Cryptanalysis Handbook

George Washington University

Spring 2017 (Graduate)

G26980825

Xuenan Xu

Table of Contents

Chapter 1. Cryptomathematics

1.1 Modular Arithmetic

1.1.1 Definition

To simplify our definition, we consider all numbers are integers.

$$a \equiv b \ (mod\ m)$$

This equation is read as "***a*** is congruent to ***b*** mod ***m***". In a simple, but not wholly correct way, we can think of the equation to mean "***a*** is the remainder when ***b*** is divided by ***m***". For instance,

$$2 \equiv 12 \ (mod\ 10)$$

means that 2 is the remainder when 12 is divided by 10.

More formally, we can define:

$$a \equiv b \ (mod\ m) \rightarrow m \mid (a - b)$$

where "|" means "evenly divides".

We can also define

$$a \equiv b \ (mod\ m)$$

as

$$a = b + km$$

where ***k*** is some integer. We sometimes have a preference for ***k*** to be a positive or negative integer such that ***a*** is the least positive value satisfying the equation.

1.1.2 Addition

We can add (and therefore subtract) congruences, that is:

$$a \equiv b\ (mod\ m)\ \&\ c \equiv d\ (mod\ m) \rightarrow a \pm c = b \pm d\ (mod\ m)$$

So, if:

$$5 \equiv 15\ (mod\ 10)\ \&\ 7 \equiv 27\ (mod\ 10)$$

then:

$$5 + 7 \equiv 15 + 27 \equiv 2\ (mod\ 10)$$

$$5 - 7 \equiv 15 - 27 \equiv 8\ (mod\ 10)$$

1.1.3 Multiplication

We can multiply congruences (and therefore also divide, but with some restrictions), that is:

$$a \equiv b\ (mod\ m)\ \&\ c \equiv d\ (mod\ m) \rightarrow ac = bd\ (mod\ m)$$

For example:

$$5 \equiv 15\ (mod\ 10)\ \&\ 7 \equiv 27\ (mod\ 10) \rightarrow 5 \times 7 \equiv 15 \times 27 \equiv 5(mod\ 10)$$

To prove this:

$$a \equiv b\ (mod\ m) \rightarrow a = b + pm$$

$$c \equiv d\ (mod\ m) \rightarrow c = d + qm$$

$$\therefore ac = bd + (qb + pd + pq)m$$

$$\therefore ac = bd\ (mod\ m)$$

1.1.4 Multiplicative Inverses

While ***r*** and ***s*** are real numbers, we have:

$$r \times s = 1 \rightarrow r = \frac{1}{s}$$

and we say that ***r*** is the multiplicative inverse of ***s***.

In modular arithmetic, we do pretty much the same thing, subject to limitations on division. So, if:

$$a \times b \equiv 1 \ (mod\ m)$$

where ***a***, ***b*** and m are integers, then we say ***b*** is the multiplicative inverse of ***a*** (mod ***m***).

However, in modular arithmetic, ***b*** may or may not exist, that is, when ***a*** and ***m*** are relatively prime, then the reverse ***b*** uniquely exists, otherwise there is no solution.

For example, if we want to find the multiplicative inverse of 4 (mod 6), we can never get a solution because GCD(4, 6) is 2 which does not divide 1. But if we want to find the multiplicative inverse of 4 (mod 7), we can find something because:

$$4 \times 2 \equiv 1 \ (mod\ 7)$$

thus the multiplicative inverse of 4 (mod 7) is 2.

1.2 Groups

1.2.1 Definition

We define groups like:

$$(\mathbb{Z}_n, \omega)$$

where ***n*** is the number range and ***ω*** specifies the operator.

1.2.2 (Z_n, +)

Let ***v*** be the corresponding value of an element ***e*** in Z_n, we have:

$$e + v \equiv 1 \ (mod\ n)$$

Take Z_5 for example:

	1	2	3	4	5
1	2	3	4	0	1
2	3	4	0	1	2
3	4	0	1	2	3
4	0	1	2	3	4
5	1	2	3	4	0

1.2.3 (Z_n, *)

Let ***v*** be the corresponding value of an element ***e*** in Z_n, we have:

$$e \times v \equiv 1 \ (mod\ n)$$

Take Z_7 for example:

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

1.3 Euler-Phi/Euler-Totient Function

We use ***ϕ*** as the symbol of the function, and define ***ϕ(n)*** as follows:

$$\phi(n) = the\ number\ of\ integers\ from\ 1\ to\ n-1\ that\ is\ relatively\ prime\ to\ n$$

If ***n*** is prime, then:

$$\phi(\mathrm{n}) = n - 1$$

If ***n*** is ***p***-prime, that is, ***n = p^m***, then:

$$\phi(\mathrm{n}) = p^m - p^{m-1}$$

If ***n = pq*** where ***p***, ***q*** are prime, then:

$$\phi(\mathrm{n}) = (p-1)(q-1)$$

ϕ is a multiplicative function, that is:

$$\phi(1) = 1$$

$$\phi(\mathrm{mn}) = \phi(\mathrm{m})\phi(\mathrm{n})$$

$$\phi(n^k) = \phi^k(\mathrm{n})$$

1.4 Jacobi Symbols

1.4.1 Quadratic residue

Consider Z_p where ***p*** is an odd prime. Let ***a*** be a nonzero element of Z_p.

We say that ***a*** is a quadratic residue mod ***p*** if:

$$\exists\, y \in \mathbb{Z}_p, a \equiv y^2 \ (mod\ p)$$

and ***a*** is a quadratic non-residue mod ***p*** if:

$$\forall\, y \in \mathbb{Z}_p, a \not\equiv y^2 \ (mod\ p)$$

1.4.2 THM (Euler's Criterion)

Let ***p*** be an odd prime, then ***a*** is a quadratic residue mod ***p*** if and only if:

$$a^{\frac{p-1}{2}} \equiv 1 \ (mod\ p)$$

1.4.3 Legendre Symbol

We write Legendre Symbol as follows:

$$\mathcal{L}(a, p)$$

For integer ***a*** given odd prime ***p***:

$$\mathcal{L}(a, p) \equiv a^{\frac{p-1}{2}} (mod\ p)$$

The value of Legendre Symbol is either 0, 1 or -1:

$$a \equiv 0\ (mod\ p) \rightarrow \mathcal{L}(a, p) = 0$$

$$a\ is\ a\ quadratic\ residue\ mod\ p \rightarrow \mathcal{L}(a, p) = 1$$

$$a\ is\ a\ quadratic\ non-residue\ mod\ p \rightarrow \mathcal{L}(a, p) = 0$$

1.4.4 Jacobi Symbol

We write Jacobi Symbol as follows:

$$\mathcal{J}(a, n) = \binom{a}{n}$$

For integer ***a*** given odd integer ***n***:

$$n = {p_1}^{e_1} \times {p_2}^{e_2} \times \cdots \times {p_k}^{e_k}$$

where $\boldsymbol{p_1}$ through $\boldsymbol{p_k}$ are primes. We have:

$$\mathcal{J}(a, n) = \prod_{i=1}^{k} \binom{a}{p_i}^{e_i}$$

So if ***n*** is prime, we have:

$$\mathcal{L}(a, p) = \mathcal{J}(a, n)$$

According to previous definition, Jacobi Symbol have following properties:

If ***n*** is a positive odd integer:

$$m_1 = m_2 \ (mod\ n) \rightarrow \left(\frac{m_1}{n}\right) = \left(\frac{m_2}{n}\right)$$

$$\left(\frac{2}{n}\right) = \begin{cases} 1 \ if \ n \equiv \pm 1 \ (mod\ 8) \\ -1 \ if \ n \equiv \pm 3 \ (mod\ 8) \end{cases}$$

$$\left(\frac{m_1 m_2}{n}\right) = \left(\frac{m_1}{n}\right)\left(\frac{m_2}{n}\right)$$

Suppose ***m*** and ***n*** are positive odd integers, then:

$$\left(\frac{m}{n}\right) = \begin{cases} -\left(\frac{n}{m}\right) \ if \ m \equiv n \equiv 3 \ (mod\ 4) \\ \left(\frac{n}{m}\right) \ otherwise \end{cases}$$

For example:

$$\left(\frac{7411}{9283}\right) = -\left(\frac{9283}{7411}\right) = -\left(\frac{1872}{7411}\right) = -\left(\frac{2}{7411}\right)^4 \left(\frac{117}{7411}\right) = -\left(\frac{7411}{117}\right) = -\left(\frac{40}{117}\right) = -\left(\frac{2}{117}\right)^3 \left(\frac{5}{117}\right)$$

$$= \left(\frac{5}{117}\right) = \left(\frac{117}{5}\right) = \left(\frac{2}{5}\right) = -1$$

1.5 Primitive Field Elements

Consider the multiplicative group ***(Z_p, *)*** where ***p*** is prime. An element ***α*** in ***Z^*_p*** is said to be primitive if it satisfies the following condition:

$$\text{The } powers\ of\ \alpha\ generate\ all\ of\ the\ elements\ of\ \mathbb{Z}^*{}_p, and\ \alpha\ is\ called\ a\ generator$$

Note that a primitive polynomial is irreducible over the polynomial field, and its root α is a primitive element. All primitive polynomials are irreducible (cannot be factored in the field), but not all irreducible elements are primitive.

For example, in Z^*_5, we can check all powers of each element:

i	1	2	3	4
power(1, i)	1	1	1	1
power(2, i)	2	4	3	1
power(3, i)	3	4	2	1
power(4, i)	4	1	4	1

So the elements $\alpha = 2, 3$ in Z_5 are primitive but $\alpha = 1, 4$ are not since the powers of 1 and 4 do not generate all elements in Z_5.

Chapter 2. Manual Cryptosystems

2.1 Substitution Ciphers

Substitution cipher uses a 1-to-1 mapping where each letter is substituted for another. So substitution is secure against "brute force" attacks as there are 26! possible keys for each attack. The crack of substitution ciphers is computationally inexpensive, because the process does not involve complex computation which require a lot of resource and time. The ciphertexts are same length as the plaintexts, so they are vulnerable due to statistical structure of message (language structure), which means that by collecting samples of ciphertext, we can generate the key table by analyzing the statistical characteristic of the letters. The key storage is vulnerable too, if the key is acquired, we can crack the message at no cost.

2.1.1 Encrypt Process

For example, if we have a message "BITEME" and key table:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	C	E	G	I	K	M	O	Q	S	U	W	Y	B	D	F	H	J	L	N	P	R	T	V	X	Z

To compute the ciphertext, we look up each character in line 1 and replace it with the corresponding value in line 2:

$$B \to C, I \to Q, T \to N, E \to I, M \to Y, E \to I$$

So the ciphertext is "CQNIYI".

2.1.2 Decrypt Process

Suppose we got a message "VLG CJMAFVCJXVG VLUJR XIACV VLUD QAFTOP UD VLXV RAAP LXIUVD XFG DA OCYL GXDUGF VA RUZG CW VLXJ IXP AJGD."

First we generate the frequency table for the sentence as names perhaps contains strange patterns that will distract our decrypting process

A	C	D	F	G	I	J	L	M	O	P	Q	R	T	U	V	W	X	Y	Z
7	5	5	4	6	3	4	7	1	1	3	1	3	1	6	11	1	8	2	1

As discovered from the frequency table, ***V***, ***L***, ***A*** and ***X*** have very high frequency, then I find that the pattern ***VL*** appeared 5 times and all of them seems to be the start of a word, so I guess they are ***TH***. After filled them into the sentence, I believe that, according to the word ***VLG***, ***VLXV*** and ***VA***, ***G*** shall be ***E***, ***X*** shall be ***A*** and ***A*** shall be ***O*** to form ***THE***, ***THAT*** and ***TO***.

A	C	D	F	G	I	J	L	M	O	P	Q	R	T	U	V	W	X	Y	Z
O				E			H								T		A		

Then I noticed the word ***VLXJ*** which looks like ***THA**** on my paper, but the * cannot be ***T***, so after I tried with the rest 25 characters, I believe that ***J*** refers to ***N*** to form ***THAN***. There is another word ***XIACV*** which is like ***A*O*T***, and the first thing comes to my mind is ***ABOUT***.

A	C	D	F	G	I	J	L	M	O	P	Q	R	T	U	V	W	X	Y	Z
O	U			E	B	N	H								T		A		

At the middle of the sentence there is two words ***LXIUVD XFG*** which looks like ***HAB*T* A*E***. I think that the second word is ***ARE*** and according to that, the last character of the first word has a great probability of being ***S***, so the perfect match for these two words is ***HABITS ARE***.

A	C	D	F	G	I	J	L	M	O	P	Q	R	T	U	V	W	X	Y	Z
O	U	S	R	E	B	N	H							I	T		A		

According to the rear part of the sentence ***RAAP ..... AGJD***, we can see a compare between habits -- ****OO* HABITS ARE SO *U*H EASIER TO *I*E U* THAN BA* ONES***, which shows that ****OO****

and ***BA**** are opposite words, so I think they are ***GOOD*** and ***BAD***. By the way, the word between ***SO*** and ***EASIER*** can easily be filled out to be ***MUCH***.

A	C	D	F	G	I	J	L	M	O	P	Q	R	T	U	V	W	X	Y	Z
O	U	S	R	E	B	N	H		M	D		G		I	T		A	C	

The longest word ***CJMAFVCJXVG*** looks like ***UN*ORTUNATE*** now, I cannot figure out any other word than UNFORTUNATE. The two words ***RUZG CW*** seems like ***GI*E U**** and I'd like to fill them with ***GIVE UP***.

A	C	D	F	G	I	J	L	M	O	P	Q	R	T	U	V	W	X	Y	Z
O	U	S	R	E	B	N	H	F	M	D		G		I	T	P	A	C	V

The last word left is ***QAFTP*** and it looks like ****OR*D***, I don't know any other word except ***WORLD***, there are minor other possibilities but I think ***WORLD*** fits this sentence perfectly. Now the assignment table is complete.

A	C	D	F	G	I	J	L	M	O	P	Q	R	T	U	V	W	X	Y	Z
O	U	S	R	E	B	N	H	F	M	D	W	G	L	I	T	P	A	C	V

2.2 Shift Ciphers

Shift ciphers contain an integer key for offset, the calculation process can be described below:

$$c = (p + key)\ mod\ 26$$

Shift ciphers are easy to implement and computationally inexpensive, because they do not contain complex computations, and can be easily implemented in hardware level. These ciphers do not require key storage, as ciphertexts are corresponded tightly with plaintexts, so we can simply try all 25 possibilities ($0 \neq k \bmod 26$), and that's why shift ciphers are vulnerable to "brute force" attacks.

2.2.1 Encrypt Process

Suppose we want to encrypt ***"YES"*** with ***k = 3***:

$$Y \to Y + 3 \; mod \; 26 = B, E \to E + 3 \; mod \; 26 = H, S \; \to S + 3 \; mod \; 26 = V$$

So we get ***"BHV"*** as result.

2.2.2 Decrypt Process

Suppose we need to decrypt ***"BHV"*** with ***k = 3***:

$$B \to B - 3 \; mod \; 26 = Y, H \to H - 3 \; mod \; 26 = E, V \; \to V - 3 \; mod \; 26 = S$$

Then we get ***"YES"***.

2.3 Multiplicative Ciphers

Similar to Shift Ciphers, Multiplicative Ciphers contain an integer key for multiplication, but the key shall be relatively prime with 26 thus the process can produce 1-to-1 result. The calculation process can be described below:

$$c = p \times key \; (mod \; 26)$$

Multiplicative ciphers are computationally inexpensive because it only includes some multiplication process. These ciphers do not require key storage, as the number of keys is limited, so it's vulnerable to "brute force" attacks.

2.3.1 Encrypt Process

Suppose we want to encrypt ***"ABC"*** with ***k = 3***:

$$A \to A \times 3 \; mod \; 26 = A, B \to B \times 3 \; mod \; 26 = \; D, C \to C \times 3 \; mod \; 26 = G$$

So we get ***"ADG"*** as output.

2.3.2 Decrypt Process

The decrypt process involves the multiplicative inverse of the key:

$$p = c \times key^{-1} \ (mod\ 26)$$

Suppose we need to decrypt ***"ADG"*** with ***k = 3***, we first need to calculate the multiplicative inverse of ***3***, which is ***9***:

$$A \to A \times 9\ mod\ 26 = A, D \to D \times 9\ mod\ 26 = B, G \to G \times 9\ mod\ 26 = C$$

So we get ***"ABC"*** as result.

2.4 Affine Ciphers

Affine Ciphers are combinations of Multiplicative Ciphers and Shift Ciphers, the process can be described below:

$$c = p \times a + b \ (mod\ 26)$$

The key of affine ciphers is a pair of integer ***(a, b)***, they are computationally inexpensive. These ciphers are vulnerable to "brute force" attack, as the ciphers require ***GCD(a, 26) = 1*** to ensure a 1-to-1 mapping.

2.4.1 Encrypt Process

Suppose we want to encrypt ***"YES"*** with ***(3, 4)***:

$$Y \to Y \times 3 + 4\ mod\ 26 = Y$$

$$E \to E \times 3 + 4\ mod\ 26 = Q$$

$$S \to S \times 3 + 4\ mod\ 26 = G$$

Then we get ***"YQG"***.

2.4.2 Decrypt Process

The decrypt process involves the multiplicative inverse of the key:

$$p = c \times a^{-1} + b \ (mod\ 26)$$

Suppose we need to decrypt ***"YQG"*** with ***(3, 4)***, we need to calculate the multiplicative inverse of ***3***, which is ***9***:

$$Y \rightarrow (Y - 4) \times 9\ mod\ 26 = Y$$

$$Q \rightarrow\ (Q - 4) \times 9\ mod\ 26 = E$$

$$G \rightarrow (G - 4) \times 9\ mod\ 26 = S$$

The result is ***"YES"***.

2.5 Vigenere Ciphers

Vigenere Ciphers use repeated words/phrases to add to plaintexts groups to get ciphertexts. They are computationally feasible, and they are more secure with long keys since for $\boldsymbol{Z_m}$, the number of keys is $\boldsymbol{m^n}$ for length ***n***, and the repeats can be eliminated.

2.5.1 Encrypt Process

Assume we need to encrypt ***"HELLO"*** with ***key = "CAT"***:

$$H \rightarrow H + C\ mod\ 26 = J$$

$$E \rightarrow E + A\ mod\ 26 = E$$

$$L \rightarrow L + T\ mod\ 26 = E$$

$$L \rightarrow L + C\ mod\ 26 = N$$

$$O \rightarrow O + A\ mod\ 26 = O$$

The result is ***"JEENO"***.

2.5.2 Decrypt Process

The decrypt process is just the reverse of encrypt process.

Assume we want to decrypt ***"JEENO"*** with ***key = "CAT"***:

$$J \to J - C \ mod\ 26 = H$$

$$E \to E - A \ mod\ 26 = E$$

$$E \to E - T \ mod\ 26 = L$$

$$N \to N - C \ mod\ 26 = L$$

$$O \to O - A \ mod\ 26 = O$$

The plaintext is ***"HELLO"***.

Chapter 3. Other Manual Cryptosystems

3.1 Permutation Ciphers

Permutation Ciphers permute n characters at a time, using an n-long permutation key with the English alphabet. The ciphers and deciphers can be described below:

$$\mathcal{P} = \mathcal{C} = (\mathbb{Z}_{26})^n$$

$$\mathcal{K} = \{\pi \mid \pi \; is \; the \; permutation \; on \; n \; symbols\}$$

$$e_\pi(x_1, x_2, \cdots, x_n) = \left(x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(n)}\right) = (y_1, y_2, \cdots, y_n)$$

$$d_\pi(y_1, y_2, \cdots, y_n) = \left(y_{\pi^{-1}(1)}, y_{\pi^{-1}(2)}, \cdots, y_{\pi^{-1}(n)}\right) = (x_1, x_2, \cdots, x_n)$$

3.1.1 Encrypt Process

Suppose we want to encrypt ***x = "ABCD"*** with ***π = (2 3 1 4)***:

$$A \rightarrow x_{\pi(1)} = x_2 = B$$

$$B \rightarrow x_{\pi(2)} = x_3 = C$$

$$C \rightarrow x_{\pi(3)} = x_1 = A$$

$$D \rightarrow x_{\pi(4)} = x_4 = D$$

The result is ***y = "BCAD"***.

3.1.2 Decrypt Process

To decrypt a message generated by Permutation Ciphers, we need to get the decryption permutation $\boldsymbol{\pi^{-1}}$. We need to reverse the rows of the two-line representation of the permutation, then rearrange the columns in order so that the top line is ordered.

$$\begin{pmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix} \rightarrow (3\ 1\ 2\ 4)$$

Then we can permute ***y = "BCAD"*** with $\boldsymbol{\pi^{-1}}$:

$$B \rightarrow y_{\pi^{-1}(1)} = y_3 = A$$

$$C \rightarrow y_{\pi^{-1}(2)} = y_1 = B$$

$$A \rightarrow y_{\pi^{-1}(3)} = y_2 = C$$

$$D \rightarrow y_{\pi^{-1}(4)} = y_4 = D$$

The result is ***"ABCD"***.

3.2 Product Ciphers

Product Ciphers are used to enhance cryptosystem security and complexity, they are composition of two or more ciphers. Suppose we have two ciphers:

$$Permutation\ Cipher\ e_1 : \pi = (2\ 1\ 4\ 3)$$

$$Vigenere\ Cipher\ e_2 : key = "XYZW"$$

And we define Product Cipher as follows:

$$y = e_1(e_2(x))$$

3.2.1 Encrypt Process

Suppose we need to encrypt ***"ABCD"***.

According to the definition, we shall use the Vigenere Cipher first, then permute the result with Permutation Cipher.

$$A \rightarrow A + X\ mod\ 26 = X$$

$$B \to B + Y \ mod \ 26 = Z$$

$$C \to C + Z \ mod \ 26 = B$$

$$D \to D + W \ mod \ 26 = Z$$

Then we need to use Permutation cipher on the result ***"XZBZ"***.

$$X \to \ x_{\pi(1)} = \ x_2 = Z$$

$$Z \to \ x_{\pi(2)} = \ x_1 = X$$

$$B \to \ x_{\pi(3)} = \ x_4 = Z$$

$$Z \to \ x_{\pi(4)} = \ x_3 = B$$

The result is ***"ZXZB"***.

3.2.2 Decrypt Process

Suppose we want to decrypt ***y = "ZXZB"***.

The decrypt process is simply the reverse of the encrypt process, that is, permute ***y*** with $\boldsymbol{\pi^{-1}}$, then use Vigenere Cipher in reverse way.

We calculate $\boldsymbol{\pi^{-1}}$ first:

$$\begin{pmatrix} 2\ 1\ 4\ 3 \\ 1\ 2\ 3\ 4 \end{pmatrix} = \begin{pmatrix} 1\ 2\ 3\ 4 \\ 2\ 1\ 4\ 3 \end{pmatrix} \to \ (2\ 1\ 4\ 3)$$

Then permute ***y***:

$$Z \to \ y_{\pi(1)} = \ y_2 = X$$

$$X \to \ y_{\pi(2)} = \ y_1 = Z$$

$$Z \rightarrow y_{\pi(3)} = y_4 = B$$

$$B \rightarrow y_{\pi(4)} = y_3 = Z$$

Finally decrypt with the Vigenere Cipher:

$$X \rightarrow X - X \; mod \; 26 = A$$

$$Z \rightarrow Z - Y \; mod \; 26 = B$$

$$B \rightarrow B - Z \; mod \; 26 = C$$

$$Z \rightarrow Z - W \; mod \; 26 = D$$

Then we get ***x = "ABCD"***.

3.3 Autokey Ciphers

In Autokey Ciphers, plaintext is written as elements from $\boldsymbol{Z_{26}}$, and the initial key is chosen from $\boldsymbol{Z_{26}}$ as well.

3.3.1 Encrypt Process

To get first ciphertext character $\boldsymbol{C_1}$, add first plaintext character $\boldsymbol{P_1}$ to $\boldsymbol{K_1}$ so that:

$$C_1 = P_1 + K_1 \; mod \; 26$$

For successive ciphertext characters $\boldsymbol{C_i}$ with ***i > 1***, we compute:

$$C_i = P_i + K_i \; mod \; 26 \; where \; K_i = P_{i-1}$$

Suppose we are encrypting ***"ABCD"*** with initial ***key = 5***:

$$A \rightarrow A + key \; mod \; 26 = F$$

$$key = A = 0$$

$$B \rightarrow B + key \; mod \; 26 = B$$

$$key = B = 1$$

$$C \rightarrow C + key \; mod \; 26 = D$$

$$key = C = 2$$

$$D \rightarrow D + key \; mod \; 26 = F$$

We get ***"FBDF"*** as result.

3.3.2 Decrypt Process

To decrypt a message generated by Autokey Ciphers, we start with initial key K_1 and compute:

$$P_1 = C_1 - K_1 \; mod \; 26$$

For successive plaintext characters P_i with $i > 1$, we compute:

$$P_i = C_i - K_i \; mod \; 26 \; where \; K_i = P_{i-1}$$

Suppose we are decrypting ***"FBDF"*** with initial ***key = 5***:

$$F \rightarrow F - key \; mod \; 26 = A$$

$$key = A = 0$$

$$B \rightarrow B - key \; mod \; 26 = B$$

$$key = B = 1$$

$$D \rightarrow D - key \; mod \; 26 = C$$

$$key = C = 2$$

$$F \rightarrow F - key \; mod \; 26 = D$$

The result is ***"ABCD"***.

3.4 Stream Ciphers

3.4.1 Definition

Stream Ciphers produce bit streams that are xored to plaintext bits to produce ciphertext:

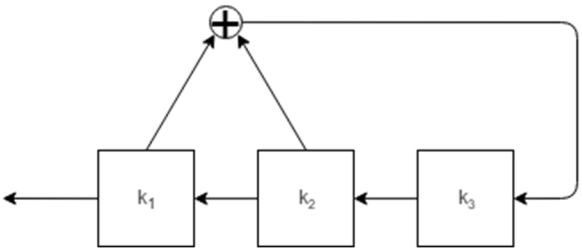
$$C_i = P_i + K_i \, mod \, 2$$

3.4.2 Linear Feedback Shift Register

LFSR has several stages, the terms $k_1, k_2, \ldots, k_n$ "initialize" the registers, and the nonzero constants in the vector $C_1, C_2, \ldots, C_n$ determine the "taps", that is:

$$\text{key} = (k_1, k_2, \cdots, k_m, c_0 k_1, c_1 k_2, \cdots, c_{m-1} k_m, \cdots)$$

The arrows and lines tell you which stages are "tapped" or added. In the sample diagram, the arrows tell us to: shift to the left to generate key, and generate the next keystream bit by xoring the first 2 bits in the register at any given time and "feed" the result back.



Chapter 4. Block Ciphers

4.1 Substitution Permutation Networks (SPN)

SPN involves three positive integers ***l***, ***m*** and N_r. The plaintext ***x*** is ***lm-bits*** long, and $N_r + 1$ subkeys are generated and used during the process. As included in its name, SPN uses two generic ciphers, Substitution and Permutation Ciphers, these ciphers use rules we call ***S-boxes*** and ***P-boxes*** to generate results.

SPN algorithm can be described as follows:

$$e_k(x) = \pi_{K_{N_r}} \circ \pi_S \circ \left(\pi_{K_{N_r-1}} \circ \pi_P \circ \pi_S\right) \circ \cdots \circ \left(\pi_{K_2} \circ \pi_P \circ \pi_S\right) \circ \left(\pi_{K_1} \circ \pi_P \circ \pi_S\right) \circ \pi_{K_0}$$

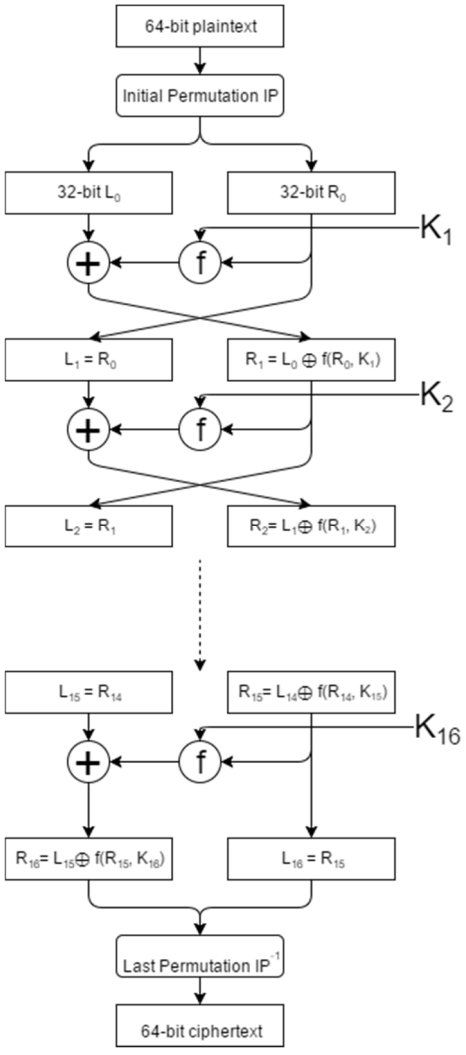
Where $\pi_{K_{N_r}}$ stands for subkey mixing, π_P stands for P-box permutation, and π_S stands for S-box substitution. We can describe algorithm for SPN as follows (u_i is the input to the i_{th} layer of S-boxes, v_i is the output, and w_i is the output of the i_{th} permutation layer):

1. Take ***lm-bits*** of plaintext ***x***, call it w_0, to work on at a time.
2. Divide plaintext into ***m*** groups of ***l*** bits and determine key schedule
3. Iterate over $N_r - 1$ rounds (for j = 1, …, N_r - 1):
   - $u_j = w_{j-1} + k_j$
   - $v_j = \pi_s(u_j)$
   - $w_j = \pi_p(v_j)$
4. Add subkey to value: $u_{N_r} = w_{N_r-1} + k_{N_r}$
5. Apply S-box substitution to output: $v_{N_r} = \pi_s(u_{N_r})$
6. Add last subkey to result to get ciphertext: $y = v_{N_r} + k_{N_r+1}$

4.2 Data Encryption Standard (DES)

DES is a kind of block ciphers in practice, which is developed by IBM and adopted by NIST in 1977, using 64-bit blocks and 56-bit keys. The input key for DES is 64-bit long, but subkeys used are only 56 bits in length. The least significant (right-most) bit in each bite is a parity bit or “check bit”, which is a bit added to the end of a string of binary code that indicates whether the number of bits in the string with the value 1 is even or odd. These parity bits are ignored so only the seven most significant bits of each byte are used, resulting in a key length of 56 bits. The problem is, with a small key space and the exploding development of compute hardware, DES is more and more feasible against exhaustive search attack since late 1990s.

We drew a flow chart to get a better understanding of DES encryption process:



It's vividly shown in the diagram that the encryption process contains mainly 4 parts:

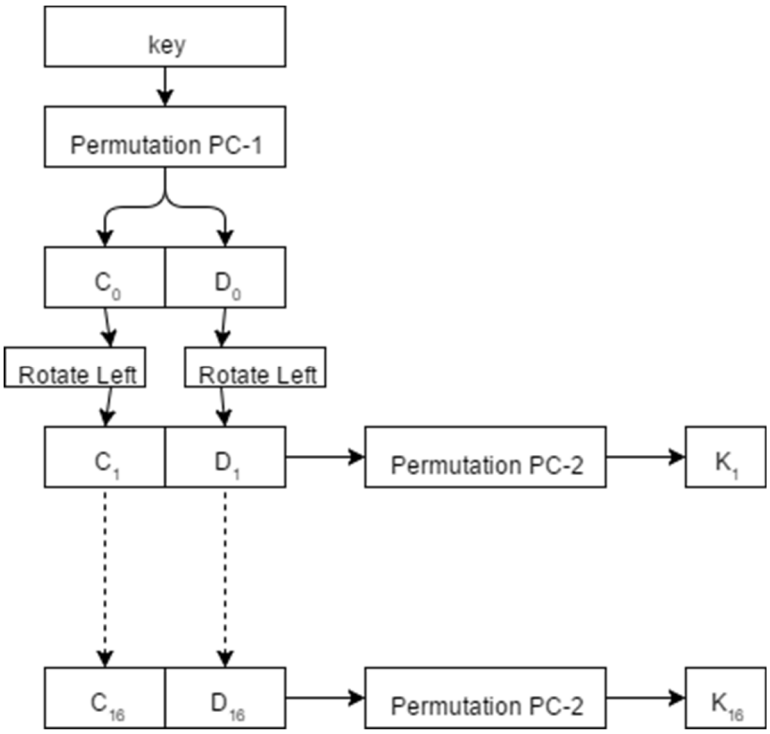
1. Initial Permutation IP

   Permute 64-bit plaintext with 8 * 8 permutation chart ***IP*** and get a 64-bit output.

2. Acquire subkeys K_i

   - Use permutation chart ***PC-1*** to turn 64-bit key (from user) into 56-bit key
   - Split 56-bit key into 2 28-bit strings C_0, D_0, rotate left to get C_1, D_1
   - Use ***PC-2*** to permute new roundkey C_1D_1, to get a 48-bit subkey K_1
   - Rotate C_1, D_1 to the left, and repeat the process. There are 16 rounds in total and we can get 16 48-bit subkeys (the rotate bits is 1 on round 1, 2, 9 and 16, otherwise 2)

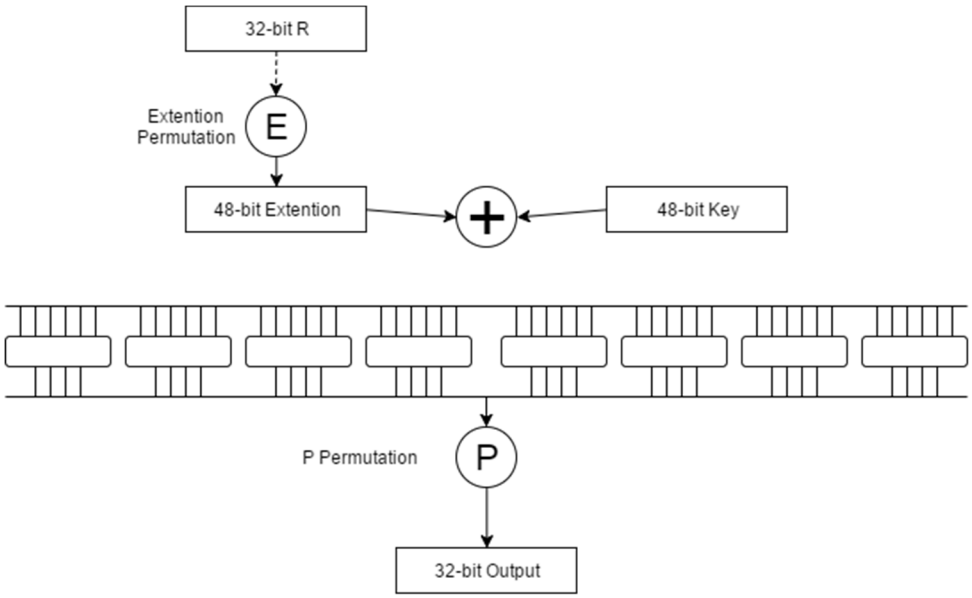
   We can draw a flow chart to demonstrate this process:



3. Encrypt function ***f***

   - The inputs are 32-bit data and 48-bit subkey

- Permute the 32-bit data into 48-bit with ***E***, and XOR it with subkey
- Split the result into 8 6-bit blocks, for each 6-bit block, generate a 4-bit output with corresponding ***S-box***: take the first and last bit of the 6-bit block and put them together to get a 2-bit binary number ***x***, and use the rest 4 digits of the 6-bit block to get a 4-bit binary number ***y***, then return the binary format of the number of ***xth*** row ***yth*** column in ***S-box***
- Put the 8 4-bit results together into a 32-bit data, and then permute it with ***P***, and the result is still 32-bit
- The following flow chart demonstrate the whole function ***f***:



4. Final Permutation IP^{-1}

   Put R_{16} and L_{16} together, and permute the 64-bit result with IP^{-1}, we can get the final ciphertext as result

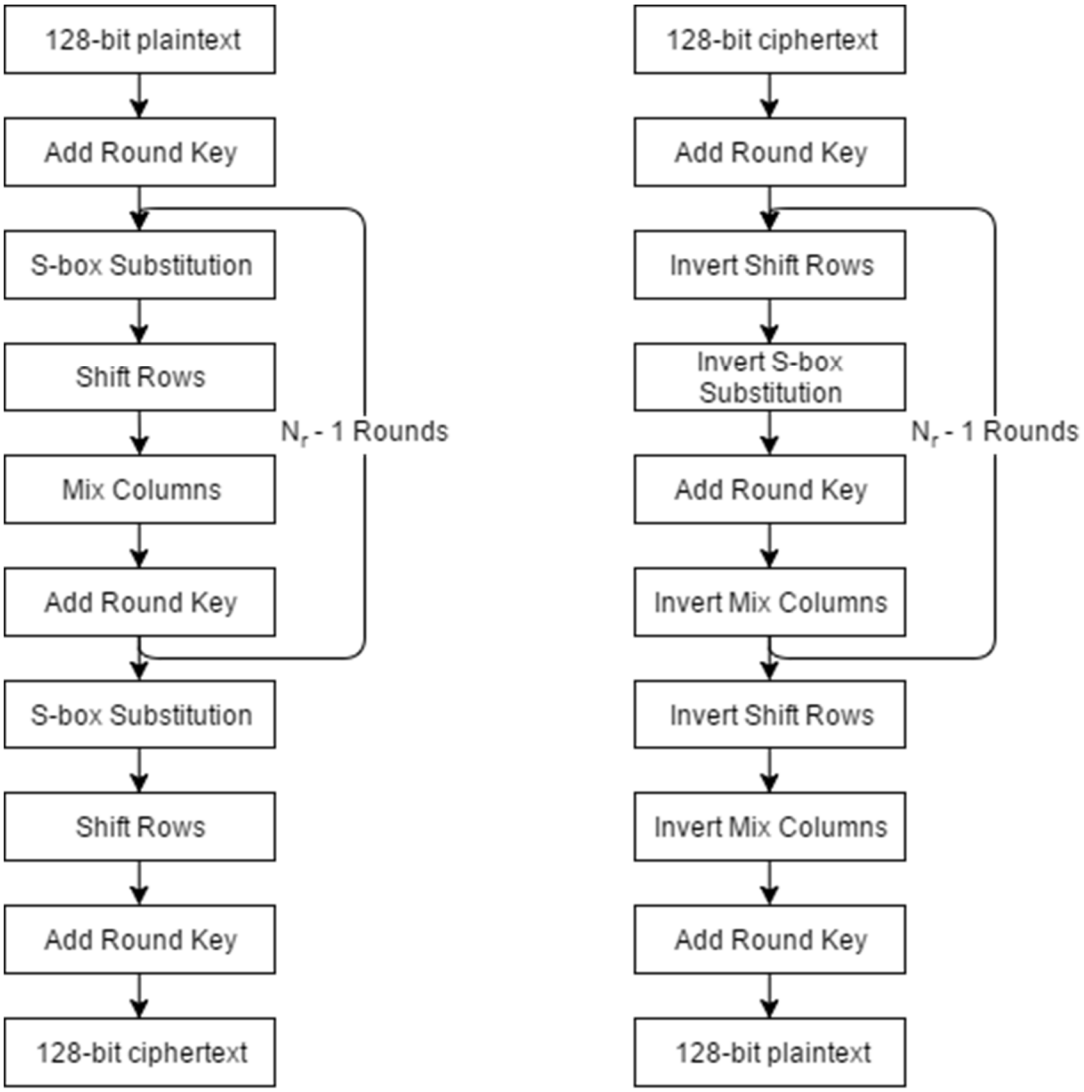
5. The data used in DES (including ***IP***, ***PC-1***, ***PC-2***, ***E***, ***S-box***, ***P*** and IP^{-1}) is public online so will not be included here.

6. The decryption process of DES is simply the same algorithm with reversed key order.

4.3 Advanced Encryption Standard (AES)

AES is selected by NIST in 2001 through open international competition and public discussion. It takes 128-bit data blocks and several possible key lengths including 128-bit, 192-bit and 256-bit. The algorithm consists of XOR with key, S-box substitution, permutation and column mixing, it's a very complex process with long key length so that exhaustive search attack is not currently possible. AES-256 is currently the most popular symmetric encryption algorithm.

To demonstrate the entire algorithm, we drew a flow chart for its encryption and decryption process:



The algorithm uses 3 symbols: N_b, the number of columns (a 32-bit word) of a state; N_k, the number of 32-bit words a key contains; and N_r, the round number, which differs among different key lengths.

	Nk	Nb	Nr
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

AES includes 3 major parts, Key Expansion, Encryption and Decryption.

4.3.1 Key Expansion

AES uses Key Expansion to extend user's key to $N_b(N_r + 1)$ words:

- Rotation: take a word $[a_0, a_1, a_2, a_3]$ as input, rotate left for 1 bit and return $[a1, a2, a3, a0]$
- S-box Substitution: take a word $[a0, a1, a2, a3]$ as input, for each a_i, take the first 4 bit as the row number x, and take the last 4 bit as the column number y, and query the S-box for the value at row x column y. Put all 4 numbers together as the result
- There are some constants called "Round Constants" used in this process
- The first N_k elements of the expansion is the user's key, and after that, every element equals to previous elements XOR previous N_k elements: $key[i] = key[i-1] \oplus key[i - N_k]$; but if N_k divides i, then:

$$key[i] = key[i - N_k] \oplus substitute(rotate\big(key[i-1]\big) \oplus roundConstant(\frac{i}{N_k} - 1)$$

4.3.2 Encryption

- S-box Substitution: same thing as in Key Expansion
- Shift Rows: take a 4*4 matrix as input, and rotate the second line 1 bit to the left, the third line 2 bits to the left and last line 3 bits to the left

A1	A2	A3	A4	→	A1	A2	A3	A4
B1	B2	B3	B4		B2	B3	B4	B1
C1	C2	C3	C4		C3	C4	C1	C2
D1	D2	D3	D4		D4	D1	D2	D3

- Mix Columns: take a 4*4 matrix as input and transform the matrix's columns with following rules (● is multiplication in Galois field):

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}.$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

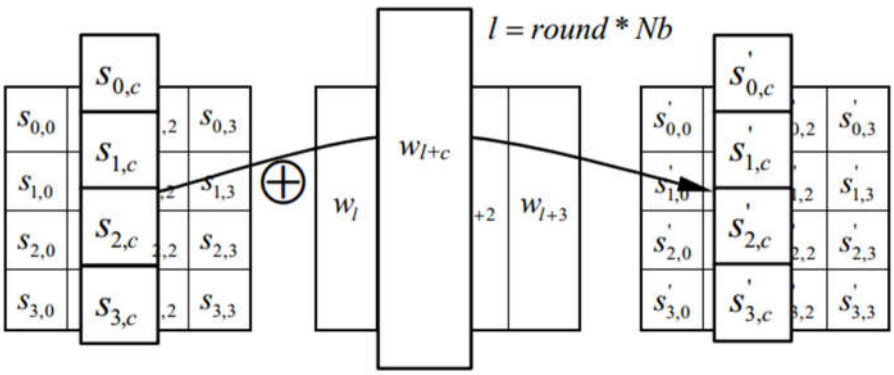
$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

- Add Round Key: in this transformation, a Round Key is added to the word by a simple bitwise XOR operation



4.3.3 Decryption

The decryption process is the reverse operation of every encryption stages.

- Invert Shift Rows: take a 4*4 matrix as input, and rotate the second line 1 bit to the right, the third line 2 bits to the right and last line 3 bits to the right

A1	A2	A3	A4	→	A1	A2	A3	A4
B1	B2	B3	B4		B4	B1	B2	B3
C1	C2	C3	C4		C3	C4	C1	C2
D1	D2	D3	D4		D2	D3	D4	D1

- Invert S-box Substitution: same thing as S-box Substitution in encryption process, with another substitution chart (S-box^{-1})
- Invert Mix Columns: same equation as in encryption, but the coefficient has changed

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}.$$

Now the result looks like

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

- Add Round Key: exactly the same process as in encryption, since it only involves XOR operation and XOR twice will get the original value

Chapter 5. Hash Functions and Public Key Cryptography

5.1 Hash Functions

5.1.1 Concept

Hash functions is to transform input (also called pre-image) of any length into fixed-length outputs. This transformation is a kind of data compression, that is, typically, the output domain is far smaller than input domain, so that different input may produce same outputs. To be short, hash functions is the functions that can compress messages of any length to a fixed-length message digest.

5.1.2 Hash family

We define hash families as follows:

$$(X, Y, K, H)$$

- ***X*** is a set of possible messages or "pre-images"
- ***Y*** is a finite set of possible hashes/message digests
- If ***X*** is finite, we assume that ***|X| ≥ |Y|***
- ***K*** is the keyspace consisting of a finite set of possible keys
- For each ***k*** in ***K***, there is a hash functionin in ***H*** with: $h_k : X \to Y$
- If ***|X| = N*** and ***|Y| = M***, then a hash family define from ***X*** to ***Y*** is called an ***(N, M) – hash family***
- For an unkeyed hash function, ***|K| = 1*** so there is only one key that is used

5.1.3 Security features

There are three kinds of security features we must consider when we evaluate the security of a hash function:

- Pre-image Resistant: given ***y*** in ***Y***, it's impossible to find ***x*** in ***X*** such that ***h(x) = y***

- Second Pre-image Resistant: given ***x*** in ***X***, it's impossible to find a different ***x'*** in ***X*** such ***h(x') = h(x)***
- Collision Free: There is no two distinct ***x*** and ***x'*** in X such ***h(x') = h(x)***

5.1.4 Random Oracle Model

An ideal hash function acts like an oracle:

- The algorithm is pre-image resistant, second pre-image resistant and collision free.
- The only efficient way to find the hash of ***x*** is to actually compute it or "query the oracle" so the algorithm appears "random"
- Knowing one hash value does not increase the ability to find others

5.2 SHA-1

SHA-1 is a iterated hash function which generates 160-bit message digests with inputs of no more than 2^{64}-1 bits. SHA-1 algorithm is word-oriented with bitwise computations, using a set of 89 functions f_i, a set of 80 word constants k_i and a set of 5 initialization constants h_i.

5.2.1 Preprocessing (Padding Input)

- Let X = input

$$X = 0\text{x}ab56f11075$$

- Write L = |X| as 64-bit value with 0-padding on the left

$$\text{L} = 0^{58}101000$$

- Compute amount of additional 0-padding by $d = (447 - |X|)\ mod\ 512$

$$d = 407$$

- Construct padded input $Y = X \,||\, 1 \,||\, 0^d \,||\, L$

$$\text{Y} = 1010\ 1011\ 0101\ 0110\ 1111\ 0001\ 0001\ 0000\ 0111\ 0101\ 1\ 0^{465}\ 10\ 1000$$

- Y is now a multiple of 512bits

5.2.2 Processing

- Assume we have M_i as input, we write it in:

$$M_i = W_0 \,||\, W_1 \,||\, \dots \,||\, W_{15}$$

So we get 16 words. We now want to get words W_{16} through W_{79}

- For 16 ≤ t ≤ 79:

$$W_t = ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16})$$

- Set:

$$A = h_0, B = h_1, \mathrm{C} = h_2, D = h_3, \mathrm{E} = h_4$$

- For 0 ≤ t ≤ 79:

$$temp = ROTL^5(A) + f_t(B, C, D) + E + W_t + K_t$$

$$E = D; D = C; C = ROTL^{30}(B); B = A; A = temp$$

- Reset:

$$H_0 += A;\ H_1 += B;\ H_2 += C;\ H_3 += D;\ H_4 += E$$

- The hash of Mi is:

$$H_0 \,||\, H_1 \,||\, H_2 \,||\, H_3 \,||\, H_4$$

5.3 RSA

5.3.1 Introduction of Public Key Cryptography

In 1976, two American scientist Whitfield Diffie and Martin Hellman, proposed a brand-new thought, which can decrypt a message without passing the “key” directly, called “Diffie-Hellman Key Exchange Algorithm”. People found out that, encryption and decryption can base on different rules, as long as there is

some correspondence between these rules. This new crypto model is called “asymmetric crypto algorithm”, which eliminates the direct transportation of keys.

- B generates 2 keys, one is public, the other one is private. The public key is open-accessed, which means anybody has access to it.
- A acquire B’s public key, and encrypt the message with it, and send the ciphertext to B
- After receiving the message, B decrypt it with private key, and get the plaintext
- If the message encrypted by the public key can only be decrypted with the corresponding private key, then the communication is secure as long as the private key is secure

In 1977, 3 mathematician Rivest, Shamir and Adleman designed an algorithm, which implemented asymmetric encryption and decryption, named “RSA”. Since then, RSA has been the most popular asymmetric cryptosystem, wherever network exists, RSA exists.

According to published papers, the longest cracked RSA key is of 768-bit long, so we believe that the RSA key of length larger than 768 is secure, and the longer the length, the harder to crack.

5.3.2 Fast Modular Exponentiation to Compute $B^N \ mod \ M$

- Write ***N*** in binary, then as a sum of powers of 2
- Compute square powers of ***B*** by successively squaring and reducing mod ***M***

$$B^2 \ mod \ M = (B \ mod \ M)^2 \ mod \ M, etc.$$

- Rewrite B^N as the product of the powers corresponding to binary ***N***

$$B^{13} = \ B^8 \times B^4 \times B, etc.$$

- Multiply and reduce mod ***M*** as necessary until arrived at last answer

$$XY \ mod \ M = (X \ mod \ M)(Y \ mod \ M) \ mod \ M$$

5.3.3 Multiplicative Inverse Algorithm to Compute $B^{-1}\ mod\ A$

- Set:

$$a_0 = A, b_0 = B, t_0 = 0, t = 1, q = \ floor\left(\frac{a_0}{b_0}\right), r = \ a_0 - qb_0$$

- While ***r > 0***:

$$temp = (t_0 - \ qt)\ mod\ A, t_0 = t, t = temp$$

$$a_0 = \ b_0, b_0 = r, q = floor\left(\frac{a_0}{b_0}\right), r = \ a_0 - qb_0$$

- If $b_0 \neq 1$, B has no inverse. Otherwise, $B^{-1}\ mod\ A = t$

5.3.4 Encryption

- Write plaintext alphabetic characters using numeric representation (2 digits, 00 ~ 25), let's use ***WXYZ = 2223 2425*** as example
- Let block size be twice the length of numeric representations (size of 4)
- Choose $n = p \times q$ where ***p***, ***q*** are LARGE primes (we take ***n = 2537***)

$$n = 2537 = 43 \times 59$$

- Calculate Euler-Phi of ***n***:

$$\phi(n) = (43^1 - 43^0)(59^1 - 59^0) = 2436$$

- Choose encryption exponent b so that $GCD(\phi(n), b) = 1$, let take ***b = 13***
- To encrypt plaintext ***P***, compute $C = P^b\ mod\ n$ for each block

$$C(2223) = 2223^{13}\ mod\ 2537 = 1359$$

$$C(2425) = 2425^{13}\ mod\ 2537 = 1604$$

- The result is ***1359 1604***

5.3.5 Decryption

- Let's use the components and result of previous encryption process for example

- The private key of previous encryption process is:

$$a = b^{-1} \, mod \, \phi(n) = 937$$

- To decrypt ciphertext C, Compute $P = C^a \, mod \, n$ for each block

$$P(1359) = 1359^{937} \, mod \, 2537 = 2223$$

$$P(1604) = 1604^{937} \, mod \, 2537 = 2425$$

5.4 El Gamal

El Gamal Algorithm is an asymmetric cryptosystem based on Diffie-Hellman Key Exchange Algorithm, proposed by Taher Elgamal in 1985. It can be used on data encryption as well as data signature, and its security depends on the problem of computing the discrete logarithm on finite fields.

The private key of El Gamal Algorithm is a chosen number ***a***, and the public key consists of 3 elements: a large prime ***p***, a primitive element ***α*** in $\boldsymbol{Z^*_p}$, and $\beta = \alpha^a \, mod \, p$.

5.4.1 Encryption

- We are using ***p = 23, α = 6, a = 6*** and plaintext ***x = 10*** for demonstration
- Compute $\beta = \alpha^a \, mod \, p = 6^6 \, mod \, 23 = 12$
- Choose random ***k*** that is relatively prime to p (say 3)
- Compute ciphertext ***(y_1, y_2)*** where:

$$y_1 = \alpha^k \, mod \, p = 6^3 \, mod \, 23 = 9$$

$$y_2 = x\beta^k \, mod \, p = 10 \times 12^3 \, mod \, 23 = 7$$

- The result is ***(9, 7)***

5.4.2 Decryption

$$x = y_2({y_1}^a)^{-1} \, mod \, p = 7 \times (9^6)^{-1} \, mod \, 23 = 7 \times 3^{-1} \, mod \, 23 = 7 \times 8 \, mod \, 23 = 10$$

Appendix: Source Code for Selected Algorithms

a. Vigenere Ciphers

```
String vigenereCipher(String text, String key) {
    StringBuilder sb = new StringBuilder();
    int keyLength = key.length();
    int[] equivalent = new int[keyLength];
    for (int i = 0; i < key.length(); i++)
        equivalent[i] = key.charAt(i) - 'A';
    for (int i = 0; i < text.length(); i++)
        sb.append((char) ((text.charAt(i) - 'A' + equivalent[i % keyLength]) % 2
6 + 'A'));
    return sb.toString();
}
```

b. Autokey Ciphers

```
String autokeyCipher(String plainText, int key) {
    char[] p = plainText.toCharArray();
    StringBuilder sb = new StringBuilder();
    for (char c : p) {
        sb.append((char) ((c - 'A' + key) % 26 + 'A'));
        key = c - 'A';
    }
    return sb.toString();
}
```

c. Data Encryption Standard

```
public class DataEncryptionStandard {

    private static final int[] SAMPLE_PLAIN_TEXT = {

            0, 0, 1, 1, 0, 1, 1, 1,

            1, 1, 1, 1, 1, 1, 1, 0,

            1, 1, 0, 0, 1, 0, 0, 1,

            0, 0, 1, 1, 0, 1, 1, 1,

            1, 1, 1, 0, 1, 1, 0, 0,

            1, 0, 0, 0, 0, 0, 0, 1,

            0, 1, 1, 1, 1, 1, 1, 0,

            0, 0, 0, 0, 1, 1, 1, 1
```

```
};
private static final int[] SAMPLE_KEY = {
        1, 0, 0, 0, 1, 1, 1, 0,
        0, 1, 1, 1, 0, 0, 0, 1,
        1, 1, 0, 0, 1, 1, 1, 1,
        0, 0, 1, 1, 1, 0, 0, 1,
        1, 1, 1, 0, 0, 1, 1, 1,
        0, 0, 1, 1, 1, 1, 0, 0,
        1, 0, 0, 1, 1, 1, 1, 0,
        1, 1, 1, 1, 0, 0, 0, 0
};
private static final int[] 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
};
private static final int IP_1[] = {
        40, 8, 48, 16, 56, 24, 64, 32,
        39, 7, 47, 15, 55, 23, 63, 31,
        38, 6, 46, 14, 54, 22, 62, 30,
        37, 5, 45, 13, 53, 21, 61, 29,
```

```
            36, 4, 44, 12, 52, 20, 60, 28,
            35, 3, 43, 11, 51, 19, 59, 27,
            34, 2, 42, 10, 50, 18, 58, 26,
            33, 1, 41, 9, 49, 17, 57, 25
    };
    private static final int[] PC_1 = {
            57, 49, 41, 33, 25, 17, 9,
            1, 58, 50, 42, 34, 26, 18,
            10, 2, 59, 51, 43, 35, 27,
            19, 11, 3, 60, 52, 44, 36,
            63, 55, 47, 39, 31, 23, 15,
            7, 62, 54, 46, 38, 30, 22,
            14, 6, 61, 53, 45, 37, 29,
            21, 13, 5, 28, 20, 12, 4
    };
    private static final int[] PC_2 = {
            14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32
    };
```

```
private static final int[] EXTEND = {
        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
};
private static final int[][][] S_BOX = {
        {
                {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}
        },
        {
                {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10},
                {3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5},
                {0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15},
                {13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9}
        },
        {
                {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8},
                {13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1},
```

```
        {13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7},
        {1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12}
    },
    {
        {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15},
        {13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9},
        {10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4},
        {3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14}
    },
    {
        {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9},
        {14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6},
        {4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14},
        {11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3}
    },
    {
        {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11},
        {10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8},
        {9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6},
        {4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13}
    },
    {
        {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1},
        {13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6},
        {1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2},
        {6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12}
```

```
        },
        {
                {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7},
                {1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2},
                {7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8},
                {2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11}
        }
};
private static final int P[] = {
        16, 7, 20, 21, 29, 12, 28, 17,
        1, 15, 23, 26, 5, 18, 31, 10,
        2, 8, 24, 14, 32, 27, 3, 9,
        19, 13, 30, 6, 22, 11, 4, 25
};
private static int[] permutateText(int[] source, int[] table, int length) {
    int[] permutationResult = new int[length];
    for (int i = 0; i < length; i++) {
        permutationResult[i] = source[table[i] - 1];
    }
    return permutationResult;
}
private static int[] exclusiveOr(int[] array_1, int[] array_2, int length) {
    int[] result = new int[length];
    for (int i = 0; i < length; i++)
        result[i] = (array_1[i] + array_2[i]) % 2;
    return result;
```

```
    }
    private static int[] keyShift(int[] key, int offset) {
        int[] newKey = new int[56];
        for (int i = 0; i < 28 - offset; i++) {
            newKey[i] = key[i + offset];
            newKey[28 + i] = key[28 + i + offset];
        }
        for (int i = 28 - offset; i < 28; i++) {
            newKey[i] = key[i + offset - 28];
            newKey[28 + i] = key[i + offset];
        }
        return newKey;
    }
    private static int[][] getKeys(int[] K0) {
        int[][] keySet = new int[16][48];
        int[] tempKey = new int[56];
        for (int i = 0; i < 16; i++) {
            tempKey = keyShift(i == 0 ? K0 : tempKey, (i == 0 || i == 1 || i ==
8 || i == 15) ? 1 : 2);
            int[] newKey = permutateText(tempKey, PC_2, PC_2.length);
            System.arraycopy(newKey, 0, keySet[i], 0, 48);
        }
        return keySet;
    }
    private static int[] f(int[] R, int[] Ki) {
        int[] extendedR = permutateText(R, EXTEND, EXTEND.length);
```

```
        int[] exclusiveOrResult = exclusiveOr(extendedR, Ki, extendedR.length);
        int[] permutationResult = new int[32];
        int temp;
        for (int i = 0; i < 8; i++) {
            int row = exclusiveOrResult[i * 6] * 2 + exclusiveOrResult[i * 6 +
5];
            int column = exclusiveOrResult[i * 6 + 1] * 8 +
                    exclusiveOrResult[i * 6 + 2] * 4 +
                    exclusiveOrResult[i * 6 + 3] * 2 +
                    exclusiveOrResult[i * 6 + 4];
            temp = S_BOX[i][row][column];
            for (int j = 0; j < 4; j++) {
                permutationResult[4 * (i + 1) - j - 1] = temp % 2;
                temp /= 2;
            }
        }
        permutationResult = permutateText(permutationResult, P,
permutationResult.length);
        return permutationResult;
    }
    public static void main(String[] args) {
        int[] plainTextPermutationResult = permutateText(SAMPLE_PLAIN_TEXT, IP,
IP.length);
        int[] keyPermutationResult = permutateText(SAMPLE_KEY, PC_1,
PC_1.length);
        int[][] keys = getKeys(keyPermutationResult);
```

```
        int[] L = new int[32];
        int[] R = new int[32];
        int[] temp;
        System.arraycopy(plainTextPermutationResult, 0, L, 0, 32);
        System.arraycopy(plainTextPermutationResult, 32, R, 0, 32);
        for (int i = 0; i < 16; i++) {
            temp = exclusiveOr(L, f(R, keys[i]), L.length);
            System.arraycopy(R, 0, L, 0, 32);
            System.arraycopy(temp, 0, R, 0, 32);
        }
        int[] cipheredText = new int[64];
        System.arraycopy(R, 0, cipheredText, 0, 32);
        System.arraycopy(L, 0, cipheredText, 32, 32);
        cipheredText = permutateText(cipheredText, IP_1, IP_1.length);
        for (int element : cipheredText)
            System.out.print(element);
    }
}
```