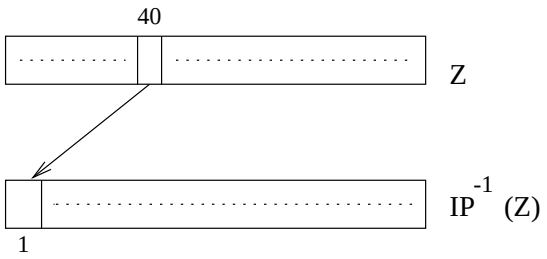


Figure 3.5: Final permutation

Note:

$IP^{-1}(IP(X)) = X$.

3.2.3 Core Iteration / f-Function

General Description:

$L_i = R_{i-1}$.

$R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$.

The core iteration is the f-function that takes the right half of the output of the previous round and the key as input.

	E	bit	table		
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

S-boxes:

Contain look-up tables (LUTs) with 64 numbers ranging from $0 \ldots 15$.

Input: Six bit code selecting one number.

Output: Four bit binary representation of one number out of 64.

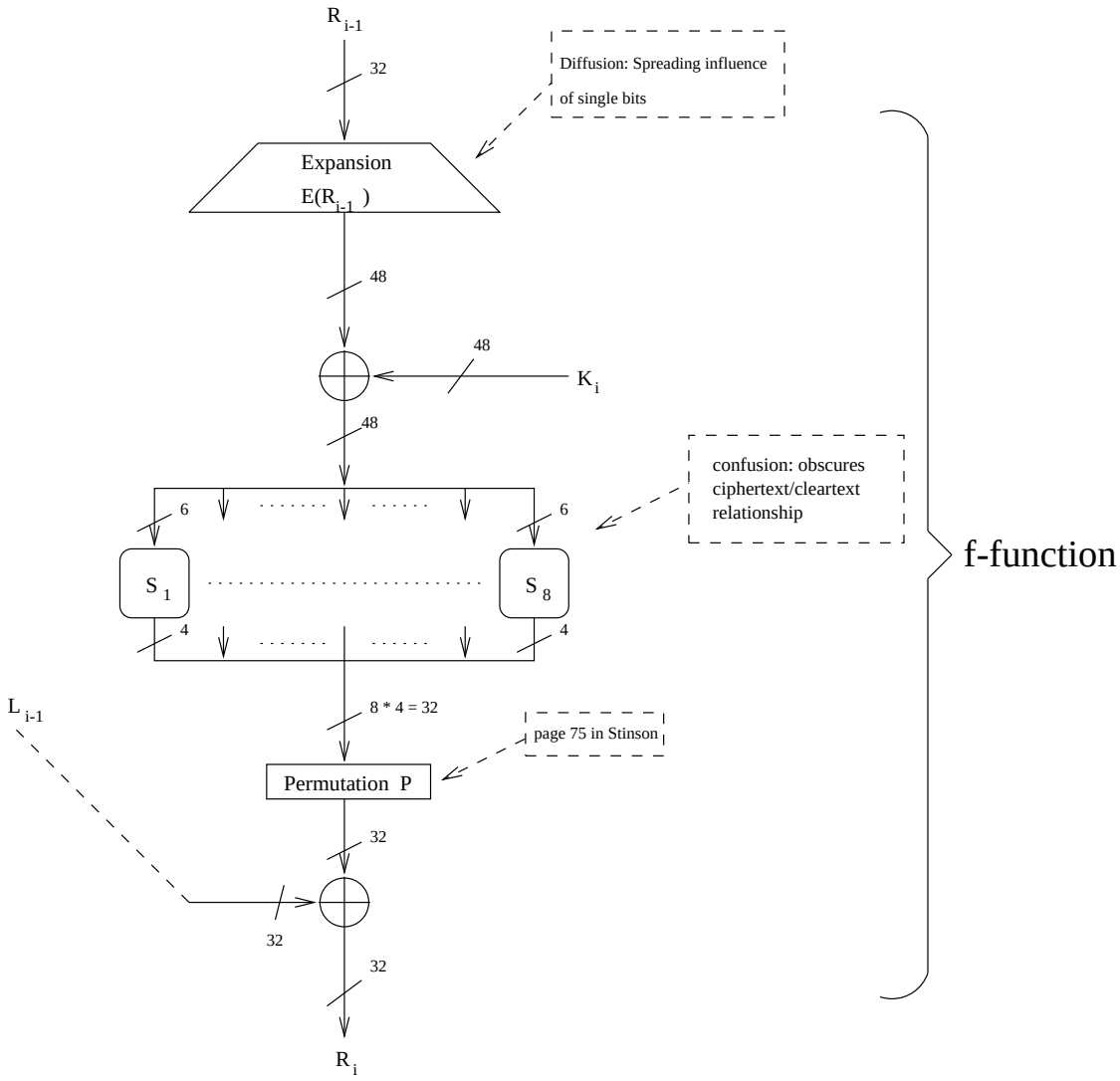


Figure 3.6: Core function of DES

Example:

S1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-Box 1

Input: Six bit vector with MSB and LSB selecting the row and four inner bits selecting column.

$b = (100101)$.

$\rightarrow$ row $= (11)_2 = 3$ (forth row).

$\rightarrow$ column $= (0010)_2 = 2$ (third column).

$S_1(37 = 100101_2) = 8 = 1000_2$.

Remark:

S-boxes are the most crucial elements of DES because they introduce a **non-linear** function to the algorithm, i.e., $S(a)$ XOR $S(b) \neq S(a$ XOR $b)$.

3.2.4 Key Schedule

Note:

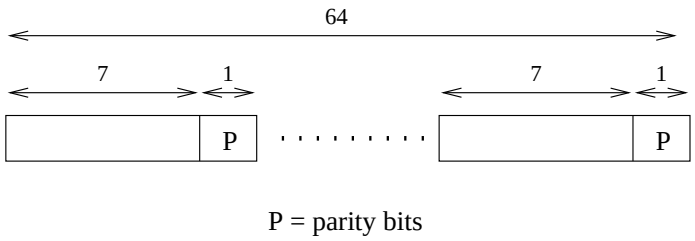


Figure 3.7: 64 bit DES block

In practice the DES key is artificially enlarged with odd parity bits. These bits are "stripped" in PC-1.

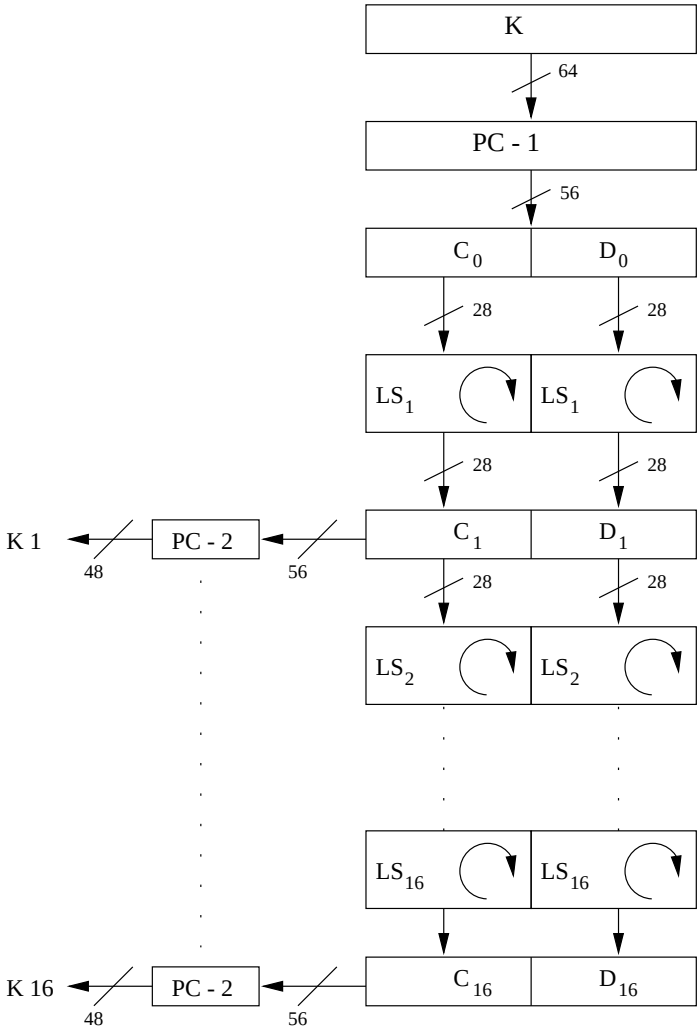


Figure 3.8: DES key scheduler

The cyclic Left-Shift (LS) blocks have two modes of operation:

a) for LS_i where $i = 1, 2, 9, 16$, the block is shifted once.

b) for LS_i where $i \neq 1, 2, 9, 16$, the block is shifted twice.

Remark:

The total number of cyclic Left-Shifts is $4 \cdot 1 + 12 \cdot 2 = 28$. As a results of this $C_0 = C_{16}$ and $D_0 = D_{16}$.

3.3 Decryption

One advantage of DES is that decryption is essentially the same as encryption. Only the key schedule is reversed. This is due to the fact that DES is based on a Feistel network.

Question: Why does decryption work essentially the same as encryption?

a) Find what happens in the initial stage of decryption!

$(L_0^d, R_0^d) = IP(Y) = IP(IP^{-1}(R_{16}, L_{16})) = (R_{16}, L_{16})$.

$(L_0^d, R_0^d) = IP(Y) = (R_{16}, L_{16})$.

$L_0^d = R_{16}$.

$R_0^d = L_{16} = R_{15}$.

b) Find what happens in the iterations!

What are (L_1^d, R_1^d) ?

$L_1^d = R_0^d = L_{16} = R_{15}$.

substitute into the above equation to get:

$R_1^d = L_0^d \oplus f(R_0^d, k_{16}) = R_{16} \oplus f(L_{16}, k_{16})$.

$R_1^d = [L_{15} \oplus f(R_{15}, k_{16})] \oplus f(R_{15}, k_{16})$.

$R_1^d = L_{15} \oplus [f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16})] = L_{15}$.

in general: $L_i^d = R_{16-i}$ and $R_i^d = L_{16-i}$;

such that: $L_{16}^d = R_{16-16} = R_0$ and $R_{16}^d = R_0$.

c) Find what happens in the final stage!

$IP^{-1}(R_{16}^d, L_{16}^d) = IP^{-1}(L_0, R_0) \doteq IP^{-1}(IP(X)) = X$ *q.e.d.*

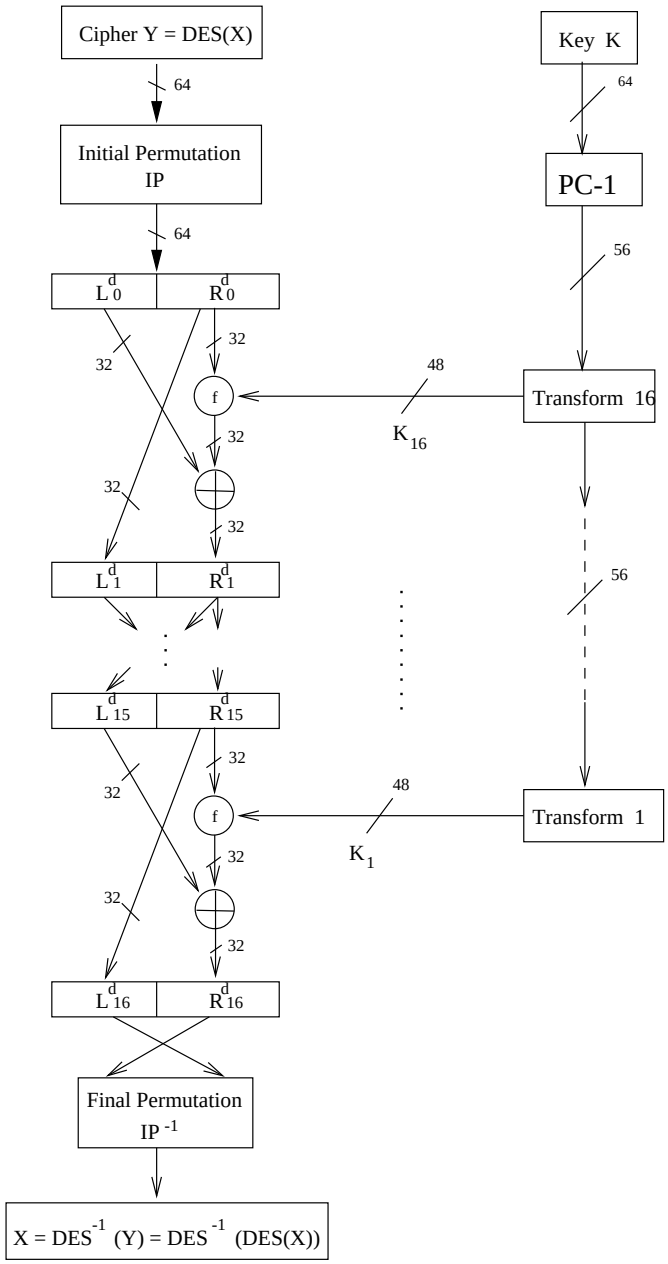


Figure 3.9: Decryption of DES

Reversed Key Schedule:

Question: Given K, how can we easily generate k_{16}?

$k_{16} = PC2(C_{16}, D_{16}) = PC2(C_0, D_0) = PC2(PC1(k))$.

$k_{15} = PC2(C_{15}, D_{15}) = PC2(RS_1(C_{16}), RS_1(D_{16})) = PC2(RS_1(C_0), RS_1(D_0))$.

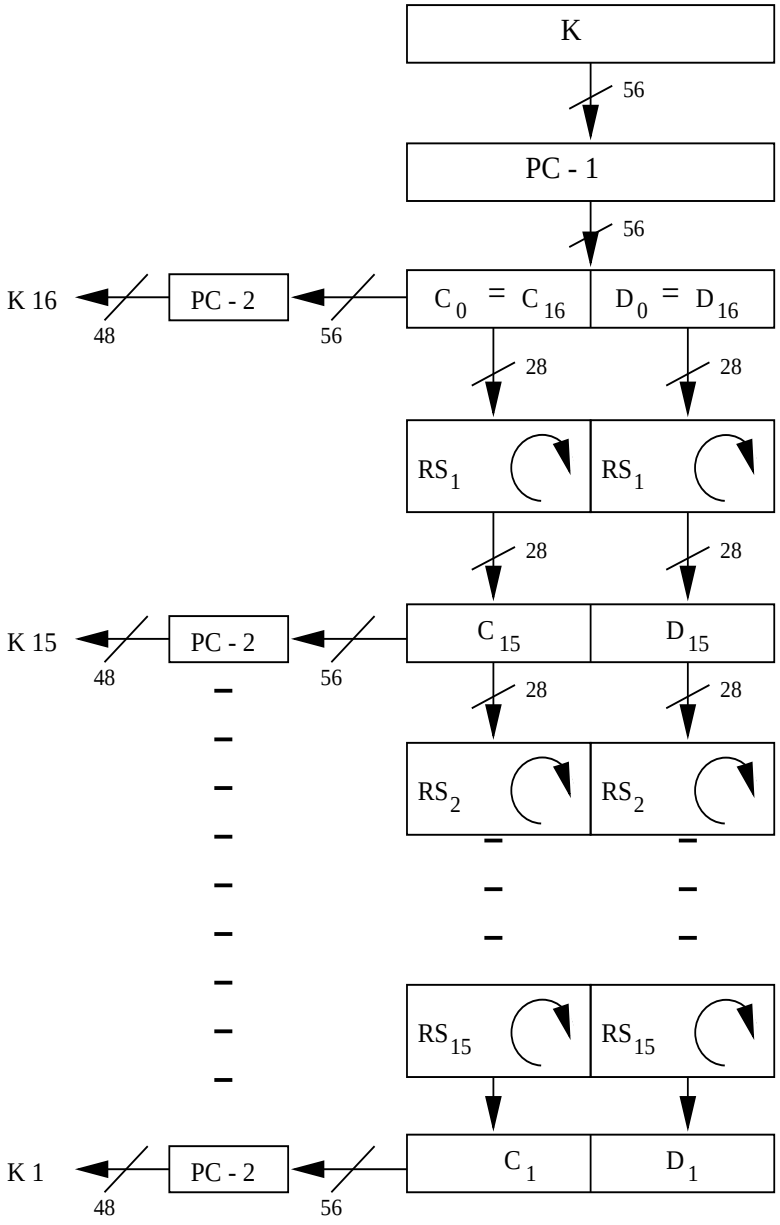


Figure 3.10: Reversed key scheduler for decryption of DES

3.4 Implementation

Note:
One design criteria for DES was fast hardware implementation.

3.4.1 Hardware

Since permutations and simple table look-ups are fast in hardware, DES can be implemented very efficiently. An implementation of a single DES round can be done with approximately 5000 gates.

1. One of the faster reported ASIC implementations: 9 Gbit/s in 0.6 μm technology with 16 stage pipeline [WPR+99].

2. A highly optimized FPGA implementation with 12 Gbit/s is described in [TPS00].

3.4.2 Software

A straightforward software implementation which follows the data flow of most DES descriptions, such as the one presented in this chapter, results in a very poor performance! There have been numerous method proposed for accelerating DES software implementations. Here are two representative ones:

1. "Bit-slicing" techniques developed by Eli Biham [Bih97]. Performance on a 300MHz DEC Alpha: 137 Mbit/sec.

2. The well known and fairly fast crypto library Crypto++ by Weidai claims a performance of about 100Mbit/sec on a 850 MHz Celeron processor. See also `http://www.eskimo.co`

3.5 Attacks

There have been two major points of criticism about DES from the beginning:
i) key size is too small (allowing a brute-force attack),
ii) the S-boxes contained secret design criteria (allowing an analytical attack).

3.5.1 Exhaustive Key Search

Known Plaintext Attack:
known: X and Y.
unknown: K, such that $Y = DES_k(X)$.
idea: test all 2^{56} possible keys $\rightarrow DES_{k_i}(X) \stackrel{?}{=} Y$; $i = 0, 1, \ldots, 2^{56} - 1$.

Date	Proposed/implemented attack
1977	Diffie & Hellman, estimate cost of key search machine (underestimate)
1990	Biham & Shamir propose differential cryptoanalysis (2^{47} chosen ciphertexts)
1993	Mike Wiener proposes detailed hardware design for key search machine: average search time of 36 h @ $100,000
1993	Matsui proposes linear cryptoanalysis (2^{43} chosen ciphertexts)
Jun. 1997	DES Challenge I broken, distributed effort took 4.5 months
Feb. 1998	DES Challenge II–1 broken, distributed effort took 39 days
Jul. 1998	DES Challenge II–2 broken, key-search machine built by the Electronic Frontier Foundation (EFF), 1800 ASICs, each with 24 search units, $250K, 15 days average (actual time 56 hours)
Jan. 1999	DES Challenge III broken, distributed effort combined with EFF's key-search machine, it took 22 hours and 15 minutes.

Table 3.1: History of full-round DES attacks

3.6 DES Alternatives

There exists a wealth of other block ciphers. A small collection of as of yet unbroken ciphers is:

Algorithm	*I/O bits*	*Key Lengths*	*Remark*
AES/Rijndael	128	128/192/256	DES "successor", US federal standard
Triple DES	64	112 (effective)	most conservative choice
Mars	128	128/192/256	AES finalist
RC6	128	128/192/256	AES finalist
Serpent	128	128/192/256	AES finalist
Twofish	128	128/192/256	AES finalist
IDEA	64	128	patented

3.7 Lessons Learned — DES

- Standard DES with 56 bits key length can relatively easily be broken nowadays through an exhaustive key search.
- DES is very robust against known analytical attacks: DES is resistant against differential and linear cryptanalysis. However, the key length is too short.
- DES is only reasonably efficient in software but very fast and small in hardware.
- The most conservative alternative to DES is triple DES which has Effective key lengths of 112 bits.

Chapter 4

Rijndael – The Advanced Encryption Standard

4.1 Introduction

4.1.1 Basic Facts about AES

- Successor to DES.
- The AES selection process was administered by NIST.
- Unlike DES, the AES selection was an open (i.e., public) process.
- Likely to be the dominant secret-key algorithm in the next decade.
- Main AES requirements by NIST:
  - Block cipher with 128 I/O bits
  - Three key lengths must be supported: 128/192/256 bits
  - Security relative to other submitted algorithms
  - Efficient software and hardware implementations
- See `http://www.nist.gov/aes` for further information on AES

4.1.2 Chronology of the AES Process

- Development announced on January 2, 1997 by the National Institute of Standards and Technology (NIST).
- 15 candidate algorithms accepted on August 20th, 1998.
- 5 finalists announced on August 9th, 1999
  - Mars, IBM Corporation.
  - RC6, RSA Laboratories.
  - Rijndael, J. Daemen & V. Rijmen.
  - Serpent, Eli Biham et al.
  - Twofish, B. Schneier et al.
- Monday October 2nd, 2000, NIST chooses Rijndael as the AES.

A lot of work went into software and hardware performance analysis of the AES candidate algorithms. Here are representative numbers:

Algorithm	Pentium-Pro @ 200 MHz (Mbit/sec)	FPGA Hardware (Gbit/sec) [EYCP01]
MARS	69	–
RC6	105	2.4
Rijndael	71	2.1
Serpent	27	4.9
Twofish	95	1.6

Table 4.1: Speeds of the AES Finalists in Hardware and Software

4.2 Rijndael Overview

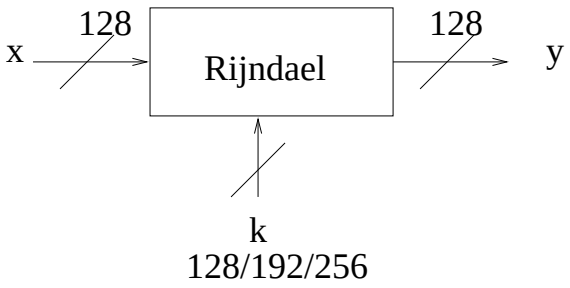


Figure 4.1: AES Block and Key Sizes

- Both block size and key length of Rijndael are variable. Sizes shown in Figure 4.2 are the ones required by the AES Standard. The number of rounds (or iterations) is a function of the key length:

Key lengths (bits)	n_r = # rounds
128	10
192	12
256	14

Table 4.2: Key lengths and number of rounds for Rijndael

- However, Rijndael also allows *block sizes* of 192 and 256 bits. For those block sizes the number of rounds must be increased.

Important: Rijndael does *not* have a Feistel structure. Feistel networks do not encrypt an entire block per iteration (e.g., in DES, $64/2 = 32$ bits are encrypted in one iteration). Rijndael encrypts all 128 bits in one iteration. As a consequence, Rijndael has a comparably small number of rounds.

Rijndael uses three different types of layers. Each layer operates on all 128 bits of a block:

1. *Key Addition Layer:* XORing of subkey.

2. *Byte Substitution Layer:* 8-by-8 SBox substitution.

3. *Diffusion Layer:* provides diffusion over all 128 (or 192 or 256) block bits. It is split in two sub-layers:

   (a) ShiftRow Layer.

   (b) MixColumn Layer.

Remark: The ByteSubstitution Layer introduces confusion with a non-linear operation. The ShiftRow and MixColumn stages form a linear Diffusion Layer.

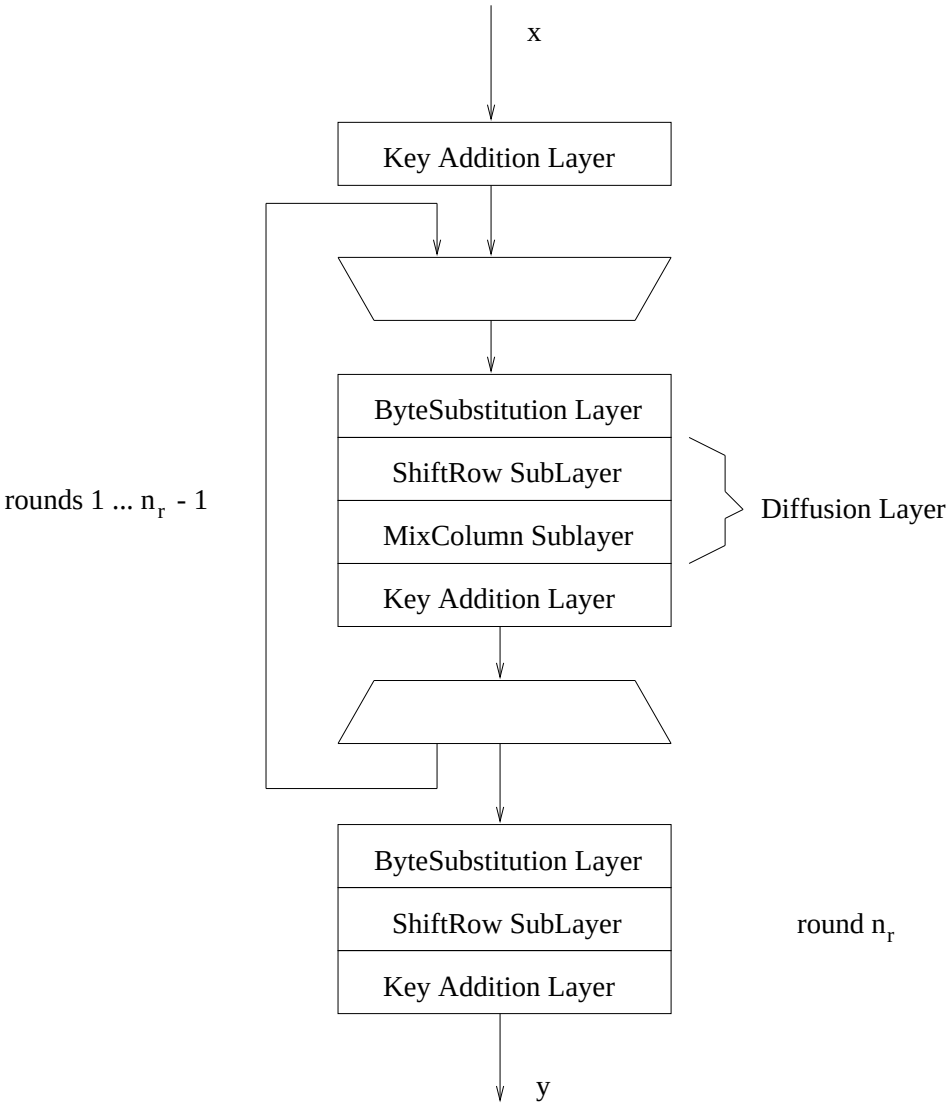


Figure 4.2: Rijndael encryption block diagram

4.3 Some Mathematics: A Very Brief Introduction to Galois Fields

"Galois fields" are used to perform substitution and diffusion in Rijndael.

Question: What are Galois fields?

Galois fields are *fields* with a finite number of elements. Roughly speaking, a field is a structure in which we ca add, subtract, multiply, and compute inverses. More exactly a field is a ring in which all elements except 0 are invertible.

Theorem 1 *Let p be a prime. $GF(p)$ is a "prime field," i.e., a Galois field with a prime number of elements. All arithmetic in $GF(p)$ is done modulo p.*

Example: $GF(3) = \{0, 1, 2\}$

addition

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

additive inverse

$-0 = 0$

$-1 = 2$

$-2 = 1$

multiplication

$\times$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

multiplicative inverse

0^{-1} does not exist

$1^{-1} = 1$

$2^{-1} = 2$, since $2 \cdot 2 \equiv 1 \bmod 3$

Theorem 4.3.1 *For every power p^m, p a prime and m a positive integer, there exists a finite field with p^m elements, denoted by $GF(p^m)$.*

Examples:

- $GF(5)$ is a finite field.
- $GF(256) = GF(2^8)$ is a finite field.
- $GF(12) = GF(3{\cdot}2^2)$ is **NOT** a finite field (in fact, the notation is already incorrect and you should pretend you never saw it).

Question: How to build "extension fields" $GF(p^m)$, $m > 1$?

Note: See also [Sti02]

1. **Represent** elements as polynomials with m coefficients. Each coefficient is an element of $GF(p)$.
   Example: $A \in GF(2^8)$
   $A \to A(x) = a_7x^7 + \cdots + a_1x + a_0, \quad a_i \in GF(2) = \{0, 1\}$

2. **Addition and subtraction in** $GF(p^m)$
   $C(x) = A(x) + B(x) = \sum_{i=0}^{i=m-1} c_ix^i, \quad c_i = a_i + b_i \bmod p$
   Example: $A, B \in GF(2^8)$

$$\begin{array}{lllllll} A(x) & = x^7+ & x^6+ & x^4+ & & & 1 \\ B(x) & = & & x^4+ & x^2+ & & 1 \\ \hline C(x) & = x^7+ & x^6+ & & x^2 & & \end{array}$$

3. **Multiplication in** $GF(p^m)$: multiply the two polynomials using polynomial multiplication rule, with coefficient arithmetic done in $GF(p)$. The resulting polynomial will have degree $2m - 2$.

$$\begin{aligned} A(x) \cdot B(x) &= (a_{m-1}x^{m-1} + \cdots + a_0) \cdot (b_{m-1}x^{m-1} + \cdots + b_0) \\ C'(x) &= c'_{2m-2}x^{2m-2} + \cdots + c'_0 \end{aligned}$$

where:

$$\begin{aligned} c'_0 &= a_0 b_0 \bmod p \\ c'_1 &= a_0 b_1 + a_1 b_0 \bmod p \\ &\vdots \\ c'_{2m-2} &= a_{m-1} b_{m-1} \bmod p \end{aligned}$$

Question: How to reduce $C'(x)$ to a polynomial of maximum degree $m-1$?

Answer: Use modular reduction, similar to multiplication in $GF(p)$. For arithmetic in $GF(p^m)$ we need an *irreducible polynomial* of degree m with coefficients from $GF(p)$. Irreducible polynomials do not factor (except trivial factor involving 1) into smaller polynomials from $GF(p)$.

Example 1: $P(x) = x^4 + x + 1$ is irreducible over $GF(2)$ and can be used to construct $GF(2^4)$.

$C = A \cdot B \Rightarrow C(x) = A(x) \cdot B(x) \bmod P(x)$

$$\begin{aligned} A(x) &= x^3 + x^2 + 1 \\ B(x) &= x^2 + x \\ C'(x) &= A(x) \cdot B(x) = (x^5 + x^4 + x^2) + (x^4 + x^3 + x) = x^5 + x^3 + x^2 + 1 \\ x^4 &= 1 \cdot P(x) + (x + 1) \\ x^4 &\equiv x + 1 \bmod P(x) \\ x^5 &\equiv x^2 + x \bmod P(x) \\ C(x) &\equiv C'(x) \bmod P(x) \\ C(x) &\equiv (x^2 + x) + (x^3 + x^2 + 1) = x^3 \\ A(x) \cdot B(x) &\equiv x^3 \end{aligned}$$

Note: in a typical computer representation, the multiplication would assign the following unusually looking operations:

$$\begin{array}{ccccc} A & \cdot & B & = & C \\ (1\ 1\ 0\ 1) & \cdot & (0\ 1\ 1\ 0) & = & (1\ 0\ 0\ 0) \end{array}$$

Example 2: $x^4 + x^3 + x + 1$ is reducible since $x^4 + x^3 + x + 1 = (x^2 + x + 1)(x^2 + 1)$.

4. **Inversion in $GF(p^m)$:** the inverse A^{-1} of $A \in GF(p^m)^*$ is defined as:
   $A^{-1}(x) \cdot A(x) = 1 \bmod P(x)$
   $\Rightarrow$ perform the Extended Euclidean Algorithm with $A(x)$ and $P(x)$ as inputs

$$\begin{aligned} s(x)P(x) + t(x)A(x) &= \gcd(P(x), A(x)) = 1 \\ \Rightarrow t(x)A(x) &= 1 \bmod P(x) \\ \Rightarrow t(x) = A^{-1}(x) & \end{aligned}$$

   Example: Inverse of $x^2 \in GF(2^3)$, with $P(x) = x^3 + x + 1$

	$t_0 = 0,\ t_1 = 1$
$x^3 + x + 1 = [x]x^2 + [x + 1]$	$t_2 = t_0 - q_1 t_1 = -q_1 = -x = x$
$x + 1 = [1]x + 1$	$t_3 = t_1 - q_2 t_2 = 1 - q_2 x = 1 - x = x + 1$
$x = [x]1 + 0$	
$\Rightarrow (x^2)^{-1} = t(x) = t_3 = x + 1$	

   Check: $(x + 1)x^2 = x^3 + x = (x + 1) + x \equiv 1 \bmod P(x)$ since $x^3 \equiv x + 1 \bmod P(x)$.

Remark: In every iteration of the Euclidean algorithm, you should use long division (not shown above) to uniquely determine q_i and r_i.

4.4 Internal Structure

In the following, we assume a block length of 128 bits. The ShiftRow Sublayer works slightly differently for other block sizes.

4.4.1 Byte Substitution Layer

- Splits the incoming 128 bits in $128/8 = 16$ bytes.
- Each byte A is considered an element of $GF(2^8)$ and undergoes the following substitution individually

1. $B = A^{-1} \in GF(2^8)$ where $P(x) = x^8 + x^4 + x^3 + x + 1$

2. Apply affine transformation defined by:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

where $(b_7 \cdots b_0)$ is the vector representation of $B(x) = A^{-1}(x)$.

- The vector $C = (c_7 \cdots c_0)$ (representing the field element $c_7x^7 + \cdots + c_1x + c_0$) is the result of the substitution:

$$C = \text{ByteSub}(A)$$

The entire substitution can be realized as a look-up in a 256×8-bit table with fixed entries.

Remark: Unlike DES, Rijndael applies the same S-Box to each byte.

4.4.2 Diffusion Layer

- Unlike the non-linear substitution layer, the diffusion layer performs a linear operation on input words A, B. That means:

$$\text{DIFF}(A) \oplus \text{DIFF}(B) = \text{DIFF}(A + B)$$

- The diffusion layer consists of two sublayers.

ShiftRow Sublayer

1. Write an input word A as $128/8 = 16$ bytes and order them in a square array:
   Input $A = (a_0, a_1, \cdots, a_{15})$

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}
a_2	a_6	a_{10}	a_{14}
a_3	a_7	a_{11}	a_{15}

2. Shift cyclically row-wise as follows:

a_0	a_4	a_8	a_{12}		0 positions
a_5	a_9	a_{13}	a_1	$---\longrightarrow$	3 positions right shift
a_{10}	a_{14}	a_2	a_6	$--\longrightarrow$	2 positions right shift
a_{15}	a_3	a_7	a_{11}	$-\longrightarrow$	1 position right shift

MixColumn Sublayer

Principle: each column of 4 bytes is individually transformed into another column.

Question: How?

Each 4-byte column is considered as a vector and multiplied by a 4×4 matrix. The matrix contains *constant* entries. Multiplication and addition of the coefficients is done in $GF(2^8)$.

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Remarks:

1. Each c_i, b_i is an 8-bit value representing an element from $GF(2^8)$.

2. The small values $\{01, 02, 03\}$ allow for a very efficient implementation of the coefficient multiplication in the matrix. In software implementations, multiplication by 02 and 03 can be done through table look-up in a 256-by-8 table.

3. Additions in the vector-matrix multiplication are XORs.

4.4.3 Key Addition Layer

Simple bitwise XOR with a 128-bit subkey.

4.5 Decryption

Unlike DES and other Feistel ciphers, all of Rijndael layers must actually be inverted.

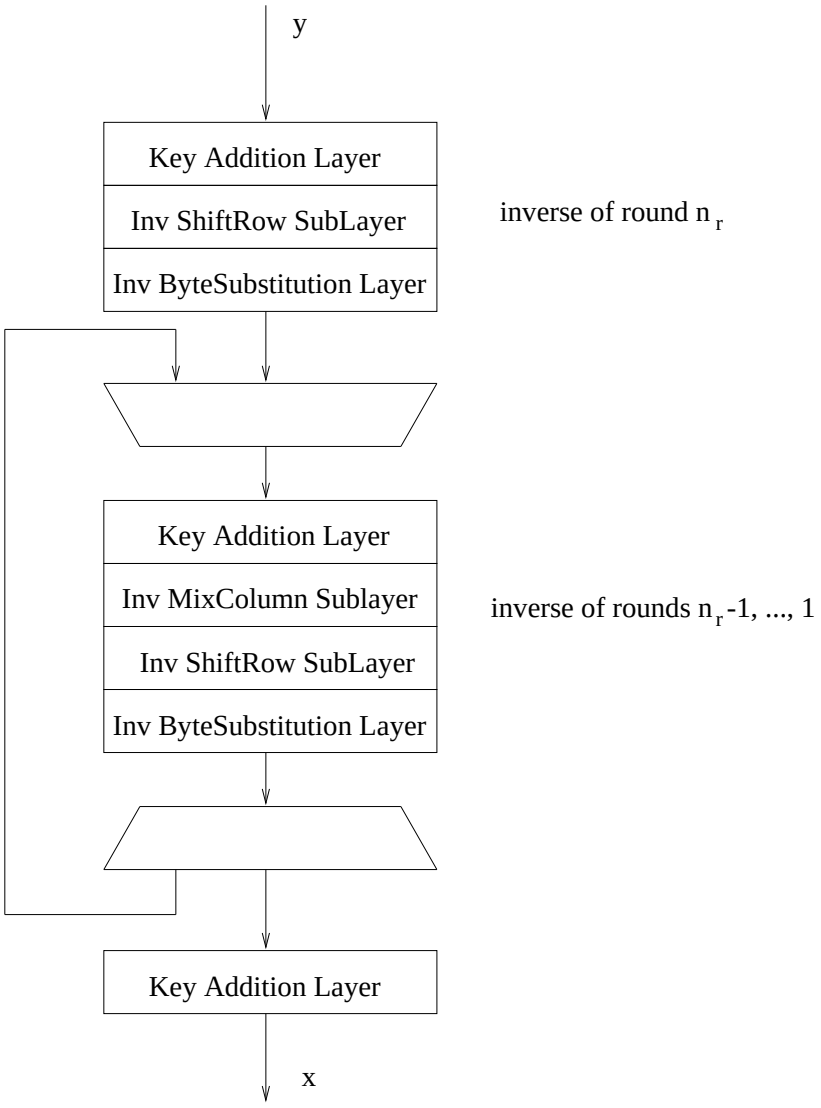


Figure 4.3: Rijndael decryption block diagram

4.6 Implementation

4.6.1 Hardware

Compared to DES, Rijndael requires considerable more hardware resources for an implementation. However, Rijndael can still be implemented with very high throughputs in modern ASIC or FPGA technology. Two representative implementation reports are:

1. A 0.6μm technology ASIC realization of Rijndael with a throughput of more than 2Gbit/sec is reported in [LTG+02]. The design encrypts four blocks in parallel.

2. Reference [EYCP01] describes an implementation (without key scheduling) of Rijndael on a Virtex 1000 Xilinx FPGA with five pipeline stages and a throughput of more than 2Gbit/sec.

4.6.2 Software

Unlike DES, Rijndael was designed such that an efficient software implementation is possible. A naive implementation of Rijndael which directly follows the data path description, such as the description given in this chapter, is not particularly efficient, though. In a naive implementation all time critical functions (Byte Substitution, Mix Row, Shift Row) operate on individual bytes. Processing 1 byte per instruction is inefficient on modern 32 or 64 bit processors.

However, the Rijndael designers proposed a method which results in fast software implementations. The core idea is to merge all round functions (except the rather trivial key addition) into one table look-up. This results in 4 tables, each of which consists of 256 entries, where each entry is 32 bit wide. These tables are named "T-Box". Four table accesses yield 32 bit output bits of one round. Hence, one round can be computed with 16 table look-ups.

A detailed description of the construction of the T-Boxes can be found in [DR98, Section 5].

Achievable throughput: 400Mbit/sec on 1.2 GHz Intel processor.

4.7 Lessons Learned — AES

- The AES selection was an open process. It appears to be extremely unlikely that the designers included hidden weaknesses (trapdoors) in Rijndael.
- AES is efficient in software and hardware.
- The AES key lengths provide long term security against brute force attacks for several decades.
- AES is a relativley new cipher. At the moment it can not be completely excluded that there will be analytical attacks against Rijndael in the future, even though this does not seem very likely.
- The fact that AES is a "standard" is currently only relevant for US Government applications.

Chapter 5

More about Block Ciphers

Further Reading:

Section 8.1 in [Sch96].

Note:

The following modes are applicable to all block ciphers $e_k(X)$.

5.1 Modes of Operation

5.1.1 Electronic Codebook Mode (ECB)

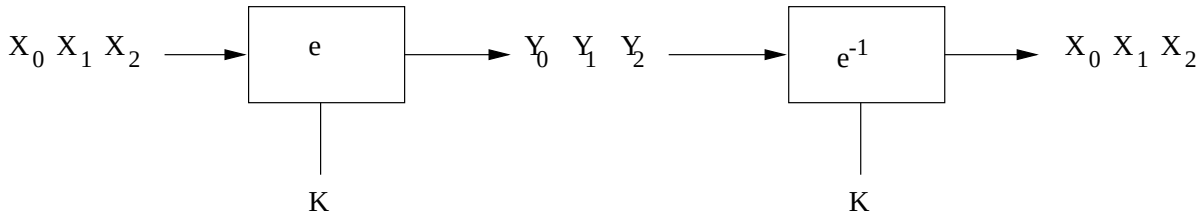


Figure 5.1: ECB model

General Description:

$e_k^{-1}(Y_i) = e_k^{-1}(e_k(X_i)) = X_i$; where the encryption can, for instance, be DES.

Problem:

This mode is susceptible to substitution attack because same X_i are mapped to same Y_i.

Example: Bank transfer.

Block #	1	2	3	4	5
	Sending Bank A	Sending Account #	Receiving Bank B	Receiving Account #	Amount $

Figure 5.2: ECB example

1. Tap encrypted line to bank B.
2. Send $1.00 transfer to own account at bank B repeatedly → block 4 can be identified and recorded.
3. Replace in all messages to bank B block 4.
4. Withdraw money and fly to Paraguay.

Note: This attack is possible only for single-block transmission.

5.1.2 Cipher Block Chaining Mode (CBC)

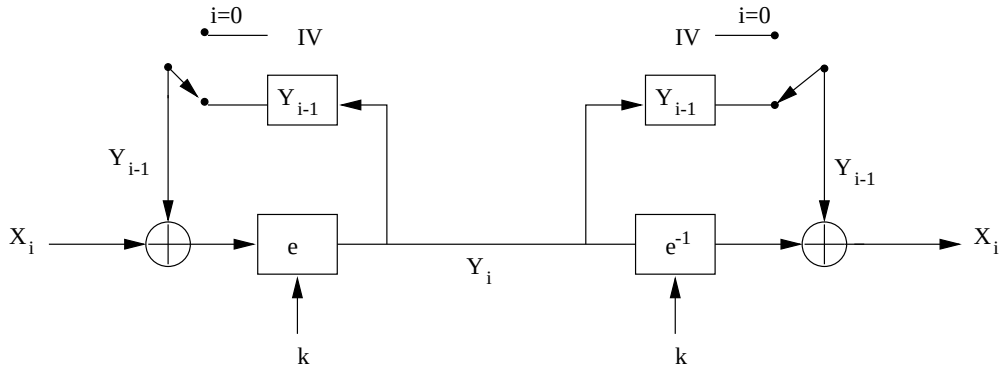


Figure 5.3: CBC model

Beginning: $Y_0 = e_k(X_0 \oplus IV)$.

$X_0 = IV \oplus e_k^{-1}(Y_0) = IV \oplus e_k^{-1}(e_k(X_0 \oplus IV)) = X_0$.

Encryption: $Y_i = e_k(X_i \oplus Y_{i-1})$.

Decryption: $X_i = e_k^{-1}(Y_i) \oplus Y_{i-1}$.

Question: How does it work?

$X_i = e_k^{-1}(e_k(X_i \oplus Y_{i-1})) \oplus Y_{i-1}$.

$X_i = (X_i \oplus Y_{i-1}) \oplus Y_{i-1}$.

$X_i = X_i$. *q.e.d.*

Remark: The Initial Vector (IV) can be transmitted initially in cleartext.

5.1.3 Cipher Feedback Mode (CFB)

Assumption: block cipher with b bits block width and message with block width l, $1 \leq l \leq b$.

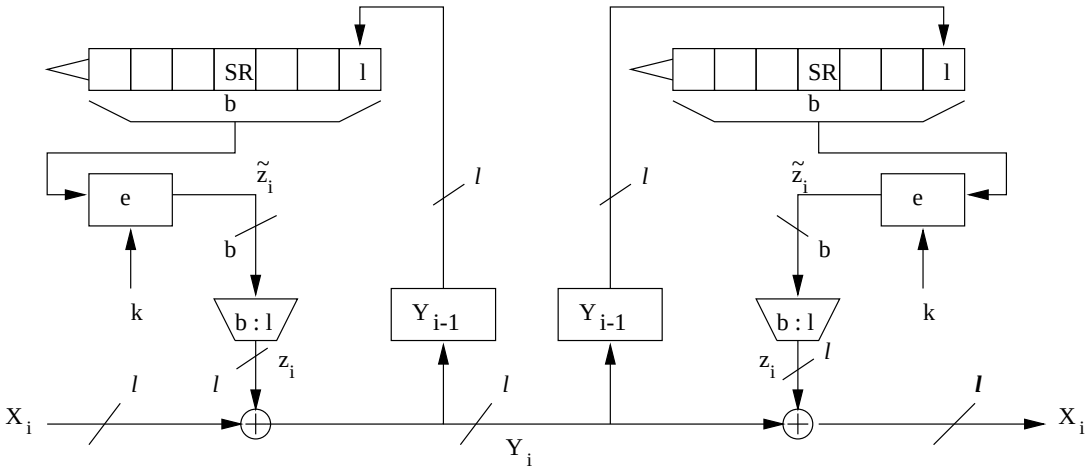


Figure 5.4: CFB model

Procedure:

1. Load shift register with initial value IV.
2. Encrypt $e_k(IV) = \tilde{z}_0$.
3. Take l leftmost bits: $\tilde{z}_0 \rightarrow z_0$.
4. Encrypt data: $Y_0 = X_0 \oplus z_0$.
5. Shift the shift register and load Y_0 into the rightmost SR position.
6. Go back to (2) substituting $e(IV)$ with $e(SR)$.

5.1.4 Counter Mode

Notes:

- Another mode which uses a block cipher as a pseudo-random generator.
- Counter Mode does not rely on previous ciphertext for encrypting the next block.
  $\Rightarrow$ well suited for parallel (hardware) implementation, with several encryption blocks working in parallel.
- Counter Mode stems from the Security Group of the ATM Forum, where high data rates required parallelization of the encryption process.

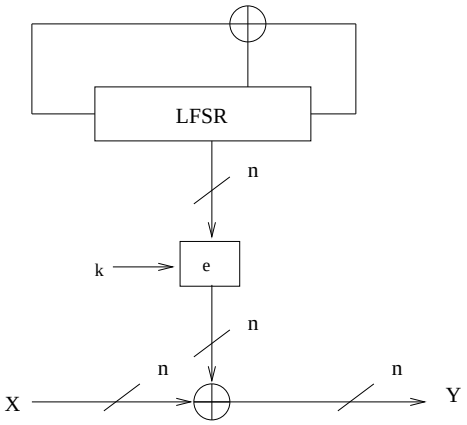


Figure 5.5: Counter Mode model

Description of Counter Mode:

1. An n-bit initial vector (IV) is loaded into a (maximum length) LFSR. The IV can be publically known, although a secret IV (i.e., the IV is considered part of the private key) turns the counter mode systems into a non-deterministic cipher which makes cryptoanalysis harder.
2. Encrypt block cipher input.
3. The block cipher output is considered a pseudorandom mask which is XORed with the plaintext.

4. The LFSR is clocked once (note: **all** input bits of the block cipher are shifted by one position).

5. Goto to Step 2.

Note that the period of a counter mode is $n \cdot 2^n$ which is very large for modern block ciphers, e.g., $128 \cdot 2^{128} = 2^{135}$ for AES algorithms.

5.2 Key Whitening

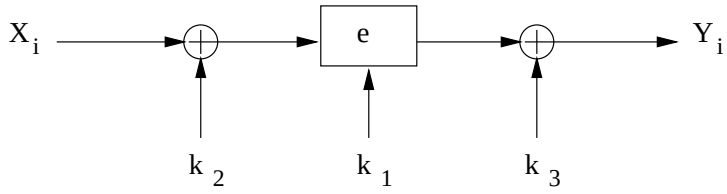


Figure 5.6: Whitening example

Encryption: $Y = e_{k_1,k_2,k_3}(X) = e_{k_1}(X \oplus k_2) \oplus k_3$.

Decryption: $X = e_{k_1}^{-1}(Y \oplus k_3) \oplus k_2$.

popular example: DESX

5.3 Multiple Encryption

5.3.1 Double Encryption

Note: The keyspace of this encryption is $|k| = 2^k \cdot 2^k = 2^{2k}$.

However, using the meet-in-the-middle attack, the key search is reduced significantly.

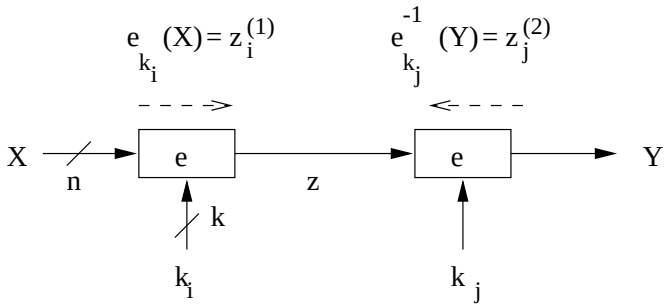


Figure 5.7: Double encryption and meet-in-the-middle attack

Meet in the middle attack:

Input → some pairs (x', y'), (x'', y''), ...

Idea → compute $z_i^{(1)} = e_{k_i}(x')$ and $z_j^{(2)} = e_{k_j}^{-1}(y')$.

Problem → to find a matching pair such that $z_i^{(1)} = z_j^{(2)}$.

Procedure:

1. Compute a look-up table for all $(z_i^{(1)}, k_i)$, $i = 1, 2, \ldots, 2^k$ and store it in memory. Number of entries in the table is 2^k with each entry being n bits wide.

2. Find matching $z_j^{(2)}$.

   (a) compute $e_{k_j}^{-1}(y') = z_j^{(2)}$

   (b) if $z_j^{(2)}$ is in the look-up table, i.e., if $z_i^{(1)} = z_j^{(2)}$, check a few other pairs (x'', y''), (x''', y'''), ... for the current keys k_i and k_j

(c) if k_i and k_j give matching encryptions stop; otherwise go back to (a) and try different key k_j.

Question: How many additional pairs $(x'', y''), (x''', y'''), \ldots$ should we test?

General system: l subsequent encryptions and t pairs $(x', y'), (x'', y''), \ldots$.

1. In the first step there are 2^{lk} possible key combinations for the mapping $E(x') = e(\cdots(e(e(x'))\cdots) = y'$ but only 2^n possible values for x' and y'. Hence, there are

$$\frac{2^{lk}}{2^n}$$

mappings $E(x') = y'$. Note that only one mapping is done by the correct key!

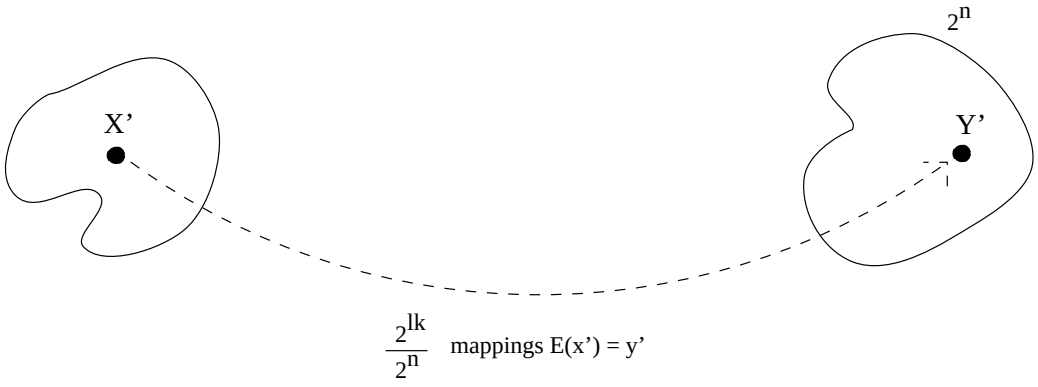


Figure 5.8: Number of mappings x' to y' under l-fold encryption

2. We use now a candidate key from step 1 and check whether $E(x'') = y''$. There are 2^n possible outcomes y for the mapping $E(x'')$. If a random key is used, the likelyhood that $E(x'') = y''$ is

$$\frac{1}{2^n}$$

If we check additionally a third pair (x''', y''') under the same "random" key from step 1, the likelyhood that $E(x'') = y''$ *and* $E(x''') = y'''$ is

$$\frac{1}{2^{2n}}$$

If we check $t-1$ additional pairs $(x'', y''), (x''', y'''), \ldots (x^{(t)}, y^{(t)})$ the likelyhood that a random key fulfills $E(x'') = y''$, $E(x''') = y''', \ldots$ is

$$\frac{1}{2^{(t-1)n}}$$

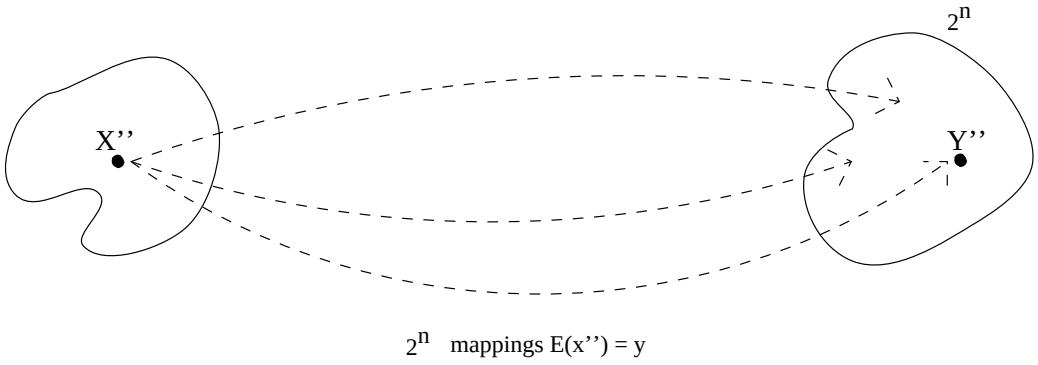


Figure 5.9: Number of mappings x'' to y

3. Since there are $\frac{2^{lk}}{2^n}$ candidate keys in step 1, the likelyhood that at least one of the candidate keys fulfills all $E(x'') = y''$, $E(x''') = y''', \ldots$ is

$$\frac{1}{2^{(t-1)n}} \frac{2^{lk}}{2^n} = 2^{lk-tn}$$

Example: Double encryption with DES. We use two pairs $(x', y'), (x'', y'')$. The likelyhood that an incorrect key pair k_i, k_j is picked is

$$2^{lk-tn} = 2^{112-128} = 2^{-16}$$

If we use three pairs $(x', y'), (x'', y''), (x''', y''')$, the likelyhood that an incorrect key pair k_i, k_j is picked is

$$2^{lk-tn} = 2^{112-192} = 2^{-80}$$

Computational complexity:

Brute force attack: 2^{2k}.

Meet in the middle attack: 2^k encryptions + 2^k decryptions = 2^{k+1} computations and 2^k memory locations.

5.3.2 Triple Encryption

Option 1:

$Y = e_{k_1}(e_{k_2}^{-1}(e_{k_3}(X)))$; if $k_1 = k_2 \rightarrow Y = e_{k_3}(X)$.

Option 2:

$Y = e_{k_3}(e_{k_2}(e_{k_1}(X)))$; where $|k| \approx 2^{2k}$

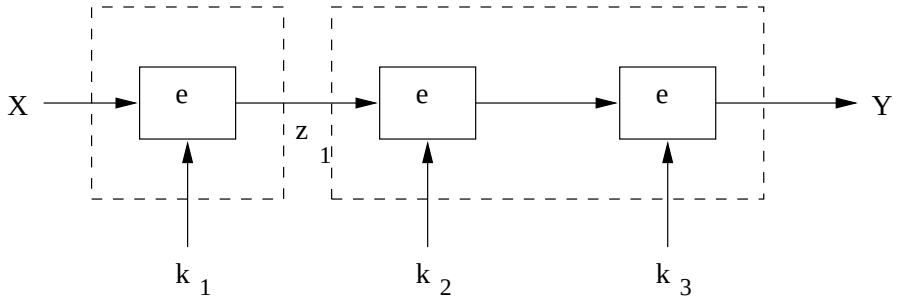


Figure 5.10: Triple encryption example

Note:

Meet in the middle attack can be used in a similar way by storing z_i results in memory. The computational complexity of this approach is $2^k \cdot 2^k = 2^{2k}$.

5.4 Lessons Learned — More About Block Ciphers

- The ECB mode has security weaknesses.
- The counter mode allows parallelization of encryption and is thus suited for high speed hardware implementations.
- Double encryption with a given block cipher only marginally improves the attack resistance against brute force attacks.
- Triple encryption with a given block cipher roughly *doubles* the key length. Triple DES ("3DES") has, thus, an effective key length of 112 bits.
- Key whithening enlarges the DES key length without too much effort.

Chapter 6

Introduction to Public-Key Cryptography

6.1 Principle

Quick review of symmetric-key cryptography

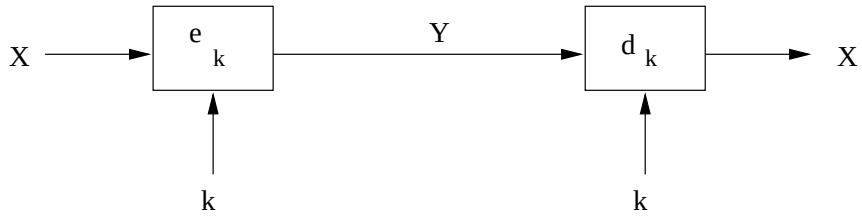


Figure 6.1: Symmetric-key model

Two properties of symmetric-key schemes:

1. The algorithm requires same secret key for encryption and decryption.

2. Encryption and decryption are essentially identical (symmetric algorithms).

Analogy for symmetric key algorithms

Symmetric key schemes are analogous to a safe box with a strong lock. Everyone with the key can deposit messages in it and retrieve messages.

Main problems with symmetric key schemes are:

1. Requires secure transmission of secret key.
2. In a network environment, each pair of users has to have a different key resulting in too many keys ($n \cdot \ (n-1) \div 2$ key pairs).

New Idea:

Make a slot in the safe box so that everyone can deposit a message, but only the receiver can open the safe and look at the content of it. This idea was proposed in [DH76] in 1976 by Diffie/Hellman.

Idea: Split key.

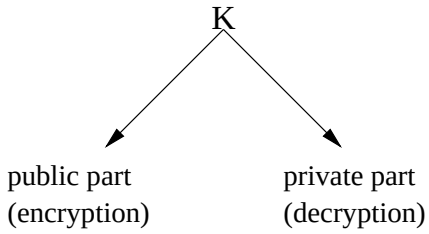


Figure 6.2: Split key idea

Protocol:

1. Alice and Bob agree on a public-key cryptosystem.
2. Bob sends Alice his public key.
3. Alice encrypts her message with Bob's public key and sends the ciphertext.
4. Bob decrypts ciphertext using his private key.

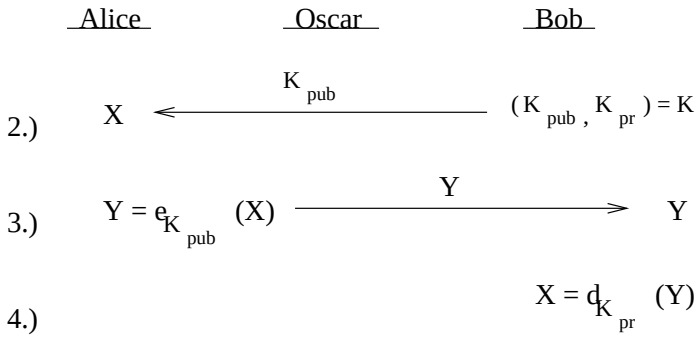


Figure 6.3: Public-key encryption protocol

Mechanisms that can be realized with public-key algorithms

1. Key establishment protocols (e.g., Diffi-Hellman key exchange) and key transport protocols (e.g., via RSA) without prior exchange of a joint secret
2. Digital signature algorithms (e.g., RSA, DSA or ECDSA)
3. Encryption

It looks as though public-key schemes can provide all functionality needed in modern security protocols such as SSL/TLS. However, the major drawback in practice is that encryption of data is extremely computationally demanding with public-key algorithms. Many block and stream ciphers can encrypt 1000 times faster in software than public-key algorithms. On the other hand, symmetric algorithms are poor at providing digital signatures and key establishment/transport functionality. Hence, most practical protocols are hybrid protocols which incorporate both symmetric and public-key algorithms.

6.2 One-Way Functions

All public-key algorithms are based on one-way functions.

Definition 6.2.1 *A function* f *is a "one-way function" if:*

(a) $y = f(x) \rightarrow$ *is easy to compute,*

(b) $x = f^{-1}(y) \rightarrow$ *is very hard to compute.*

Example: Discrete Logarithm (DL) one-way Function

$2^x \bmod 127 \equiv 31$

$x = ?$

Definition 6.2.2 *A* trapdoor one function *is a one-way function whose inverse is easy to compute given a side information such as the private key.*

6.3 Overview of Public-Key Algorithms

There are three families of Public-Key (PK) algorithms of practical relevance:

1. Integer factorization algorithms (RSA, ...)
2. Discrete logarithms (Diffie-Hellman, DSA, ...)
3. Elliptic curves (EC)

In addition, there are many other public-key schemes, such as NTRU or systems based on hidden field equations, which are not in wide spread use. Often, their security is not very well understood.

Algorithm Family	Bit length of the operands
Integer Factorization (RSA)	1024
Discrete Logarithm (D-H, DSA)	1024
Elliptic curves	160
Block cipher	80

Table 6.1: Bit lengths of public-key algorithms for a security level of **approximately** 2^{80} computations for a successful attack.

Remark: The long operands lead to a high computationally complexity of public-key algorithms. This **can** be a bottleneck in applications with constrained microprocessors (e.g., mobile applications) or on the server side of networks, where many public-key operations per time unit have to be executed.

6.4 Important Public-Key Standards

a) IEEE P1363. Comprehensive standard of public-key algorithms. Collection of IF, DL, and EC algorithm families, including in particular:

– Key establishment algorithms

– Key transport algorithms

– Signature algorithms

Note: IEEE P1363 does not recommend any bit lengths or security levels.

b) ANSI Banking Security standards.

ANSI#	Subject
X9.30–1	digital signature algorithm (DSA)
X9.30–2	hashing algorithm for RSA
X9.31–1	RSA signature algorithm
X9.32–2	hashing algorithms for RSA
X9.42	key management using Diffe-Hellman
X9.62 (draft)	elliptic curve digital signature algorithm (ECDSA)
X9.63 (draft)	elliptic curve key agreement and transport protocols

c) U.S. Government standards (FIPS)

FIPS#	Subject
FIPS 180-1	secure hash standard (SHA-1)
FIPS 186	digital signature standard (DSA)
FIPS JJJ (draft)	entity authentication (asymetric)

6.5 More Number Theory

6.5.1 Euclid's Algorithm

Basic Form

Given r_0 and r_1 with one larger than the other, compute the $\gcd(r_0, r_1)$.

Example 1:

$r_0 = 22, r_1 = 6$.

$\gcd(r_0, r_1) = ?$

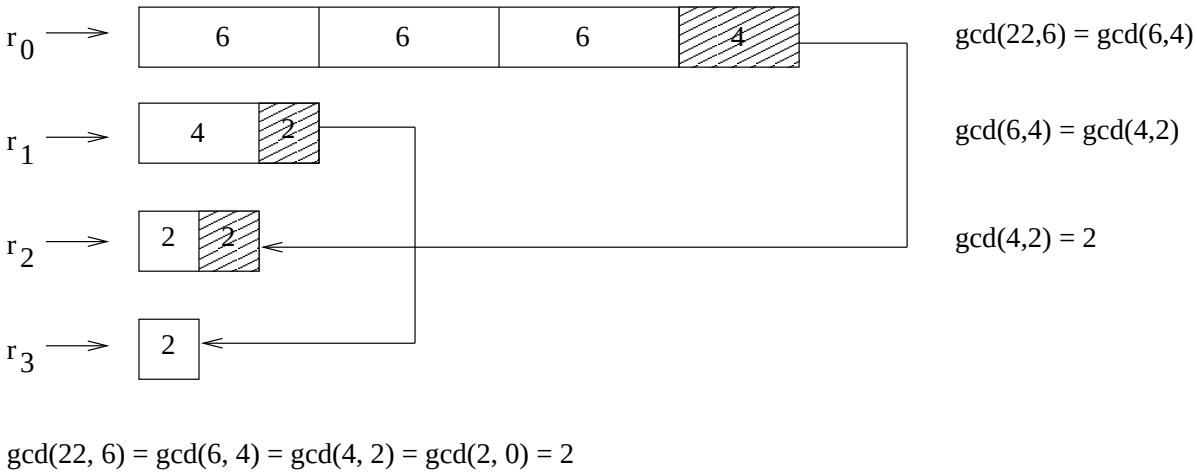


Figure 6.4: Euclid's algorithm example

Example 2:

$r_0 = 973$; $r_1 = 301$.

$973 = 3 \cdot 301 + 70$.

$301 = 4 \cdot 70 + 21$.

$70 = 3 \cdot 21 + 7$.

$21 = 3 \cdot 7 + 0$.

$\gcd(973, 301) = \gcd(301, 70) = \gcd(70, 21) = \gcd(21, 7) = 7$.

Algorithm:

input: r_0, r_1

$$
\begin{array}{ll}
r_0 = q_1 \cdot r_1 + r_2 & \gcd(r_0, r_1) = \gcd(r_1, r_2) \\
r_1 = q_2 \cdot r_2 + r_3 & \gcd(r_1, r_2) = \gcd(r_2, r_3) \\
\vdots & \vdots \\
r_{m-2} = q_{m-1} \cdot r_{m-1} + r_m & \gcd(r_{m-2}, r_{m-1}) = \gcd(r_{m-1}, r_m) \\
r_{m-1} = q_m \cdot r_m + 0 \leftarrow \dagger & \gcd(r_0, r_1) = \gcd(r_{m-1}, r_m) = r_m
\end{array}
$$

$\dagger$ - termination criteria

Extended Euclidean Algorithm

Theorem 6.5.1 *Given two integers r_0 and r_1, there exist two other integers s and t such that $s \cdot r_0 + t \cdot r_1 = \gcd(r_0, r_1)$.*

Question: How to find s and t?

Use Euclid's algorithm and express the current remainder r_i in every iteration in the form $r_i = s_i r_0 + t_i r_1$. Note that in the last iteration $r_m = \gcd(r_0, r_1) \stackrel{!}{=} s_m r_0 + t_m r_1 = s r_0 + t r_1$.

index	Euclid's Algorithm	$r_j = s_j \cdot r_0 + t_j \cdot r_1$
2	$r_0 = q_1 \cdot r_1 + r_2$	$r_2 = r_0 - q_1 \cdot r_1 = s_2 \cdot r_0 + t_2 \cdot r_1$
3	$r_1 = q_2 \cdot r_2 + r_3$	$r_3 = r_1 - q_2 \cdot r_2 = r_1 - q_2(r_0 - q_1 \cdot r_1)$
		$= [-q_2]r_0 + [1 + q_1 \cdot q_2]r_1 = s_3 \cdot r_0 + t_3 \cdot r_1$
$\vdots$	$\vdots$	$\vdots$
i	$r_{i-2} = q_{i-1} \cdot r_{i-1} + r_i$	$r_i = s_i \cdot r_0 + t_i \cdot r_1$
$i+1$	$r_{i-1} = q_i \cdot r_i + r_{i+1}$	$r_{i+1} = s_{i+1} \cdot r_0 + t_{i+1} \cdot r_1$
$i+2$	$r_i = q_{i+1} \cdot r_{i+1} + r_{i+2}$	$r_{i+2} = r_i - q_{i+1} \cdot r_{i+1}$
		$= (s_i \cdot r_0 + t_1 \cdot r_1) - q_{i+1}(s_{i+1} \cdot r_0 + t_{i+1} \cdot r_1)$
		$= [s_i - q_{i+1}] \cdot s_{i+1}]r_0 + [t_1 - q_{i+1} \cdot t_{i+1}]r_1$
		$= s_{i+2} \cdot r_0 + t_{i+2} \cdot r_1$
$\vdots$	$\vdots$	$\vdots$
m	$r_{m-2} = q_{m-1} \cdot r_{m-1} + r_m$	$r_m = \gcd(r_0, r_1) = s_m \cdot r_0 + t_m \cdot r_1$

Now: $s = s_m$, $t = t_m$

Recursive formulae:

$s_0 = 1,$	$t_0 = 0$
$s_1 = 0,$	$t_1 = 1$
$s_i = s_{i-2} - q_{i-1} \cdot s_{i-1},$	$t_i = t_{i-2} - q_{i-1} \cdot t_{i-1};\ i = 2, 3, 4 \ldots$

Remark:

a) Extended Euclidean algorithm is commonly used to compute the inverse element in Z_m. If $\gcd(r_0, r_1) = 1$, then $t = r_1^{-1} \bmod r_0$.

b) For fast software implementation, the "binary extended Euclidean algorithm" is more efficient [AM97] because it avoids the division required in each iteration of the extended Euclidean algorithm shown above.

6.5.2 Euler's Phi Function

Definition 6.5.1 *The number of integers in Z_m relatively prime to m is denoted by $\Phi(m)$.*

Example 1:

$m = 6$; $Z_6 = \{0, 1, 2, 3, 4, 5\}$

$\gcd(0, 6) = 6$

$\gcd(1, 6) = 1 \ \leftarrow$

$\gcd(2, 6) = 2$

$\gcd(3, 6) = 3$

$\gcd(4, 6) = 2$

$\gcd(5, 6) = 1 \ \leftarrow$

$\Phi(6) = 2$

Example 2:

$m = 5$; $Z_5 = \{0, 1, 2, 3, 4\}$

$\gcd(0, 5) = 5$

$\gcd(1, 5) = 1 \ \leftarrow$

$\gcd(2, 5) = 1 \ \leftarrow$

$\gcd(3, 5) = 1 \ \leftarrow$

$\gcd(4, 5) = 1 \ \leftarrow$

$\Phi(5) = 4$

Theorem 6.5.2 *If* $m = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_n}$, *where* p_i *are prime numbers and* e_i *are integers, then:*

$$\Phi(m) = \prod_{i=1}^{n} (p_i^{e_i} - p_i^{e_i - 1})$$

.

Example: $m = 40 = 8 \cdot 5 = 2^3 \cdot 5 = p_1^{e_1} \cdot p_2^{e_2}$

$\Phi(m) = (2^3 - 2^2)(5^1 - 5^0) = (8 - 4)(5 - 1) = 4 \cdot 4 = 16$

Theorem 6.5.3 Euler's Theorem

If $\gcd(a, m) = 1$, *then:*

$$a^{\Phi(m)} \equiv 1 \bmod m$$

.

Example: $m = 6$; $a = 5$

$\Phi(6) = \Phi(3 \cdot 2) = (3 - 1)(2 - 1) = 2$

$5^{\Phi(6)} = 5^2 = 25 \equiv 1 \bmod 6$

6.6 Lessons Learned — Basics of Public-Key Cryptography

- Public-key algorithms have capabilities that symmetric ciphers don't have, in particular digital signature and key establishment functions.
- Public-key algorithms are computationally intensive (= slow), and are hence poorly suited for bulk data encryption.
- Most modern protocols are hybrid protocols which use symmetric as well as public-key algorithms.
- There are considerably fewer established public-key algorithms than there are symmetric ciphers.
- The extended Euclidean algorithm provides an efficient way of computing inverses modulo an integer.
- Computing Euler's phi function of an integer number is easy if one knows the factorization of the number. Otherwise it is very hard.

Chapter 7

RSA

A few general remarks:

1. Most popular public-key cryptosystem.
2. Invented by Rivest/Shamir/Adleman in 1977 at MIT.
3. Was patented in the USA (not in the rest of the world) until 2000.
4. The main application of RSA are:
   (a) encryption and, thus, for key transport
   (b) digital signature (see Chapter 11)

7.1 Cryptosystem

Set-up Stage

1. Choose two large primes p and q.
2. Compute $n = p \cdot q$.
3. Compute $\Phi(n) = (p-1)(q-1)$.
4. Choose random b; $0 < b < \Phi(n)$, with $\gcd(b, \Phi(n)) = 1$.
   Note that b has inverse in $Z_{\Phi(n)}$.
5. Compute inverse $a = b^{-1} \bmod \Phi(n)$:
   $$b \cdot a \equiv 1 \bmod \Phi(n).$$
6. Public key: $k_{pub} = (n, b)$.
   Private key: $k_{pr} = (p, q, a)$.

Encryption: done using public key, k_{pub}.

$y = e_{k_{pub}}(x) = x^b \bmod n.$
$x \in Z_n = \{0, 1, \ldots, n-1\}.$

Decryption: done using private key, k_{pr}.

$x = d_{k_{pr}}(y) = y^a \bmod n.$

Example:

Alice sends encrypted message ($x = 4$) to Bob after Bob sends her the public key.

Alice		Bob
		(1) choose $p = 3$; $q = 11$
		(2) $n = p \cdot q = 33$
		(3) $\Phi(n) = (3-1)(11-1) = 2 \cdot 10 = 20$
		(4) choose $b = 3$; $\gcd(20, 3) = 1$
$x = 4$	$\overset{k_{pub}(3,33)}{\longleftarrow}$	(5) $a = b^{-1} = 7 \bmod 20$
$y = x^b \bmod n = 4^3 = 64 \equiv 31 \bmod 33$	$\overset{y=31}{\longrightarrow}$	$x = y^a = 31^7 \equiv 4 \bmod 33$

Why does RSA work?

We have to show that: $d_{k_{pr}}(y) = d_{k_{pr}}(e_{k_{pub}}(x)) = x$.

$d_{k_{pr}} = y^a = x^{ba} = x^{ab} \bmod n$.

$a \cdot b \equiv 1 \bmod \Phi(n) \iff a \cdot b = 1 + t \cdot \Phi(n)$, where t is an integer

$d_{k_{pr}} = x^{ab} = x^{t \cdot \Phi(n)} \cdot x^1 = (x^{\Phi(n)})^t \cdot x \bmod n$.

1. Case: $\gcd(x, n) = \gcd(x, p \cdot q) = 1$

Euler's Theorem: $x^{\Phi(n)} \equiv 1 \bmod n$,

$d_{k_{pr}} = (x^{\Phi(n)})^t \cdot x \equiv 1^t \cdot x = x \bmod n$. *q.e.d.*

2. Case: $\gcd(x, n) = \gcd(x, p \cdot q) \neq 1$

either $x = r \cdot p$ or $x = s \cdot q$, where r, s are integers such that: $r < q$, $s < p$.

assume $x = r \cdot p \Rightarrow \gcd(x, q) = 1$

$(x^{\Phi(n)})^t = (x^{(q-1)(p-1)})^t = (x^{\Phi(q)(p-1)})^t = ((x^{\Phi(q)})^{p-1})^t \equiv 1^{(p-1)t} = 1 \bmod q$

$(x^{\Phi(n)})^t \equiv 1 + c \cdot q$, where c is an integer

$x \cdot (x^{\Phi(n)})^t \equiv x + x \cdot c \cdot q = x + r \cdot p \cdot c \cdot q = x + r \cdot c \cdot p \cdot q = x + r \cdot c \cdot n$

$x \cdot (x^{\Phi(n)})^t \equiv x \bmod n$

$d_{k_{pr}} = (x^{\Phi(n)})^t \cdot x \equiv x \bmod n$. *q.e.d.*

7.2 Computational Aspects

7.2.1 Choosing p and q

Problem: Finding two large primes p, q (for instance, each ≈ 512 bits).

Approach: Choose a random large integer and apply a *primality* test. In practice, a "Monte Carlo" test, for instance the Miller-Rabin [Sti02] test, is used. Note that a primality test does *not* require factorization, and is in fact enormously faster than factorization.

Input-output behavior of the **Miller-Rabin Algorithm:**

Input: p (or q) and an arbitrary number $r < p$.
Output 1: Statement "p is composite" $\rightarrow$ always true
Output 2: Statement "p is prime" $\rightarrow$ true with high probability

In practice, the above algorithm is run 3 times (for a 1000 bit prime) and upto 12 times (for a 150 bit prime) [AM97, Table 4.4 page 148] with different parameters r. If the answer is always "p is prime", then p is with very high probability a prime.

Question: What is the likelihood that a randomly picked integer p (or q) is prime?
Answer: $\mathcal{P}(p \text{ is prime }) \approx \frac{1}{ln(p)}$.

Example: $p \approx 2^{512} \rightarrow$ (512 bits).

$$\mathcal{P}(p \text{ is prime}) \approx \frac{1}{\ln(2^{512})} \approx \frac{1}{355}$$

This means that on average about 1 in 355 random integers with a length of 512 bit is a prime. Since we can spot even numbers right away, we only have to generate and test on average $355/2 \approx 173$ numbers before we find a prime of this bit length.

Conclusion: Primes are relatively frequent, even for numbers with large bit lengths. Together with an efficient primality test, this results in a very practical way of finding random prime numbers.

7.2.2 Choosing a and b

$k_{pub} = b$; condition: $\gcd(b, \Phi(n)) = 1$; where $\Phi(n) = (p-1) \cdot (q-1)$.

$k_{pr} = a$; where $a = b^{-1} \bmod \Phi(n)$.

Pick b (does not have be full length of n!) and compute:

1. Euclidean Algorithm: $s \cdot \Phi(n) + t \cdot b = \gcd(b, \Phi(n))$
2. Test if $\gcd(b, \Phi(n)) = 1$
3. Calculate a:

   Question: What is $t \cdot b \bmod \Phi(n)$?

$$\begin{aligned} t \cdot b &= (-s)\Phi(n) + 1 \\ \Rightarrow t \cdot b &\equiv 1 \bmod \Phi(n) \\ \Rightarrow t &= b^{-1} = a \bmod \Phi(n) \end{aligned}$$

Remark:

It is not necessary to find s for the computation of a.

7.2.3 Encryption/Decryption

encryption: $e_{k_{pub}}(x) = x^b \bmod n = y$.

decryption: $d_{k_{pr}}(y) = y^a \bmod n = x$.

Observation: Both encryption and decryption are exponentiations.

The goal now is to find an efficient way of performing exponentiations with very large numbers. Note that all parameters n, x, y, a, b are in general very large numbers[1]. Nowadays, in actual implementations, these parameters are typically chosen in the range of 1024–4096 bit! The straightforward way of exponentiation:

$$x, x^2, x^3, x^4, x^5, \ldots$$

does not work here, since the exponents a, b have in actual applications values in the range of 2^{1024}. Straightforward exponentiation would thus require around 2^{1024} multiplications. Since the number of atoms in the visible universe is estimated to be around 2^{150}, computing 2^{1024} multiplications for setting up one secure session for our web browser is not too tempting. The central question is whether there are considerably faster methods for exponentiation available. The answer is, luckily, yes (otherwise we could forget about RSA and pretty much all other public-key cryptosystem in use today.) In order to develop the method, let's look at some absurdly small example of an exponentiation:

Question: How many multiplications are required for computing x^8?

Answer: With the straightforward method $(x, x^2, x^3, x^4, x^5, x^6, x^7, x^8)$ we need 7 multiplications. However, alternatively we can do something much smarter:

$\underbrace{x \cdot x = x^2}_{\text{1. MUL}}$; $\underbrace{x^2 \cdot x^2 = x^4}_{\text{2. MUL}}$; $\underbrace{x^4 \cdot x^4 = x^8}_{\text{3. MUL}}$.

[1]The only exception is the public exponent b, which is often chosen to be a short number, e.g., $b = 17$.

Question: OK, that worked fine, but the exponent 8 is a very special case since it's a power of two ($2^3 = 8$) after all. Is there are fast way of computing an exponentiation with an arbitrary exponent? Let's look at the exponent 13. How many multiplications are required for computing x^{13}?

Answer: $\underbrace{x \cdot x = x^2}_{\text{SQ}}$; $\underbrace{x^2 \cdot x = x^3}_{\text{MUL}}$; $\underbrace{x^3 \cdot x^3 = x^6}_{\text{SQ}}$; $\underbrace{x^6 \cdot x^6 = x^{12}}_{\text{SQ}}$; $\underbrace{x^{12} \cdot x = x^{13}}_{\text{MUL}}$.

Observation: Apparently, we have to perform squarings (of the current result) and multiplying by x (of the current result) in order to achieve the over-all exponentiation.

Question: Is there a systematic way for finding the sequence in which we have to perform squarings and multiplications (by x) for a given exponent B?

Answer: Yes, the method is the square-and-multiply algorithm.

Square-and-multiply Algorithm

The square-and-multiply algorithm (also called *binary method* or *left-to-right exponentiation*) provides a chain of squarings and multiplications for a given exponent B so that an exponentiation x^B is being computed. The algorithms is based on scanning the bits of the exponents from left (the most significant bit) to the right (the least significant bit). Roughly speaking, the algorithm works as follows: in every iteration, i.e., for every exponent bit, the current results is squared. If (and only if) the currently scanned exponent bit has the value 1, a multiplication of the current result by x is also executed. Let's revisit the example from above but let's pay attention to the exponent bits:

Example: $x^{13} = x^{1101_2} = x^{(b_3,b_2,b_1,b_0)_2}$

#1	$(x^1)^2 = x^2 = x^{10_2}$	SQ, bit processed: b_2
#2	$x^2 \cdot x = x^3 = x^{11_2}$	MUL, since $b_2 = 1$
#3	$(x^3)^2 = x^6 = x^{110_2}$	SQ, bit processed: b_1
#4	$x^6 \cdot 1 = x^6 = x^{110_2}$	no MUL operation since $b_1 = 0$
#5	$(x^6)^2 = x^{12} = x^{1100_2}$	SQ, bit processed: b_0
#6	$x^{12} \cdot x = x^{13} = x^{1101_2}$	MUL, since $b0 = 1$

Why the algorithms works becomes clear if we look at a more general form of a 4-bit exponentiation:

Binary representation of the exponent $\rightarrow x^B$; $B \leq 15$

$B = b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0$

$B = (b_3 \cdot 2 + b_2)2^2 + b_1 \cdot 2 + b_0 = ((b_3 \cdot 2 + b_2)2 + b_1)2 + b_0$

$x^B = x^{((b_3 \cdot 2 + b_2)2 + b_1)2 + b_0}$

Step	x^B
#1	$x^{b_3 \cdot 2}$
#2	$(x^{b_3 \cdot 2} \cdot x^{b_2})$
#3	$(x^{b_3 \cdot 2} \cdot x^{b_2})^2$
#4	$(x^{b_3 \cdot 2} \cdot x^{b_2})^2 \cdot x^{b_1}$
#5	$((x^{b_3 \cdot 2} \cdot x^{b_2})^2 \cdot x^{b_1})^2$
#6	$((x^{b_3 \cdot 2} \cdot x^{b_2})^2 \cdot x^{b_1})^2 \cdot x^{b_0}$

Of course, the algorithm also works for general exponents with more than 4 bits. In the following, the square-and-multiply algorithms is given in pseudo code. Compare this pseudo code with the verbal description in the first paragraph after the headline *Square-and-multiply Algorithm*.

Algorithm [Sti02]: computes $z = x^B \bmod n$, where $B = \sum_{i=0}^{l-1} b_i 2^i$

1. $z = x$
2. `FOR` $i = l - 2$ `DOWNTO` 0:
   (a) $z = z^2 \bmod n$
   (b) `IF` $(b_i = 1)$
   $z = z \cdot x \bmod n$

Average complexity of the square-and-multiply algorithms for an exponent B:
$[\log_2 n] \cdot \mathrm{SQ} + [\frac{1}{2} \log_2 n] \cdot \mathrm{MUL}$.
Average complexity comparison for an exponent with about 1000 bits
Straightforward exponentiation: $2^{1000} \approx 10^{300}$ MUL $\Rightarrow$ impossible (before sun cools down)
Square-and-multiply: $1.5 \cdot 1000 = 1500$ MUL + SQ $\Rightarrow$ relatively easy

Remark 1: Remember to apply modulo reduction after every multiplication and squaring operation, in oder to keep the intermediate results small.

Remark 2: Bear in mind that each individual multiplication and squaring involves in practice a number with 1024 or more bits. Thus, a single multiplication (or squaring) consists typically of 100s of 32 integer multiplications on a desktop PC (and even more integer multiplications on an 8 bit smart card processor.)

Remark 3 (exponenent lengths in practice): The public exponent b is often chosen to be a short integer, for instance, the value $b = 17$ is popular. This makes encryption of a message (and verification of an RSA signature) a very fast operation. However, the private exponent a needs to have full length, i.e., the same length as the modulus n, for security reasons. Note that a short exponent b does not cause a to be short.

7.3 Attacks

There have been several attacks proposed against RSA implementations. They typically exploited *weaknesses in the way RSA was implemented* rather than breaking the actual RSA algorithms. The following is a list of attacks against the actual algorihtm that could, in theory, be exploited. However, the only known method for breaking the RSA algorithm is by factoring the modulus.

7.3.1 Brute Force

Given $y = x^b \bmod n$, try all possible keys a; $0 \leq a < \Phi(n)$ to obtain $x = y^a \bmod n$. In practice $|\mathcal{K}| = \Phi(n) \approx n > 2^{500} \Rightarrow$ impossible.

7.3.2 Finding $\Phi(n)$

Given $n, b, y = x^b \bmod n$, find $\Phi(n)$ and compute $a = b^{-1} \bmod \Phi(n)$.
$\Rightarrow$ computing $\Phi(n)$ is believed to be as difficult as factoring n.

7.3.3 Finding a directly

Given $n, b, y = x^b \bmod n$, find a directly and compute $x = y^a \bmod n$.
$\Rightarrow$ computing a directly is believed to be as difficult as factoring n.

7.3.4 Factorization of n

Factoring attack: Given $n, b, y = x^b \bmod n$, factor $p \cdot q = n$ and compute:
$\Phi(n) = (p-1)(q-1)$
$b = a^{-1} \bmod \Phi(n)$
$x = y^a \bmod n$

Factoring Algorithms:

1. Quadratic Sieve (QS): speed depends on the size of n; record: in 1994 factoring of n =RSA129, $\log_{10} n = 129$ digits, $\log_2 n = 426$ bits.

2. Elliptic Curve: similar to QS; speed depends on the size of the smallest prime factor of n, i.e., on p and q.

3. Number Field Sieve: asymptotically better than QS; record: in 1999 factoring of n =RSA155; $\log_{10} n = 155$ digits; $\log_2 n = 512$ bits.

Complexities of factoring algorithms:

Algorithm	*Complexity*
Quadratic Sieve	$\mathcal{O}(e^{(1+o(1))\sqrt{\ln(n)\ln(\ln(n))}})$
Elliptic Curve	$\mathcal{O}(e^{(1+o(1))\sqrt{2\ln(p)\ln(\ln(p))}})$
Number Field Sieve	$\mathcal{O}(e^{(1.92+o(1))(\ln(n))^{1/3}(\ln(\ln(n)))^{2/3}})$

number	**month**	**MIPS-years**	**algorithm**
RSA-100	April 1991	7	quadratic sieve
RSA-110	April 1992	75	quadratic sieve
RSA-120	June 1993	830	quadratic sieve
RSA-129	April 1994	5000	quadratic sieve
RSA-130	April 1996	500	generalized number field sieve
RSA-140	February 1999	1500	generalized number field sieve
RSA-155	August 1999	8000	generalized number field sieve

Table 7.1: RSA factoring challenges

7.4 Implementation

Some representative performance numbers:

- Hardware (FPGA): 1024 bit decryption in less that 5 ms.
- Software (Pentium at a few 100MHz): 1024 bit decryption in 43 ms; 1024 bit encryption with short public exponent in 0.65 ms.

In practice, hybrid systems consisting of public-key and symmetric-key algorithms are commonly used:

1. key exchange and digital signatures are performed with (slow) public-key algorithm
2. bulk data encryption is performed with (fast) block ciphers or stream ciphers

7.5 Lessons Learned — RSA

- RSA is the most widely used public-key cryptosystems. In the future, elliptic curves cryptosystems will probably catch up in popularity.
- RSA is mainly used for key transport (i.e., encryption of keys) and digital signatures.
- The public key b can be a short integer. The private key a needs to have the full length of the modulus.
- Decryption with the long private key is computationally demanding and can be a bottleneck on small processors, e.g., in mobile applications.
- Encryption with a short public key is very fast.
- RSA relies on the integer factorization problem:
  1. Currently, 1024 bit (about 310 decimal digits) numbers cannot be factored.
  2. Progress in factorization algorithms and factorization hardware is hard to predict. It is advisable to use RSA with 2048 bit modulus if one needs reasonable long term security or is concerned about extremely well funded attackers.

Chapter 8

The Discrete Logarithm (DL) Problem

- DL is the underlying one-way function for:
  1. Diffie-Hellman key exchange.
  2. DSA (digital signature algorithm).
  3. ElGamal encryption/digital signature scheme.
  4. Elliptic curve cryptosystems.
  5. .....
- DL is based on cyclic groups.

8.1 Some Algebra

Further Reading: [Big85].

8.1.1 Groups

Definition 8.1.1 *A* group *is a set $\mathcal{G}$ of elements together with a binary operation "o" such that:*

1. *If $a, b \in \mathcal{G}$ then $a \circ b = c \in \mathcal{G} \rightarrow$ (closure).*
2. *If $(a \circ b) \circ c = a \circ (b \circ c) \rightarrow$ (associativity).*
3. *There exists an identity element $e \in \mathcal{G}$:*
   $e \circ a = a \circ e = a \rightarrow$ (identity).
4. *There exists an inverse element $\tilde{a}$, for all $a \in \mathcal{G}$:*
   $a \circ \tilde{a} = e \rightarrow$ (inverse).

Examples:

1. $\mathcal{G}= Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
   $\circ$ = addition
   $(Z, +)$ is a group with $e = 0$ and $\tilde{a} = -a$
2. $\mathcal{G}= Z$
   $\circ$ = multiplication
   $(Z, \times)$ is NOT a group since inverses $\tilde{a}$ do not exist except for $a = 1$
3. $\mathcal{G}=\mathcal{C}$ (complex numbers $u + iv$)
   $\circ$ = multiplication
   $(\mathcal{C}, \times)$ is a group with $e = 1$ and

$$\tilde{a} = a^{-1} = \frac{u - iv}{u^2 + v^2}$$

Definition 8.1.2 *"Z_n^*" denotes the set of numbers i, $0 \leq i < n$, which are relatively prime to n.*

Examples:

1. $Z_9^* = \{1, 2, 4, 5, 7, 8\}$

2. $Z_7^* = \{1, 2, 3, 4, 5, 6\}$

Multiplication Table

$*$ mod 9	1	2	4	5	7	8
1	1	2	4	5	7	8
2	2	4	8	1	5	7
4	4	8	7	2	1	5
5	5	1	2	7	8	4
7	7	5	1	8	4	2
8	8	7	5	4	2	1

Theorem 8.1.1 *Z_n^* forms a group under modulo n multiplication. The identity element is $e = 1$.*

Remark:

The inverse of $a \in Z_n^*$ can be found through the extended Euclidean algorithm.

8.1.2 Finite Groups

Definition 8.1.3 *A group $(\mathcal{G}, \circ)$ is **finite** if it has a finite number of g elements. We denote the cardinality of $\mathcal{G}$ by $|\mathcal{G}|$.*

Examples:

1. $(Z_m, +)$: $a + b = c \bmod m$

   Question: What is the cardinality $\rightarrow |Z_m| = m$

   $Z_m = \{0, 1, 2, \ldots, m-1\}$

2. $(Z_p^*, \times)$: $a \times b = c \bmod p$; p is prime

   Question: What is the cardinality $\rightarrow |Z_p^*| = p - 1$

   $Z_p^* = \{1, 2, \ldots, p-1\}$

Definition 8.1.4 *The **order** of an element $a \in (\mathcal{G}, \circ)$ is the smallest positive integer o such that $a \circ a \circ \ldots \circ a = a^o = 1$.*

Example: $(Z_{11}^*, \times)$, $a = 3$

Question: What is the order of $a = 3$?

$a^1 = 3$

$a^2 = 3^2 = 9$

$a^3 = 3^3 = 27 \equiv 5 \bmod 11$

$a^4 = 3^4 = 3^3 \cdot 3 = 5 \cdot 3 = 15 \equiv 4 \bmod 11$

$a^5 = a^4 \cdot a = 4 \cdot 3 = 12 \equiv 1 \bmod 11$

$\Rightarrow \text{ord}(3) = 5$

Definition 8.1.5 *A group $\mathcal{G}$ which contains elements α with maximum order $ord(\alpha) = |\mathcal{G}|$ is said to be* ***cyclic****. Elements with maximum order are called* ***generators*** *or* ***primitive elements****.*

Example: 2 is a primitive element in Z_{11}^*

$|Z_{11}^*| = |\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}| = 10$

$a = 2$

$a^2 = 4$

$a^3 = 8$

$a^4 = 16 \equiv 5$

$a^5 = 10;$

$a^6 = 20 \equiv 9$

$a^7 = 18 \equiv 7$

$a^8 = 14 \equiv 3;$

$a^9 = 6$

$a^{10} = 12 \equiv 1$

$\underline{a^{11} = 2 = a}.$

$\Rightarrow \text{ord}(a = 2) = 10 = |Z_{11}^*|$

$\Rightarrow$ (1) $|Z_{11}^*|$ is cyclic

$\Rightarrow$ (2) $a = 2$ is a primitive element

Observation (important): 2^i; $i = 1, 2, \ldots, 10$ **generates** all elements of Z_{11}^*

i	1	2	3	4	5	6	7	8	9	10
2^i	2	4	8	5	10	9	7	3	6	1

Some properties of cyclic groups:

1. The number of primitive elements is $\Phi(|\mathcal{G}|)$.
2. For every $a \in \mathcal{G}$: $a^{|\mathcal{G}|} = 1$.
3. For every $a \in \mathcal{G}$: ord(a) divides $|\mathcal{G}|$.

Proof only for (2): $a = \alpha^i$

$a^{|\mathcal{G}|} = (\alpha^i)^{|\mathcal{G}|} = (\alpha^{|\mathcal{G}|})^i \doteq 1^i = 1.$

Example: Z_{11}^*; $|Z_{11}^*| = 10$

1. $\Phi(10) = (2-1)(5-1) = 1 \cdot 4 = 4$
2. $a = 3 \rightarrow 3^{10} = (3^5)^2 = 1^2 = 1$
3. homework ...

8.1.3 Subgroups

Definition 8.1.6 *A subset $\mathcal{H}$ of a group $\mathcal{G}$ is called a **subgroup of** $\mathcal{G}$ if the elements of $\mathcal{H}$ form a group under the group operation of $\mathcal{G}$.*

Example: $\mathcal{G} = Z_{11}^\star$:

$3^1 = 3$

$3^2 = 9$

$3^3 \equiv 5$

$3^4 \equiv 4$

$3^5 \equiv 1$

$\Rightarrow 3$ is a generator of $\mathcal{H} = \{1, 3, 4, 5, 9\}$ which is a subgroup of $\mathcal{G}$

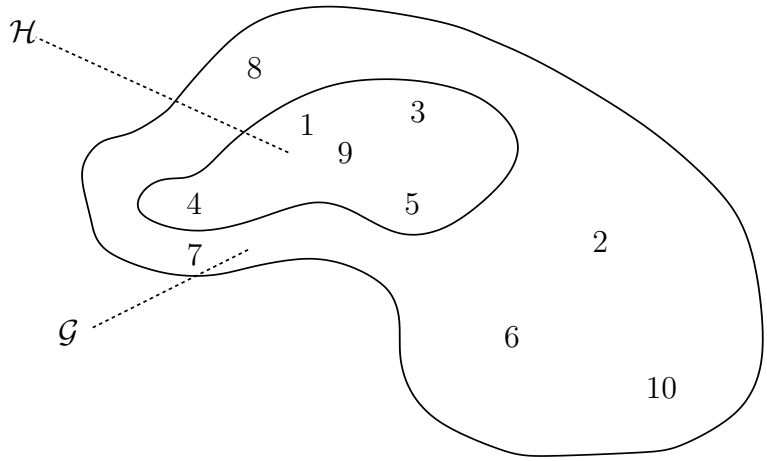


Figure 8.1: Subgroup $\mathcal{H}$ of $\mathcal{G} = Z_{11}^{\star}$

Multiplication Table

$\mathcal{H}$	1	3	4	5	9
1	1	3	4	5	9
3	3	9	1	4	5
4	4	1	5	9	3
5	5	4	9	3	1
9	9	5	3	1	4

Observation: Multiplication of elements in $\mathcal{H}$ is closed!

Theorem 8.1.2 *Any subgroup of a cyclic group is cyclic.*

Example: $\mathcal{H} = \{1, 3, 4, 5, 9\}$ is cyclic (see above).

Theorem 8.1.3 *An element α of a cyclic group with*

$$\text{ord}(\alpha) = t$$

generates a cyclic subgroup with t elements.

Example (1): $\text{ord}(3) = 5$ in $Z_{11}^{\star}$

$\Rightarrow$ 3 generates $\mathcal{H}$ with 5 elements.

Remarks:

- Since the possible element order is t, t must divide $|\mathcal{G}|$.
- The possible subgroup orders also divide $|\mathcal{G}|$.

Example (2): $|Z_{11}^{\star}| = 10$

$\Rightarrow$ possible subgroup orders: 1, 2, 3 (,10).

$\{1\}$	$\alpha = 1$
$\{1, 10\}$	$\alpha = 10$
$\{1, 3, 4, 5, 9\}$	$\alpha = 3, 4, 5, 9$

8.2 The Generalized DL Problem

Given a cyclic subgroup $(\mathcal{G}, \circ)$ and a primitive element α. Let

$$\beta = \underbrace{\alpha \circ \alpha \ldots \alpha}_{i \quad times} = \alpha^i$$

be an arbitrary element in $\mathcal{G}$.

General DL Problem:

Given $\mathcal{G}$, $\alpha, \beta = \alpha^i$, find i.
$i = \log_\alpha(\beta)$

Examples:

1. $(Z_{11}, +)$; $\alpha = 2$; $\beta = \underbrace{2 + 2 + \ldots + 2}_{i \quad times} = i \cdot 2$

i	1	2	3	4	5	6	7	8	9	10	11
2i	2	4	6	8	10	1	3	5	7	9	0

   Let $i = 7$: $\beta = 7 \cdot 2 \equiv 3 \bmod 11$

   Question: given $\alpha = 2$, $\beta = 3 = i \cdot 2$, find i

   Answer: $i = 2^{-1} \cdot 3 \bmod 11$

   Euclid's algorithm can be used to compute i thus this example is NOT a one-way function.

2. $(Z_{11}^*, \times)$; $\alpha = 2$; $\beta = \underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{i \quad times} = 2^i$

   $\beta = 3 = 2^i \bmod 11$

   Question: $i = \log_2(3) = \log_2(2^i) = ?$

   Very hard computational problem!

8.3 Attacks for the DL Problem

1. **Brute force**:

   check:

   $\alpha^1 \stackrel{?}{=} \beta$

   $\alpha^2 \stackrel{?}{=} \beta$

   $\vdots$

   $\alpha^i \stackrel{?}{=} \beta$

   Complexity: $\mathcal{O}(|\mathcal{G}|)$ steps.

   Example: DL in Z_p^* $\approx \frac{p-1}{2}$ tests

   minimum security requirement $\Rightarrow p - 1 = |\mathcal{G}| \geq 2^{80}$

2. **Shank's algorithm (Baby-step giant-step) and Pollard's-ρ method**:

   Further reading: [Sti02]

   Complexity: $\mathcal{O}(\sqrt{|\mathcal{G}|})$ steps (for both algorithms).

   Example: DL in Z_p^* $\approx \sqrt{p}$ steps

   minimum security requirement $\Rightarrow p - 1 = |\mathcal{G}| \geq 2^{160}$

3. **Pohlig-Hellman algorithm**:

   Let $|\mathcal{G}| = p_1 \cdot p_2 \cdots \underbrace{p_l}_{\substack{largest \\ prime}}$

   Complexity: $\mathcal{O}(\sqrt{p_l})$ steps.

   Example: DL in Z_p^*: p_l of $(p-1)$ must be $\geq 2^{160}$

   minimum security requirement $\Rightarrow p_l \geq 2^{160}$

4. **Index-Calculus method**:

   Further reading: [AM97].

   Applies only to Z_p^* and Galois fields GF(2^k)

   Complexity: $\mathcal{O}\ (e^{(1+\mathcal{O}(1))\sqrt{\ln(p)\ln(\ln(p))}})$ steps.

   Example: DL in Z_p^*: minimum security requirement $\Rightarrow p \geq 2^{1024}$

   Remark: Index-Calculus is more powerful against DL in Galois Fields GF(2^k) than

against DL in Z_p^*.

8.4 Diffie-Hellman Key Exchange

Remarks:

- Proposed in 1976 in by Diffie and Hellman [DH76].
- Used in many practical protocols.
- Can be based on any DL problem.

8.4.1 Protocol

Set-up:

1. Find a large prime p.
2. Find a primitive element α of Z_p^* or of a subgroup of Z_p^*.

Protocol:

Alice		Bob
pick $k_{prA} = a_A \in \{2, 3, \ldots, p-1\}$		pick $k_{prB} = a_B \in \{2, 3, \ldots, p-1\}$
compute $k_{pubA} = b_A = \alpha^{a_A} \bmod p$		compute $k_{pubB} = b_B = \alpha^{a_B} \bmod p$
	$\xrightarrow{b_A}$	
	$\xleftarrow{b_B}$	
$k_{AB} = b_B^{a_A} = (\alpha^{a_B})^{a_A}$		$k_{AB} = b_A^{a_B} = (\alpha^{a_A})^{a_B}$

Session key $k_{ses} = k_{AB} = \alpha^{a_B \cdot a_A} = \alpha^{a_A \cdot a_B} \bmod p$.

8.4.2 Security

Question: Which information does Oscar have?

Answer: α, p, b_A, b_B.

Diffie-Hellman Problem:

Given $b_A = \alpha^{a_A} \bmod p, b_B = \alpha^{a_B} \bmod p$, and α find $\alpha^{a_A \cdot a_B} \bmod p$.

One solution to the D-H problem:

1. Solve DL problem: $a_A = \log_\alpha(b_A) \bmod p$.
2. Compute: $b_B^{a_A} = (\alpha^{a_B})^{a_A} = \alpha^{a_A \cdot a_B} \bmod p$.
   Choose $p \geq 2^{1024}$.

Note:

There is no proof that the DL problem is the only solution to the D-H problem! However, it is conjectured.

8.5 Lessons Learned — Diffie-Hellman Key Exchange

- The Diffie-Hellman protocol is a widely used method for key exchange. It is based on cyclic groups.
- In practice, the multiplicative group of the prime field Z_p or the group of an elliptic curve are most often used.
- If the parameters are chosen carefully, the Diffie-Hellman protocol is secure against passive (i.e., attacker can only eavesdrop) attacks.
- For the Diffie-Helmann protocol in $Z_p^\star$, the prime p should be at least 1024 bit long. This provides a security roughly equivalent to an 80 bit symmetric cipher. For a better long term security, a prime of length 2048 bit should be chosen.

Chapter 9

Elliptic Curve Cryptosystem

Further Reading:

Chapter 6 in [Kob94].

Book by Alfred Menezes [Men93].

Remarks:

- Relatively new cryptosystem, suggested independently:
  → 1987 by Koblitz at the University of Washington,
  → 1986 by Miller at IBM.
- It is believed to be more secure than RSA/DL in Z_p^*, but uses arithmetic with much shorter numbers ($\approx$ 160 – 256 bits vs. 1024 – 2048 bits).
- It can be used instead of D-H and other DL-based algorithms.

Drawbacks:

- Not as well studied as RSA and DL-base public-key schemes.
- It is conceptually more difficult.
- Finding secure curves in the set-up phase is computationally expensive.

9.1 Elliptic Curves

Goal: To find another instance for the DL problem in cyclic groups.

Question: What is the equation $x^2 + y^2 = r^2$ over reals?

Answer: It is a circle.

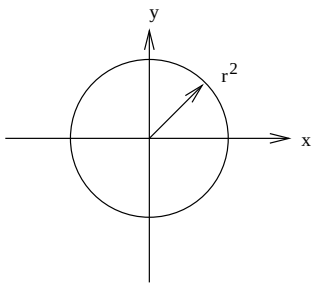


Figure 9.1: $x^2 + y^2 = r^2$ over reals

Question: What is the equation $a \cdot x^2 + b \cdot y^2 = c$ over reals?

Answer: It is an ellipsis.

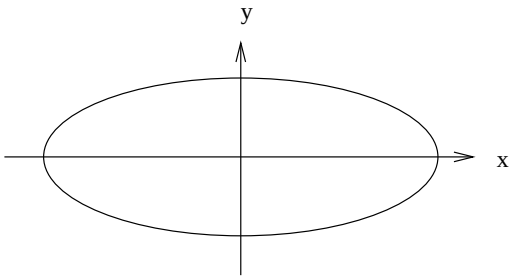


Figure 9.2: $a \cdot x^2 + b \cdot y^2 = c$ over reals

Note:

There are only certain points (x,y) which fulfill the equation. For example the point $(x = r, y = 1)$ fulfills the equation of a circle.

Definition 9.1.1 *The* elliptic curve *over Z_p, $p > 3$, is a set of all pairs $(x, y) \in Z_p$ which fulfill:*

$$y^2 \equiv x^3 + a \cdot x + b \bmod p$$

where

$$a, b, \in Z_p$$

and

$$4 \cdot a^3 + 27 \cdot b^2 \neq 0 \bmod p$$

Question: How does $y^2 = x^3 + a \cdot x + b$ look over reals?

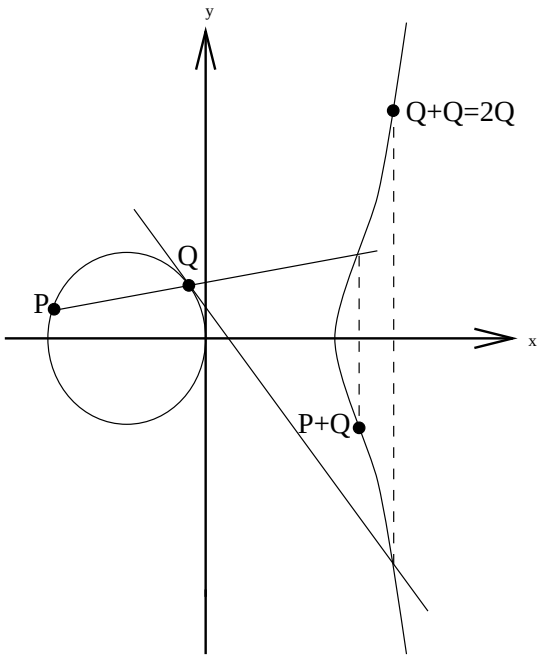


Figure 9.3: $y^2 = x^3 + a \cdot x + b$ over the reals

Goal: Finding a (cyclic) group $(\mathcal{G}, \circ)$ so that we can use the DL problem as a one-way function.

We have a set (points on the curve). We "only" need a group operation on the points.

Group $\mathcal{G}$: Points on the curve given by (x, y).

Operation $\circ$: $P + Q = (x_1, y_1) + (x_2, y_2) = R = (x_3, y_3)$.

Question: How do we find R?

Answer: First geometrically.

a) $P \neq Q \rightarrow$ line through P and Q and mirror point of third interception along the x-axis.

b) $P = Q \Rightarrow P + Q = 2Q \rightarrow$ tangent line through Q and mirror point of second intersection along the x-axis.

Point Addition (group operation):

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \bmod p \\ y_3 &= \lambda(x_1 - x_3) - y_1 \bmod p \end{aligned}$$

where

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \bmod p & ; \text{if } P \neq Q \\ \frac{3x_1^2 + a}{2y_1} \bmod p & ; \text{if } P = Q \end{cases}$$

Remarks:

- If $x_1 \equiv x_2 \bmod p$ and $y_1 \equiv -y_2 \bmod p$, then $P + Q = \mathcal{O}$ which is an abstract point at infinity.
- $\mathcal{O}$ is the neutral element of the group: $P + \mathcal{O} = P$; for all P.
- Additive inverse of any point $(x, y) = P$ is $P + (-P) = \mathcal{O}$ such that $(x, y) + (x, -y) = \mathcal{O}$.

Theorem 9.1.1 *The points on an elliptic curve together with $\mathcal{O}$ have cyclic subgroups.*

Remark: Under certain conditions all points on an elliptic curve form a cyclic group as the following example shows.

Example: Finding all points on the curve E: $y^2 \equiv x^3 + x + 6 \bmod 11$.

$\#E = 13$.

primitive element $\rightarrow \alpha = (2, 7) \Rightarrow$ generates all points.

$2\alpha = \alpha + \alpha = (2, 7) + (2, 7) = (x_3, y_3)$

$\lambda = \frac{3x_1^2+a}{2y_1} = (2 \cdot 7)^{-1}(3 \cdot 4 + 1) = 3^{-1} \cdot 13 \equiv 4 \cdot 13 \equiv 4 \cdot 2 = 8 \bmod 11$

$x_3 = \lambda^2 - x_1 - x_2 = 8^2 - 2 - 2 = 60 \equiv 5 \bmod 11$

$y_3 = \lambda(x_1 - x_3) - y_1 = 8(2 - 5) - 7 = -24 - 7 = -31 \equiv 2 \bmod 11$

$2\alpha = (2, 7) + (2, 7) = (5, 2)$

$3\alpha = 2\alpha + \alpha = \ldots$

$\vdots$

$12\alpha = 11\alpha + \alpha = (2, 4)$

$13\alpha = 12\alpha + \alpha = (2, 4) + (2, 7) = (2, 4) + (2, -4) = \mathcal{O}$

$14\alpha = 13\alpha + \alpha = \mathcal{O} + \alpha = \alpha$

$\vdots$

All 12 non-zero elements together with $\mathcal{O}$ form a cyclic group.

$\alpha = (2, 7)$	$2\alpha = (5, 2)$	$3\alpha = (8, 3)$
$4\alpha = (10, 2)$	$5\alpha = (3, 6)$	$6\alpha = (7, 9)$
$7\alpha = (7, 2)$	$8\alpha = (3, 5)$	$9\alpha = (10, 9)$
$10\alpha = (8, 8)$	$11\alpha = (5, 9)$	$12\alpha = (2, 4)$

Table 9.1: Non-zero elements of the group over $y^2 \equiv x^3 + x + 6 \bmod 11$

Remark: In general, finding of the group order $\#E$ is computationally very complex.

9.2 Cryptosystems

9.2.1 Diffie-Hellman Key Exchange

The cryptosystem is completely analogous to D-H in Z_p^*.

Set-up:

1. Choose E: $y^2 \equiv x^3 + a \cdot x + b \bmod p$.
2. Choose primitive element $\alpha = (x_\alpha, y_\alpha)$.

Protocol:

Alice		Bob
choose $k_{prA} = a_A \in \{2, 3, \ldots, \#E - 1\}$		choose $k_{prB} = a_B \in \{2, 3, \ldots, \#E - 1\}$
compute $k_{pubA} = b_A = a_A \cdot \alpha = (x_A, y_A)$		compute $k_{pubB} = b_B = a_B \cdot \alpha = (x_B, y_B)$
	$\xrightarrow{b_A}$	
	$\xleftarrow{b_B}$	
compute $a_A \cdot b_B = a_A \cdot a_B \cdot \alpha = (x_k, y_k)$		compute $a_B \cdot b_A = a_B \cdot a_A \cdot \alpha = (x_k, y_k)$
$k_{AB} = x_k \in Z_p$		$k_{AB} = x_k \in Z_p$

Security:

Diffie-Hellman problem for elliptic curves $\begin{cases} \text{Oscar knows:} & E, p, \alpha, b_A = a_A \cdot \alpha, b_B = a_B \cdot \alpha \\ \text{Oscar wants to know:} & k_{AB} = a_A \cdot a_B \cdot \alpha \end{cases}$

One possible solution to the D-H problem for elliptic curves:

1. Compute *discrete logarithm*:
   Given α and $\underbrace{\alpha + \alpha + \ldots + \alpha}_{a_A \quad times} = b_A$, find a_A.
2. Compute $a_A \cdot b_B = a_A \cdot a_B \cdot \alpha$.

Attacks:

- Only possible attacks against elliptic curves are the Pohlig-Hellman scheme together with Shank's algorithm or Pollard's-Rho method.
  $\Rightarrow \#E$ must have one large prime factor p_l
  $\Rightarrow 2^{160} \leq p_l \leq 2^{250}$.
- So-called "Koblitz curves" (curves with $a, b \in \{0, 1\}$)
- For *supersingular* elliptic curves over GF(2^n), DL in elliptic curves can be solved by solving DL in GF($2^{k \cdot n}$); $k \leq 6$.
  $\Rightarrow$ stay away from supersingular curves despite of possible faster implementations.
- Powerful index-calculus method attacks are not applicable (as of yet).

9.2.2 Menezes-Vanstone Encryption

Set-up:

1. Choose E: $y^2 \equiv x^3 + a \cdot x + b \bmod p$.
2. Choose primitive element $\alpha = (x_\alpha, y_\alpha)$.
3. Pick random integer $a \in \{2, 3, \ldots, \#E - 1\}$.
4. Compute $a \cdot \alpha = \beta = (x_\beta, y_\beta)$.
5. Public Key: $k_{pub} = (E, p, \alpha, \beta)$.
6. Private Key: $k_{pr} = (a)$.

Encryption:

1. Pick random $k \in \{2, 3, \ldots, \#E - 1\}$. Compute $k \cdot \beta = (c_1, c_2)$.

2. Encrypt $e_{k_{pub}}(x, k) = (Y_0, Y_1, Y_2)$.
   $Y_0 = k \cdot \alpha \rightarrow$ point on the elliptic curve.
   $Y_1 = c_1 \cdot x_1 \bmod p \rightarrow$ integer.
   $Y_2 = c_2 \cdot x_2 \bmod p \rightarrow$ integer.

Decryption:

1. Compute $a \cdot Y_0 = (c_1, c_2)$.
   $a \cdot Y_0 = a \cdot k \cdot \alpha = k \cdot \beta = (c_1, c_2)$.

2. Decrypt: $d_{k_{pr}}(Y_0, Y_1, Y_2) = (Y_1 \cdot c_1^{-1} \bmod p, Y_2 \cdot c_2^{-1} \bmod p) = (x_1, x_2)$.

Remark: The disadvantage of this scheme is the message expansion factor:

$$\frac{\#\ bits\ \ y}{\#\ bits\ \ x} = \frac{4\lceil \log_2 p \rceil}{2\lceil \log_2 p \rceil} = 2$$

9.3 Implementation

1. Hardware:

   - Approximatly 0.2 msec for an elliptic curve point multiplication with 167 bits on an FPGA [OP00].

2. Software:

   - One elliptic curve point multiplication $a \cdot P$ in less than 10 msec over GF(2^{155}).
   - Implementation on 8-bit smart card processor without coprocessor available

Chapter 10

ElGamal Encryption Scheme

10.1 Cryptosystem

Remarks:

- Published in 1985.
- Based on the DL problem in Z_p^* or GF(2^k).
- Extension of the D-H key exchange for encryption.

Principle:

Alice		Bob
choose private key $k_{prA} = a_A$		choose private key $k_{prB} = a_B$
compute $k_{pubA} = \alpha^{a_A} \bmod p = b_A$		compute $k_{pubB} = \alpha^{a_B} \bmod p = b_B$
	$\xrightarrow{b_A}$	
	$\xleftarrow{b_B}$	
$k_{AB} = b_B^{a_A} = \alpha^{a_A a_B} \bmod p$		$k_{AB} = b_A^{a_B} = \alpha^{a_B a_A} \bmod p$
$y = x \cdot k_{AB} \bmod p$	$\xrightarrow{y}$	
		$x = y \cdot k_{AB}^{-1} \bmod p$

ElGamal:

Set-up:

1. Choose large prime p.
2. Choose primitive element $\alpha \in Z_p^*$.
3. Choose secret key $a \in \{2, 3, \ldots, p-2\}$.
4. Compute $\beta = \alpha^a \bmod p$,
   Public Key: $K_{pub} = (p, \alpha, \beta)$,
   Private Key: $K_{pr} = (a)$.

Encryption:

1. Choose $k \in \{2, 3, \ldots, p-2\}$.
2. $Y_1 = \alpha^k \bmod p$.
3. $Y_2 = x \cdot \beta^k \bmod p$.
4. Encryption: $= e_{k_{pub}}(x, k) = (Y_1, Y_2)$.

Decryption:

$x = d_{k_{pr}}(Y_1, Y_2) = Y_2(Y_1^a)^{-1} \bmod p$.

Question: How does the ElGamal scheme work?

$$\begin{aligned} d_{k_{pr}}(Y_1, Y_2) &= Y_2(Y_1^a)^{-1} \\ &= x \cdot \beta^k((\alpha^k)^a)^{-1} \rightarrow \text{ but } \beta = \alpha^a \\ &= x(\alpha^a)^k((\alpha^k)^a)^{-1} \\ &= x \cdot \alpha^{ak} \cdot \alpha^{-ak} \\ &= x \end{aligned}$$

Protocol:

Alice		Bob
message $x < p$		set-up phase steps 1-4
		$k_{pub} = (p, \alpha, \beta)$
		$k_{pr} = (a)$
	$\xleftarrow{k_{pub}=(p,\alpha,\beta)}$	
choose $k \in \{2, 3, \cdots, p-2\}$		
$Y_1 = \alpha^k \bmod p$		
$Y_2 = x \cdot \beta^k \bmod p$		
	$\xrightarrow{(Y_1,Y_2)}$	
		$x = Y_2(Y_1^a)^{-1}$

Remarks:

- ElGamal is essentially an extension of the D-H key exchange protocol (α^k corresponds to Alice's public key b_A and β^k corresponds to the derived session key K_{AB}).
- $\left.\begin{array}{l} Y_2 = x_1 \cdot \beta^k \\ Y_3 = x_2 \cdot \beta^k \end{array}\right\}$ if x_1 is known, β^k can be found from Y_2.
  Thus for every message block x_i choose a new k!
- Message expansion factor
$$\frac{\#\ of\ \ y\ \ bits}{\#\ of\ \ x\ \ bits} = \frac{2\lceil \log 2p_y \rceil}{\lceil \log 2p_x \rceil} = 2.$$
- ElGamal is non-deterministic.

10.2 Computational Aspects

10.2.1 Encryption

$\left.\begin{array}{l} Y_1 = \alpha^k \bmod\ p \\ Y_2 = x \cdot \beta^k \bmod\ p \end{array}\right\}$ apply the square-and-multiply for exponentiation

10.2.2 Decryption

$x = d_{k_{pr}}(Y_1, Y_2) = Y_2 (Y_1^a)^{-1} \bmod\ p.$

Question: How can $(Y_1^a)^{-1}$ be computed efficiently?

Derivation: $b \in Z_p^*$:

$$\begin{aligned} b^e = b^{q(p-1)+r} &= (b^{p-1})^q \cdot b^r \\ &= 1^q \cdot b^r \bmod\ p \\ &= b^r \bmod\ p \\ \Rightarrow e &= r \bmod\ (p-1) \end{aligned}$$

Thus, $b^e \equiv b^{e \bmod (p-1)} \bmod p$, where $b \in Z_p^*$ and $e \in Z$

The above derivation can be used for decryption:

$$\begin{aligned}(Y_1^a)^{-1} = Y_1^{-a} &= Y_1^{-a \bmod (p-1)} \bmod p \\ &= Y_1^{p-1-a} \bmod p\end{aligned}$$

Note: $Y_1^{p-1-a} \bmod p$ can be computed using the square-and-multiply algorithm.

10.3 Security of ElGamal

Oscar knows: $p, \alpha, \beta = \alpha^a, Y_1 = \alpha^k, Y_2 = x \cdot \beta^k$.

Oscar wants to know: x

- He attempts to find the secret key a:
  1. $a = \log_\alpha \beta \bmod p \leftarrow$ hard, DL problem.
  2. $x = Y_2 (Y_1^a)^{-1} \bmod p \leftarrow$ easy.
- He attempts to find the random exponent k:
  1. $k = \log_\alpha Y_1 \bmod p \leftarrow$ hard, DL problem.
  2. $Y_2 \cdot \beta^{-k} = x \leftarrow$ easy.
- In both cases Oscar has to compute the DL problem in finite fields (Z_p^* or GF(2^k)). He can use index-calculus method which forces us to implement schemes with at least 1024 bits.

Chapter 11

Digital Signatures

Protocols use:

- Symmetric-key algorithms.
- Public-key algorithms.
- Digital Signatures.
- Hash functions.
- Message Authentication Codes.

as building blocks. In practice, protocols are often the most vulnerable part of a cryptosystem. The following chapters deal with digital signature, message authentication codes (MACs), and hash functions.

11.1 Principle

The idea is similar to a conventional signature on paper: Given a message x, a digital signature ist appended to the message. As with conventional signatures, only the person who sends the message must be capable of generating a valid signature. In order to achieve this with cryptography, we make the signature a function of a private key, so that only the holder of the private key can sign a message. In order to make sure that a signature changes with each document, we also make the signature a function of the message that is being signed.

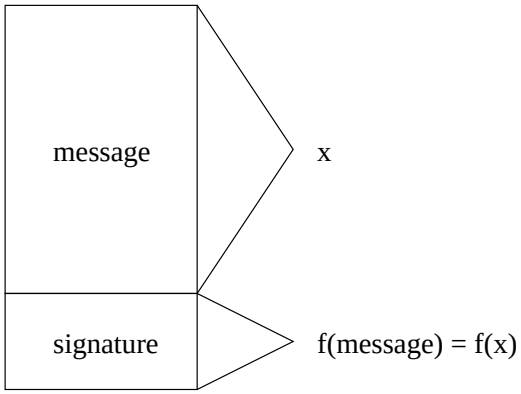


Figure 11.1: Digital signature and message block

The main advantage which digital signatures have is that they enable communication parties to **prove** that one party has actually generated the message. Such a "proof" can even have legal meaning, for instance as in the German Signaturgesetz (signature law.)

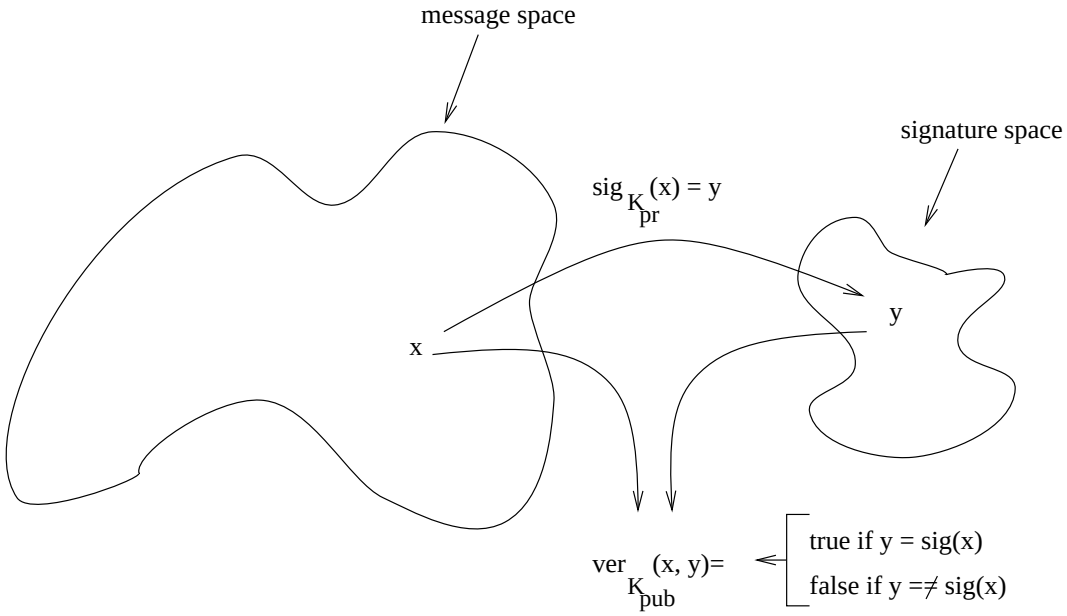


Figure 11.2: Digital signature and message domain

Basic protocol:

1. Bob signs his message x with **his** private key k_{pr}:
   $\Rightarrow y = sig_{k_{pr}}(x)$.

2. Bob sends (y, x) to Alice.

3. Alice runs the verification function $ver_{k_{pub}}(x, y)$ with **Bob's** public key.

Properties of digital signatures:

- Only Bob can sign his document (with k_{pr}).
- Everyone can verify the signature (with k_{pub}).
- Authentication: Alice is sure that Bob signed the message.
- Integrity: Message x cannot be altered since that would be detected through verification.
- Non-repudiation: The receiver of the message can *prove* that the sender had actually send the message.

It is important to note that the last property, sender non-repudiation, can only be achieved with public-key cryptography. Sender authentication and integrity can also be achieved via symmetric techniques, i.e., through message authentication codes (MACs).

11.2 RSA Signature Scheme

Set-up: $k_{pr} = (p, q, a)$; $k_{pub} = (n, b)$.

General Protocol:

1. Bob computes: $y = sig_{k_{pr}}(x) = e_{k_{pr}}(x) = x^a \bmod n$.
2. Bob sends (x, y) to Alice.
3. Alice verifies:

$$ver_{k_{pub}}(x, y) = d_{k_{pub}}(y) = y^b \begin{cases} = x & \Rightarrow \text{ true} \\ \neq x & \Rightarrow \text{ false} \end{cases}$$

Question: Why does it work?

$$d_{k_{pub}}(y) = d_{k_{pub}}(e_{k_{pr}}(x)) = x.$$

Remark:

- The role of public/private key are exchanged if compared with RSA public-key encryption.
- This algorithm was standardized in ISO/IEC 9796.

Drawback/possible attack:

Oscar can generate a valid signature for a *random* message x:

1. Choose signature $y \in Z_n$.
2. Encrypt: $x = e_{k_{pub}}(y) = y^b \bmod n \rightarrow$ outcome x **cannot** be controlled.
3. Send (x, y) to Alice.
4. Alice verifies: $ver_{k_{pub}}(x, y)$: $y^b \equiv x \bmod n \Rightarrow$ true.

The attack above can be prevented by formatting rules for the message x. For instance, a simple rule could be that the first and last 100 bits of x must all be zero (or one or any other specific bit pattern.) It is extremely unlikely that a random message x shows this bit pattern. Such a formatting scheme imposes a rule which distinguishes between valid and invalid messages.

11.3 ElGamal Signature Scheme

Remarks:

- ElGamal signature scheme is different from ElGamal encryption.
- Digital Signature Algorithm (DSA) is a modification of ElGamal signature scheme.
- This scheme was published in 1985.

Set-up:

1. Choose a prime p.
2. Choose primitive element $\alpha \in Z_p^*$.
3. Choose random $a \in \{2, 3, \ldots, p-2\}$.
4. Compute $\beta = \alpha^a \bmod p$.
   Public key: $k_{pub} = (p, \alpha, \beta)$.
   Private key: $k_{pr} = (a)$.

Signing:

1. Choose random $k \in \{0, 1, 2, \ldots, p-2\}$; such that $\gcd(k, p-1) = 1$.
2. Compute signature:

$$\begin{aligned} sig_{k_{pr}}(x, k) &= (\gamma, \delta), \quad \text{where} \\ \gamma &= \alpha^k \bmod p \\ \delta &= (x - a \cdot \gamma) k^{-1} \bmod p - 1 \end{aligned}$$

Public verification:

$$ver_{k_{pub}}(x, (\gamma, \delta)) = \beta^\gamma \cdot \gamma^\delta \begin{cases} = \alpha^x \bmod p & \text{valid signature} \\ \neq \alpha^x \bmod p & \text{invalid signature} \end{cases}$$

Question: Why does this scheme work?

$$\begin{aligned} \beta^\gamma \cdot \gamma^\delta &= (\alpha^a)^\gamma (\alpha^k)^{(x - a\cdot\gamma)k^{-1} \bmod (p-1)} \bmod p \\ &= \alpha^{a\cdot\gamma} \cdot \alpha^{k \cdot k^{-1}(x - a\cdot\gamma)} \bmod p \\ &= \alpha^{a\cdot\gamma - a\cdot\gamma + x} = \alpha^x \end{aligned}$$

11.4 Lessons Learned — Digital Signatures

- Digital signatures provide message integrity, sender authentication, and non-repudiation.
- One of the main application areas of digital signatures are certificates.
- RSA is the currently most widely used digital signature algorithm. Competitors are the Digital Signature Standard (DSA) and the Elliptic Curve Digital Signature Standard (ECDSA.)
- RSA verification can be done with short public keys b, whereas the signature key a must have the full length of the modulus n. Hence, RSA verification is fast and signing is slower.
- RSA digital signature is almost identical to RSA encryption, except that the private key is applied first to the message (signing), and the public key is applied to the signed message in the second step (verification.)
- As with RSA encryption, the modulus n should be at least 1024 bit long. This provides a long-term security roughly equivalent to an 80 bit symmetric cipher. For a better long-term security, a prime of length 2048 bit should be chosen.

Chapter 12

Error Coding (Channel Coding)

12.1 Cryptography and Coding

There are three basic forms of coding in modern communication systems: source coding, error coding (also called channel coding), and encryption. From an information theoretical and practical point of view, the three forms of coding should be applied as follows:

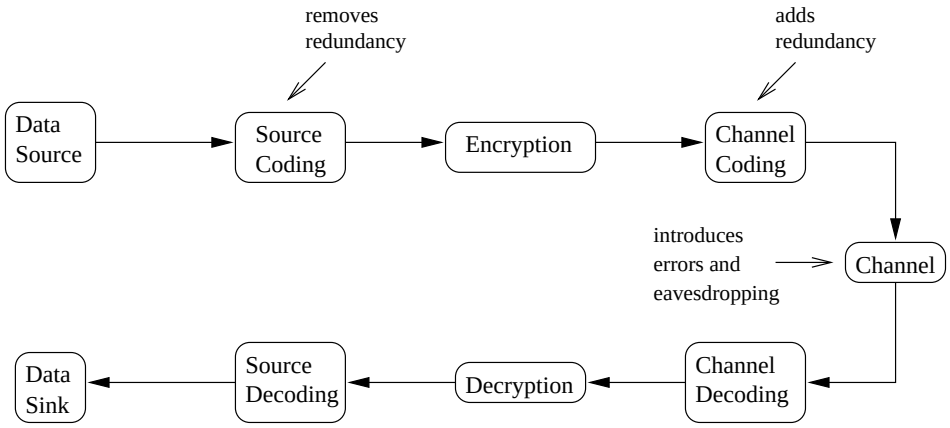


Figure 12.1: Coding in digital communication systems

Source Coding (Data Compression) Most data, such as text, has redundancy in it. This means the standard representation of the message, e.g., English text, uses more bits than necessary to uniquely represent the message contents. Source coding techniques extract the redundany and, thus, reduce the message length.

Encryption (Reminder: Pretty much everything in these lecture notes) The goal of encryption is to disguise the contents of a message. Only the owner of cryptographic keys should be able to recover the original content. Encryption can be viewed as a form of coding.

Channel Coding (Error Coding) The purpose of channel codes is to make the data robust against errors introduced during transmission over the channel.

It is very important for an understanding of cryptography to distinguish between these three forms of coding. In particular, error codes and encryption should not be confused. Roughly speaking, error codes protect against non-intentional malfunction (i.e., transmission errors due to noise), and enryption protects against malfunction due to human attackers, e.g., someone who tries to read or alter a message. Obviously "attacks by nature" (noise) are quite different than attacks by a smart and well-funded eavesdropper. In order to understand the difference between error coding and encryption better and in order to understand the requirements of hash functions, this chapter gives a brief introduction to error codes.

12.2 Basics of Channel Codes

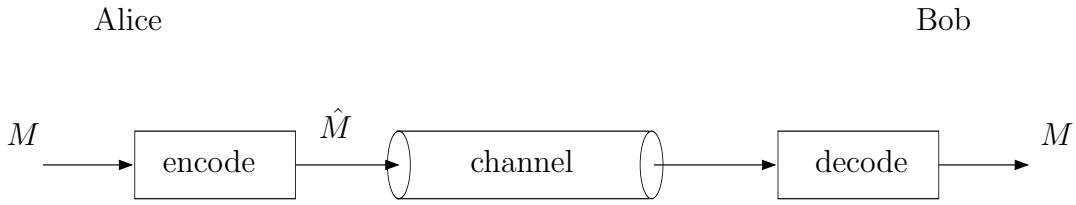


Figure 12.2: Simple Channel En-/ Decoding

The goal of channel codes it to make the data robust against errors on the transmission channel. The basic idea of channel codes is to introduce extra information (i.e., to add extra bits) to the data before transmission, so that there is a certain functional relationship between the message bits and the extra bits.

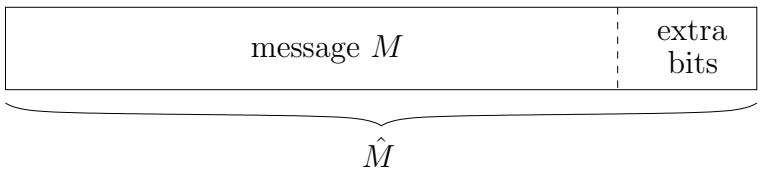


Figure 12.3: Encoded Message $\hat{M}$ with Redundant Bits

If this is done in a smart way it is unlikely (but not impossible) that a random error during data transmission changes the bits in such a way that the relationship between the bits are destroyed. The receiver (Bob) checks for the relationship between the message bits and the extra bit. Note that channel coding adds redundancy to the message, i.e., makes the transmitted data longer, which is the opposite of source coding.

There are two types of channel codes:

1. Error detection codes
2. Error correction codes

In the remainder of this chapter, only error detection codes are discussed.

12.3 Simple Parity Check Codes

To the message M a single parity check bit P is added:

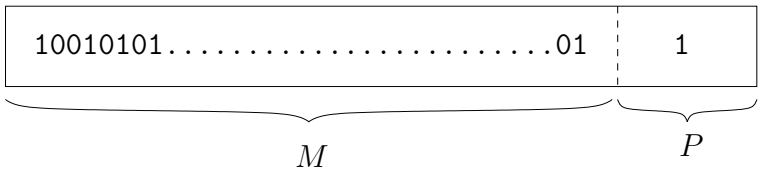


Figure 12.4: Message with Parity Check Bit

Let $M = m_1, m_2, \ldots, m_l$ be the message. Then the functional relationship between the message and P is:

$$\sum_{i=1}^{l} m_i + P \equiv 0 \bmod 2$$

From this, the construction of P for a given message follows trivally as:

$$P \equiv \sum_{i=1}^{l} m_i \bmod 2$$

A consequence of this coding scheme is that the number of bits that have the value "1" in a message together with the parity bit is always even. Hence, this coding scheme is also called *even parity*.

Example: Transmission of the ASCII character "s"= 1010 011

parity bit $P = 0$

transmitted $(M|P) = 1010\ 0110$

received $(M|P)' = 1010\ 0010$ (bit error in position 6)

error will be detected since the mod 2 sum of the bits received is not equal to 0.

Properties of simple parity check codes:

- they detec all odd number of bit errors, i.e., single bit errors, 3-bit errors, 5-bit errors, ...
- they do not detect any even number of bit errors
- they do not detect swapping of bits
  $1010\ 0010 \Rightarrow 1010\ 0100$

12.4 Weighted Parity Check Codes: The ISBN Book Numbers

Simple parity check codes are well suited for random errors introduced by white noise (i.e., errors are equally likely at any bit position and bit errors are independent of each other.) However, many real-world channels have a different error characteristic, for instance reordering of bits may occur. In order to detect reordering of bits on the channel, weighted parity check codes are used. A "channel" where this occurs frequently are transmissions of data by humans. An extremely wide spread example for a coding scheme which protects against many reordering errors is the ISBN (international standard book notation) numbering system.

Example:

ISBN: 3 – 540 – 59353 – 5

The functional relationship between the message (the first 9 digits) and the check sum is the following. Let

$$M = m_{10}, m_9, \ldots, m_2$$

be the message and P be the check sum, then:

$$\sum_{i=2}^{10} i\, m_i + P \equiv 0 \bmod 11$$

where $P \in Z_{11} = \{0, 1, \ldots, 9, X\}$.

12.5 Cyclic Redundancy Check (CRC)

Principle: We divide the message by a *generator* and consider the remainder of the division as a checksum (CRC), which is attached to the message.

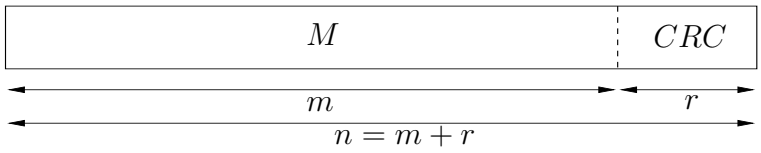


Figure 12.5: Message with CRC

Encoding:

1. Consider the message as a polynomial with binary coefficients:
   Example: $M = (1101011011) \rightarrow M(x) = x^9 + x^8 + x^6 + x^4 + x^3 + x + 1$

2. Shift the polynomial r positions to the left: $x^r \cdot M(x)$
   Example: $r = 4$
   $x^4 \cdot M(x) = x^{13} + x^{12} + x^{10} + x^8 + x^7 + x^5 + x^4$

3. Divide $x^r \cdot M(x)$ by the generator poylnomial $G(x) = x^4 + x + 1$. The remainder of the division is considered as checksum.

Example:

$$
\begin{array}{llllllll:lll}
(x^{13} & +x^{12} & +x^{10} & & +x^{8} & +x^{7} & +x^{5} & +x^{4}) & : (x^{4} & +x & +1) \\
-(x^{13} & & +x^{10} & +x^{9}) & & & & & & & \\
\hline
 & (x^{12} & & +x^{9} & +x^{8} & +x^{7} & +x^{5} & +x^{4}) & & & \\
 & -(x^{12} & & +x^{9} & +x^{8}) & & & & & & \\
\hline
 & & & & & (x^{7} & +x^{5} & +x^{4}) & & & \\
 & & & & & -(x^{7} & & +x^{4} & +x^{3}) & & \\
\hline
 & & & & & & (x^{5} & & +x^{3}) & & \\
 & & & & & & -(x^{5} & & & +x^{2} & +x) \\
\hline
 & & & & & & & & (x^{3} & +x^{2} & +x)
\end{array}
\quad = x^9 + x^8 + x^3 + x + H(x)/G(x)
$$

$$= H(x)$$

4. Build the transmission polynomial $T(x) = x^4 \cdot M(x) + H(x)$:

$$T(x) = (x^{13} + x^{12} + x^{10} + x^8 + x^7 + x^5 + x^4) + (x^3 + x^2 + x)$$

Remark:

a) $\deg H(x) < \deg G(x)$

b) $T(x)$ is divisible by $G(x)$:

$$\begin{aligned} T(x)/G(x) &= (x^r \cdot M(x))/G(x) + H(x)/G(x) \\ &= Q(x) + H(x)/G(x) + H(x)/G(x) = Q(x) \\ \Rightarrow T(x) &\equiv 0 \bmod G(x) \end{aligned}$$

c) The behavior of the code is completely determined by the generator polynomial

Decoding: Divide the received polynomial $R(x)$ by $G(x)$. If the remainder is *not* zero, an error occured. Otherwise, we *assume* no error occured:

$$R(x) \bmod G(x) = \begin{cases} = 0 & \text{error free} \\ \neq 0 & \text{error occured} \end{cases}$$

- An error at position i is represented by the error polynomial $E(x) = x^i$
  Example: error at position $0, 1, 8 \rightarrow E(x) = x^8 + x + 1$

- Channel: $R(x) = T(x) + E(x)$ (all bits at error positions are flipped)
  Example: $R(x) = x^{13} + x^{12} + x^{10} + x^7 + x^5 + x^4 + x^3 + x^2 + x + 1$

- Decoder:

$$\begin{aligned} R(x) \bmod G(x) &= (T(x) + E(x)) \bmod G(x) \\ &= T(x) \bmod G(x) + E(x) \bmod G(x) \\ &= 0 + E(x) \bmod G(x) \end{aligned}$$

  Condition for error detection: $E(x) \bmod G(x) \neq 0$

  or: error detection fails iff: $E(x) = Q(x)G(x)$

Chapter 13

Hash Functions

13.1 Introduction

The problem with digital signatures is that long messages require very long signatures. We would like for performance as well as for security reasons to have **one** signature for a message of arbitrary length. The solution to this problem are hash functions.

Note: There are many other applications of hash functions in cryptography beyond digital signatures. In particular, hash functions have become very popular for message authentication codes (MACs.)

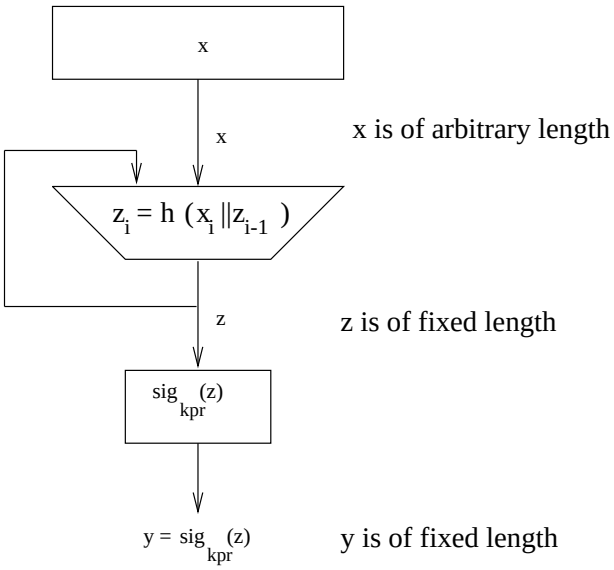


Figure 13.1: Hash functions and digital signatures

Remarks:

- z, x don't have the same length.
- $h(x)$ has no key.
- $h(x)$ is public.

Basic Protocol:

Alice		Bob
		1) $z = h(x)$
		2) $y = sig_{k_{pr}}(z)$
	3) $\xleftarrow{(x,y)}$	
4) $z = h(x)$		
5) $ver_{k_{pub}}(z, y)$		

Naïve approach: Use of error detection codes as hash functions

Principle of error correction codes: Given a message x, the sender computes $f(x)$, where $f()$ is a publically known function and sends $x||f(x)$. The receiver obtains x' and checks $f(x') \stackrel{?}{=} f(x)$.

Sender		Receiver
		1) x
		2) compute $f(x)$
	3) $\xleftarrow{(x,f(x))}$	
4) check if $f(x') = f(x)$		

Important: Error detection codes are designed to detect errors introduced during transmission on the channel, e.g., errors due to noise.

Let's try to use a column-wise parity check code. In this method, we compute an even parity check bit for every bit position. An even parity bit is defined such that the sum of all bits in the column is "1" if the XOR sum of all column bits is "1", and "0" if the XOR of sum of all column bits is "0".
E.g., consider a text $x = (x_1, x_2, ..., x_l)$ consisting of ASCII symbols x_i. We can compute the parity bits $P = (p_1, p_2, ..., p_l)$ by bitwise XOR of the column entries:

$$\begin{aligned} x_1 &= \texttt{00101010} \\ \oplus\, x_2 &= \texttt{01010011} \\ &\vdots \\ \oplus\, x_l &= \texttt{11101000} \\ \hline P = (p_1, p_2, ..., p_l) &= \texttt{10010100} \end{aligned}$$

The problem with error detection codes is, that they were designed to detect *random* errors, and not "errors" introduced by an intelligent opponent such as Oscar. Oscar can easily alter the message and additionally alter the parity bits such that the changes go undetected.

Requirements for a hash function

1. $h(x)$ can be applied to x of any size.
2. $h(x)$ produces a fixed length output.
3. $h(x)$ is relatively easy to compute in software and hardware.
4. *One-way:* for (almost) all given output z, it is impossible to find any input x such that $h(x) = z$.is one-way.
5. *Weak collision resistant:* given x, and thus $h(x)$, it is impossible to find any x' such that $h(x) = h(x')$.
6. *Strong collision resistant:* it is impossible to find any two pairs x, x' such that
   $h(x) = h(x')$.

Discussion:

- (1) — (3) are practical requirements
- (4) if $h(x)$ is not one-way, Oscar can compute x from $h(x)$ in cases where x is encrypted.

- (5) if $h(x)$ is not weak collision free, Oscar can replace x with x'.

Alice	Oscar	Bob
		$z = h(x)$
	$\xleftarrow{(x,y)}$	$y = sig_{K_{pr}}(z)$
	$\xleftarrow{(y,x')}$	
$z = h(x') = h(x)$		
$ver_{K_{pub}}(z, y) = true$		

- (6) if $h(x)$ is not strong collission free, Oscar runs the following attack:

  a) Choose legitimate message x_1 and fraudulent message x_2

  b) Alter x_1 and x_2 at "non-visible" location, i.e. replace tabs through spaces, append returns, etc., until $h(x_1') = h(x_2')$ (*Note:* e.g. 64 alteration locations allow 2^{64} versions of a message with 2^{64} different hash values).

  c) Let Bob sign $x_1' \rightarrow (x_1', sig_{K_{pr}}(h(x_1'))$

  d) Replace $x_1' \rightarrow x_2'$ and $(x_2', sig_{K_{pr}}(h(x_2'))$

Question: Why is there no collision free hash function?

Answer: There exist far more x than z!

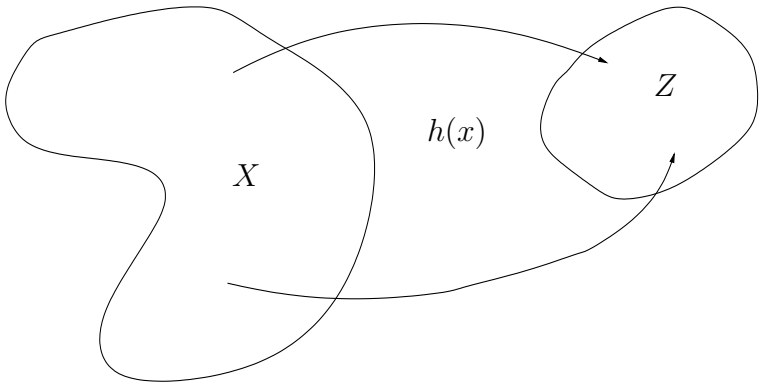


Figure 13.2: Map $h(x)$ from $X = \{x\}$ to $Z = \{z\}$

$h(x)$ is the map from X to Z, where $|X| >> |Z|$ with $x \in X, z \in Z$. A minimum in possible collisions is the objective of any hash function. The function $h(x)$ (and the size of Z) has to assure that a construction of two different x's with the same hash value z is time-consuming and, thus, not feasible.

13.2 Security Considerations

Question: How many people are needed at a party so that there is a 50% chance that at least two people have the same birthday?

In general, given a large set with n different values:

$$P(\text{no collission among } k \text{ random elements}) = \underbrace{\underbrace{\left(1-\frac{1}{n}\right)}_{k=2 \text{ elt.}}\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)}_{k \text{ elt.}} = \prod_{i=1}^{k-1}\left(1-\frac{i}{n}\right)$$

Often n is large ($n = 365$ in birthday paradox, $n = 2^{160}$ in hash functions).

Recall:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$$

if $x << 1$

$$e^{-x} \approx 1 - x$$

Thus,

$$\begin{aligned} P(\text{no collision}) \approx \prod_{i=1}^{k-1} e^{-\frac{i}{n}} &= e^{-\frac{1}{n}} e^{-\frac{2}{n}} e^{-\frac{3}{n}} \cdots e^{-\frac{k-1}{n}} \\ \prod_{i=1}^{k-1} e^{-\frac{i}{n}} &= e^{-\frac{1+2+3+\cdots+k-1}{n}} \end{aligned}$$

Rewriting the exponent with the help of the following identity:

$$1 + 2 + 3 + \cdots + k - 1 = k(k-1)/2$$

We obtain,

$$P(\text{no collission}) \approx e^{-\frac{k(k-1)}{2n}}$$

Define ϵ as

$$\begin{aligned} P(\text{at least one collission}) &\overset{DEF}{=} \epsilon \approx 1 - e^{-\frac{k(k-1)}{2n}} \\ 1 - \epsilon &\approx e^{-\frac{k(k-1)}{2n}} \\ \ln(1-\epsilon) &\approx -\frac{k(k-1)}{2n} \\ k(k+1) &\approx -2n\ln(1-\epsilon) = 2n\ln\left(\frac{1}{1-\epsilon}\right) \end{aligned}$$

If $k >> 1$, then

$$\begin{aligned} k^2 &\approx k(k-1) \approx 2n \ln\left(\frac{1}{1-\epsilon}\right) \\ k &\approx \sqrt{2n \ln\left(\frac{1}{1-\epsilon}\right)} \end{aligned}$$

Example:

$$k(\epsilon = 0.5) \approx \sqrt{2n \ln\left(\frac{1}{1-0.5}\right)} = \sqrt{2 \ln 2}\sqrt{n} = 1.18\sqrt{n}$$

$\Rightarrow$ A collission in a set of n values is found after about $\sqrt{n}$ trials with a probability of 0.5. In other words, given a hash function with 40 bit output $\Rightarrow$ collission after approximately $\sqrt{2^{40}} = 2^{20}$ trials.

$\Rightarrow$ In order to provide collision resistance in practice, the output space of the hash function should contain at least 2^{160} elements, that is, the hash function should have at least 160 output bits. Finding a collision takes then roughly $\sqrt{2^{160}} = 2^{80}$ steps.

13.3 Hash Algorithms

Overview:

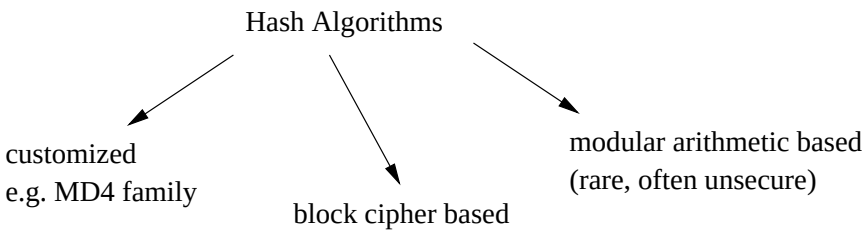


Figure 13.3: Family of Hash Algorithms

a) MD4–family

1. **SHA-1**

Output: 160 bits $\Rightarrow$ input size for DSS.
Input: 512 bit chunks of message x.
Operations: bitwise AND, OR, XOR, complement and cyclic shift.

2. **RIPE-MD 160**

Output: 160 bits.
Input: 512 bit chunks of message x.
Operations: same as SHA but runs two algorithms in parallel whose outputs are combined after each round.

b) Hash functions from block ciphers

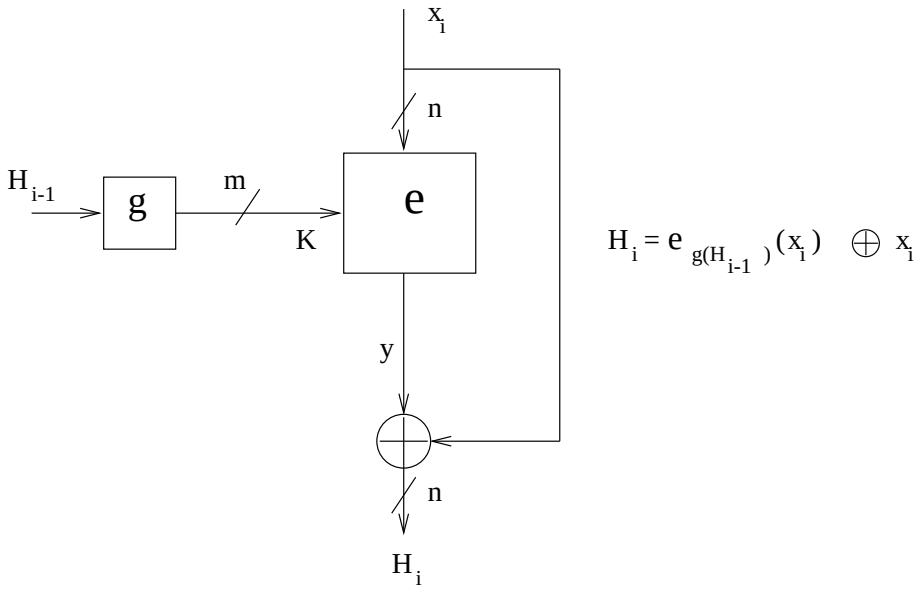


Figure 13.4: Hash Functions from Block Ciphers

where g is a simple n-to-m bit mapping function (if $n = m$, g can be the identity mapping)

Last output H_l is the hash of the whole message x_1,x_2,. . .,x_l

Also secure are:

- $H_i = H_{i-1} \oplus e_{x_i}(H_{i-1})$
- $H_i = H_{i-1} \oplus x_i \oplus e_{g(H_{i-1})}(x_i)$

Remark:

For block ciphers with less than 128 bit block length, different techniques must be used (Sec. 9.4.1 (ii) in [AM97])

13.4 Lessons Learned — Hash Functions

- Hash functions are key-less. They serve as auxiliary functions in many cryptographic protocols.
- Among the two most important applications of hash functions are: support function for digital signatures and core function for building message authentication codes, e.g., HMAC.
- Hash functions should have at least 160 bit output length in order to withstand collision attacks. 256 or more bits are better.
- SHA-1 and RIPEMD160 are considered to be secure hash functions.

Chapter 14

Message Authentication Codes (MACs)

Other names: "cryptographic checksum" or "keyed hash function".

Message authentication codes are widely used in practice for providing message integretiy and message authentication in cases where the two communication parties share a secret key. MACs are much faster than digital signatures since they are based on symmetric ciphers or hash functions.

14.1 Principle

Similar to digital signatures, MACs append an "authentication tag" to a message. The main difference is that MACs use a symmetric key on both the sender and receiver side.

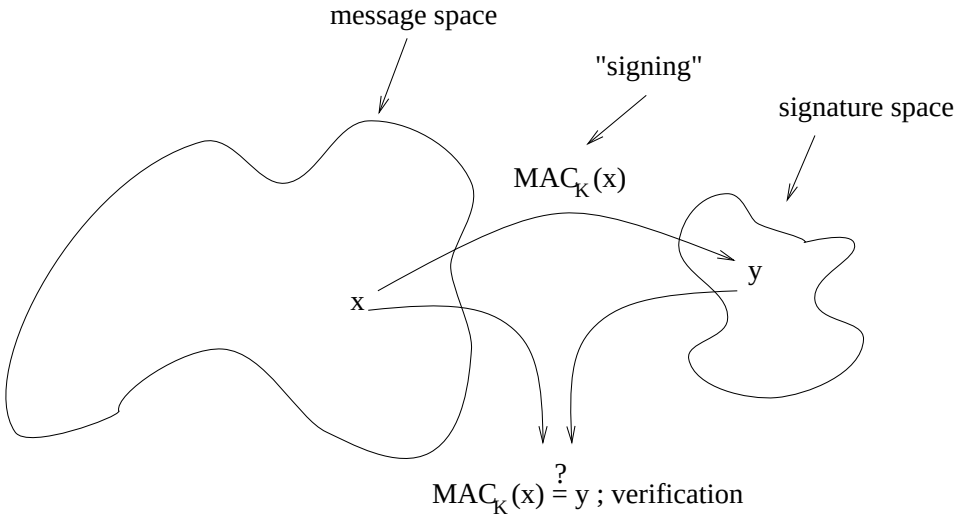


Figure 14.1: Message authentication codes

Protocol:

Alice		Bob
		1) $y = \text{MAC}_K(x)$
	2) $\xleftarrow{(x,y)}$	
3) $y' = \text{MAC}_K(x)$		
$y' \stackrel{?}{=} y$		

Note: For MAC verification, Alice performs exactly the same steps that Bob used for generating the MAC. This is quite different from digital signatures.

Properties of MACs:

1. Generate signature for a given message.
2. Symmetric-key based: signing and verifying party must share a secret key.
3. Accepts messages of arbitrary length and generates fixed size signature.
4. Provides message integrity.
5. Provides message authentication.
6. Does not provide non-reputation.

Note: Properties 2, 3, and 6 are different from digital signatures.

14.2 MACs from Block Ciphers

MAC generation: Run block cipher in CBC mode

$y_0 = e_k(x_0 \oplus IV) = e_k(x_0 \oplus 0000\ldots)$

$y_i = e_k(x_i \oplus y_{i-1})$

$X = x_0, x_1, \ldots, x_{m-1}$

$MAC_k(x) = y_{m-1}$

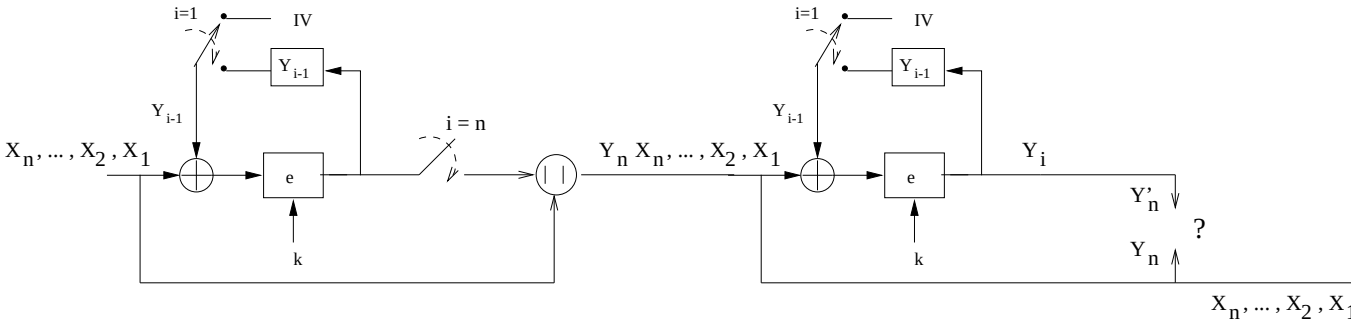


Figure 14.2: MAC built from a block cipher in CBC mode

MAC Verification: Run the same process that was used for MAC generation on the receiving end.

Remark: CBC with DES is standardized (ANSI X9.17).

14.3 MACs from Hash Functions: HMAC

- Popular in modern protocols such as SSL.
- **Attractive property**: HMAC can be *proven* to be secure under certain assumptions about the hash function. "Secure" means here that the hash function has to be broken in order to break the HMAC.
- **Basic idea**: Hash a secret key K together with the message M and consider the hash output the authentication tag for the message: $H(K||M)$.
- **Details**

  $\text{HMAC}_K(M) = H\left[(K^+ \oplus \text{opad})||H\left[(K^+ \oplus \text{ipad})||M\right]\right]$

  where

  $K^+ = K$ padded with zeros on the left so that the result is b bits in length (where b is the number of bits in a block).

  ipad = 00110110 repeated $b/8$ times.

  opad = 01011010 repeated $b/8$ times.

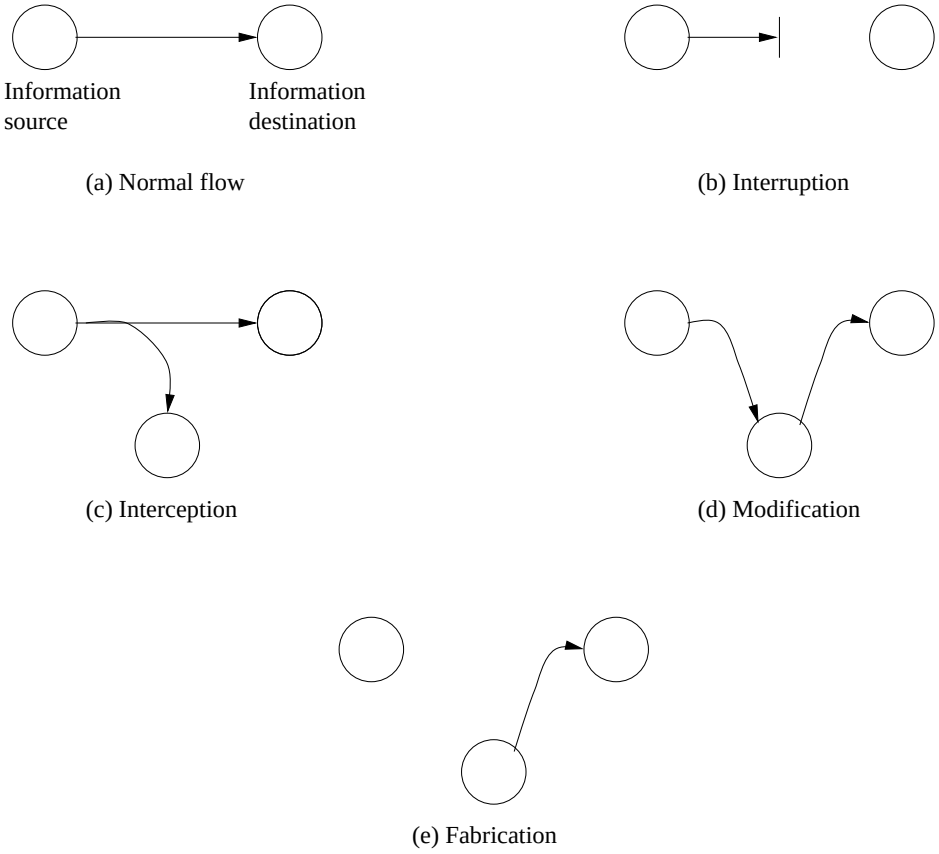
14.4 Lessons Learned — Message Authentication Codes

- MACs provide the two security services message integrity and message authentication using symmetric techniques. MACs are widely used in protocols in practice.
- Both of these services are also provided by digital signatures but MACs are much faster.
- MACs do not provide non-repudiation.
- In practice, MACs are either based on block ciphers or on hash functions.
- HMAC is a popular MAC used in many practical protocols such as SSL.

Chapter 15

Security Services

15.1 Attacks Against Information Systems



(a) Normal flow

(b) Interruption

(c) Interception

(d) Modification

(e) Fabrication

Remarks:

- Passive attacks: (c) → interception.
- Active attacks: (b) → interruption, (d) → modification, (e) → fabrication.

15.2 Introduction

Security Services are **goals** which information security systems try to achieve. Note that cryptography is only **one module** in information security systems.

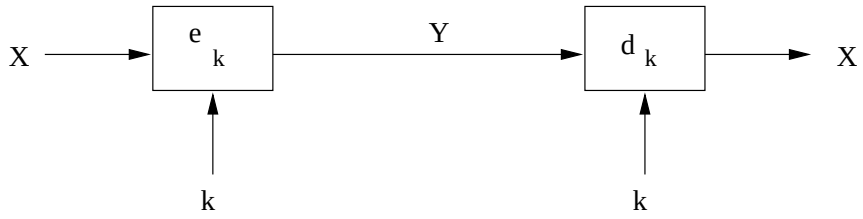
The main security services are:

- *Confidentiality/Privacy.* Information is kept secret from all but authorized parties.
- *(Message/Sender) Authentication.* Ensures that the sender of a message is who she/he claims to be.
- *Integrity.* Ensures that a message has not been modified in transit.
- *Non-repudiation.* Ensures that the sender of a message can not deny the creation of the message.
- *Identification/Entity Authentication.* Establishing of the identity of an entity (e.g. a person, computer, credit card).
- *Access Control.* Restricting access to the resources to privileged entitites.

Remark: Message Authentication implies data integrity; the opposite is not true.

15.3 Privacy

Tool: Encryption algorithm.



a) Symmetric-Key

Provides:

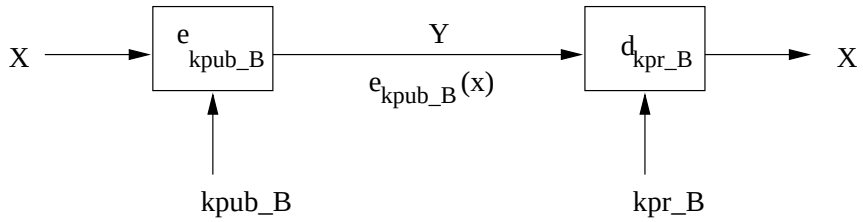
−privacy
−message authentication and thus
−integrity
−no non-repudiation

} only if Bob can distinguish between valid and invalid X and if there are only two parties.

Remark:

In practice, authentication and integrity are often achieved with MACs (Chapter 14)

b) Public-Key



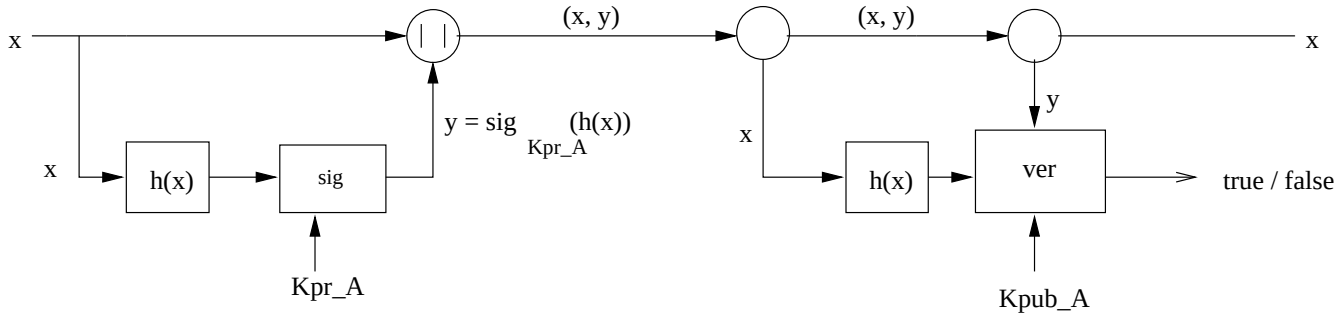
Provides:

- privacy
- integrity (if invalid x can e detected)
- no message authentication

15.4 Integrity and Sender Authentication

Recall: Sender authentication implies integrity.

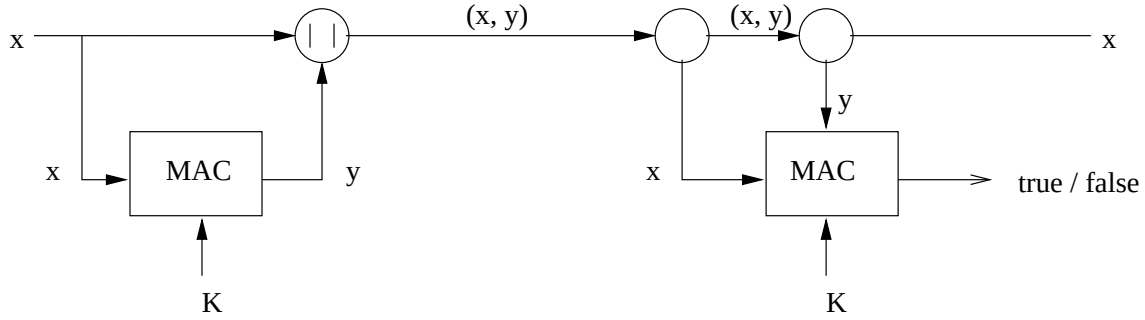
15.4.1 Digital Signatures



Provides:

- integrity
- sender authentication
- non-repudiation (only Alice can construct valid signature)

15.4.2 MACs



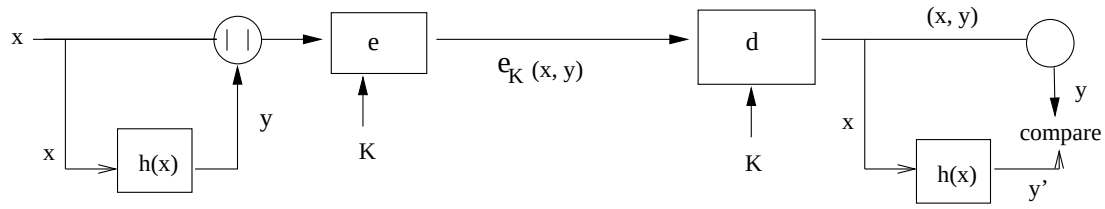
Provides:

- integrity

- authentication

- no non-repudiation

15.4.3 Integrity and Encryption



Provides:

- privacy

- integrity

- authentication

- no non-repudiation

Remark:

- Instead of hash functions, MACs are also possible. In this case: $c = e_{K_1}(x, \mathrm{MAC}_{K_2}(y))$.

- This scheme adds strong authentication and integrity to an encryption-protocol with very little computational overhead.

Chapter 16

Key Establishment

16.1 Introduction

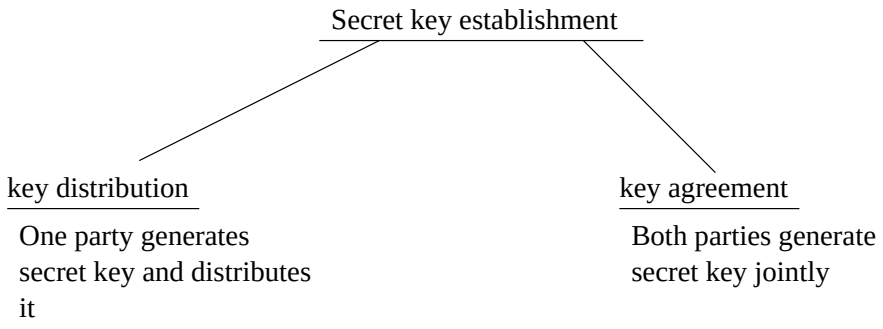


Figure 16.1: Key establishment schemes

Remark:

Some schemes make use of trusted authority (TA) which is trusted by and can communicate with all users.

16.2 Symmetric-Key Approaches

16.2.1 The n^2 Key Distribution Problem

TA generates a key for every pair of users:

Example: $n = 4$ users.

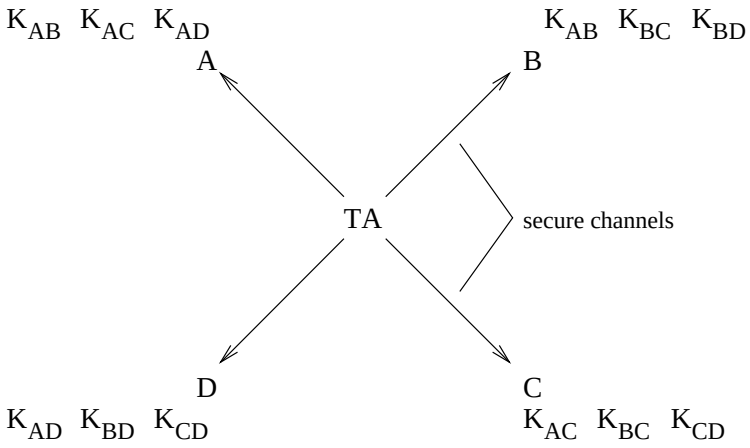


Figure 16.2: The role of the Trusted Authority

Drawbacks:

- n secure channels are needed
- each user must store $n - 1$ keys
- TA must transmit $n(n - 1)$ keys
- TA must generate $\frac{n(n-1)}{2} \approx \frac{n^2}{2}$ keys
- every new network user makes updates at all other user as of necessary $\Rightarrow$ scales badly

16.2.2 Key Distribution Center (KDC)

TA is a KDC: TA shares secret key with each user and generates session keys.

a) Basic protocol:

- k_s = session key between Alice and Bob
- $k_{A,KDC}$ = secret key between Alice and KDC (Key encryption key, KEK)
- $k_{B,KDC}$ = secret key between Bob and KDC (Key encryption key, KEK)

Alice		KDC		Bob
	$\xleftarrow{e_{k_{A,KDC}}(k_s)=y_A}$		$\xrightarrow{e_{k_{B,KDC}}(k_s)=y_B}$	
$k_s = d_{k_{A,KDC}}(y_A)$				$k_s = d_{k_{B,KDC}}(y_B)$
$y = e_{k_s}(x)$		$\xrightarrow{y}$		$x = d_{k_s}(y)$

Remarks:

– TA stores only n keys

– each user U stores only one key

b) Modified (advanced) protocol:

Alice		KDC		Bob
		1a) $y_A = e_{k_A}(k_s)$		
		1b) $y_B = e_{k_B}(k_s)$		
	2) $\xleftarrow{(y_A,y_B)}$			
3) $k_s = d_{k_A}(y_A)$				
4) $y = e_{k_s}(x)$		5) $\xrightarrow{(y,y_B)}$		7) $k_s = d_{k_B}(y_B)$
				6) $x = d_{k_s}(y)$

Remark: This approach is the basis for Kerberos.

16.3 Public-Key Approaches

16.3.1 Man-In-The-Middle Attack

D-H key exchange revised

Set-up:

- find large prime p
- find primitive element $\alpha \in Z_p$

Protocol:

Alice		Bob
pick $k_{prA} = a_A \in \{2, 3, \ldots, p-2\}$		pick $k_{prB} = a_B \in \{2, 3, \ldots, p-2\}$
compute $k_{pubA} = b_A = \alpha^{a_A} \bmod p$		compute $k_{pubB} = b_B = \alpha^{a_B} \bmod p$
	$\xrightarrow{b_A}$	
	$\xleftarrow{b_B}$	
$k_{AB} = b_B^{a_A} = \alpha^{a_A a_B} \bmod p$		$k_{AB} = b_A^{a_B} = \alpha^{a_A a_B} \bmod p$

Security:

1. passive attacks

   $\Rightarrow$ security relies on Diffie-Hellman problem thus $p > 2^{1000}$.

2. active attack

   $\Rightarrow$ *Man-in-the-middle attack:*

Alice		Oscar		Bob
	$\xrightarrow{\alpha^a}$		$\xrightarrow{\alpha^o}$	
	$\xleftarrow{\alpha^o}$		$\xleftarrow{\alpha^b}$	
$k_{AO} = (\alpha^o)^a = \alpha^{ao}$		$k_{AO} = (\alpha^a)^o$		$k_{BO} = (\alpha^o)^b = \alpha^{bo}$
		$k_{BO} = (\alpha^b)^o$		
$y' = e_{k_{AO}}(x)$	$\xrightarrow{y'}$	$x = d_{k_{AO}}(y')$		
		$y'' = e_{k_{BO}}(x)$	$\xrightarrow{y''}$	$x = d_{k_{BO}}(y'')$

Remarks:

- Oscar can read and alter x without detection.
- Underlying Problem: *public keys are not authenticated.*
- **Man-in-the-middle attack applies to all Public-key schemes.**

16.3.2 Certificates

Problem: Public keys are not authenticated!

Solution:

1. Digital signatures (asymmetric)
2. MACs (symmetric)

Review: Digital signatures

Alice		Bob
$y = sig_{K_{prA}}(x)$	$\xrightarrow{(x,y)}$	$ver_{K_{pubA}}(x, y)?$

Idea: Sign public key together with identification.

$$[K_{pubA}, ID(A)], sig[K_{pubA}, ID(A)] = \text{Certificate}$$

Question: Who issues certificates?

Answer: "CA" = Certification Authority

Certificates bind ID information (e.g., name, social security number) to a public key through digital signatures.

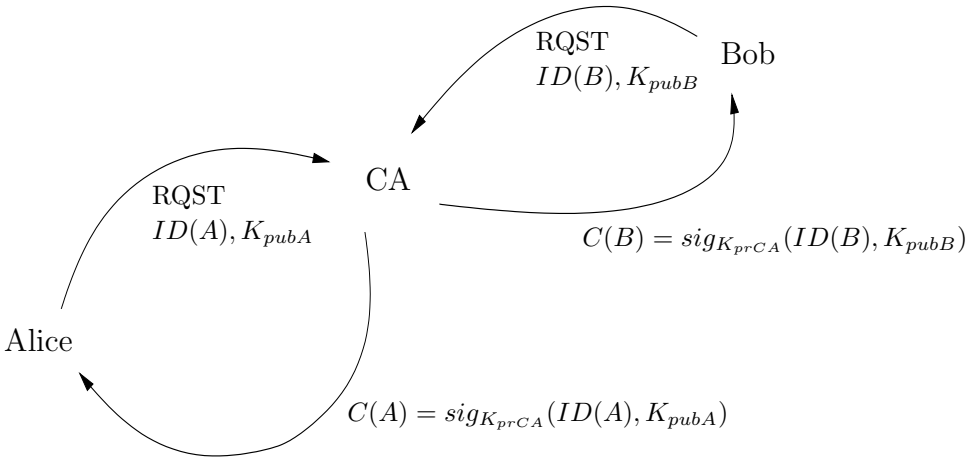


Figure 16.3: Certification workflow

General structure of certificates:

1. Each user U:
   - ID(U) = ID information such as user name, e-mail address, SS#, etc.
   - private key: K_{prU}
   - public key: K_{pubU}
2. Certifying Authority (CA):
   - secret signature algorithm sig_{TA}
   - public verification algorithm ver_{TA}
   - certificates for each user U:
     $C(U) = (ID(U), K_{pubU}, sig_{TA}(ID(U), K_{pubU}))$

General requirement: all users have the correct verification algorithm ver_{TA} with TA's public key.

Remarks:

- Certificate structures are specified in X.509, authentication services for the X.500 directory recommendation (CCITT).

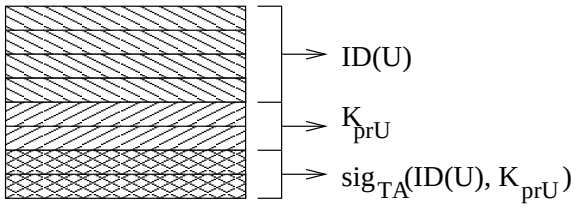


Figure 16.4: General structure of the certificate C(U)

Version
Serial Number
Algorithm Identifier: - Algorithm - Parameters
Issuer
Period of Validity: - Not Before Date - Not After Date
Subject
Subject's Public Key: - Algorithm - Parameters - Public Key
Signature

Figure 16.5: Detailed structure of an X.509 certificate

16.3.3 Diffie-Hellman Exchange with Certificates

Idea: Same as standard Diffie-Hellman key exchange, but each users's public key is authenticated by a certificate.

Alice		Bob
$K_{pubA} = b_A$		$K_{pubB} = b_B$
$K_{prA} = a_A$		$K_{prB} = a_B$
	$C(B)=(ID(B),b_B,sig_{CA}(ID(B),b_B))$ $\longleftarrow$	
	$C(A)=(ID(A),b_A,sig_{CA}(ID(A),b_A))$ $\longrightarrow$	
1.) $ver_{CA}(ID(B), b_B)$		1.) $ver_{CA}(ID(A), b_A)$
2.) $k_{AB} = b_B^{a_A} = \alpha^{a_B a_A} = \alpha^{a_A a_B}$		2.) $k_{AB} = b_A^{a_B} = \alpha^{a_A a_B}$

Question: Does Oscar have any further possibilities for an attack?

Answer:

1. Oscar impersonates Alice to obtain a certificate of the CA (with his key but Alices' identity)

2. Oscar replaces the CA's public key by his public key:

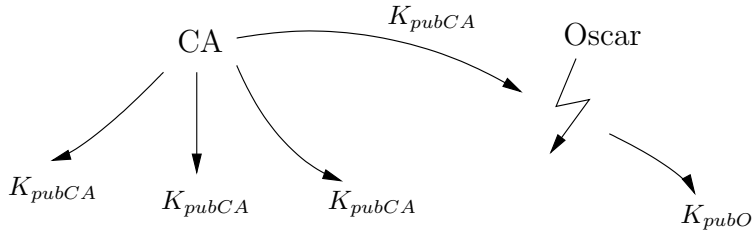


Figure 16.6: Simple attack on a CA

Remaining major problems with CAs:

1. The CA's public key must initially be distributed in an authenticated manner!

2. Identity of user must be established by CA.

3. Certificate Revocation Lists (CRLs) must be distributed.

16.3.4 Authenticated Key Agreement

Idea: Alice and Bob sign their own public keys. Signatures can be correctly verified through certificates.

Set-up:

- public verification key for ver_{TA}
- public prime p
- public primitive element $\alpha \in Z_p$

Protocol:

Alice		TA		Bob
		$C(A)=(ID(A),ver_A,sig_{TA}(ID(A),ver_A))$ $\xleftarrow{}$		
		$C(B)=(ID(B),ver_B,sig_{TA}(ID(B),ver_B))$ $\xrightarrow{}$		
1.) $k_{prA} = a_A$				
2.) $k_{pubA} = b_A = \alpha^{a_A} \bmod p$	$\xrightarrow{b_A}$			
				3.) $k_{prB} = a_B$
				4.) $k_{pubB} = b_B = \alpha^{a_B} \bmod p$
				5.) $k_{AB} = b_A^{a_B} = \alpha^{a_A a_B} \bmod p$
		$(C(B),b_B,y_B)$ $\xleftarrow{}$		6.) $y_B = sig_B(b_B, b_A)$
7.) $ver_{TA}(C(B))$: true/false				
8.) $ver_B(y_B)$: true/false				
9.) $k_{AB} = b_B^{a_A} = \alpha^{a_A a_B} \bmod p$				
10.) $y_A = sig_A(b_A, b_B)$	$\xrightarrow{(C(A),y_A)}$			
				11.) $ver_{TA}(C(A))$: true/false
				12.) $ver_A(y_A)$: true/false

Remark:

This scheme is also known as station-to-station protocol and is the basis for ISO 9798-3.

Chapter 17

Case Study: The Secure Socket Layer (SSL) Protocol

Note:

This chapter describes the most important security mechanisms of the SSL Protocol. For more details references [Sta02] and Netscape's SSL web page are recommended.

17.1 Introduction

- SSL was developed by Netscape.
- TLS (Transport Layer Security) is the IETF standard version of SSL. TLS is very close to SSL.
- SSL provides security services for end-to-end applications.
- Most applications must be SSL enabled, i.e., SSL is **not** transparent.
- SSL is *algorithm independent*: for both public-key and symmetric-key operations, several algorithms are possible. Algorithms are negotiated on a per-session basis.

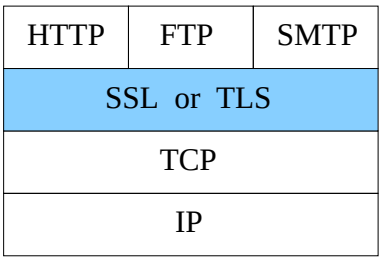


Figure 17.1: Location of SSL in the TCP/IP protocol stack.

- SSL consists of two main phases:

  Handshake Protocol : provides shared secret key using public-key techniques and mutual entity authentication.

  Record Protocol : provides confidentiality and message integrity for application data, using the shared secret established during the Handshake Protocol.

17.2 SSL Record Protocol

The SSL Record Protocol provides two main services:

1. *Confidentiality*: SSL payloads are encrypted with a symmetric cipher. The keys are for the symmetric cipher and they must be established during the preceding handshake protocol.

2. *Message Integrity*: the integrity of the message is provided through HMAC, a message authentication code.

17.2.1 Overview of the SSL Record Protocol

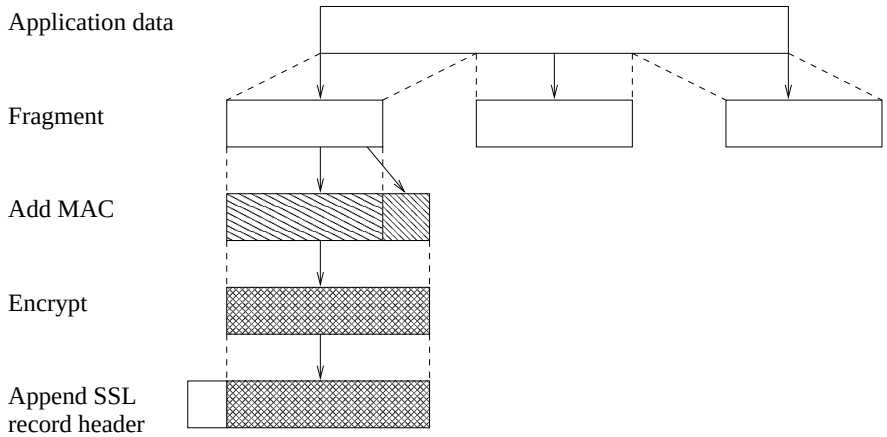


Figure 17.2: Simplified operations of the SSL Record Protocol

Description:

- **Fragmentation**: the message is devided into blocks of 2^{14} bytes.

- **MAC**: a derivative of the popular HMAC message authentication code. HMACs are based on hash functions.

  MAC = H(secret-key || pad2 ||
  H(secret-key || pad1 || seq-num || fragment-length || fragment))

where:

H = hash algorithm; either MD5 or SHA-1.

secret-key = shared secret session key.

pad1 = the byte 0x36 (0011 0110) repeated 48 times (384 bits) for MD5 and 40 times (320 bits) for SHA-1.

pad2 = the byte 0x5C (0101 1100) repeated 48 times for MD5 and 40 times for SHA-1.

seq-num = the sequence number of the message.

fragment-length = length of the fragment (plaintext).

fragment = the plaintext block for which the MAC is computed.

- **Encrypt**: the following algorithms are allowed:

  1. Block ciphers:
     - IDEA (128-bit key)
     - RC-2 (40-bit key)
     - DES-40 (40-bit key)
     - DES (56-bit key)
     - 3DES (168-bit key)
     - Fortezza (80-bit key)
  2. Stream ciphers:
     - RC4-40 (40-bit key)
     - RC4-128 (128-bit key)

17.3 SSL Handshake Protocol

Remark: Most complex part of SSL, requires costly public-key operations

17.3.1 Core Cryptographic Components of SSL

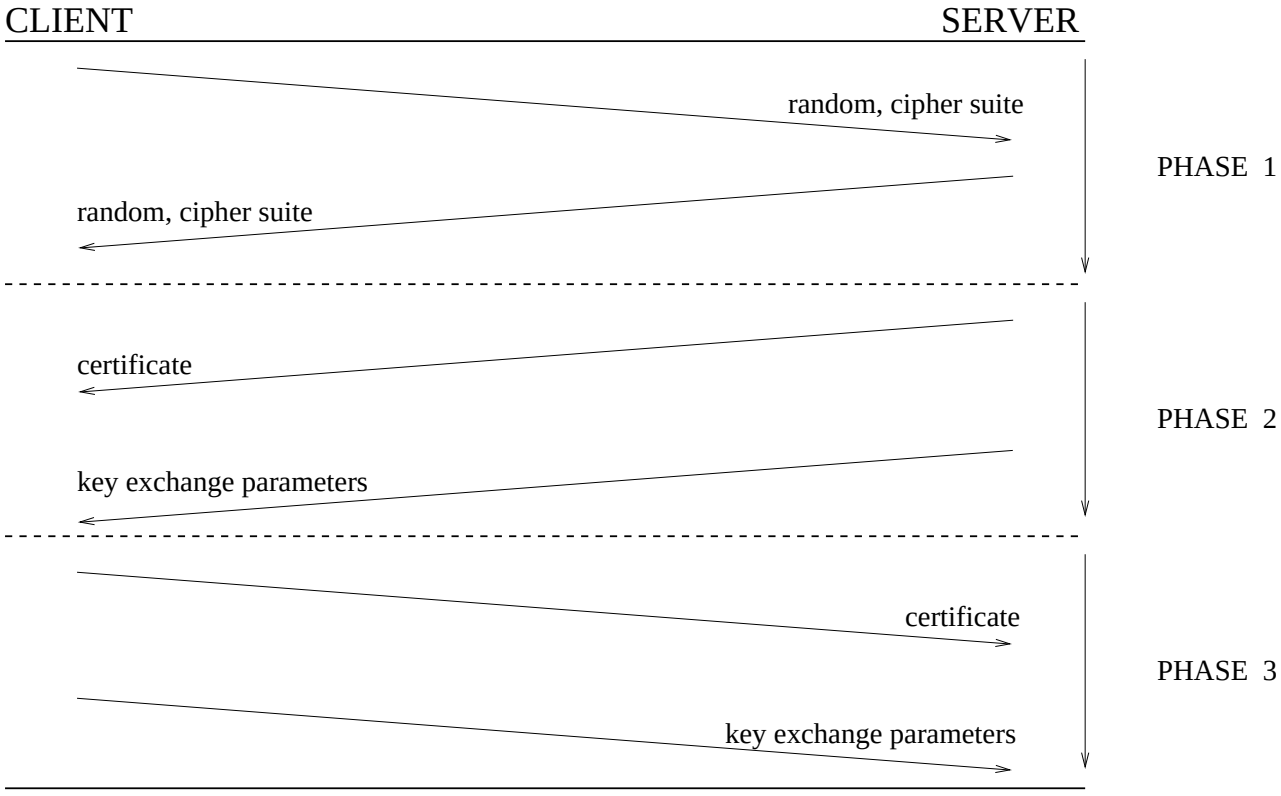


Figure 17.3: Simplified SSL Handshake Protocol

Explanation:

- **Phase 1**: establish security capabilities.

  random : 32-bit timestamp concatenated with 28-byte random value. Used as nonces and to prevent replay attacks during the key exchange.

  cipher suite : several fields, in particular:

  1. Key exchange method.

(a) RSA: the secret key is encrypted with the receiver's public RSA-key. Certificates are required.

(b) Authenticated Diffie-Hellman: Diffie-Hellman with certificate.

(c) Anonymous Diffie-Hellman: Diffie-Hellman without authentication.

(d) Fortezza

2. Secret-key algorithm (see Section 17.2).

3. MAC algorithm (MD5 or SHA-1).

- **Phase 2**: server authentication and key exchange.

  Certificate : authenticated public key for any key exchange method except anonymous Diffie-Hellman.

  Key exchange parameters : signed public-key parameters, depending on the key exchange method.

- **Phase 3**: see Phase 2.

Chapter 18

Introduction to Identification Schemes

Examples for electronic identification situation:

1. Money withdrawal from ATM machine (PIN).
2. Credit card purchase over telephone (card number).
3. Remote computer login (user name and password).

Distinction between identification (or entity authentication) and message authentication:

- Identification schemes are performed online.
- Identification schemes do not require a meaningful message.

Basis for identification techniques:

1. Something known (password, PIN) } cryptography based
2. Something possessed (chipcard) }
3. Something inherent to a human individual (fingerprint, retina pattern)

Overview:

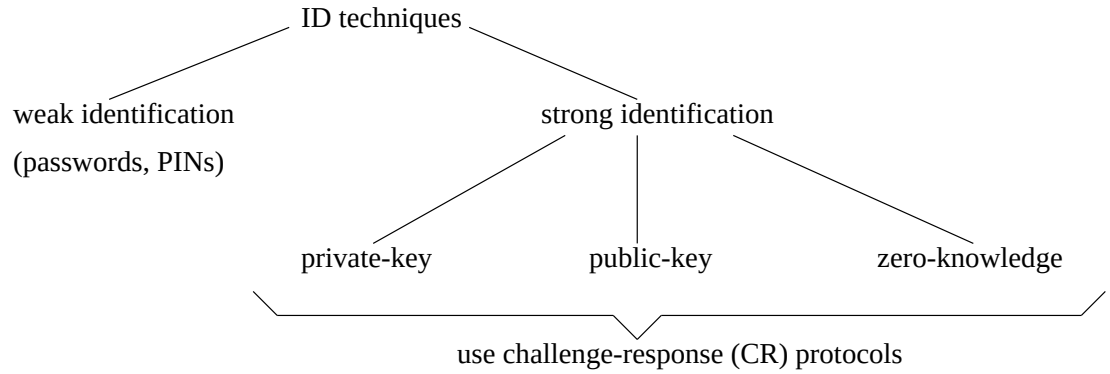


Figure 18.1: Identification Techniques

$\Rightarrow$ passwords and PINs are weak since they violate requirement 1 below.

Goals (informal definition):

1. Alice wants to prove her identity to Bob without revealing her identifying information to a listening Oscar. ("strong identification")

2. Also, Bob should not be able to impersonate Alice.

To achieve these goals, Alice has to perform a *proof of knowledge* which in general involves a *challenge-and-response* protocol.

18.1 Symmetric-key Approach

Challenge-and-response (CR) protocol:

Assumption: Alice and Bob share a secret key k_{AB} and a keyed one-way function $f(x)$.

Alice		Bob
		1) generate challenge$x$
	$\xleftarrow{x}$	
2) $y = f_{k_{AB}}(x)$	$\xrightarrow{y}$	
		3) $y' = f_{k_{AB}}(x)$
		4) verification: $y \stackrel{?}{=} y'$

Example:

a) $f_k(x) = DES_k(x)$.

b) $f_k(x) = H(k||x)$.

c) $f_k(x) = x^k \bmod p$.

Remarks:

- CR protocols are standardized in ISO/IEC 9798.
- There are many variations to the above protocol, e.g., including time stamps or serial numbers in the response.
- Instead of block ciphers, public-key algorithms and keyed hash functions can be used.

Variant with time stamp (TS)

Alice		Bob
1) $y = e_{k_{AB}}(TS, ID(\text{Bob}))$		
	$\xrightarrow{y}$	
		2) $(TS', ID'(\text{Bob}) = e_{k_{AB}}^{-1}(y)$
		$TS \overset{?}{\leq} \text{time} \overset{?}{\leq} TS + \epsilon$

Bibliography

[AM97] S.A. Vanstone A.J. Menezes, P.C. Oorschot. *Handbook of Applied Cryptography.* CRC Press, 1997.

[Big85] N.L. Biggs. *Discrete Mathematics.* Oxford University Press, New York, 1985.

[Bih97] E. Biham. A Fast New DES Implementation in Software. In *Fourth International Workshop on Fast Software Encryption*, volume LNCS 1267, pages 260–272, Berlin, Germany, 1997. Springer-Verlag.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[DR98] J. Daemen and V. Rijmen. AES Proposal: Rijndael. In *First Advanced Encryption Standard (AES) Conference*, Ventura, California, USA, 1998.

[EYCP01] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on VLSI Design*, 9(4):545, 2001.

[Kah67] D. Kahn. *The Codebreakers. The Story of Secret Writing.* Macmilian, 1967.

[Kob94] N. Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, New York, second edition, 1994.

[LTG+02] A. K. Lutz, J. Treichler, F.K. Gurkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, , and W. Fichtner. 2Gbit/s Hardware Realiza-

tions of RIJNDAEL and SERPENT: A comparative analysis. In Çetin K. Koç Burt Kaliski and Christof Paar, editors, *Proceedings of the Fourth Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Berlin, Germany, August 2002. Springer-Verlag.

[Men93] A.J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, USA, 1997.

[OP00] Gerardo Orlando and Christof Paar. A High-Performance reconfigurable Elliptic Curve Processor for $GF(2^m)$. In Cetin K. Koc and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES'2000)*, pages 41–56, Berlin, 2000. Springer-Verlag. Lecture Notes in Computer Science Volume.

[Sch96] B. Schneier. *Applied Cryptography*. John Wiley & Sons Inc., New York, New York, USA, 2nd edition, 1996.

[Sim92] G.J. Simmons. *Contemporary Cryptology*. IEEE Press, 1992.

[Sta02] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, Upper Saddle River, New Jersey, USA, 3rd edition, 2002.

[Sti02] D. R. Stinson. *Cryptography, Theory and Practice*. CRC Press, 2nd edition, 2002.

[TPS00] S. Trimberger, R. Pang, and A. Singh. A 12 Gbps DES encryptor/decryptor core in an FPGA. In *Workshop on Cryptographic Hardware and Embedded Systems - CHES 2000*, volume LNCS 1965, Worcester, Massachusetts, USA, August 2000. Springer-Verlag.

[WPR+99] D. C. Wilcox, L. Pierson, P. Robertson, E. Witzke, and K. Gass. A DES ASIC Suitable for Network Encryption at 10 Gbps and Beyond. In Ç. Koç and C. Paar,

editors, *Workshop on Cryptographic Hardware and Embedded Systems - CHES '99*, volume LNCS 1717, pages 37–48, Worcester, Massachusetts, USA, August 1999. Springer-Verlag.