4-24-2025

Cryptology with Bitcoin and Blockchain Applications

Seth D. Bergmann
Rowan University, bergmann@rowan.edu

Cryptology

with Bitcoin and Blockchain Applications

Seth D. Bergmann

April 24, 2025

Preface

This book is intended to be used for a first course in cryptography for computer science students. It assumes that the student has had at least one programming course and a discrete structures or discrete math course. This book places an emphasis on algorithms and the internals of cryptographic systems.

The book begins with some classical cryptographic algorithms used for confidentiality. Then it exposes modern cryptographic algorithms for confidentiality, integrity, and authenticity. Both private key (symmetric) and public key (asymmetric) algorithms are covered. This book also describes the workings of the most common cryptocurrency, Bitcoin, as well as blockchain technology.

Most sections conclude with a list of exercises (solutions are available to instructors who have adopted this text in a course). Some exercises refer to a *repository* of data and/or computer software. This repository can be found at: `http://cs.rowan.edu/~bergmann/books/cryptology`

Chapter 4 consists of several topics from elementary discrete mathematics which are necessary to understand cryptographic algorithms. The more advanced reader may skip this chapter and refer to it as needed. Each subsequent chapter which requires an understanding of these topics will list the relevant sections at the beginning of the chapter.

This book is an open source book. This means that not only is the pdf version available (to potential students and teachers) for free download, but that the original (LaTeX) source files are also available (to potential authors and contributors). Based on the model of open source software, open source for textbooks is a relatively new paradigm in which many authors and contributors can cooperate to produce a high quality product, for no compensation. For details on the rationale of this new paradigm, and citations for other open source textbooks, see the journal *Publishing Research Quarterly*, Vol. 30, No. 1, March 2014. The source materials and pdf files of this book are licensed with the Creative Commons NonCommercial license, which means that they may be freely used, copied, or modified, but not for financial gain.

This book is available at `rdw.rowan.edu` (search for Bergmann). The source files for this book are available at `cs.rowan.edu/~bergmann/books`

Secondary Authors

Contributors

Technical Consultant

Joshua Grochowski, Rowan University

Contents

Chapter 1

Introduction and Terminology

For centuries there have been many times when the need for *confidential* communication has been critical. Certainly in time of war commanding officers need to communicate with their troops in such a way that the enemy is not privy to the content of the communication. There can be many other areas in which confidentiality is important, including government, business, health care, and even personal communication.

In addition to confidentiality, we may also be interested in ensuring *integrity* and *authenticity*. Communication integrity is the process of ensuring that a communication is received in an unaltered state; i.e. integrity ensures that no one has tampered with the communication. Communication authenticity ensures that the sender of the communication is really who they claim to be, and not an impostor.

There are two aspects to cryptology: cryptography and cryptanalysis. *cryptography* involves ensuring confidentiality, integrity, and/or authenticity in our communication with friends. In many situations we have a need to intercept our enemies' communications, for our own protection. This is known as *cryptanalysis*. Our government and military are the primary users of cryptanalyis to ensure our national security.

Using popular language we might say that cryptography is code-making, and cryptanalysis is code-breaking.

1.1 Terminology for cryptography

The terminology described here is used throughout this text. We organize these terms within the three areas of rationale for cryptography described above.

- A diagram representing confidentiality is shown in Figure 1.1. Terminology related to confidentiality is shown below:

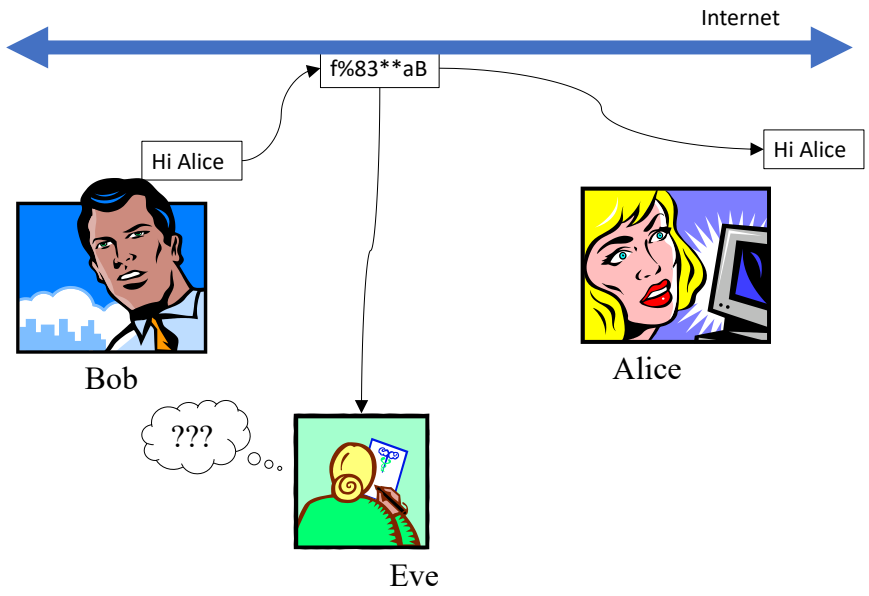


Figure 1.1: Bob wishes to send a confidential message to Alice, so he encrypts the message. Alice decrypts the message. (Evil) Eve intercepts the encrypted message but cannot decrypt it.

  - *Encryption* is the process of altering a communication message in such a way that only the intended recipients can use it to restore the original message.
  - *Decryption* is the process of retrieving the original message from the encrypted version (this is typically done by the intended recipient(s)).
  - *Plaintext* is the communication, or message, which is not encrypted. In modern parlance we use the phrase plain *text* even when the message is not strictly text, but could be any digital information, such as images, sound clips, video clips, and even binary keys.[1]
  - *Ciphertext* is the encrypted form of the communication. It is presumably a garbled version of the plaintext, and is unintelligible to anyone other than the intended recipient(s). As with plaintext, the ciphertext need not be text in the normal sense, but can be binary data with no textual interpretation.
  - A *key* is information which is needed to encrypt plaintext and/or decrypt ciphertext.
  - A *cipher* is essentially an algorithm which precisely specifies the encryption and decryption processes.

- A diagram representing a violation of integrity is shown in Figure 1.2. Bob sends a message, unencrypted, to Alice, but Eve intercepts the message, changes it and forwards the modified message to Alice. Alice thinks she received the message directly from Bob. Terminology related to integrity is shown below:

  - *Tampering* is the unauthorized modification of a communication message, generally done for illicit purposes. Tampering is usually done in such a way that the intended recipient of the message is unaware that it has been changed.
  - A message *hash* also known as a *message digest* or *fingerprint* is a fixed size sequence of bits which in some way is (almost perfectly) unique to that message, and can be used to identify the message. The original message could be just a few bytes, or it could be gigabytes in size, but the hash is a fixed size, typically a few hundred bits. The hash can be used to detect violations of integrity.

- A diagram representing a violation of authenticity is shown in Figure 1.3. Eve composes a message to Alice and makes it appear to be coming from Bob. Alice does not realize that the message is really coming from Eve, and Bob has no knowledge that any of this is happening. Terminology related to authenticity is shown below:

[1] Confusion often arises from this usage of *plaintext*. It is often conflated with *plain text*, as in 'My data is in a plain text file', i.e. a file created with Notepad or TextEdit which contains no markups. Here a plaintext is not necessarily plain text.

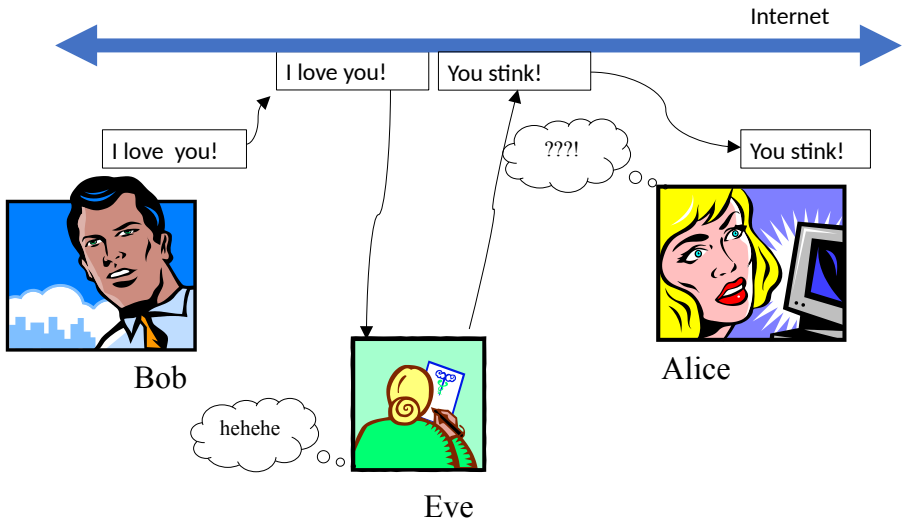


Figure 1.2: A violaton of integrity: Bob sends a message to Alice, but Eve intercepts the message, tampers with it, and sends it on to Alice. Alice thinks the message came directly from Bob.

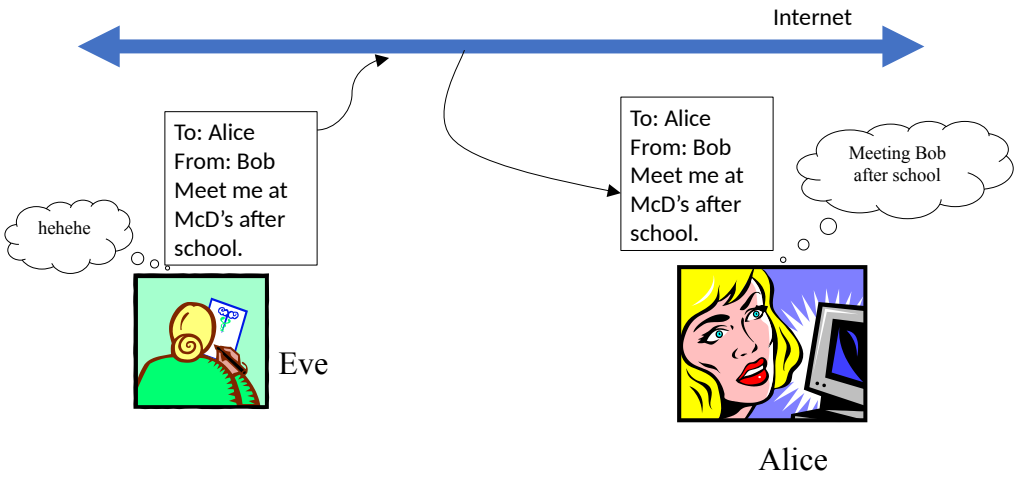


Figure 1.3: A violaton of authenticity: Eve composes a message to Alice, but it appears to be sent by Bob.

- A digital *signature* associated with a message is a sequence of bits which can be used to determine the message was truly sent by the person or entity shown in the `from:` field.[2]
- *Verification* is the process which a recipient would use to authenticate the sender of the message using the digital signature.
- A *digital certificate* is a document which can be used to authenticate the identity of an individual person, organization, or other entity involved with communication.
- A *certificate authority* is a trusted organization which can issue digital certificates and verify the identity of its clients.

1.2 Cryptanalysis

As described above, the process of breaking the enemy's codes is known as *cryptanalysis*. We speak of an 'attack' on an enemy's communication to mean an attempt at foiling the enemy's confidentiality, integrity, or authenticity. Prior to the advent of computers cryptographers would often keep their algorithms secret, to afford better protection from the enemy cryptanalysts. However, once the enemy broke the code and derived the algorithm, all was lost, and the cryptographers needed to develop a new algorithm and find a way to communicate the algorithm to the intended recipients.

Today we understand that the algorithms should be public. Security is achieved by using secret keys which we distribute to our friends but not to our enemies, or by using algorithms which do not require the distribution of keys (see chapter 3 and chapter 6). In what follows it is assumed that the cryptographic algorithms are public.

It should be clear that we are teaching cryptanalysis to our students not because we want them to break into legitimate computing systems. A cryptographer can build secure systems only if he/she has a good understanding of the methods of attack which could be launched against it.

Cryptanalysis techniques can be classified as:

- Known plaintext attack: The cryptanalyst has somehow obtained a plaintext and the corresponding ciphertext. This information is used to decrypt other plaintext messages, perhaps by deriving the decryption key from the pair of texts.
- Chosen plaintext attack: The cryptanalyst chooses a plaintext and is able to obtain the corresponding ciphertext. If this information can be used to derive the decryption key the cryptanalyst will be able to decrypt other ciphertexts, and the code is broken.[3]

[2] A digital signature is *not* a bitmap image of a written signature, such as one would obtain from a digital scanner.

[3] One of the most famous chosen plaintext attacks took place during World War II prior to the battle of Midway Island in the Pacific Ocean. Allied cryptanalysts suspected that

- Ciphertext only attack: This is the most common, and most difficult kind of attack. The cryptanalyst has intercepted ciphertext, with no corresponding plaintext. There are two common methods to obtain the decryption key and/or plain text:

  - Brute force: Try every possible decryption key by using it with the decryption algorithm to produce plaintext. The cryptanalyst must somehow be able to distinguish between sensible plaintext (if the correct key is used) versus garbage (produced by an incorrect key). When the plaintext consists of standard ASCII text, this is feasible, but if the plaintext is an image, sound, or other digital information, it is not that easy. Even if detecting a good key is feasible, the huge number of possibilities for a key of more than 100 bits makes the brute force approach unlikely to succeed. [4]

  - Statistical attack: After accumulating a large quantity of ciphertext resulting from the same encryption key,[5] the cryptanalyst will search for patterns which may provide clues to the plaintext and/or key. If the plaintext is known to be true text in a natural language, such as English, the expected distribution of letters in the plaintext can help the cryptanalyst use statistics to break the code (this is covered in more detail in chapter 2).

- Chosen ciphertext: The cryptanalyst is able to obtain the plaintext for a ciphertext of his own choice. With several such pairs of texts the cryptanalyst is able to derive the encryption key.

- Chosen text: The cryptanalyst is able to obtain the plaintext for a ciphertext of his own choice. The cryptanalyst is also able to obtain the ciphertext for a plaintext of his own choice. With several such dual pairs of texts the cryptanalyst is able to derive the encryption key.

- Man-in-the-middle attack: The attacker is able to intercept messages, tamper with them and forward to the intended recipient. Typically the attacker will substitute his own public encrytion key (see chapter 3) for the sender's public encryption key.

The most commonly used attacks are the brute force attack, statistical attack, and man-in-the-middle attack.

Japanese cryptographers were using a code word for 'Midway'. The Allies broadcast a message in the clear: 'Midway low on water', though there was plenty of water at Midway. Japanese headquarters alerted their navy by broadcasting this message, and using the code word for 'Midway'. The Allies were now certain of the code word for 'Midway', and were able to prepare for the planned Japanese attack on the naval forces at Midway Island, which was a turning point in the war in the Pacific.

[4] The number of different keys, with length 100 bits is 2^{100} which is about 10^{30}.

[5] For security, cryptographers recommend using a different key for each communication session, known as a *session key*.

1.3 Exercises

1. Briefly define each of these: Cryptology, Cryptography, Cryptanalysis

2. (a) Bob sends Alice a message. He does not care who reads the message, but he wants to assure Alice that the message is really from him so he attaches a digital signature to the message. Bob is concerned with which (one or more) of the following:
      i. Confidentiality
      ii. Integrity
      iii. Authenticity

   (b) Bob sends Alice a message. He wants to make sure that only Alice can read the message, so he encrypts the message and sends the encrypted version to Alice. Bob is concerned with which (one or more) of the following:
      i. Confidentiality
      ii. Integrity
      iii. Authenticity

   (c) Bob sends Alice a message. He wants to assure Alice that no one has tampered with the message, so he sends a hash of the message along with the message. Bob is concerned with which (one or more) of the following:
      i. Confidentiality
      ii. Integrity
      iii. Authenticity

3. An encryption algorithm needs which (one or more) of the following to encrypt a plaintext:
   (a) Plaintext
   (b) Ciphertext
   (c) Encryption key
   (d) IP address of the recipient

4. A decryption algorithm needs which (one or more) of the following to encrypt a plaintext:
   (a) Plaintext
   (b) Ciphertext
   (c) Encryption key
   (d) IP address of the recipient

5. Which method of attack may rely on the distribution of letters in English language text?

6. During World War II, the German military was using a machine known as the Enigma Machine to encrypt and decrypt messages to its forces in the field. Allied cryptanalysts captured some decrypted messages and noticed that they always ended with 'Heil Hitler!'. They were able to use this information to obtain the Enigma keys used for future transmitions. What kind of attack was this?

7. Lately a new computing paradigm, known as *quantum computing* has been the subject of much research. Quantum computing uses quantum theory from modern physics to perform computations far faster than conventional computers. If our enemies develop quantum computers before we do, what kind of attack will they be able to launch against our cryptosystems?

Chapter 2

Classical Cryptography

In this chapter we discuss only a few of the cryptographic systems that were used prior to the advent of computers. These systems are not used today, because of the convenience, power, and security of modern computers. Nevertheless, these early cryptographic systems form a good introduction to some of the techniques that are still used today. In addition they will give us an opportunity to delve into cryptanalysis, the process of breaking the enemy's codes, without overly technical obstacles.

2.1 Mathematics for classical cryptography

The ciphers described here can be described with or without using mathematics. We will do both, to help the student grasp the concepts firmly.

2.1.1 Modular arithmetic

Modular arithmetic is used throughout many cryptosystems, and in public key cryptosystems in particular. A modulus is simply the remainder obtained by an integer division. Integer division has two results: a quotient and a remainder. For example, when 17 is divided by 5, the quotient is 3, and the remainder is 2. We can check this by verifying that $3 * 5 + 2 = 17$.

In this text we will use the forward slash, '/', to indicate the remainder, and the operator `MOD` to indicate the remainder. Thus
$17 \ / \ 5 \ = \ 3$
$17 \ MOD \ 5 \ = \ 2$[1]
If the left operand is negative, we add the right operand repeatedly until we have a non-negative result. For example,
$-3 \ MOD \ 5 \ = \ 2 \ MOD \ 5 \ = \ 2$
$-14 \ MOD \ 5 \ = \ -9 \ MOD \ 5 \ = \ -4 \ MOD \ 5 \ = \ 1 \ MOD \ 5 \ = 1$

[1] Many programming languages use the % symbol for MOD.

Modulus	Generator	Congruence class
5	1	{...-9,-4,1,6,11,16,21,...}
5	0	{...-10,-5,0,5,10,15,20,...}
5	3	{...-2,3,8,13,18,23,...}
7	0	{...-7,0,7,14,21,28,...}
7	6	{...-8,-1,6,13,20,27,...}
1000	3	{...-997,3,1003,2003,...}

Figure 2.1: Modular arithmetic: Some examples of congruence classes

$-10\ MOD\ 5\ =\ -5\ MOD\ 5\ =\ 0\ MOD\ 5\ =\ 0$
We will assume that the right operand for modulus must be positive.[2]

It is now apparent that the MOD operator effectively groups numbers into what we call *congruence classes*. Since
$-9\ MOD\ 5\ =\ -4\ MOD\ 1\ =\ 6\ MOD\ 5\ =\ 11\ MOD\ 5\ = 1$
we say that the values {-9, -4, 1, 6, 11} are all *congruent* to 1 (mod 5). The notation we use for congruence is:
$-9\ \equiv\ 1\ (mod\ 5)$
$-4\ \equiv\ 1\ (mod\ 5)$
$1\ \equiv\ 1\ (mod\ 5)$
$6\ \equiv\ 1\ (mod\ 5)$
Consequently we will be writing equivalence expressions such as the one below:
$3 \times 8\ \equiv\ 4\ (mod\ 5)$

The congruence classes defined by mod 5 are infinite, but in cryptography we will generally focus our attention on the member of the congruence class which is in the range $[0..m-1]$ where m represents the modulus. We call this the *generator* for the congruence class. Figure 2.1 shows some examples of congruence classes for a few different moduli, showing the generator for each congruence class.

Multiplicative inverses

In ordinary arithmetic a *mulitplicative inverse* of a given number is that number which, when multiplied by the given number, gives a result of 1. More formally, if $x \times y = 1$, then we say that y is a multiplicative inverse of x (also x is a multiplicative inverse of y). The notation we use for an inverse is a superscript of -1.
$x \times x^{-1} = 1$
For example, $3^{-1} = 1/3$ and $2.5^{-1} = 0.4$.

In cryptography we are working with modular arithmetic, using only whole numbers.[3] However, we can still find multiplicative inverses. To find the inverse

[2]There is disagreement on the meaning of the % operator for negative operands on various computing platforms.

[3]Mathematicians speak of ordinary arithmetic as *continuous* and modular arithmetic as *discrete*.

Modulus	x	x^{-1}	Modulus	x	x^{-1}	Modulus	x	x^{-1}
5	0	-	7	2	4	10	2	-
5	1	1	7	3	5	10	3	7
5	2	3	7	4	2	10	4	-
5	3	2	7	5	3	10	5	-
5	4	4	7	6	6	10	7	3
5	7	3	7	8	1	10	9	9

Figure 2.2: Modular arithmetic: Some examples of multiplicative inverses

of 3 (mod 5), we ask what number in the range [0..4] will produce 1 when multiplied by 3? I.e.
$3\times ? \equiv 1(mod\ 5)$
Since $3 \times 2 = 6$ and $6 \equiv 1\ (mod\ 5)$, the solution is 2.
$3 \times 2 \equiv 1\ (mod\ 5)$. The multiplicative inverse of 3 (mod 5) is 2, and the multiplicative inverse of 2 (mod 5) is 3. The numbers 2 and 3 are multiplicative inverses of each other. Figure 2.2 shows some examples of multiplicative inverses. We notice a few interesting properties in Figure 2.2.

- In modular arithmetic some numbers do not have inverses.
- The number 0 never has an inverse.
- It can be shown that if n shares a factor with m, then n has no multiplicative inverse (mod m). The factors of 10 are 2,5. Since even numbers share the factor 2 with 10, no even numbers will have an inverse (mod 10). Also multiples of 5 will have no inverse (mod 10).

At this point we rely on the student's ingenuity to find multiplicative inverses. This could involve much trial-and-error, i.e. a brute-force search. In chapter 6 we will expose a fast algorithm for multiplicative inverses.

2.1.2 Permutation vectors

A *vector* is, in general, an ordered list of values. A *permutation vector* is a vector of integer values which specifies a reordering of a list of values of the same length. A permutation vector consists of a list of indices which select values from the vector being permuted. In a permutation vector the indices will be unique non-negative integers, with no duplicates. For example, if we have the list of values to be reordered:
`myList = [a,b,c,d,e]`
and a permutation vector:
`perm = [3,1,2,4,0]`
the permutation vector can be applied to the list:
`myList[perm] = [d,b,c,e,a]`
Note that the values in a permutation vector are the integers in [0..len-1], in some order, where `len` is the length of the vector. In this chapter, we will use

a permutation vector of length 26 to specify a substitution cipher. In chapter 3 we will use permutation vectors extensively in private-key cryptosystems.

Inverse permutation vector

In mathematics an *inverse* function is a function which 'undoes' the effect of a given function. If f and g are the names of two functions, and `f(g(x)) = x`, and `g(f(x)) = x`, then f and g are inverse functions of each other. For example, if `f(x) = 2x + 3` then the inverse function would be `g(x) = (x-3)/2`. `f(7) = 17`, and `g(17) = 7`, the original argument for f. We often use a superscript of -1 to indicate an inverse function: f^{-1} is the inverse of the function f.

An *inverse permutation* is a permutation which 'undoes' the effect of a given permutation. If v is a vector, and p and q are permutation vectors of the same length, then if `(v[p])[q] = v` and `(v[q])[p] = v`, then p and q are inverse permutations of each other. They are mutual inverse permutations.

For example, if we have the permutation vector `p = [3,1,0,2]`, the inverse p^{-1} would be `[2,1,3,0]`. To see this assume `v = [a,b,c,d]`. Call the result `inv = v[p] = [d,b,a,c]`. Applying the inverse permutation, `inv[2,1,3,0] = [a,b,c,d] = v`.

Given any permutation vector, how can we find the inverse permutation vector? For each position,p, we will need to do a search for p in the given permutation vector, copying its position to the inverse permutation. In our example:

1. Search for 0, it is found at position 2. inv = [2,?,?,?]
2. Search for 1, it is found at position 1. inv = [2,1,?,?]
3. Search for 2, it is found at position 3. inv = [2,1,3,?]
4. Search for 3, it is found at position 0. inv = [2,1,3,0]

2.1.3 Frequency distributions

The mathematical concept of a *frequency distribution* will be important in studying cryptanalysis. A distribution of a list of items tells us how many times each item is found in the given list. In cryptanalysis we will deal primarily with a list of letters (presumably drawn from a given ciphertext). For each letter in the list, the distribution tells us how many times it occurs in the list.

As an example, let's use the previous sentence as our list of letters. The frequency distribution of that set is shown in Figure 2.3.

Finding the frequency distribution in a given text can be a daunting task. One popular tool that can be used is a spreadsheet. Here is how it can be done:

1. Enter the letters of text, one letter per cell, into the first column of the spreadsheet.
2. Sort the column in ascending order

T	12
I	9
E	8
S	7
H	5
L	5
N	4
O	4
R	4
C	3
U	3
M	2
B	1
F	1
W	1
Y	1

Figure 2.3: Frequency distribution of the letters in the sentence `FOR EACH LETTER IN THE LIST THE DISTRIBUTION TELLS US HOW MANY TIMES IT OCCURS IN THE LIST`

3. In the second column
   (a) In the first row place a 1
   (b) In the next row place a formula that increments the number in the cell directly above it by 1 if the letters in the first column are equal, otherwise it will be 1. If the 1 is in cell B1, then in cell B2 you will have the formula =IF(A1=A2,B1+1,1).
   (c) Copy that formula to all the other cells in the same column.
4. Copy what you have to a pair of separate columns, but when pasting, choose Paste Special to past just text and numbers, but not formulas.
5. Sort the separate columns using the column of numbers as the sort key.
6. It should now be easy to pick out a frequency count for each letter in the given text.

2.1.4 Exercises

1. Evaluate each of the following:
   (a) 17 MOD 5
   (b) 23 MOD 4
   (c) 125 MOD 5
   (d) -7 MOD 10

(e) -23 MOD 4

(f) 123080320903098 MOD 10

(g) 123080320903098 MOD 5

2. Evaluate each of the following, if possible. If there is more than one solution, choose the generator.

(a) $23\ (mod\ 7)\ \equiv\ ?$

(b) $21 + 9\ (mod\ 3)\ \equiv\ ?$

(c) $1011\ (mod\ 10)\ \equiv\ ?$

(d) $47\ (mod\ 5)\ \equiv\ ?$

(e) $203219943\ (mod\ 10)\ \equiv\ ?$

(f) $29851923939393932948\ (mod\ 5)\ \equiv\ ?$

(g) $3^{-1}\ (mod\ 11)\ \equiv\ ?$

(h) $7^{-1}\ (mod\ 11)\ \equiv\ ?$

(i) $8^{-1}\ (mod\ 10)\ \equiv\ ?$

(j) $14^{-1}\ (mod\ 91)\ \equiv\ ?$

3. Given the vector v = [a,c,w,a,d,e,a] and the permutation vector p = [0,3,6,1,4,5,2] find the vector v[p].

4. Given the vector v = [w,c,a,c,x,b,a] Find the permutatation vector which will arrange the letters in the vector v in alphabetic order.

5. Given the permutation vector p = [0,3,6,1,4,5,2] find the permutation vector, inv, which will 'undo' the permutation performed by p. (v[p])[inv] should be equal to v, for any vector v of length 7. The permutation vector inv is said to be the *inverse* permutation vector of p, and is denoted as inv = p^{-1}.

6. Write a computer program which will find the inverse permutation for any given permutation vector.

7. Find the frequency distribution of letters in this sentence. Ignore all spaces and punctuation.

8. Find the frequency distribution of letters in the Gettysburg Address. Ignore all spaces and punctuation.

2.2 Shift and substitution ciphers

Shift and substitution ciphers are conceptually the easiest to understand and implement. They are also easily broken (and even easier with the aid of a computer). Both of these ciphers, as well as the Viegenere cipher use the 26

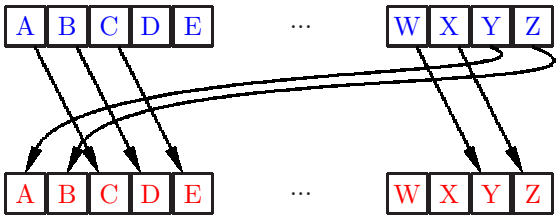


Figure 2.4: Diagram of a shift cipher, with a displacement of 2

letters of the English alphabet. In this section we will have no other symbols in the plaintext; this means that there will be no spaces, nor punctuation, when the intended recipient decrypts the ciphertext to obtain the original plaintext.[4]

2.2.1 Shift ciphers

A shift cipher encrypts plaintext by shifting each letter a fixed number of positions in the alphabet. Let's call this number of positions a shift *displacement*. Then the ciphertext can be decrypted by shifting the letters of the ciphertext by the displacement in the *opposite* direction. If the shift reaches the end or beginning of the alphabet, it wraps around to the other end. Shifting for encryption, we have the sequence:
$A \to B \to C \to D... \to Y \to Z \to A$
Shifting for decryption, we have:
$D \to C \to B \to A \to Z \to Y$

We can think of the displacement as the *key* for a shift cipher. In order to decrypt the ciphertext, the intended recipient needs to know that key. A diagram representing a shift cipher with a displacement of 2 is shown in Figure 2.4.

We can now give an example of a shift cipher with a displacement of 3. In this cipher to encrypt plaintext each letter is shifted 3 positions in the alphabet:
$A \to D$
$B \to E$
$C \to F$
...
$W \to Z$
$X \to A$
$Y \to B$
$Z \to C$

To encrypt the plaintext:
`I VISIT THE ZOO AND AXE THROW`
we produce the ciphertext:

[4] Of course the fact that we are ignoring spaces between words opens the possibility of ambiguous plaintexts - plaintexts in which the spaces could be inserted in different ways to obtain a sensible message.

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

Figure 2.5: Numeric codes for the 26 alphabetic letters

L YLVLW WKH CRR DQG DAH WKURZ
Note that the Z encrypts as C and the X encrypts as A.

The recipient of the ciphertext would decrypt by shifting in the opposite direction to obtain the original plaintext. Each letter in the ciphertext is shifted toward the *beginning* of the alphabet:
$A \rightarrow X$
$B \rightarrow Y$
$C \rightarrow Z$
$D \rightarrow A$
...
$Y \rightarrow V$
$Z \rightarrow W$

To decrypt our example:
L → I
Y → V L → I V → S L → I W → T
W → T K → H H → E
C → Z R → O R → O
D → A Q → N G → D
D → A A → X H → E
W → T K → H U → R R → O Z → W

Of course we can use mathematics to provide a much more succinct description of a shift cipher. We use the numeric codes described above for the letters of the alphabet, where A=0, B=1, C=2, ... Z=25, as shown in Figure 2.5.

Encryption of each letter is simply:
```
Encr(ltr) = (ltr + d) mod 26
```
where ltr is the code for the letter being encrypted and d is the displacement. To decrypt a letter we subtract the displacement instead of adding:
```
Decr(ltr) = (ltr - d) mod 26
```
Encryption and decryption of our example using these mathematical forumlas is shown in Figure 2.6.

2.2.2 Substitution ciphers

A *substitution cipher* is a cipher which substitutes a unique letter of the plaintext with some other letter, throughout the encryption process. This kind of cipher is commonly seen as a puzzle in many newspapers, and is called a 'cryptogram'.

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

Figure 2.6: A shift cipher with displacement = 3. Ignoring spaces, the plaintext is `I VISIT THE ZOO AND AXE THROW`. The ciphertext is `LYLVLWWKHCRRDQGDAHWKURZ`.

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

Figure 2.7: An example of a substitution cipher.

Common cryptograms differ from our substitution ciphers in two ways:

- Common cryptograms include spaces between words, and often include punctuation.
- Common cryptograms normally do not substitute a letter for itself.[5]

We can specify a substitution cipher by showing the substitution as a table with two rows. In Figure 2.7 the top row shows every letter of the alphabet, and the bottom row shows an example of the letters which could be substituted in the ciphertext. To encypt a plain text using the substitution of Figure 2.7, every
A is encrypted as M
B is encrypted as O
C is encrypted as R
...
Y is encrypted as U
Z is encrypted as T

To encrypt the plaintext
`I VISIT THE ZOO AND AXE THROW`
using the cipher of Figure 2.7 we apply the substitution and produce the ciphertext
`B XBLBN NPA THH MSV MQA NPJHY`

The table specifying the substitution, as in Figure 2.7, is the *key* for a substitution cipher. In order for the intended recipients to decrypt the ciphertext, they need to know the substitution specified in the table.

As with shift ciphers, the substitution cipher can be specified mathematically, though it is not as simple. For substitutions we can use a *permutation*

[5] This feature of cryptograms actually make them easier to solve.

0	1	2	3	4	5	6	7	8	9	10	11	12
12	14	17	21	0	22	6	15	1	3	2	5	4
13	14	15	16	17	18	19	20	21	22	23	24	25
18	7	8	10	9	11	13	25	23	24	16	20	19

Figure 2.8: Permutation vector specifying the substitution cipher of Figure 2.7

3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	0	1	2

Figure 2.9: Permutation vector for the shift cipher of Figure 2.6, shown in two rows.

vector to specify the substitution. Just as the table in Figure 2.7 specifies an encryption key, the permutation vector is an equivalent way of specifying the substitution. Position 0 of the permutation vector gives the numeric code of the ciphertext letter corresponding to an 'A' in plaintext. Position 1 of the permutation vector gives the numeric code of the ciphertext letter corresponding to a 'B' in plaintext. Position 25 of the permutation vector gives the numeric code of the ciphertext letter corresponding to a 'Z' in plaintext. The permutation vector that corresponds to Figure 2.7 is shown in Figure 2.8.

We note that a shift cipher is a special case of substitution cipher in which the indices in the permutation vector are sequentially increasing, wrapping around to the beginning, when reaching the end. The permutation vector for the shift cipher with displacement = 3, shown in Figure 2.6, is shown in Figure 2.9; it is shown in two rows due to lack of space on the page.

2.2.3 Cryptanalysis of the shift and substitution ciphers

In cryptanalysis we are attempting to 'break the enemy's codes'. We are either trying to decrypt a given ciphertext, or we are trying to find the key, in order to quickly decrypt other ciphertexts.

Cryptanalysis of the shift cipher

With the shift cipher, as presented here, there are only 26 possible keys, 0..25.[6] This suggests a brute force approach: try decrypting the ciphertext with every possbile key, until a sensible plaintext is obtained. Eventually we will find the key.

However, we may wish to expedite the cryptanalysis, so that we do not have to try all 26 keys; this can be done by using statistics. By looking at large samples of English text, we can determine that the three most common letters are E, T, A, with E much more frequent than T, and T slightly more frequent than A. If we can find the most frequently occurring letter in the ciphertext,

[6]There are actually only 25 keys that need to be tried, because a displacement of 0 would afford no encryption at all.

D	7
Z	3
S	2
Q	2
U	1
X	1
V	1
J	1
M	1
C	1
O	1

Figure 2.10: Letter distribution from the ciphertext DZSDUDQXVDDJZMCQDODZS. The plaintext is `EAT EVERY WEEK AND REPEAT`.

that would probably correspond to a plaintext E. If so, that immediately gives us the key, or displacement. For example, in the ciphertext DZSDUDQXVDDJZMCQDODZS the most frequent letter is D. If that corresponds to an E in the plaintext, the displacement would be 25 (or -1), and the plaintext would be EATEVERYWEEKANDREPEAT. Estimating where spaces should be inserted, we get EAT EVERY WEEK AND REPEAT

This strategy is not guaranteed to succeed; it is possible that the ciphertext D does not correspond to plaintext E. We take a closer look at the distribution of letters in the ciphertext in Figure 2.10.

If the key of 25 does not work, we could either:

- Look for the second-most common letter in the ciphertext, which happens to be Z, and assume that letter corresponds to a plaintext E, indicating a displacement of 21, or
- Assume the ciphertext D corresponds to the second-most common letter in English text, T. This would be a displacement of 10.

We would continue in this way until a sensible decryption is found. In this way our knowledge of the English text frequency distribution provides a speedup of the brute force attack; we probably would not need to try very many keys before finding the true key.

Cryptanalysis of the substitution cipher

Cryptanalysis of the substitution cipher is a bit more difficult. The substitution cipher key is a permutation vector, as described above. The length of the vector is 26. The number of possible keys would be the number of permutations of a vector of length 26, which is 26 factorial. 26! is approximately 4×10^{26}. This is a huge number; a brute force attack would take an eternity, even with a computer.

D	723	E
Z	513	T
S	495	A
Q	411	O
U	388	I
X	352	N
V	312	S
J	288	H
M	250	R
C	185	D
O	120	L

Figure 2.11: Letter distribution from a hypothetical ciphertext, shown in the left column, with probable letters from the plaintext shown in the right column.

Therefore, cryptanalysis of the substitution cipher will need to use statistics, as described above for cryptanalysis of the shift cipher. However, with the substitution cipher we will need to make several attempts at key values. For example, suppose the most common letter in the ciphertext is W. It would be reasonable to assume that this corresponds to an E in the plaintext. If this is correct we are not finished; unlike the shift cipher, we still need to find the other 25 parts of the key.

Figure 2.11 shows a possible distribution of of the eleven most frequently occurring letters in a hypothetical ciphertext. It also shows a reasonable guess for the corresponding letters of the plaintext in the right column. This guess is based on the fact that the eleven most commonly occurring letters in English text are E,T,A,O,I,N,S,H,R,D,L in that order.[7] Though Figure 2.11 does not give us the complete key, it gives enough information that we may be able to construct the plain text from the ciphertext, and then fill in many of the missing parts of the key (the other 15 letters, not shown in the left column of Figure 2.11).

Graphic representations of frequency distribution: Scrawl and Signature

In many areas of computer science it is helpful to provide pictorial or graphic images to gain a better understanding of concepts. This is certainly the case with cryptanalysis. We have seen that the frequency distribution of letters in a ciphertext is often critical to our efforts to break the enemy's code. Here we look at graphic representations of that distribution.

Before doing so, we should obtain a fairly representative distribution of letters taken from a large English text. Such a distribution is shown in Figure 2.12. Here we see, as previously noted, that the letter E is by far the most frequently

[7] Various sources might report slight variations in this distribution, depending on the text which is sampled.

Letter	Count	Frequency	Letter	Count	Frequency
E	21912	12.02%	M	4761	2.61%
T	16587	9.10%	F	4200	2.30%
A	14810	8.12%	Y	3853	2.11%
O	14003	7.68%	W	3819	2.09%
I	13318	7.31%	G	3693	2.03%
N	12666	6.95%	P	3316	1.82%
S	11450	6.28%	B	2715	1.49%
R	10977	6.02%	V	2019	1.11%
H	10795	5.92%	K	1257	0.69%
D	7874	4.32%	X	315	0.17%
L	7253	3.98%	Q	205	0.11%
U	5246	2.88%	J	188	0.10%
C	4943	2.71%	Z	128	0.07%

Figure 2.12: Frequency distribution of the letters in a sample of English text (40,000 words). Source: Cornell University Math Explorers Club

occurring letter in English. graph

Figure 2.12 is simply a table of numbers. It may be more useful to view these numbers in the form or a graph. This is easily done using a spreadsheet, but there are a few different ways of doing this.

Scrawl

A graph known as the *scrawl* of English text shows the frequency of each letter, and is shown in Figure 2.13. Note that the letters on the horizontal axis are in alphabetic order, with the letter A on the left and Z on the right.

The scrawl graph is useful in analyzing a shift cipher. By using a frequency distribution of the ciphertext, and producing the scrawl (i.e. a graph) of the ciphertext, we should get a bar graph similar to the one shown in Figure 2.13, but displaced to the left or right. The displacement of this graph is the key! An example is shown in Figure 2.14. To see this we notice that there is a sharp peak for the letter J in the ciphertext. This means that since J is the most frequent letter in the ciphertext, it probably correspondes to an E in the plaintext. Subtracting, J-E (i.e. 9-4) we get a displacement of 5 for the key. The value 5 was added to each letter in the plaintext to get the corresponding letter in the ciphertext.

Signature

Figure 2.12 is simply a table of numbers. It may be more useful to view these numbers in the form or a graph. This is easily done using a spreadsheet, and the resulting graph is shown in Figure 2.15. This graph is known as the *signature* of English text. Note that in a signature the horizontal axis is sorted by the frequency of occurrence of each letter. It shows the most common letter, E, on

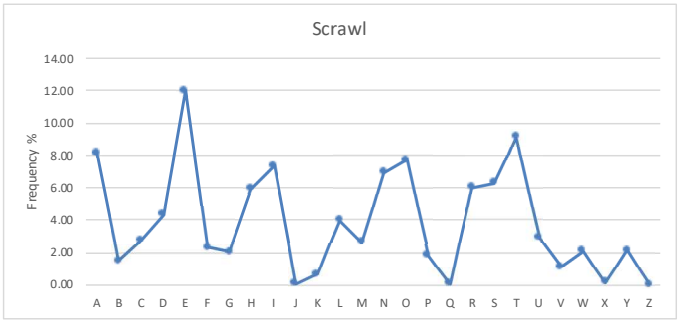


Figure 2.13: Scrawl of English text, showing the percent frequency for each letter of the alphabet. The horizontal axis of the graph is sorted alphabetically by letter.

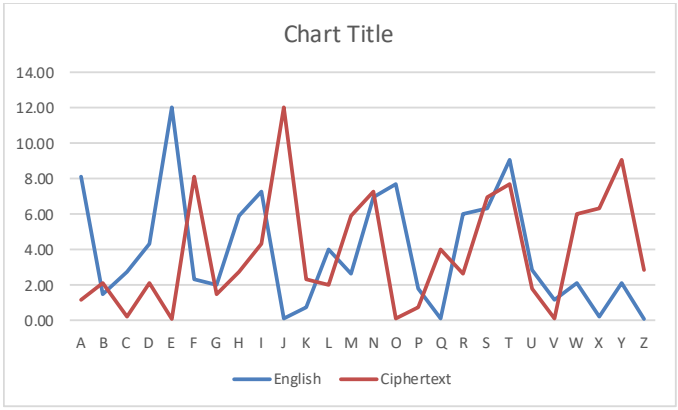


Figure 2.14: Scrawl of a hypothetical ciphertext, along with the scrawl of English text. The displacement, which appears to be 5, is the encryption key.

the left and the least common letter, Z, on the right. A graph of the signature of a ciphertext created with a shift or substitution cipher will have a similar shape, and can be used as an aid in determining the key during cryptanalysis.

To see this we show the signature of a hypothetical ciphertext in Figure 2.16. The shape of this graph appears to be the same as the shape of the graph in Figure 2.15. This suggests that a substitution cipher was used for the encryption. The horizontal axis in Figure 2.16 shows the letters of the ciphertext, and also the corresponding letters of the plain text, from Figure 2.15. Presumably this forms the key to the encryption. However, we are assuming that the the ciphertext is large enough to make this kind of statistical analysis valid. Also, notice that many of the points on the graph are very close to neighboring points to the left or right. This suggests that some trial and error will be required before the ciphertext can be completely determined. For example Figure 2.16 indicates that a plaintext `U` is encrypted as a ciphertext `M`, and that a plaintext `C` is encrypted as a ciphertext `U`. But since the frequencies for these two cases are so close, we may get an incorrect result, in which case we may wish to reverse the `U` and `C` in Figure 2.16 to see if that provides a reasonable decryption.

Summary of signature and scrawl

We emphasize that signature and scrawl are simply two different ways of viewing the same information: the frequency distribution of letters in English (or encrypted) text. Scrawl can be used in the cryptanalysis of a shift cipher and signature can be used in the cryptanalysis of a substitution cipher.

2.2.4 Exercises

1. Encrypt the following plaintext, using a shift cipher with displacement = 9:
   `FOUR SCORE AND SEVEN YEARS AGO`
   Ignore spaces.

2. The following ciphertext was produced with a shift cipher using a displacement = 5. Decrypt, and show the plaintext.
   `XMFQQSTYUJWNXMKWTRYMJJFWYM`

3. Encrypt the following plaintext with a substitution cipher (ignore spaces)
   `FOUR SCORE AND SEVEN YEARS AGO`

   (a) Using the substitution specified in Figure 2.17.

   (b) Using the permutation vector:
   `[5,8,12,17,23,4,1,21,19,16,2,15, 3,18,24,7,10,0,6,13,9,20,11,22,14]`

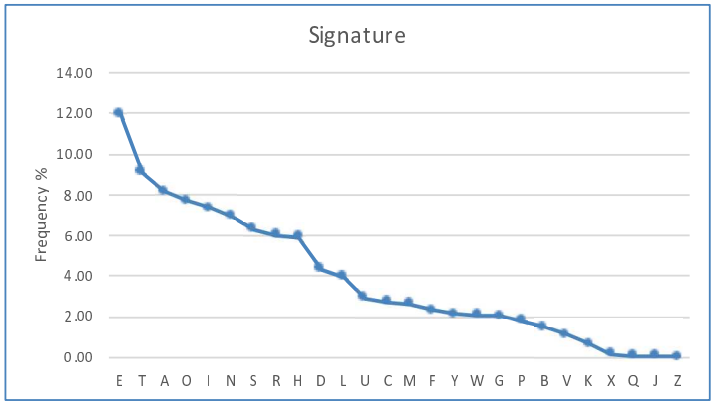


Figure 2.15: Signature of English text, showing the percent frequency for each letter of the alphabet. The horizontal axis of the graph is sorted by frequency.

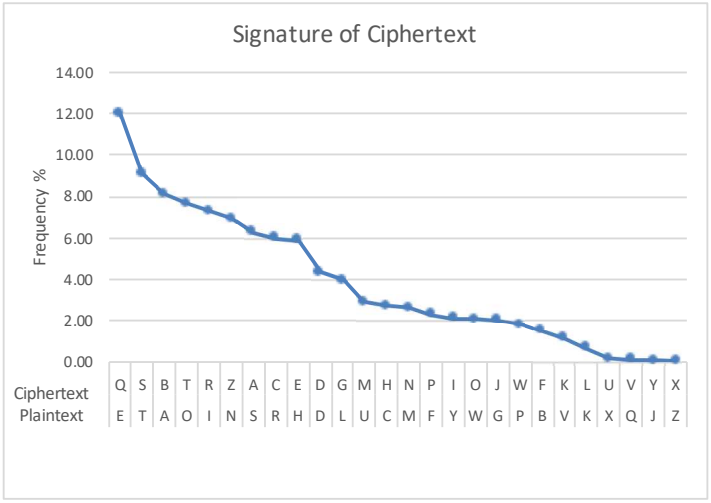


Figure 2.16: Signature of a hypothetical ciphertext. The horizontal axis shows possible letters from the plaintext, based on frequency.

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

Figure 2.17: A substitution cipher for an exercise.

4. Decrypt the following ciphertext with a substitution cipher
   GVFPPSZNHXATGVEAZDNVXXFANV

   (a) Using the substitution specified in Figure 2.17.

   (b) Using the permutation vector:
   `[5,8,12,17,23,4,1,21,19,16,2,15, 3,18,24,7,10,0,6,13,9,20,11,22,14]`

5. The following ciphertext was encrypted with a shift cipher.
   XLSWIALSIEXERHJIIHEVIVIEHCXSIEXQIEXERHIEXERHIEX

   (a) Derive the frequency distribution of letters in the ciphtertext.

   (b) Find the displacement, using statistics.

   (c) Decrypt the ciphertext.

6. There is a ciphertext at
   `http://cs.rowan.edu/~bergmann/books/Cryptology/Ciphertext.txt`
   It was produced by a substitution cipher.

   (a) Derive the frequency distribution of letters in the ciphtertext.

   (b) Use statistics and trial-and-error to derive the plaintext

   (c) Show the substitution table or permutation vector.

7. In discussing the cryptanalysis for the substitution cipher we said that 26! is approximately 4×10^{26}.

   (a) Write a program to find a better approximation of 26!

   (b) Write a program to find the exact value of 26!

2.3 Affine ciphers

2.3.1 Encryption with affine ciphers

In this section we explore another kind of substituion cipher, known as an *affine* cipher. In this cipher we use a simple mathematical equation to define the cipher:
$y = ax + b \ (mod\ 26)$

Plaintext			Ciphertext	
Letter	Code	Calculation	Code	Letter
H	7	$5 \times 7 + 3 \ (mod\ 26)$	12	M
E	4	$5 \times 4 + 3 \ (mod\ 26)$	23	X
L	11	$5 \times 11 + 3 \ (mod\ 26)$	6	G
L	11	$5 \times 11 + 3 \ (mod\ 26)$	6	G
O	14	$5 \times 14 + 3 \ (mod\ 26)$	21	V

Figure 2.18: Encryption of the plaintext HELLO using the affine cipher y = 5x +3. The ciphertext is MXGGV

in which a plain text character is represented by x and a ciphertext letter is represented by y, and a and b are integer constants. Each letter is represented by its ordinal number, 0..25, as described in the section on substitution ciphers.

As an example, we use the affine cipher:
$y = 5x + 3 \ (mod\ 26)$
To encrypt the plaintext `HELLO` we multiply each letter by 5 and add 3, as shown in Figure 2.18 which results in the ciphertext `MXGGV`.

2.3.2 Decryption with affine ciphers

To decrypt a ciphertext that has been encrypted with an affine cipher we must first derive the decryption formula. To do that we start with the encryption formula and solve for x, which will give us the plaintext letter as a function of the ciphertext letter. In doing so we must remember that the affine cipher is *discrete*; i.e. we are using modular arithmetic with whole numbers.

$y = a \times x + b \ (mod\ 26)$	Encryption formula
$y - b = a \times x \ (mod\ 26)$	Subtract b from both sides
$(y - b) \times a^{-1} = a \times x \times a^{-1} \ (mod\ 26)$	Multiply both sides by the inverse of a
$(y - b) \times a^{-1} = x \ (mod\ 26)$	Decryption formula

Note on the third step that we cannot divide both sides by `a` because we are using discrete modular arithmetic; instead we must multiply both sides by the multiplicative inverse of `a`.

We return to the previous example, the affine cipher y = 5x+3. In order to decrypt a ciphertext we must first find the multiplicative inverse of 5:
$5^{-1} \equiv 21 \ (mod\ 26)$ because $5 \times 21 = 105$ and $105 \equiv 1 \ (mod\ 26)$. Since b = 3, the decryption formula will be
$x = (y - 3) \times 21 \ (mod\ 26)$ In Figure 2.19 we decrypt the ciphertext that was derived in Figure 2.18 by applying this decryption formula to the ciphertext MXGGV to obtain the original plaintext HELLO.

2.3.3 Choosing an affine cipher

We note that for the affine cipher $y = a \times x + b \ (mod\ 26)$ if we choose `a` = 1, we have a simple shift cipher, which is easy to break, so it is best to choose a value in the range [2..25]

Plaintext			Ciphertext	
Letter	Code	Calculation	Code	Letter
M	12	$(12-3) \times 21 \ (mod\ 26)$	7	H
X	23	$(23-3) \times 21 \ (mod\ 26)$	4	E
G	6	$(6-3) \times 21 \ (mod\ 26)$	11	L
G	6	$(6-3) \times 21 \ (mod\ 26)$	11	L
V	21	$(21-3) \times 21 \ (mod\ 26)$	14	O

Figure 2.19: Decryption of the ciphertext MXGGV which was encrypted in Figure 2.18, to yield the plaintext HELLO

However, some of the values in the range [2..25] cannot be used; in order to derive a decryption formula, the value of `a` must be a number which has a multiplicative inverse (mod 26). As we saw previously a number, n, has a multiplicative inverse (mod m) if m and n do not share factors. Since $26 = 2 \times 13$ the factors of 26 are 2 and 13. This means for an affine cipher we cannot choose an even number, nor 13, for the multiplier `a`.

As we continue our study of cryptography, it is important to remember that when designing encryption systems, we must always be able to derive a corresponding decryption system.

2.3.4 Cryptanalysis of the affine cipher

Can a brute force attack be used to break the affine cipher? How many possible keys are there? The value of `a` must be an odd number in the range [3..25] and the value of `b` must be any number in the range [0..25]. Thus, there are $12 \times 26 = 312$ possible keys. Alternatively, since the affine cipher is a special case of substitution cipher, a frequency distribution, or signature, could be used to find the plaintext, and consequently the key. Once we have a good guess for only two letters, we can solve for `a` and `b`. For example, suppose the signature of the ciphertext tells us that the letter E (4) is likely to be encrypted as a W (22), and that the letter T (19) is likely to be encrypted as a B. We now have two equations, with two unknowns:

$a \times 4 + b = 22 \ (mod\ 26)$	E encrypts as W
$a \times 19 + b = 1 \ (mod\ 26)$	T encrypts as B
$a \times -15 = 21 \ (mod\ 26)$	Subtracting the equations
$a \times 11 = 21 \ (mod\ 26)$	Because $-15 \equiv 11 \ (mod\ 26)$
$11^{-1} \equiv 19 \ (mod\ 26)$	Because $11 \times 19 \equiv 1 \ (mod\ 26)$
$a \times 11 \times 19 = 21 \times 19 \ (mod\ 26)$	Multiplying both sides by 19
$a = 399 \equiv 9 \ (mod\ 26)$	
Now we can solve for `b` using either	of our original equations.
$a \times 4 + b = 22 \ (mod\ 26)$	Use the first equation
$9 \times 4 + b = 22 \ (mod\ 26)$	We know that a is 9
$36 + b = 22 \ (mod\ 26)$	
$b = -14 \equiv 12 \ (mod\ 26)$	

So the key of the affine cipher is:
$y = 9 \times x + 12 \ (mod\ 26)$

2.3.5 Exercises

1. Use the affine cipher $y = 3 \times x + 8 \ (mod\ 26)$
   (a) Encrypt the plaintext `GOOD EVENING`
   (b) Derive the decryption formula
   (c) Decrypt your solution to part (a)
2. Is the affine cipher $y = 8 \times x + 1 \ (mod\ 26)$ a good cipher? Why or why not?
3. Is it possible that, for some affine cipher two different plaintext letters will encrypt to the same ciphertext letter? Explain.
4. You have intercepted a message that was encrypted with an affine cipher: `LKIOSEIMEPKLLKXGFKXGFKVGKOKKL`
   (a) Find the frequency distribution of letters in the ciphertext.
   (b) Derive the key, i.e. the encryption formula.
   (c) Show the original plaintext.

2.4 Polyalphabetic ciphers

The substitution ciphers described above share the property that each letter of the plaintext is consistently encrypted as the same letter of the ciphertext. This is also called a monoalphabetic cipher.

position = 0 (mod 3)	A	B	C	D	E	F	G	H	I	J	K	L	M
	E	Z	A	O	F	P	G	B	H	I	C	D	Q
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	R	J	X	S	K	T	L	U	Y	V	M	W	N
position = 1 (mod 3)	A	B	C	D	E	F	G	H	I	J	K	L	M
	F	N	W	O	V	G	Y	X	P	H	Z	I	Q
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	D	J	A	R	K	D	S	L	B	T	U	C	M
position = 2 (mod 3)	A	B	C	D	E	F	G	H	I	J	K	L	M
	T	B	A	K	U	L	V	C	M	W	D	X	E
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	N	F	O	Y	G	P	Z	H	Q	R	I	S	J

Figure 2.20: A polyalphabetic cipher, using a different substitution key every third position

Plaintext	G	O	O	D	M	O	R	N	I	N	G
Substitution cipher	0	1	2	0	1	2	0	1	2	0	1
Ciphertext	G	J	F	O	Q	F	K	D	M	R	Y

Figure 2.21: Encryption of the plaintext "GOOD MORNING" using the polyalphabetic cipher of Figure 2.20

2.4.1 General polyalphabetic ciphers

We now consider a broader range of ciphers, in which each letter of the ciphertext is determined not only by the corresponding letter of plaintext, but also its position in the plaintext. A different substitution key (i.e. permutation vector) is used for successive letters in the plaintext. After the last key is used, it cycles back to the first key. If there are 4 substitution keys, the first key is used for plaintext letters at positions 0,4,8,12,..., the second key is used for positions 1,5,9,13, ..., the third key is used for plaintext letters at positions, 2, 6,10,14, ..., and the fourth key is used for plaintext letters at positions 3,7,11,15,.... If there are n substitution ciphers, the plaintext letter at position p will be encrypted using the cipher chosen by p (mod n). For example, a polyalphabetic cipher could consist of three separate substitution ciphers, as shown in Figure 2.20. After encrypting the plaintext letter at position 2 with the last substitution cipher, it encrypts the plaintext letter at position 3 with the first substitution cipher.

Figure 2.21 shows an encryption of the the plaintext `GOOD MORNING` using the polyalphabetic cipher of Figure 2.20. The ciphertext is `GFQOQFKDMRY`. We notice that the consecutive O's in the word `GOOD` are encrypted as different letters in the ciphertext: `JF`, whereas with a monoalphabetic cipher this would not have been the case, making it a little more difficult to break this polyalphabetic code.

2.4.2 One-time pad

A substitution cipher with a sufficiently long and random key is known as a *one-time pad*. Statistical attacks on a one-time pad are unlikely to succeed. If the substitution vectors are perfectly random, the one-time pad is unbreakable.[8] In order to resist statistical attacks, the key must be longer than the sum total of all plaintexts to be encrypted, and the letters of the key must be randomly chosen. If we are to encrypt plaintexts of sizes 3K bytes, 5K bytes, and 9K bytes, the key must be at least 17K bytes, so that there is no need to cycle back to the first letter of the key.

Historically, during World War II, British spies airlifted behind enemy lines were equipped with encryption keys written on silk scarves. After using each portion of the key, the spies were ordered to burn the portion of the scarf used; no part of the key was re-used.[9]

2.4.3 Exercises

1. Use the polyalphabetic cipher of Figure 2.20 to
   (a) encrypt the plaintext `HAPPYBIRTHDAY`.
   (b) decrypt the ciphertext `VFDFLOAFXD`.
2. Try to construct a random number generator.

2.5 The Vigenere cipher

We now turn to a famous polyalphabetic cipher known as the *Vigenere* cipher.[10] The Vigenere cipher is a polyalphabetic cipher in which the component substitution ciphers are all shift ciphers. The shift ciphers are determined by a key word, or key, for the cipher. A Vigenere cipher is often described in the form of a table, in which the letters labeling the columns represent letters of the key, and the letters labeling the rows represent letters of the plaintext. Then to encrypt a letter of the plaintext, the ciphertext letter is selected from the letter in the table at the selected row and column. For example, if the letter of the key is 'G' and the plaintext letter is 'Z', then the ciphertext letter is 'F'. After the last letter of the key is used, it cycles back to the first letter, as described above for general polyalphabetic ciphers. The Vigenere table is shown in Figure 2.22 in which the plaintext letter is P and the key letter is S, selecting H as the ciphertext letter.

[8] Perfectly random data cannot be generated by a computer, which is a deterministic machine. Every code can be broken.

[9] Silk was used because it leaves very little ash when burned. See the historical account *Between Silk and Cyanide* by Leo Marks.

[10] This cipher was acually invented in the sixteenth century by the Italian cryptologist, Giovan Bellaso, but was later attributed, incorrectly, to his contemporary, Blaise de Vigenere, of France. Source: Wikipedia

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

Figure 2.22: The Vigenere table. A row is selected by a plaintext letter; a column is selected by a key letter. The letter in the selected row and column is the ciphertext letter. If the plaintext letter is P, and the key letter is S, the ciphertext letter is H

```
plaintext =  G O O D M O R N I N G
key =        C R Y P T O C R Y P T
-----------------------------------
ciphertext = I F M S F C T E G C Z
```

Figure 2.23: Encryption of the plaintext GOODMORNING with the key word CRYPTO using the Vigenere cipher

```
plaintext =  G  O  O  D  M  O  R  N  I  N  G
  code =     6 14 14  3 12 14 17 13  8 13  6
key =        C  R  Y  P  T  O  C  R  Y  P  T
 +code =     2 17 24 15 19 14  2 17 24 15 19  (mod 26)
-----------------------------------------------
 code =      8  5 12 18  5  2 19  4  6  2 25
ciphertext = I  F  M  S  F  C  T  E  G  C  Z
```

Figure 2.24: Encryption of the plaintext GOODMORNING with the key word CRYPTO using the Vigenere cipher with modular arithmetic

To decrypt a ciphertext, we use the reverse procedure. In the column selected by the key letter, look for the ciphertext letter. The row in which it occurs is the plaintext letter. In Figure 2.22 if the key letter is S, we look for the ciphertext letter, H, in the column for S. It is found in the row for P, so the plaintext letter is P.

As an example, we encrypt the plaintext GOODMORNING using the key CRYPTO, as shown in Figure 2.23. The ciphertext is: IFMSFCTEGCZ. Note that when reaching the end of the key, we wrap around to the beginning, so the key is, in effect, CRYPTOCRYPTOCRYPTOCRY... for a length as long as needed to encrypt the entire plaintext.

2.5.1 Vigenere with modular arithmetic

The Vigenere cipher can be specified with modular arithmetic, eliminating the need for Figure 2.22. Encryption is done by adding the plaintext letter code and the key letter code (mod 26). In the formula shown below x is a letter from the plaintext, k is the corresponding letter from the key word, and y is the corresponding letter from the ciphertext.

y = x + k (mod 26)

Our previous example is then done as shown in Figure 2.24 in which we again use the key word CRYPTO. In this example we add the codes for the letters to get the code for the ciphertext.

To decrypt we take our encryption formula and solve for x:

x = y - k (mod 26)

Continuing with the same example decryption of the ciphertext IFMSFCTEGCZ

```
ciphertext = I  F  M  S  F  C  T  E  G  C  Z
  code =     8  5 12 18  5  2 19  4  6  2 25
key =        C  R  Y  P  T  O  C  R  Y  P  T
 -code =     2 17 24 15 19 14  2 17 24 15 19   (mod 26)
-----------------------------------------------
 code =      6 14 14  3 12 14 17 13  8 13  6
plaintext =  G  O  O  D  M  O  R  N  I  N  G
```

Figure 2.25: Decryption of the ciphertext IFMSFCTEGCZ with the key word `CRYPTO` using the Vigenere cipher with modular arithmetic

is shown in Figure 2.25.

Note that when we subtract 5-17, the result -12 is congruent to 14 (mod 26).

2.5.2 Probabilities

Before discussing cryptanalysis of the Vigenere cipher, we need to establish some concepts from mathematical *probability*. Think of probability as a way of expressing the likelihood of an event, using mathematics. For example, if there are `n` possible equally likely outcomes of an experiment, the probability of any one outcome is `1/n`. Some examples are shown below:

- The probability of a result of 3, when rolling a die with 6 sides is 1/6.
- The probability of an even result, when rolling a die with 6 sides is 3/6 or 1/2.
- When rolling two dice, there are 6 * 6 = 36 possible outcomes. The probability of a result of 12 is 1/36.
- When rolling two dice, there are 6 ways of rolling a 7:
  1,6
  6,1
  2,5
  5,2
  3,4
  4,3
  Thus the probability of rolling a 7 is 6/36 = 1/6.

Combinatorics is the area of mathematics which deals primarily with counting. Often we would like to know "How many different ways can something be done?". We denote the number of combinations of things, taken r at a time as C(n,r). It can be shown that
$C(n,r) = \frac{n!}{r!(n-r)!}$
As an example we ask how many ways can we select 3 objects from a group of 5 distinct objects?
$C(5,3) = \frac{5!}{3!(5-3)!} = \frac{5 \cdot 4}{2!} = 10$

Choosing three distinct letters from [a,b,c,d,e] we have 10 possibilities:

```
[a,b,c]
[a,b,d]
[a,b,e]
[a,c,d]
[a,c,e]
[a,d,e]
[b,c,d]
[b,c,e]
[b,d,e]
[c,d,e]
```

Mathematically, we are choosing 3 of 5, so r = 3 and n = 5. $C(5,3) = 10$

In the cryptanalysis of the Vigenere cipher we will focus on:
$C(n,2) = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$

2.5.3 Index of coincidence

In this section we will need to discuss the probability of selecting two matching items from a collection of items. For example, if we have the text: ABABCA, and we choose two letters from this text, there are 15 ways this can be done:
$AB, AA, AB, AC, AA, BA, BB, BC, BA, AB, AC, AA, BC, BA, CA$
How many times did we select identical letters? Foure times: AA, AA, BB, AA. Thus the probability of selecting identical letters from the text $ABABCA$ is 4/15, or about 2.7%. We define the *Index of coincidence* of a given text to be the probability of selecting two letters the same. For the text, ABABCA, the index of coincidence is 26.7%.

In general to find the index of coincidence for a given text, we should first find the distribution of letters in that text:
n_A is the number of A's in the text
n_B is the number of B's in the text
n_C is the number of C's in the text
.
.
.
n_Z is the number of Z's in the text
The probability of selecting two A's will be the number of ways of selecting two A's divided by the total number of ways of selecting two letters. This will be $C(n_A,2)/C(n,2)$ where n is the total number of letters in the text. The probability of selecting two letters the same will be the sum of these probabilities for all 26 letters. This is the *index of coincidence*:

$$I = \sum_{i=A}^{Z} \frac{C(n_i,2)}{C(n,2)}$$

$$\begin{aligned} &= \frac{\sum_{i=A}^{Z} C(n_i, 2)}{C(n, 2)} \\ &= \frac{\sum_{i=A}^{Z} n_i \cdot (n_{i-1})/2}{n \cdot (n-1)/2} \\ &= \frac{\sum_{i=A}^{Z} n_i \cdot (n_{i-1})}{n \cdot (n-1)} \end{aligned}$$

Suppose the text has a flat distribution, i.e. the number of A's equals the number of B's equals the number of C's:
$n_A = n_B = n_C ... = n_Z$
and let s represent that number, i.e.
$s = n_A = n_B = n_C ... = n_Z$
The index of coincidence for this flat distribution is:

$$\begin{aligned} I &= \frac{\sum_{i=A}^{Z} n_i \cdot (n_{i-1})/2}{n \cdot (n-1)/2} \\ &= \frac{26 \cdot s \cdot (s-1)}{26 \cdot s \cdot (26 \cdot s - 1)} \\ &= \frac{s-1}{26 \cdot s - 1} \\ &\approx 1/26 \end{aligned}$$

For large values of s this is a good approximation, thus this would be the index of coincidence for a large amount of purely random text:
$I \approx 1/26 \approx 3.85\%$

What would the index of coincidence be if the text were English words? Figure 2.12 shows a frequency distribution for the letters in a fairly large sample of English text. To find the probability of selecting matching letters, we can weight each letter by its frequency in English text (there are far more many E's than there are Q's, for example). We designate these frequencies from Figure 2.12 $p_A = 8.12\%, p_B = 1.49\%., ... p_Z = 0.07\%$. The probability of selecting an A followed by another A is $8.12\% \times 8.12\%$. The probability of selecting a B followed by another B is $1.49\% \times 1.49\%$.... The probability of selecting two identical letters will be the sum of all these probabilities:
$p_A^2 + p_B^2 + p_C^2 + p_D^2 + ... p_Z^2 \approx 6.5\%$

We will need these two numbers in the cryptanalysis of the Vigenere cipher, and we define the *index of coincidence* to be the probability of selecting two identical letters from a large set of letters, and denote this with the letter `I`. The probability of selecting two identical letters from:

- Random text: $I_R = 3.85\%$
- English text: $I_E = 6.5\%$

Note that if English text were to be encrypted with a shift cipher, the distribution of letters would be the same, but shifted by the displacement of the encryption key. For example, if the key were the letter `C`, each letter would be encrypted as the letter two positions toward the end of the alphabet, as shown in Figure 2.4. Then the scrawl would look like the scrawl of English text, but shifted to the right, as shown in Figure 2.14. It should be clear that the index of coincidence for ciphertext produced by a shift cipher should be the same as for English text:

- Shift cipher: $I_S = 6.5\%$

2.5.4 Cryptanalysis of the Vigenere cipher

The Vigenere cipher is a polyalphabetic cipher, which means that successive letters of the plaintext can be can be encrypted with different key letters. If the length of the key word is 1, the Vigenere cipher becomes a simple shift cipher.

We can visualize the plaintext as being separated into groups by the key word. For example, if the key word is `DOG`, then we can see that:

- The plaintext letters at positions 0,3,6,9,... will be encrypted by adding `D`.
- The plaintext letters at positions 1,4,7,10,... will be encrypted by adding `O`.
- The plaintext letters at positions 2,5,8,11,... will be encrypted by adding `G`.

Since the length of the key word is 3, we essentially have 3 shift ciphers. It should soon be clear that a longer key word will make the cipher more difficult to break.

To break the enemy's Vigenere cipher, we approach the problem with two steps:

1. Find the key word length
2. Find the key word

Finding the keyword length: The Friedman test

To find the length of the keyword we discuss a strategy first presented by William Friedman in the 1920's. We begin by imagining that the ciphertext is arranged in rows, in which the length of each row is the length of the keyword, as shown in Figure 2.26. In that diagram there are a total of `n` letters in the ciphertext, arranged in rows of length `k`; thus there are `n/k` rows, where `n` is the length of the ciphertext.

We note that all letters in a column of Figure 2.26. have been encrypted by the same letter of the key. As noted above this is like a shift cipher. The scrawl

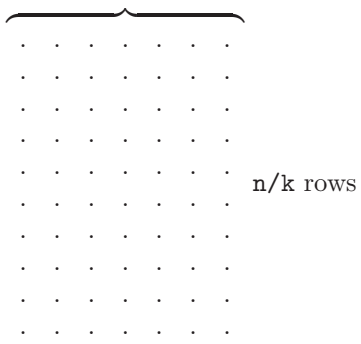


Figure 2.26: Hypothetical arrangement of the n letters of a ciphertext produced with a Vigenere key of length k

of the letters in a column will resemble the scrawl of English text, but shifted by a displacement determined by the key letter. For example, if the first letter of the key is D, then every letter in the first column is encrypted by D, and those letters are essentially the result of a shift cipher. Consequently the incidence of coincidence for the letters in a column will be approximately $I_E = 6.5\%$.

We also note that the letters in a row resulted from an encryption by (mostly) different letters of the key. These letters will have a flat frequency distribution; this is essentially a substitution cipher (and not a shift cipher). Consequently the incidence of coincidence for the letters in a row will be approximately $I_R = 3.85\%$.

This information can be used to find the keyword length. We wish to count the number of ways that we can select two identical letters from the ciphertext (in order to find the incidence of coincidence). We have two cases:

- Choose two letters from the *same* column. How many ways can this be done?
  - There are n/k letters in a column. The number of ways of selecting two letters from one column is:
    $C(\frac{n}{k}, 2) = \frac{\frac{n}{k}(\frac{n}{k}-1)}{2}$
  - Since there are k columns we get the number of ways of selecting letters from the same column:
  - $k \cdot \frac{\frac{n}{k}(\frac{n}{k}-1)}{2} = \frac{n(\frac{n}{k}-1)}{2} = \frac{n(n-k)}{2k}$
  - Since letters in the same column result from a shift cipher, we know that the chance of selecting two identical letters is 6.5%. Thus the number of identical pairs, selected from the same column, will be:

$.065 \cdot \frac{n(n-k)}{2k}$

- Choose two letters from *different* columns. How many ways can this be done?
  - In a single column there are `n/k` letters:
    $\frac{n}{k} \cdot \frac{n}{k}$ pairs from different columns
  - How many pairs of columns are there? There are `k` columns:
    $C(k,2) = \frac{k(k-1)}{2}$ pairs of columns
  - We multiply these to get the total number of ways that two letters can be chosen from different columns:
    $\frac{n}{k} \cdot \frac{n}{k} \cdot \frac{k(k-1)}{2} = \frac{n^2(k-1)}{2k}$
  - Since letters in different columns result from a substitution cipher, we know that the chance of selecting two identical letters is 3.85%. Thus the number of identical pairs, selected from different columns, will be:
    $.0385 \cdot \frac{n^2(k-1)}{2k}$

- We can now find an expression for the index of coincidence of the ciphertext. We get this by adding the number of identical letters selected from the *same* column plus the number of identical letters selected from *different* columns, and divide this by the total number of ways of selecting two letters from the ciphertext:
  $I = \frac{.065 \cdot \frac{n(n-k)}{2k} + .0385 \cdot \frac{n^2(k-1)}{2k}}{\frac{n(n-1)}{2}}$

  This can be simplified by multiplying both the numerator and denominator by 2k:
  $I = \frac{.065 \cdot n(n-k) + .0385 \cdot n^2(k-1)}{nk(n-1)}$

  This can be simplified further by factoring out the `n` in the numerator; it cancels with an `n` in the denominator:
  $I = \frac{.065 \cdot (n-k) + .0385 \cdot n(k-1)}{k(n-1)}$

- This is just an approximation for the index of coincidence as a function of the length of the keyword, `k`.
  - Since the value of I can be calculated from the actual ciphertext, we can solve for `k` by multiplying both sides by $k(n-1)$:
    $I \cdot k(n-1) = .065 \cdot (n-k) + .0385 \cdot n(k-1)$

- To solve for `k` first isolate coefficients of `k`:
  $I \cdot kn - Ik = .065 \cdot n - .065k + .0385 \cdot nk - .0385n$

- And group terms involving k on the left:
  $I \cdot kn - Ik + .065 \cdot k - .0385 \cdot nk = .065n - .0385n$

- And factor out the `k` to get:
  $k \cdot (I \cdot n - I + .065 - .0385 \cdot n) = .065n - .0385n$

- And finally, solve for `k`:
  $k = \frac{.0265n}{I \cdot n - I + .065 - .0385 \cdot n}$

Note that this approximation of the keyword length is not an integer. It would make sense to try the closest integer to this value, or a few of the closest integers, when attempting to find the keyword.

Finding the keyword

Once we have a good guess for the keyword length, we need to find the keyword. First we separate the ciphertext into *cosets*. Each letter at position i in the ciphertext will be assigned to coset $i \pmod{k}$ where k is the length of the keyword. For example, if the length of the keyword is 3, then
the letters at positions 0, 3, 6, 9, ... will be in coset 0
the letters at positions 1, 4, 7, 10, ... will be in coset 1
the letters at positions 2, 5, 8, 11, ... will be in coset 2

Since the letters in a particular coset were all encrypted with the same keyword letter, they should have the same frequency distribution as English. We have essentially a shift cipher for each coset. We can now use the scrawls of the cosets to find the keyword. Having separated the ciphertext into cosets, we display the scrawl of each coset alongside the scrawl of English text (shown in Figure 2.13). By noting the displacement of the cipher scrawl from the English scrawl, we determine the keyword letter for that coset. It will look similar to Figure 2.14.

As always, this may not immediately provide a solution to the cryptanalysis, i.e. the plaintext. It is often the case that we must resort to trial and error. For example, if two letters in a coset have a very high frequency count, it is likely that one of them is an encrypted `E`. It may be necessary to try using the second letter as an encrypted `E` if using the first letter does not produce a solution.

Example of Vigenere cryptanalysis

An example of the cryptanalysis is shown at
`http://cs.rowan.edu/~bergmann/books/crypto`.
The example is a small package of Java classes for encryption/decryption and cryptanalysis.

In that example the USA Declaration of Independence is encrypted with a Vigenere cipher.[11] To perform the cryptanalysis, we:

- Determine the probable key length using the Index of Concidence (Friedman test). If the probable key length is not close to an integer, this may require some trial and error, with key lengths near the probable key length. For example, if the probable key length is 4.2, we would try key lengths of 4, 5, and 3, in that order, until a reasonable decryption is found.
- For each possible key length we then:
  - Find a coset for each letter of the key.
  - Find the distribution of letters for each coset.
  - Choose the most frequent letter of each distribution as the encryption of E, which yields the letters of the key.
  - If a reasonable decryption is not obtained, repeat the previous step for T and A, for each coset individually.

In this example, our strategy correctly finds the key when the key is fairly short, such as: `AXE`. This strategy is weak when there are duplicated letters in the key, such as `CHEESE`. In a case like that the strategy will underestimate the key length significantly.

2.5.5 Exercises

1. Encrypt the plaintext: `WE ATTACK AT DAWN` using a Vigenere cipher with the key word: `APE`.
2. Decrypt the ciphertext: `RXWEPRDHLICI` using a Vigenere cipher with the key word: `APE`.
3. Write a computer program to encrypt and decrypt strings of uppercase letters with a Vigenere cipher.
4. Attempt to decrypt the ciphertext, vigenereCipherText.txt, in the repository for this book.
   (a) Find the length of the keyword using the index of coincidence of the ciphertext (Friedman's procedure).
   (b) Attempt to find the keyword using the cosets. It is an actual English word.
5. Write a computer program to crack a Vigenere ciphertext using the strategy outlined in the previous problem.

[11] We actually use the Declaration of Independence, concatenated with itself several times, to generate enough data for a statistical attack to succeed.

2.6 Key distribution

Throughout the history of cryptology, and even today, one of the most critical issues facing secure communication is the problem of *key distribution*.[12] Simply stated, you have a secret key which you will use to encrypt transmissions, and you would like to share it with your friends, but not your enemies. If you have neither a secure communications channel (this is what you are trying to establish) nor a physical means of distributing your key, such as a trusted courrier, you face a serious hurdle. Moreover, even if you are able to distribute the key to your friends, the more it is used to encrypt plaintext, the more likely your enemies will be able to use statistical methods, such as the Friedman test, to obtain your key. You are then faced, once again, with the original problem of key distribution.

In chapter 6 we will see some modern solutions to this problem. Prior to the computer age, an example of a solution to the key distribution problem is:

> Buy the New York Times every day. On page 5 of the first section, go to the third complete paragraph of the second column. Take the first 10 words of that paragraph as the key for the day.

This enables you to use a different encryption key every day, which would discourage statistical attacks.

2.6.1 Exercises

1. Explain why it is helpful to use a different encryption key every day.

2.7 Historic ciphers and events

2.7.1 Enigma

Prior to the computer age one of the most successful cipher devices was the German Enigma machine, an electro-mechanical device. Originally developed for commercial use in the German business community, it was soon acquired, and adapted, by the German military. It came to play an important role in World War II.

An Enigma machine is shown in Figure 2.27.[13]

It consists of:

- 26 display lights at the top for data output
- 3 rotors, to scramble the letters of a plaintext
- 26 alphabetic keys for data entry

[12] Here we use the word "distribution" in a different sense than it was used in the section on cryptanalysis, to mean a frequency distribution.

[13] There are Enigma machines in the USA which still work at the National Cryptologic Museum, in Ft Meade, MD.

Figure 2.27: A German Enigma machine with, from top to bottom, 26 display lights, 3 rotors, 26 alpha keys, and 8 patch cords

- 8 patch cords to scramble information further as it passes through the machine.

Each German officer in the field was also provided with a codebook containing the initial rotor settings for the day.[14] When a communication was received by radio signal, the machine operator would set the rotors to the initial settings, then enter the received ciphertext on the keypad.[15] As he did so, the rotors would change position after each letter, unscrambling the ciphertext, and showing the plaintext letter on the display lights.

When encrypting a message to be transmitted, the operator typed the plaintext on the keypad, and, as he did so, the corresponding letters of the ciphertext appeared on the display lights, which were copied to paper and then transmitted by radio.

The Enigma machine was very successful and widely used by all branches of the German military.[16]

After a few Enigma machines were captured by the Allies, codebreakers working at Bletchley Park[17] near London eventually constructed a machine which was capable of breaking the Enigma codes.[18] The British command took great pains to conceal the fact that they were receiving the German communications daily.

[14] Officers were instructed to destroy the codebooks as well as the machines if in danger of being captured by the Allies. Allied officers were praised and rewarded for capturing a German codebook and/or Enigma machine.

[15] The Enigma machine was strictly alphabetic upper case; no lower case letters, numeric, or other characters, including spaces, were involved.

[16] The German navy used an even more secure version of the Enigma machine with four rotors.

[17] Bletchley Park is now a cryptology museum well worth visiting.

[18] One of the key people at Bletchley Park was the mathematician Alan Turing, considered by many to be the father of Computer Science. A biography well worth reading is by Andrew Hodges.

2.7.2 Breaking the Japanese code - a known plaintext attack

In chapter 1 we described various kinds of attacks which are included in cryptanalysis. One such attack is a known plaintext attack. In a known plaintext attack the cryptanalyst has somehow obtained a plaintext and the corresponding ciphertext.

Perhaps the most famous example of a known plaintext attack occurred in the Pacific theatre of World War II. The Americans were fairly successful in breaking the Japanese encrypted communications; however, certain important aspects, such as location names, were coded further for security. One such code word was used for Midway Island, where part of the American fleet was stationed. The Americans suspected an imminent attack on that fleet. They sent out a message in the clear: "Midway is low on water." when in fact Midway was not low on water. A few days later the Americans intercepted a Japanese message to the effect that Midway was low on water, and the Americans were now certain of the code word for Midway; they knew when and where the Japanese were planning to attack. The ensuing battle of Midway was a clear victory for the American naval forces, and is judged by many historians to be the turning point in the war in the Pacific.

Chapter 3

Private Key Algorithms

At this point we enter the age of digital computers (circa 1960). Most communication is now digital, i.e. we can think of a message as a binary sequence of 1's and 0's. We will refer to the *unencrypted* message as the plaintext. The plaintext could be ordinary text, or it could be a marked-up document (such as a Microsoft Word document), a PDF document, a digital image, a sound or video clip, etc. What we call the plaintext is not necessarily *plain text*, and what we call ciphertext is not necessarily *text*, i.e. printable characters. Anything which is digital can be encrypted using the algorithms presented here.

In this chapter we present *private key* algorithms used for confidentiality, i.e. encryption/decryption. These algorithms are often called *symmetric* algorithms because encryption and decryption are accomplished using the same key, as opposed to public key algorithms which use different keys for encryption and decryption (see chapter 6).

3.1 Boolean algebra and bit manipulation

Boolean algebra[1] is a mathematical system involving calculations which operate on only two possible values. These values may be thought of as True and False. In computer science we use 1 and 0 instead of True and False, respectively. Remember:
1 = True
0 = False
For a more complete discussion of boolean algebra the reader is referred to the open source textbook on Computer Organization:
`http://cs.rowan.edu/~bergmann/books`

[1] Named for the British logician, George Boole

Logic			Computer Science		
x	y	x OR y	x	y	x + y
False	False	False	0	0	0
False	True	True	0	1	1
True	False	True	1	0	1
True	True	True	1	1	1

Figure 3.1: The boolean OR operation, shown using logic notation and computer science notation

3.1.1 Boolean operations

There are four primary boolean operations which we will discuss here: OR, AND, NOT, and Exclusive OR (XOR). In cryptology we use the Exclusive OR operation extensively, for reasons to be shown below.

OR

The boolean OR operation is defined in Figure 3.1. The result of the OR operation is True (1) if and only if either (or both) operands are True (1). The result of the OR operation is False (0) if and only if *both* operands are False (0). Note that in computer science we use the plus operator (+) for the boolean OR operation. In this context it does *not* represent addition of numbers. [2]

An example of a true logical sentence using the OR operation is:
"Elephants are pink OR 3+4 = 7"

A few identities for the boolean OR operation are:
x + 1 = 1
x + 0 = x
x + x = x
These identities will always be true, regardless of the value of x.

AND

The boolean AND operation is defined in Figure 3.2. The result of the AND operation is True (1) if and only if *both* operands are True (1). The result of the AND operation is False (0) if and only if *either* or *both* operands are False (0). Note that in computer science we use the raised dot operator (·) for the boolean AND operation. In this context it does *not* represent muliplication of numbers. [3]

An example of a false logical sentence using the AND operation is:
"Elephants are pink AND 3+4 = 7"

[2] In programming languages such as C++ or Java, The || symbol is used for logical OR, and the | symbol is used for bitwise OR.

[3] In programming languages such as C++ or Java, The && symbol is used for logical AND, and the & symbol is used for bitwise AND.

Logic			Computer Science		
x	y	x AND y	x	y	x · y
False	False	False	0	0	0
False	True	False	0	1	0
True	False	False	1	0	0
True	True	True	1	1	1

Figure 3.2: The boolean AND operation, shown using logic notation and computer science notation

Logic		Computer Science	
x	NOT x	x	x'
False	True	0	1
True	False	1	0

Figure 3.3: The boolean NOT operation, which has only one operand, shown using logic notation and computer science notation

A few identities for the boolean AND operation are:
$x \cdot 0 = 0$
$x \cdot 1 = x$
$x \cdot x = x$
These identities will always be true, regardless of the value of x.

NOT

The boolean NOT operation is a *unary* operation - it has only one operand, and is defined in Figure 3.3. The result of the NOT operation is True (1) if and only if its operand is False (0). The result of the NOT operation is False (0) if and only if its operand is True (1). In computer science we use an overbar ($\overline{x}$) or a prime (x') to represent the NOT operation. [4]

An example of a true logical sentence using the NOT operation is:
"NOT (Elephants are pink)"
An identity for the boolean NOT operation is:
x" = x
This identity will always be true, regardless of the value of x.

Exclusive OR

The boolean Exclusive OR (XOR) operation is defined in Figure 3.4. The result of the XOR operation is True (1) if and only if the two operands are *different*. The result of the XOR operation is False (0) if and only if the two operands are *the same*. Note that in computer science we use the circled plus operator ($\oplus$)

[4] In programming languages such as C++ or Java, The ! symbol is used for logical NOT and $\sim$ is used for bitwise NOT.

Logic			Computer Science		
x	y	x XOR y	x	y	x ⊕ y
False	False	False	0	0	0
False	True	True	0	1	1
True	False	True	1	0	1
True	True	False	1	1	0

Figure 3.4: The boolean Exclusive OR (XOR) operation, shown using logic notation and computer science notation

for the boolean XOR operation.[5] The XOR operation is called *exclusive* OR because it *excludes* the case where both operands are True from the set of True cases in Figure 3.4.

An example of an incorrectly formed logical English sentence using the the word OR in the sense of Exclusive OR is:
"Only those students who have completed Calculus I or Discrete Math may sign up for Introduction to Programming".
This would imply that those students who have taken both Calculus I and Discrete Math may not sign up for Introduction to Programming.

The boolean OR operation described above is sometimes referred to as *inclusive* OR, to distinguish it from *exclusive* OR.

A few identities for the boolean XOR operation are:
x ⊕ 0 = x
x ⊕ 1 = x'
x ⊕ x = 0
x ⊕ x' = 1
These identities will always be true, regardless of the value of x.

3.1.2 Associativity of Exclusive OR

The boolean operations with two variables described above (OR, AND, XOR) are all associative operations. An associative operation, call it `op`, is associative if it satisfies:
`(x op y) op z = x op (y op z)`
If `op` is an associative operation in the expression shown below, it does not matter which operator is evaluated first:
`x op y op z`
Addition and multiplication are associative operations but subtraction and division are not associative.

We can prove that the Exclusive OR operation is associative with a simple truth table. The fact that the Exclusive OR operation is associative, combined with the identities shown above, make it very useful in cryptography.

[5] In programming languages such as C++ or Java, The != symbol is used for logical XOR, and the ∧ symbol is used for bitwise XOR.

Precedence of operations

As we write boolean expressions algebraically, the order of operations is important. We observe the same conventions as most programming languages and mathematics textbooks:

- AND has precedence over OR or XOR:
  $x \cdot y + z = (x \cdot y) + z$
  $x \cdot y \oplus z = (x \cdot y) \oplus z$
- NOT has precedence over AND:
  $x \cdot y' = x \cdot (y')$

Parentheses can always be used to clarify the desired order of operations.

3.1.3 Bitwise boolean operations

In computer science we often work with blocks of bits at a time, as opposed to a single bit. The boolean operations described here can be applied in *bitwise* fashion, meaning that corresponding bits of two blocks form the operands of a boolean operation. For example:

```
        0 0 1 1
OR      1 0 1 0
--------------
        1 0 1 1
```

Algebraically, we could write this as:
`0011 + 1010 = 1011`

3.1.4 Bit manipulation

Many encryption algorithms use operations on the binary digits (bits) of a plaintext to 'scramble' the data to confuse the enemy. Then the intended recipient would need to 'unscramble' bits in the ciphertext to retrieve the original plaintext. These operations are collectively known as *bit manipulation* operations.

Shifting of bits

In the process of encrypting plaintext (or decrypting ciphertext) some algorithms will *shift* the bits either to the left or right. This shift operation can be applied in either direction (left or right) and is generally applied to segments, or blocks, of the text being manipulated. It is also possible for a shift operation to shift the bits by several positions. For example, if an 8-bit block of bits:
`10110010`
is shifted left by 3 bit positions, the result is:
`10010000`
As bits are shifted off the left end of the block, 0's are shifted into the right end.

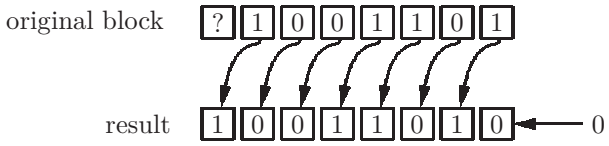


Figure 3.5: Diagram of a left shift operation on an 8-bit block

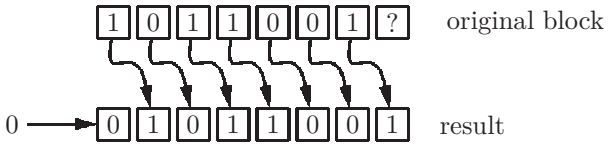


Figure 3.6: Diagram of a right shift operation on an 8-bit block

Another example, the block:
`01100101`
is shifted 1 bit position to the right, resulting in:
`00110010`
because a 0 is shifted in at the left end. A diagram of a left shift operation is shown in Figure 3.5. A diagram of a right shift operation is shown in Figure 3.6.

Related to bit shifting is the operation of bit *rotation*. With a rotate operation, the bit(s) shifted off the end of the block are shifted in at the other end. For example, if the block:
`10110001`
is rotated two bit positions to the left, the result is:
`11000110`

Permutation of bits

A *permutation* is simply a reordering of a list.[6] For example, given the list of numbers `[2,4,3,2,7]`, some permutations are:
`[7,2,3,4,2]`
`[2,2,3,4,7]`
`[3,2,7,2,4]`
A *permutation vector* is a list of indices or positions of a list which define a permutation.[7] The permutation vector `[0,3,2,1,4]` when applied to the list
`[2,4,3,2,7]`
produces the list
`[2,2,3,4,7]`

In cryptography permutations are used to scramble the bits in a block of plaintext (or ciphertext) to confuse the enemy. If we apply the permutation

[6] permutations of English words which are also English words are called *anagrams*.
[7] In computer science the position indices begin with 0.

vector:
`[3,0,7,2,1,5,4,6]` when applied to the 8-bit block:
`01110010`
produces the result:
`10011001`

A permutatation vector which reverses the effect of a permutation is called an *inverse* permutation. An inverse permutation, when applied to the result of a permutation produces the original list or block. Given the permutation vector:
`[3,0,7,2,1,5,4,6]`
the inverse permutation vector is:
`[1,4,3,0,6,5,7,2]`

Here is an algorithm to find the inverse permutation vector:

1. Write the given permutation vector, and its indices in two rows:

3	0	7	2	1	5	4	6
0	1	2	3	4	5	6	7

2. Sort the columns using the top row as the sort field:

0	1	2	3	4	5	6	7
1	4	3	0	6	5	7	2

3. The bottom row is the inverse permutation vector.

Selection of bits

An operation which is related to permutation is the *selection* operation. An algorithm may select bits from a given block to produce a smaller (or bigger) block. For example, if we are working with 8-bit blocks, a selection vector could be:
`S = [1,3,5,6]`.
This would select bits 1,3,5,6, in that order from a given 8-bit block, producing a 4-bit block. Given the 8-bit block:
`B = [0,1,1,0,1,1,0,0]`
The selection shown above would result in the following 4-bit block:
`B[S] = [1,0,1,0]`
Selection can also permute the selected bits:
`B[2,4,0,7,5] = [1,1,0,0,1]`

The size of the result will always equal the size of the selection vector. Unlike permutation vectors, the selection vector may have duplicate indices.
`B[2,4,2,7,4] = [1,1,1,0,1]`

The selection operation may also produce a block which is larger than the given block. This is called an *expansion*:

```
B[2,4,0,7,5,1,3,4,6,0,4,2] = [1,1,0,0,1,1,0,1,0,0,1,1]
```

It should be apparent that the selection operation is a generalization of the permutation operation. Anything that can be done with a permutation operation can also be done with a selection operation.[8]

3.1.5 Exercises

1. Evaluate each of the following boolean expressions involving bitwise operations:
   (a) $0011 + 1010$
   (b) $0011 \cdot 1010$
   (c) $(0011 + 1100) \oplus 1010$
   (d) $(010101 \oplus 111111)'$
2. Prove that the Exclusive OR operation is associative, i.e. prove that $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ for all possible values of x,y,z. (x,y,z represent single bits)
3. Prove each of De Morgan's laws (x and y represent single bits):
   (a) $(x + y)' = x' \cdot y'$
   (b) $(x \cdot y)' = x' + y'$
4. Find a boolean expression which is equivalent to each of the following but which is simpler (involves fewer operations).
   (a) $x \cdot y + x$
   (b) $x'y'z' + x'yz'$
   (c) $x'y'z + x'yz + xy'z + xyz$

   Hint: Use the distributive property:
   $a \cdot (b + c) = a \cdot b + a \cdot c$
5. Given the 8-bit block [0,0,1,0,1,1,1,0], show the result obtained by each of the following shift operations.
   (a) Shift the given block 1 bit to the right
   (b) Shift the given block 3 bits to the left
   (c) Shift the given block 7 bits to the right
6. Given the 8-bit block:
   `B = 01011101`
   Find each of the following permutations

[8] This generalized selection operation is provided as array subscripting in the programming language APL.

(a) `B[7,6,5,4,3,2,1,0]`

(b) `B[2,4,6,0,1,3,7,5]`

(c) `B[7,0,6,1,5,2,4,3]`

7. Find the inverse permutation vector for each of the following permutation vectors

   (a) `[7,6,5,4,3,2,1,0]`

   (b) `[2,4,6,0,1,3,7,5]`

   (c) `[7,0,6,1,5,2,4,3]`

   (d) `[0,1,2,3,4,5,6,7]`

8. Given the 8-bit block:
   `B = 01011101`
   Find each of the following selections

   (a) `B[0,1,2,3,4,5,6,7]`

   (b) `B[2,4,6,0,1,3,7,5]`

   (c) `B[2,4,2,0,1,3,1,5]`

   (d) `B[2,4,3,6]`

   (e) `B[2,4,3,6,5,1,2,7,1,3,4,5,7,6,0,0]`

3.2 Encryption with exclusive OR

The exclusive OR (XOR) operator has properties and identities which make it particularly useful for encryption/decryption:

- The associative property:
  (1) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- (2) $x \oplus x = 0$
- (3) $x \oplus 0 = x$

Suppose we have a digital message, i.e. the so-called plaintext, msg, which we wish to encrypt with a binary key. All we need to do is form the bitwise exclusive OR to obtain the encrypted message, i.e. the ciphertext:
$ciphertext = msg \oplus key$
The recipient will need the same key to decrypt the ciphertext:
$ciphertext \oplus key$
To see how this works, we substitute the formula shown above for the ciphertext:
$(msg \oplus key) \oplus key$
Then we apply the associative property (1):
$msg \oplus (key \oplus key)$

Using the identity (2) we get:
$msg \oplus 0$
Using the identity (3) we get:
msg
which is the original plaintext. This shows that any message encrypted by forming the bitwise exclusive OR with a key can be decrypted by forming the bitwise exclusive OR again using the same key. More briefly:
$msg \oplus key \oplus key = msg$

As an example, suppose the plaintext is the 8-bit block `00110101` and the shared key is `10110111`. The ciphertext is formed with a bitwise exclusive OR:

```
     00110101    = plaintext
XOR  10110111    = key
     --------
     10000010    = ciphertext
```

The ciphertext seems to bear no resemblance to the original plaintext. The recipient will obtain the plaintext (i.e. decrypt) using the same key:

```
     10000010    = ciphertext
XOR  10110111    = key
     --------
     00110101    = plaintext
```

How does this work if the plaintext message is longer than the key? We use the same strategy as that used in the Vigenere cipher. The key wraps around to the beginning to accommodate longer plaintexts. For example, suppose the plaintext message is 21 bits, and the key is only 8 bits:

```
     00110101 10101110 10001    = plaintext
XOR  10110111 10110111 10110    = key
     -------- -------- -----
     10000010 00011001 00111    = ciphertext
```

Decryption uses the same key:

```
     10000010 00011001 00111    = ciphertext
XOR  10110111 10110111 10110    = key
     -------- -------- -----
     00110101 10101110 10001    = plaintext
```

If the key is short, this crypto system is vulnerable to a *brute force* attack. There are only $2^8 = 256$ possible keys of length 8 bits. A computer could easily try all possible keys until a reasonable plaintext is found. A key length of about 300 bits is more typical.[9] Even with a long key, statistical methods such as those used in chapter 2 can be used if there is a substantial amount of ciphertext.

[9] $2^{300} \approx 10^{90}$. It would require a lifetime to try every possible key.

If it is possible, a key with length equal to the total length of plaintext, would be most effective in guarding against attack. This is called a *one-time pad*. The following statements about crypto systems in general are both true (but appear to be contradictory):

- There is no such thing as a perfect crypto system. All codes can be broken.
- An infinite one-time pad with a perfectly random key is a perfect crypto system.

The statements are not contradictory because there cannot be an infinite number of bits in the key, and because computers cannot generate perfectly random numbers.[10]

If we continue to use the same key day after day, our adversaries will accumulate a large quantity of ciphertext, making statisical attacks more likely to succeed. Thus it is wise to distribute new keys to our friends periodically. But how can this be done safely? Without a secure channel to distribute keys, our codes will eventually be broken. This is called the *key distribution problem*, and we address this problem in chapter 6.

This simple, yet effective, process - exclusive OR - for encryption and decryption is universally used in private key cryptographic algorithms. Many industrial-strength standard encryption algorithms will use substantial bit manipulation (permuations, shifting, selection, etc.) to confuse the adversary. However, these algorithms nearly always use exclusive OR with a shared key at some point in the algorithm.

3.2.1 Exercises

1. Using the 8-bit key `0101 1110` and the exclusive OR operation:
   (a) Encrypt the plaintext `0011 0101`.
   (b) Decrypt the ciphertext `0000 1111`.
   (c) Encrypt the plaintext `0010 0`.
   (d) Encrypt the plaintext `0110010111101`.
2. Bob would like to send a diamond ring to Alice, through the US Postal Service. Alice is expecting the ring. The problem is that postal service employees are dishonest and may steal the ring. Bob and Alice have decided to use a security box that can be locked with one or more padlocks. They each have several padlocks, each with its own key, but they cannot share keys, nor meet in person. How can Bob safely send the ring to Alice?
3. Bob and Alice wish to communicate confidential information with each other over the Internet. However, Eve is capable of intercepting their messages. There is no way Bob and Alice can share a cryptographic key, without exposing it to Eve. Bob and Alice have decided to use the following algorithm:

[10] A more correct term for the random numbers generated by computer is *pseudo-random*.

(a) Bob uses the exclusive OR operation to encrypt the message with his own key, and sends the ciphertext to Alice:
$ciphertext_1 = msg \oplus Key_{Bob}$

(b) Alice further encrypts the ciphertext with her own key and sends it to Bob:
$ciphertext_2 = ciphertext_1 \oplus Key_{Alice}$

(c) Bob 'removes' his key by encrypting again, and sends it to Alice:
$ciphertext_3 = ciphertext_2 \oplus Key_{Bob}$
$= (ciphertext_1 \oplus Key_{Alice}) \oplus Key_{Bob}$
$= ((msg \oplus Key_{Bob}) \oplus Key_{Alice}) \oplus Key_{Bob}$
$= msg \oplus Key_{Alice}$

(d) Alice decrypts with her own key to obtain the original message:
$msg \oplus Key_{Alice} \oplus Key_{Alice}$
$= msg$

Have Bob and Alice discovered a simple solution to the key distribution problem? If not, show how Eve can easily discover their keys.

4. Write a computer program to generate a sequence of pseudo-random integers, starting with a given seed.

5. Develop software which will prompt the user for a 16-bit key and:

   (a) Encrypt a plaintext given as a String of characters, producing an array or List of bytes of the same length. Use the exclusive OR operation as described in this section.

   (b) Decrypt a ciphertext given as an array or List of bytes, producing a String of characters. Use the exclusive OR operation as described in this section.

3.3 Data Encryption Standard (DES and SDES)

3.3.1 DES

In the 1970's as more communication was done by digital computers, it became apparent that confidentiality could be ensured with cryptographic techniques. At the time most encryption was done with hardware, but researchers were beginning to implement the algorithms in software as well. At this time it also became apparent that there was a need for the algorithms to be public, with a private key providing the security.

In 1975 the National Bureau of Standards (now known as the National Institute of Standards Technology) adopted an algorithm from IBM named *Lucifer*. This was the first public data encryption standard, and therefore the NIST named it DES.

DES had a 56-bit key, which was considered secure against brute force attacks at the time. It was widely adopted and implemented in both hardware

and software. It continued to be used until the turn of the millenium, at which time the 56-bit key was thought to be vulnerable to brute-force attacks.

Encryption with DES

A diagram of the DES cipher is shown in Figure 3.7. It blocks the plaintext into 64-bit blocks and encrypts one block at a time. DES uses an initial permutation (IP) on the bits of the plaintext. Then a series of *rounds* is used to manipulate the result of the initial permutation. In each round the input is split into a left and right half, each of which is 32 bits. Each round involves a function, *f*, which manipulates the right half (32 bits) of the result of the previous round, using a selected portion (48 bits) of the key. The result of the f function is used as an operand for an exclusive OR operation; the other operand is the left half of the previous round. The result of the exclusive OR is the right half of the input to the next round. After the 16 rounds are complete, the result is used as input to the inverse of the IP permutation (IP^{-1}), producing the ciphertext. Figure 3.7 is known as a *Feistel* network. In order to derive the decryption algorithm, we will describe the operation of each round algebraically:

- The left half of round `n` is the same as the right half of round `n-1`:
  $Left_n = Right_{n-1}$
- The right half of round `n` is the result of the exclusive OR of the left half of round `n-1` with the result of the f function applied to the right half of round `n-1` and selected bits of the key for round `n`:
  $Right_n = Left_{n-1} \oplus f(Right_{n-1}, K_{n-1})$

We should emphasize that DES is a *public* algorithm; the permutation vector, the sub-key selections, the f function, the Feistel network, everything is publicly known, except for the 56-bit key.

Decryption with DES

The recipient would then decrypt the ciphertext using the reverse of the process shown in Figure 3.7. It would begin by applying the IP permutation, then rounds 15, 14, 13,... 1 in that order. We can use the algebraic equations from the section on encryption, above, to derive the decryption formulas. We will need to derive formulas for the left and right halves of round `n-1` as a function of round `n`, since we are working backwards.

- We begin with the second encryption formula:
  $Right_n = Left_{n-1} \oplus f(Right_{n-1}, K_{n-1})$
- Form the exclusive OR of both sides with $Left_{n-1}$:
  $Right_n \oplus Left_{n-1} = Left_{n-1} \oplus f(Right_{n-1}, K_{n-1}) \oplus Left_{n-1}$
- Since the exclusive OR operation is associative and $x \oplus x = 0$ *and* $x \oplus 0 = x$, we are left with:
  $Right_n \oplus Left_{n-1} = f(Right_{n-1}, K_{n-1})$

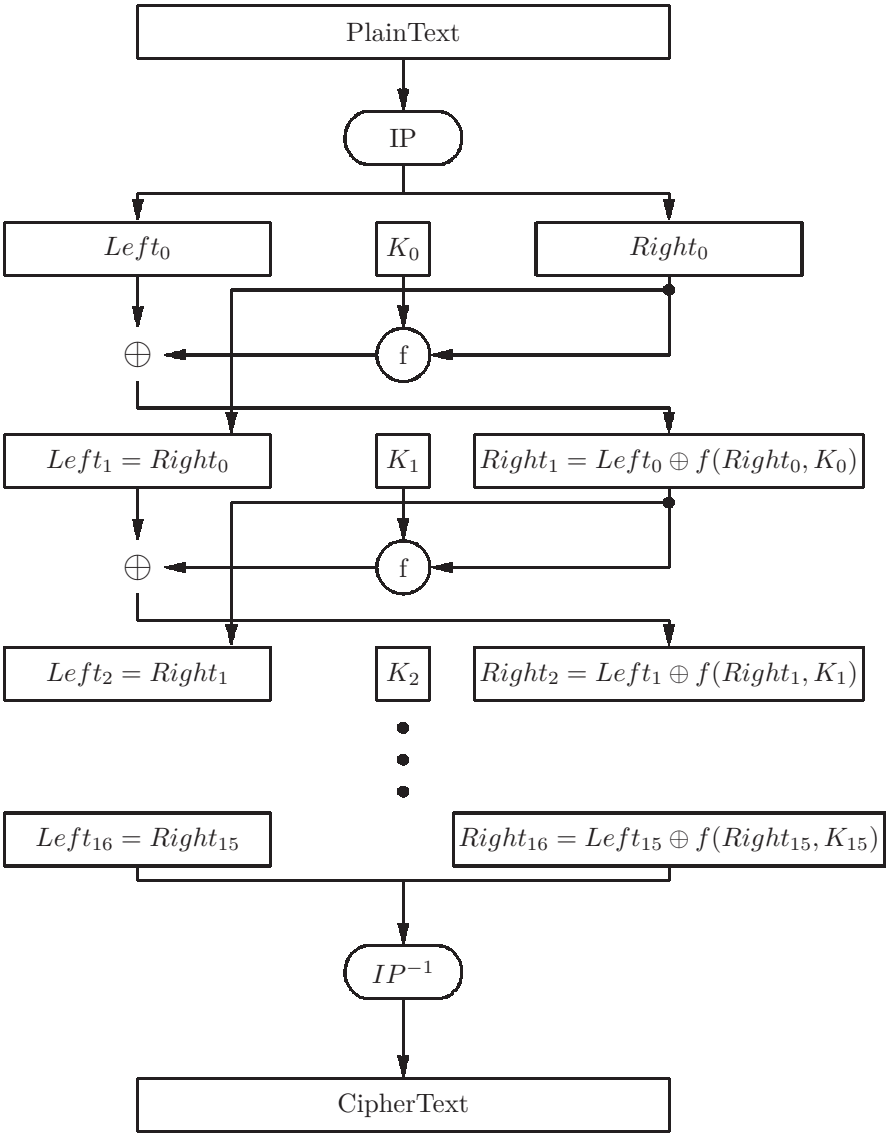


Figure 3.7: Diagram of the DES cipher

- Form the exclusive OR of both sides with $Right_n$:
  $Left_{n-1} = Right_n \oplus f(Right_{n-1}, K_{n-1})$
- From the first encryption formula above, we know that we can substitute $Left_n$ for $Right_{n-1}$:
  $Left_{n-1} = Right_n \oplus f(Left_n, K_{n-1})$
- This formula, together with the first encryption formula from above, give us formulas for the left and right halves of round `n-1` as a function of the left and right halves of round `n`. If we know the left half and we know the right half, then we know the entire value for round `n-1`:
  $Left_{n-1} = Right_n \oplus f(Left_n, K_{n-1})$
  $Right_{n-1} = Left_n$

AES

DES was ultimately replaced in 2001 by an algorithm known as AES (Advanced Encryption Standard)[11]. The disadvantage of a long key is that it increases the time required for encryption or decryption. The developers of AES recognized the fact that computers would be faster in the future.[12] A key considered long, and secure, today would be too short a few years hence. For this reason AES was designed with the option to choose from three key lengths: 128 bits, 192 bits, and 256 bits.

SDES

A simplified version of DES, named SDES (Simplified DES), was published in the journal *Cryptologia* in 1970. SDES is strictly used for pedagogic purposes; it is not recommended for security in real systems.

SDES features the same Feistel network that we saw in DES, but there are only two rounds. A diagram for SDES encryption is shown in Figure 3.8.

In this section we give a complete description of the sub-key selections and the f function, which are similar to those of DES. For convenience SDES takes as input an 8-bit plaintext and a 10-bit key, to produce an 8-bit ciphertext. It processes the complete message in 8-bit blocks, one block at a time. If k is the 10-bit key, the two sub-keys are defined by the selections:

- `K0 = k[0,6,8,3,7,2,9,5]`
- `K1 = k[7,2,5,4,9,1,8,0]`

The inital permutation vector is:
`IP = [1,5,2,0,3,7,4,6]`

[11] Originally called Rijndael, from the Netherlands

[12] Moore's Law: The density of components in an integrated circuit will double every two years because of improving technology. As components are placed closer, the time for data to move between components is lessened, and the processing speed of the computer is increased. Predicted by Gordon Moore in 1965, it has held true until about 2010-2020.

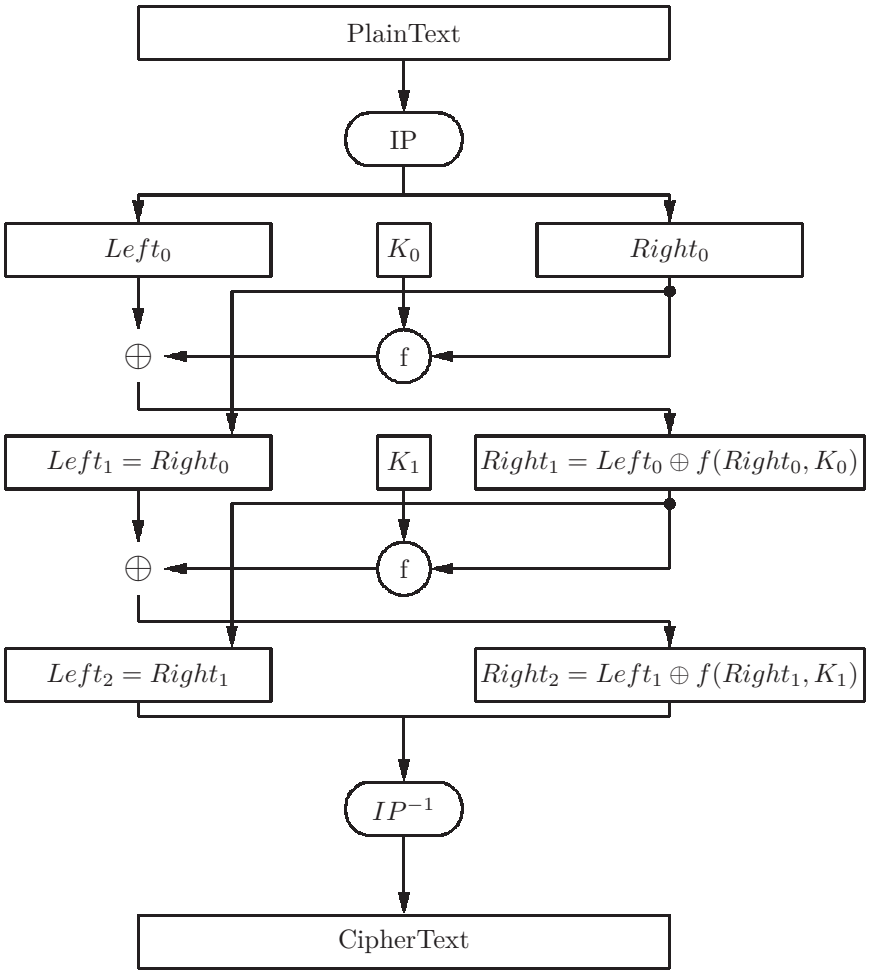


Figure 3.8: Diagram of the SDES cipher

x_0	x_1	x_2	x_3	$s_0(x)$	$s_1(x)$
0	0	0	0	01	00
0	0	0	1	11	10
0	0	1	0	00	01
0	0	1	1	10	00
0	1	0	0	11	10
0	1	0	1	01	11
0	1	1	0	10	11
0	1	1	1	00	11
1	0	0	0	00	11
1	0	0	1	11	10
1	0	1	0	10	00
1	0	1	1	01	01
1	1	0	0	01	01
1	1	0	1	11	00
1	1	1	0	11	00
1	1	1	1	10	11

Figure 3.9: Definition of the two s-box functions for SDES encryption.

SDES will also use:

- An expansion permutation, EP, which expands and permutes a 4-bit value to 8 bits:
  `EP = [3,0,1,2,1,2,3,0]`
- A 4-bit permutation, P4, which permutes a 4-bit value:
  `P4 = [1,3,2,0]`
- Two 4-bit s-boxes. Each s-box performs a substitution on a 4-bit value, producing a 2-bit result. The two s-boxes, $s_0(x)$ and $s_1(x)$ are defined in Figure 3.9.
- A *round function*, $f(x, k)$. This function takes as arguments, a 4-bit value, x, and an 8-bit sub-key. It uses the tools defined above to produce a 4-bit result:
  $f(x, k) = P4(s_0(L(k \oplus EP(x)))||s_1(R(k \oplus EP(x))))$
  In this function:
  - The || operator performs concatenation of bits:
    `0110 || 1010 = 01101010`
  - L and R represent the left and right halves[13] of a string of bits, respectively. If `x = 01101110`, then:
    `L(x) = 0110`
    `R(x) = 1110`

[13] We always use L and R on bit strings of even length.

3.3.2 Decryption with SDES

To derive the decryption algorithm corresponding to a given encryption algorithm, we need to 'reverse' the effects of the encryption algorithm to arrive at the original plaintext. With the exclusive OR operation, this was easy because exclusive OR is its own inverse. For SDES it is more detailed.

Imagine that we are beginning at the bottom of Figure 3.8 and working in an upward direction to obtain the original plaintext. How can we reverse the sequence of computations? To reverse the application of IP^{-1} we simply use IP, so that is the first step. To reverse the effect of each round, we use the formulas shown above in the section on decryption with DES. Note that it is not necessary to find an inverse of the function $f(x, k)$, where x is a Left (or Right) half, and k is a subkey. Finally we arrive at the IP permuation at the top, so to reverse that we use IP^{-1}.

3.3.3 Exercises

1. Show a diagram similar to Figure 3.7 for the decryption algorithm of DES.
2. The initial permutation vector for SDES is:
   `IP = [1,5,2,0,3,7,4,6]`
   Find the inverse permutation vector, IP^{-1}.
3. Refer to the function, f, in the description of SDES. Find $f(0011, 10010101)$.
4. Using the SDES algorithm, show how to encrypt the plaintext `00100100`, with the key `0111111101`.
5. Show a diagram similar to Figure 3.8 for the decryption algorithm of SDES.
6. Using your solution to the previous exercise, show how to decrypt the ciphertext produced in the exercise previous to that. Use the same key that was given for encryption.
7. Implement the SDES algorithm in software.

3.4 Blocking Modes

In all cryptographic systems we will need the ability to deal with large quantities of data. Most algorithms, however, can deal with a limited amount of data at a time. This is called a data *block*. The size of a block is the number of bits in the block. Thus if a message consisting of 256 bits is to be encrypted by an algorithm that uses an 8-bit block, the message would be encrypted (or decrypted) by applying the algorithm 32 times, since $256/8 = 32$.

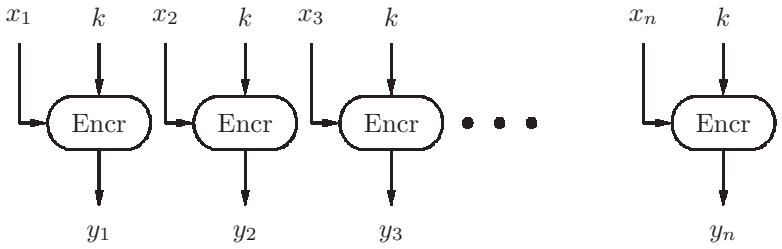


Figure 3.10: Diagram of the Electronic Codebook (ECB) blocking mode. There is no interaction between successive blocks.

If the message length is not a multiple of the block size, additional bits are *padded* at the end of the last block to form a full block. For example, if the message length is 517 bits, and the block size is 8 bits, an extra 5 bits would be padded to the end of the last block to form 65 complete blocks. The values of the padded bits are generally not all 0's nor all 1's, but some string of bits determined by the algorithm.[14] Padding algorithms are discussed later in this chapter.

The blocks of input to the algorithm can be processed separately and independently. More commonly there is interaction between successive blocks, which is used to defend against statistical attacks. These interactions are called *blocking modes* and are described below.

3.4.1 Electronic Codebook Blocking (ECB)

Encryption

ECB blocking is the simplest to describe. It involves no interaction between successive blocks. When encrypting the plaintext, each block of plaintext is encrypted independently, producing a block of ciphertext. During decryption, the process is reversed and as each block of ciphertext is decrypted a block of plaintext is produced, again with no interaction between successive blocks. Algebraically, if x_i is the i^{th} block of plaintext, and y_i is the i^{th} block of ciphertext, k is the key, and $Encr$ represents the encryption function, then:
$y_i = Encr(x_i, k)$
A diagram of ECB blocking is shown in Figure 3.10.

Decryption

For decryption the process is reversed, so:
$x_i = Decr(y_i, k)$
where $Decr$ represents the decryption function.

[14] If the padded bits were all the same, the cryptanalyst (i.e. the enemy) might be able to determine the message length.

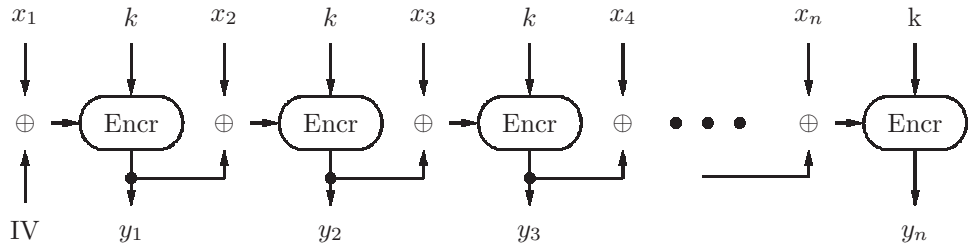


Figure 3.11: Diagram of the Cipher Block Chaining (CBC) blocking mode. The input to each stage is the exclusive OR of the plain text,x, and the ciphertext,y, of the previous stage. The key is k.

3.4.2 Cipher Block Chaining Blocking (CBC)

Encryption

CBC blocking involves interaction between blocks during encryption. The input to the encryption function is the exclusive OR of the plaintext with the ciphertext of the *previous* block. For the first block, there is no previous block, so an *initial vector*, IV, a constant, public block of bits is used instead.[15] Algebraically:

$$\begin{aligned} y_1 &= Encr(IV \oplus x_1, k) \\ y_i &= Encr(x_i \oplus y_{i-1}, k) \end{aligned}$$

A diagram of CBC blocking is shown in Figure 3.11.

Decryption

To derive the decryption formulas, we will need to revise our notation so that the Encryption and Decryption functions have just one argument. To do that we use the key, k, as a subscript on the function rather than an argument:

$$\begin{aligned} Encr(x, k) &= Encr_k(x) \\ Decr(y, k) &= Decr_k(y) \end{aligned}$$

so our encryption formulas become:

$$y_1 = Encr_k(IV \oplus x_1) \quad (3.1)$$

$$y_i = Encr_k(x_i \oplus y_{i-1}) \quad (3.2)$$

Starting with the encryption formulas we derive the decryption formulas by solving for x. Note that encryption and decryption are inverse functions. For

[15] The initial vector is sometimes omitted, in which case there is no exclusive OR at the first block.

the first block, apply the decryption function to both sides of equation 3.1:

$$\begin{aligned} Decr_k(y_1) &= Decr_k(Encr_k(IV \oplus x_1)) && (3.3)\\ &= IV \oplus x_1 && (3.4)\end{aligned}$$

because *Decr* and *Encr* are inverse functions.

Form the exclusive OR of both sides of equation 3.4 with IV to solve for x_1:

$$Decr_k(y_1) \oplus IV = x_1 \quad (3.5)$$

For the remaining blocks apply the decryption function to both sides of equation 3.2:

$$\begin{aligned} Decr_k(y_i) &= Decr_k(Encr_k(x_i \oplus y_{i-1})) && (3.6)\\ &= x_i \oplus y_{i-1} && (3.7)\end{aligned}$$

Form the exclusive OR of both sides of equation 3.7 with y_{i-1} to solve for x_i:

$$Decr_k(y_i) \oplus y_{i-1} = x_i \quad (3.8)$$

3.4.3 Cipher Feedback Mode (CFB)

Encryption

Cipher Feedback Mode (CFB) offers even more interaction between blocks, as compared with other blocking modes. It involves two temporary double-blocks of 2n bits each, which we call z and *temp*.[16] For each block of plaintext, x_i, we form the exclusive OR with the left half of the double-block, *temp*, and this is the block of ciphertext, y_i. To compute the next double-block, z_i, the left half is taken from the right half of *temp*, and the right half is taken from y_i. The double-block z_i forms the input to the encryption algorithm, $Encr_k$, for which the input and output are double-blocks, and which uses the key, k. The output of $Encr_k$ is stored in the *temp* double-block for the next block of input. The initial double-block, z_0 is a given, public, Initial Vector, IV.

For the encryption algorithm, $Encr_k$, the input and output are double blocks. This means that if your encryption algorithm uses a block size of 128 bits, the plaintext and ciphertext would be processed in 64-bit blocks.

A diagram for the CFB blocking mode is shown in Figure 3.12.

Algebraically, the CFB mode is:

$$\begin{aligned} z_0 &= IV \\ temp &= Encr_k(z_{i-1}) \\ y_i &= x_i \oplus L(temp) \\ L(z_i) &= R(z_{i-1}) \\ R(z_i) &= y_i \end{aligned} \quad (3.9)$$

[16] More generally the temporary blocks are of size m, where $m > n$, but we assume $m = 2n$, for simplicity.

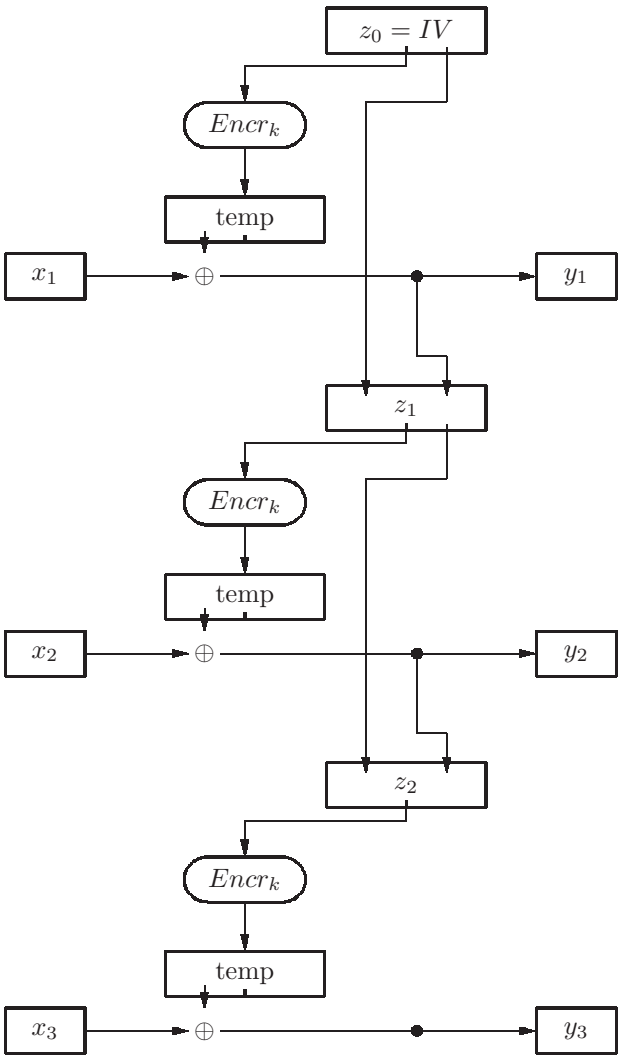


Figure 3.12: Diagram of the Cipher Feedback (CFB) The input to the encryption function *Encr* is a double block, and the output of *Encr* is a double block. The temporary results, `temp` and z_i are also double blocks

in which $L(d)$ is the left half of a double-block d, and $R(d)$ is the right half of a double-block d.

Decryption

We now derive the decryption formulas for the CFB blocking mode. This is easier than the previous blocking modes because the temporary double-blocks can be computed in the same way they were computed for encryption. This is a result of the fact that only equation 3.9 involves a block of plain text, x_i. To compute the decryption formula corresponding to equation 3.9 we solve for x_i by forming the exclusive OR of both sides of equation 3.9 with the left side of the *temp* double-block:

$$\begin{aligned} y_i \oplus L(temp) &= x_i \oplus L(temp) \oplus L(temp) \\ &= x_i \end{aligned}$$

We can now write the decryption formulas for the CFB blocking mode:

$$\begin{aligned} z_0 &= IV \\ temp &= Encr_k(z_{i-1}) \\ x_i &= y_i \oplus L(temp) \\ L(z_i) &= R(z_{i-1}) \\ R(z_i) &= y_i \end{aligned}$$

3.4.4 Output Feedback mode (OFB)

Encryption

Output Feedback Mode (OFB) avoids propagation of errors from one block to the next.[17] As with the CFB mode, it uses a double-block for intermediate results.[18] The Initial Vector (IV), a double block, is the input to the encryption algorithm, $Encr_k$, where k is the encryption/decryption key. The output double-block then forms the input to the encryption algorithm of the next stage. In this manner the sequence of double-blocks, z_i, acts as a stream of different keys for each stage and is sometimes called a *keystream*. In each stage the ciphertext block, y_i, is the exclusive OR of the plaintext block, x_i, with the left half of the temporary block, z_i. A diagram of OFB blocking mode is shown in Figure 3.13.

[17] OFB mode has slightly different descriptions in various sources. We use the version supplied by Menezes.

[18] More generally the temporary blocks are of size m, where n is the block size and $m > n$, but we assume $m = 2n$, for simplicity.

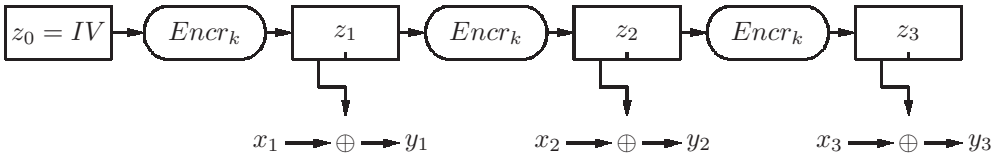


Figure 3.13: Diagram of the Output FeedBack (OFB) blocking mode. The double-block z acts as a stream of different keys.

Algebraically, OFB mode can be described as:

$$z_0 = IV \tag{3.10}$$
$$z_i = Encr_k(z_{i-1}) \tag{3.11}$$
$$y_i = x_i \oplus L(z_i) \tag{3.12}$$

As with the CFB blocking mode, for the encryption algorithm, $Encr_k$, the input and output are double blocks. This means that if your encryption algorithm uses a block size of 128 bits, the plaintext and ciphertext would be processed in 64-bit blocks.

Output Feedback Mode (OFB) avoids propagation of errors from one block to the next.

Decryption

We can now derive the decryption formulas for the OFB blocking mode. The forumulas for the double-block z remain the same as in equations 3.10 and 3.11:

$$\begin{aligned} z_0 &= IV \\ z_i &= Encr_k(z_{i-1}) \end{aligned}$$

To decrypt a block of ciphertext, we need to solve equation 3.12 for x_i. We do that by forming the exclusive OR of both sides of equation 3.12 with the left half of z_i:

$$\begin{aligned} y_i &= x_i \oplus L(z_i) \\ y_i \oplus L(z_i) &= x_i \oplus L(z_i) \oplus L(z_i) \\ &= x_i \end{aligned}$$

3.4.5 Padding

A block cipher works with fixed-size blocks of data. The final block will almost never be a complete block because the plaintext being encrypted will likely have

Figure 3.14: Diagram showing how the last block is padded to build a full block of data for a block cipher

a length, in bits, which is not a multiple of the block size. In this case extra bits must be added to the last block of plaintext to obtain a full block. However, some care must be taken; if we simply pad with all zeros, we may be vulnerable to an attack which focusses on the last block, which has a small amount of true data. Instead, there are *padding* algorithms which produce *nonce* data for the padding. A few examples of standard padding algorithms are `ANSI X9.23`, `ISO 10126` , `PKCS5`, and `ISO 7816-4`. Figure 3.14 diagrams a plaintext with the padding bits shown as question marks to complete the final block.

On the receiving end the recipient must know the padding algorithm to avoid confusing the padding with the plaintext.

3.4.6 Exercises

1. Show a diagram for decryption using ECB blocking mode.
2. Show a diagram for decryption using CBC blocking mode.
3. Show a diagram for decryption using CFB blocking mode.
4. Show a diagram for decryption using OFB blocking mode.
5. Consider a block cipher with an 8-bit block size. The encryption function is:
   $Encr_k(x) = P(x \oplus k)$
   where P(x) is the permutation vector: $P(x) = [4, 2, 6, 7, 0, 3, 1, 5]$
   and the 8-bit key is:
   k = `01101010`
   The initial vector, if needed, is:
   `IV = 1101 0110`
   For the 16-bit plaintext:
   plaintext = `1111 0000 1010 0101`
   find the 16-bit ciphertext.

   (a) Use the ECB blocking mode
   (b) Use the CBC blocking mode
   (c) Use the CFB blocking mode
   Hint: Since the Encryption algorithm requires 8-bit blocks, here 8-bits is a double-block, and the plaintext will be broken into 4-bit blocks.
   (d) Use the OFB blocking mode

Algorithm	Key size	Block size
DES	56 bits	64 bits
AES	128, 192, or 256 bits	128 bits
FEAL	64 bits	64 bits
IDEA	128 bits	64 bits
SAFER	64 bits	64 bits
RC5	Variable	Variable

Figure 3.15: Comparison of private key encryption algorithms

6. Decrypt each of the ciphertexts obtained in the previous exercise to retrieve the original plaintexts.

3.5 Current private key algorithms

In addition to the DES and AES algorithms mentioned above,[19] some commonly used private key (symmetric key) algorithms, with their block size and key size, are shown in Figure 3.15.

3.5.1 FEAL

The Fast data Encipherment ALgorithm (FEAL) was developed at Japan's Nippon Telegraph and Telephone corporation in 1987. It originally had 4 rounds and was later expanded to 8 rounds. It was soon discovered that it could be broken by examining the differences between two or more ciphertexts, leading to the field of *differential cryptanalysis*. This led to the development of FEAL-N, permitting the user to determine N, the number of rounds. FEAL-NX, an extension of FEAL-N, used a 128-bit key.

3.5.2 IDEA

The International Data Encryption Algorithm (IDEA) was first developed in 1991 at ETH Zurich, as a potential replacement for DES. It was used in Phil Zimmerman's PGP package (see chapter 9).

IDEA uses 64-bit blocks and a 128-bit key, with 8 rounds. Each round uses 16-bit subkeys. At the time it was developed it was considered one of the most secure private-key algorithms available. It was not vulnerable to differential cryptanalysis.

3.5.3 SAFER

The Secure and Fast Encryption Routine (SAFER) is actually a family of encryption algorithms developed at the CYLINK corporation. The first version,

[19] The 256-bit version of AES is still considered strong encryption as of 2024

SAFER K-64, was published in 1993 with a 64-bit key. A few years later SAFER K-128 expanded the key to 128 bits. Both of these algorithms utilized four rounds, which soon became vulnerable and were expanded to SAFER+ and SAFER++. SAFER+ was used in the Bluetooth wireless short-range communication standard.

3.5.4 RC5

RC5 was published in 1994 by Ron Rivest of MIT. The 'RC' could be 'Rivest Cipher', or 'Ron's Code', and it was the successor to previous variants proposed by Rivest.

RC5 is known for its simplicity, as well as for its variability in construction. The block size can be 32, 64, or 128 bits. The key size can be variable, up to 2040 bits. The number of rounds, also variable, is typically 12. Although the algorithm itself is a simple Feistel function, it uses a more complex schedule of subkeys for each round. The RC5 algorithm is usually described as RC5-w/r/b, where w is the word (i.e. block) size, r is the number of rounds, and b is the key size (in bytes).

3.5.5 Exercises

1. Describe each of the private key algorithms introduced in this section.

Chapter 4

Discrete Mathematics

In our mathematics classes we study Calculus and related subjects which deal primarily with *continuous* functions. These are functions which are defined at every point along the real number line. In crypto systems we generally work with integers only, consequently we will not be using continuous math. We call non-continuous math *discrete math*. Several chapters in this book will need to use topics from discrete math, and we have decided to collect these topics in one chapter. We do not recommend that this chapter be studied independently. Rather, as each other chapter in the book is studied, refer to the relevant sections in this chapter. These will be listed at the beginning of each chapter that uses discrete math.

4.1 Exponents

In this section we point out some basic properties of exponent notation in mathematics. The notation x^e represents the repeated multiplication of x with itself, e times. For example,
$3^5 = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 243$

Some basic properties of exponents which we will be using throughout this textbook are:

- if $a = b$ then $a^e = b^e$
  Example: if $x = 3$ then $x^5 = 3^5$
  This property has a modular counterpart:
  if $a \equiv b \pmod{m}$ then $a^e \equiv b^e \pmod{m}$
  Mod arithmetic is discussed below

- $a^e \cdot a^d = a^{e+d}$
  Example: $3^7 \cdot 3^5 = 3^{12}$

- $a^e / a^d = a^{e-d}$
  Example: $3^7 / 3^5 = 3^2$

- $a^e \cdot b^e = (a \cdot b)^e$
  Example: $3^5 \cdot 4^5 = 12^5$

- $(a^d)^e = a^{d \cdot e}$
  Example: $(3^4)^5 = 3^{20}$

4.1.1 Exercises

1. Use the basic properties of exponents to evaluate each of the following:

   (a) $2^4 \cdot 2^6$

   (b) $2^7/2^5$

   (c) $99^{17} \cdot 99^5/99^{21}$

2. Fill in the missing information:

   (a) if $w + x = y + z$ then $(w + x)^7 = (y + z)^?$

   (b) $5^6 \cdot 7^6 = ?^6$

   (c) $(17^5)^6 = 17^?$

4.2 Modular arithmetic

Recall that division of integers produces *two results*: a quotient and a remainder. In many programming languages the quotient is obtained with a forward slash:
/
The remainder is obtained with a percent symbol:
% [1]

Thus 17 / 5 is 3, and 17 % 5 is 2.[2] In mathematics we would say either of the following:

- $17\ MOD\ 5 \ = \ 2$
  "Seventeen mod five equals two"

- $17 \equiv 2\ (mod\ 5)$
  "Seventeen is congruent to two, mod five"

[1] The percent symbol seems to make no sense here, and we don't know why it was chosen to represent remainder, except that it was one of the few symbols on the keyboard available.

[2] In this text we will avoid using this operator with negative operands, because of nonstandard implementations of the % operator on negative numbers. See the discussion below on congruence classes for a better way of working with negative integers.

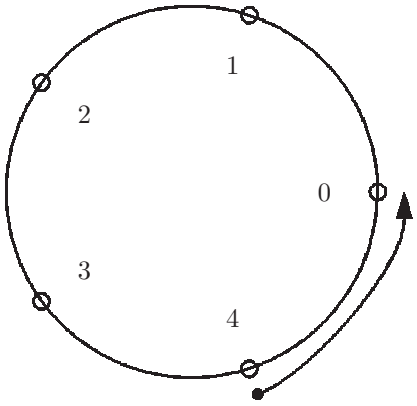


Figure 4.1: A diagram of a mod 5 counter.

4.2.1 Modular counters

A modular counter, working only with non-negative integers, is a counter which goes in circles as shown in Figure 4.1. This happens to be a mod 5 counter, because after 4, the next number is 0. Mathematically, $4 + 1 \equiv 0 \pmod 5$. Thus a mod 5 counter would never go beyond 4:
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 ...

4.2.2 Congruence classes

Viewing the integers on an infinite number line, we can describe subsets of the integers known as *congruence classes*. Here we will be able to include negative numbers. Given any integer on the number line, and a modulus, we can define a *congruence class*. It consists of all the values that can be obtained by adding or subtracting the modulus any number of times.

For example, given the integer 10, and the modulus 7, we can find all the integers in that congruence class, which includes all the integers obtained by adding 7:
17, 24, 31, 38, ...
and all the numbers obtained by subtracting 7:
3, -4, -11, -18, -25, ...
and don't forget the given integer:
10

Thus the congruence class for 10 (mod 7) is the infinite set: {...-25,-18,-11,-4,3,10,17,24,31,38,...}

We are often interested primarily in one integer from a congruence class, the one which is less than the modulus but not negative. In this example that would be 3, as shown in Figure 4.2.

When we say that

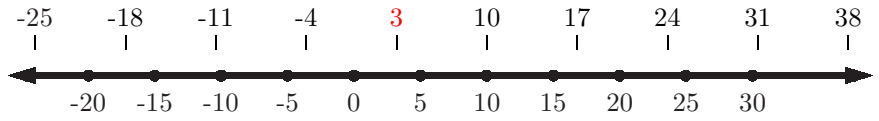


Figure 4.2: A diagram of the congruence class defined by the given integer 10, and the modulus of 7

$x \equiv y \ (mod\ m)$
we mean that x and y belong to the same congruence class, modulo m.

4.2.3 Mod product

An important property of modular arithmetic is: $(a \cdot b)\ MOD\ m\ =\ (a\ MOD\ m) \cdot (b\ MOD\ m)$
In other words, when working mod m, we can reduce the result at any step in the calculation. Thus
$12 \cdot 93 \equiv 2 \cdot 3 \equiv\ 6\ (mod\ 10)$

When forming the discrete product of many large integers, we may have a choice in the order of operations. For example, to find the product
$a \cdot b \cdot c\ (mod\ m)$
we could first multiply $a \cdot b \cdot c$ and then divide by `m` to get the remainder. Or we could divide by m after each multiplication, potentially reducing the size of the numbers to be multiplied.
$((a \cdot b)\ MOD\ m) \cdot c\ MOD\ m$
For example, to calculate
$4 \cdot 7 \cdot 8 \cdot 9\ (mod\ 10)$
we could do all the multiplications first:
$4 \cdot 7 \cdot 8 \cdot 9 = 28 \cdot 8 \cdot 9 = 224 \cdot 9 = 2016$
and then take the modulus
2016 MOD 10 = 6.

It might be simpler to reduce the products by taking the modulus after each multiplication:
$4 \cdot 7 \cdot 8 \cdot 9 = 28 \cdot 8 \cdot 9 = 8 \cdot 8 \cdot 9 = 64 \cdot 9 = 4 \cdot 9 = 36$
and 36 MOD 10 = 6
This is simpler in that we are multiplying smaller numbers.

4.2.4 Mod power

Related to modular multiplication is the mod power calculation.[3] Here we wish to calculate $x^p\ (mod\ m)$. Rather than performing p-1 multiplications, followed by a division, we can reduce the partial products at each step. For example, to calculate
$5^7\ (mod\ 7)$
we can reduce the result after each multiplication:
$5^2 \equiv 4\ (mod\ 7)$

[3] Also known as *modular exponentiation*.

$5^3 = 4 \cdot 5 \equiv 6 \ (mod\ 7)$
$5^4 = 6 \cdot 5 \equiv 2 \ (mod\ 7)$
$5^5 = 2 \cdot 5 \equiv 3 \ (mod\ 7)$
$5^6 = 3 \cdot 5 \equiv 1 \ (mod\ 7)$
$5^7 = 1 \cdot 5 \equiv 5 \ (mod\ 7)$

These techniques can be helpful in speeding up calculations. Also, if working with primitive data types (`int` or `long`), these techniques can help prevent fixed-point overflow.

Our mod power algorithm can be even more efficient, if we use products already computed. Notice that

$$\begin{aligned} 5^{15} &= 5^{8+4+2+1} \\ 5^2 &= 5 \cdot 5 \\ 5^4 &= 5^2 \cdot 5^2 \\ 5^8 &= 5^4 \cdot 5^4 \\ 5^{15} &= 5^8 \cdot 5^4 \cdot 5^2 \cdot 5^1 \end{aligned}$$

Note that we will need to write the exponent in binary to determine which powers of the base are to be selected. For example, to calculate 5^{13} we see that $5^{13} = 5^8 \cdot 5^4 \cdot 5^1$
because $13 = 1101_2$

Our algorithm will begin with the base, square it, and decide whether to include that square on each iteration. We call this the *binary* algorithm for modPower.

An Efficient algorithm for mod power: Binary modPower

Given x, e, and m, we wish to calculate $y = x^e \ (mod\ m)$ with a minimal number of multiplications.

1. Begin with y = 1, power=x
2. Repeat until e = 0
   (a) If e is odd, $y = y \cdot power \ (mod\ m)$
   (b) $power = power \cdot power \ (mod\ m)$
   (c) $e = e/2$

Note the following:

- The division e/2 is an integer division; we take the integer quotient. This is a right shift of the bits of e.
- Checking whether e is odd, is examining the low order bit of e. This determines whether we wish to include the power in the result.

Bisection algorithm for mod power

Here we expose another algorithm to calculate mod power which is comparable in running time to the previous algorithm, but which is simpler to implement. In this algorithm we work with square roots, which are calculated recursively: If n is even, the square root of x^n is $x^{n/2}$, so $x^{n/2} \cdot x^{n/2} = x^n$. If n is odd, $x^{n/2} \cdot x^{n/2} \cdot x = x^n$. This is all we need to calculate x^n.

Here is a recursive algorithm to calculate $x^n \pmod m$.

1. If n is 0, $x^0 = 1$; terminate, the result is 1.
2. The result is $x^{n/2} \cdot x^{n/2} \cdot x^{n \bmod 2}$

We use the same example, $5^{13} \pmod 7$

$$\begin{aligned} 5^{13} &= 5^6 \cdot 5^6 \cdot 5 \\ 5^6 &= 5^3 \cdot 5^3 \cdot 1 \\ 5^3 &= 5 \cdot 5 \cdot 5 \\ &= 25 \cdot 5 \\ &\equiv 4 \cdot 5 \pmod 7 \\ &= 20 \\ &\equiv 6 \pmod 7 \end{aligned}$$

We can now substitute

$$\begin{aligned} 5^6 &= 5^3 \cdot 5^3 \cdot 1 \\ &\equiv 6 \cdot 6 \pmod 7 \\ &\equiv 1 \pmod 7 \end{aligned}$$

And finally,

$$\begin{aligned} 5^{13} &= 5^6 \cdot 5^6 \cdot 5 \\ &\equiv 1 \cdot 1 \cdot 5 \pmod 7 \\ &\equiv 5 \pmod 7 \end{aligned}$$

This algorithm cuts the exponent in half on each iterative call, hence the name *bisection* algorithm for modPower.

4.2.5 Exercises

1. Show at least 5 integers, including at least one negative integer, in each of the following congruence classes defined below. In each case underline the preferred member of the congruence class.

(a) 3 (mod 5)

(b) -12 (mod 5)

(c) 0 (mod 7)

(d) 13 (mod 40)

(e) 0 (mod 401)

(f) 1 (mod 401)

2. Show how to calculate each of the following. Try to work with small numbers, when possible, and minimize the number of multiplications. Show your work.

   (a) $2 \cdot 3 \cdot 4$ (mod 5)

   (b) $6 \cdot 3 \cdot 4$ (mod 7)

   (c) $6 \cdot 3 \cdot 4$ (mod 8)

   (d) $2204095 \cdot 302222 \cdot 99999$ (mod 5)

   (e) $-52 \cdot 54 \cdot -52 \cdot 54$ (mod 53)

3. Show how to calculate each of the following. Try to work with small numbers, when possible, and minimize the number of multiplications. Show your work.

   (a) 3^5 (mod 7)

   (b) 2^8 (mod 11)

   (c) 3^{17} (mod 11)

4. Write a computer program to calculate x^e ($mod\ m$) where x, e, and m are integers supplied by the user.

   (a) Use the binary modPower algorithm provided in this section.

   (b) Use the bisection modPower algorithm provided in this section.

5. The modPower algorithms discussed in this section are used to calculate the value of $x^e \pmod m$. We wish to compare the relative efficiencies of the binary and bisection modPower algorithms. For each of the three expressions shown below.

   i $x^7 \pmod{23}$

   ii $x^{17} \pmod{39}$

   iii $x^{73} \pmod{47}$

   (a) How many multiplications are required in each case using the binary algorithm?

   (b) How many multiplications are required in each case using the bisection algorithm?

decimal (base 10)	hexadecimal (base 16)	binary (base 2)
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	a	1010
11	b	1011
12	c	1100
13	d	1101
14	e	1110
15	f	1111

Figure 4.3: Table showing the 16 hexadecimal digits, and their corresponding bit values

(c) Find a formula for the number of multiplications required by the binary algorithm for modPower, as a function of the exponent, e.

(d) Find a formula for the number of multiplications required by the bisection algorithm for modPower, as a function of the exponent, e.

4.3 Representation of bit strings

4.3.1 Hexadecimal

A string of bits can be represented with hexadecimal (base 16 digits). Each group of 4 bits is represented as a single hexadecimal digit as shown in Figure 4.3. Since we need 16 distinct symbols in base 16, and decimal supplies us with only 10 symbols (0..9) we use 6 alphabetic characters (a..f) to represent the decimal values 10..15, respectively

.

As an example, we use the string of 16 bits:
$0001 - 1100 - 1010 - 0011_2$
It can be represented in hexadecimal as:
$1ca3_{16}$
Thus hexadecimal can be used as a shorthand notation for bit strings; each hex digit represents four bits.

decimal (base 10)	Base-64	binary (base 2)
0..25	A..Z	000000..011001
26..51	a..z	011010..110011
52..61	0..9	110100..111101
62	+	111110
63	/	111111

Figure 4.4: Table showing the 64 base-64 digits, and their corresponding bit values

4.3.2 Base-64 encoding

We have seen that hexadecimal (base 16) provides a quick shorthand notation for bit strings. We now describe another shorthand notation in which each digit represents 6 bits, called *base-64 encoding*. In this notation each digit represents 6 bits; thus we will need 64 possible symbols for a single digit ($64 = 2^6$). Where can we find these 64 symbols? We have 10 decimal digits (0..9), 26 lower case letters (a..z), and 26 upper case letters (A..Z), but that provides a total of 62 symbols; we need two more, so we will use the '+' character and the '/' character, to get a total of 64 symbols. Figure 4.4 shows the values assigned to these 64 symbols.

As an example, we take $2b+_{64}$.
This will represent the bit string:
$110101\ 011011\ 111110_2$

Base 64 is used in some cryptographic applications, such as GPG (see chapter 9) to provide a concise representation of binary keys.

4.3.3 Base-58 encoding

The encoding schemes described above give us three different ways of representing a binary field as a string of characters:

- Octal (base 8) uses the decimal digits '0'..'7', and is more compact than binary.
- Hexadecimal (base 16) uses the digits '0'..'9', 'a'..'f', and is more compact than octal.
- Base 64 uses the digits 'A'..'Z', 'a'..'z', '0'..'9', '+', '/', and is more compact than hexadecimal

Cryptocurrencies, such as Bitcoin, often use a different encoding scheme. Bitcoin secret keys need to be stored securely because they provide access to the owner's holdings, potentially millions of dollars worth of Bitcoin. Backup copies of these keys are sometimes printed to paper, which is then stored in a secure safe. If the backup copy of the key is needed, the owner merely enters the key

decimal (base 10)	base-58	decimal (base 10)	base-58
0	'1'	21	'N'
1	'2'	22	'P'
2	'3'		
3	'4'	32	'Z'
		33	'a'
8	'9'		
9	'A'	43	'k'
10	'B'	44	'm'
11	'C'		
		57	'z'
16	'H'		
17	'J'		

Figure 4.5: Table showing the correspondence between decimal numbers and base-58 encoded characters

as shown on the paper. Since keys are often hundreds of bits in length, a compact encoding scheme, such as base-64 is used. However, there are potential problems with base-64. Since the encoded characters may be copied manually, it is possible that some of the characters can be conflated:

- The numeric digit `0` and the upper-case letter, `O` can be hard to distinguish.
- The numeric digit `1` and the lower-case letter, `l` can be hard to distinguish

If we choose to drop these four characters from our encoding scheme, as well as the '+' and '/' characters, we are left with 58 encoding characters. This gives rise to base-58 encoding, which is used by Bitcoin when converting long binary fields to printable form. The exact coding is shown in Figure 4.5.

We should note that the character '1' represents 0, and, unlike octal, hex, and base-64, each digit does not correspond to a fixed-length sub-field in the binary field which is being represented.

- In octal, each digit represents 3 bits because $2^3 = 8$.
- In hexadecimal, each digit represents 4 bits because $2^4 = 16$.
- In base-64 , each digit represents 6 bits because $2^6 = 64$.

Unfortunately, base-58 does not provide such a nice correspondence between binary fields and base-58 digits because 58 is not a power of two. Some examples of base-58 encoded values are shown in Figure 4.6.

Base-58 encoding with checksum, used in Bitcoin

Because Bitcoin user addresses and keys are routinely transmitted over the internet among Bitcoin users, it is important that transmission errors are detected,

decimal (base 10)	base-58
0	'1'
10	'B'
100	'2j'
1000	'JF'
58	'21'
3364	'211'
583	'B4'

Figure 4.6: Examples of numbers encoded with base-58

so that the information can be resent until it is delivered intact. Also, this information is often entered by a Bitcoin user on the keyboard, and typing errors occur frequently. Many such error-checking (and error-correcting) schemes, such as parity bits and SECDED (Single-error Correction, Double-Error Detection) have been used since the early days of communication technology. Bitcoin uses an error-detetion scheme known as Base-58Check encoding. This encoding scheme will detect (essentially) any number of incorrect bits in a transmission.

In Base58-Check encoding a 32-bit *checksum* is appended to the data being transmitted. The checksum is derived from the data (which we call the *payload*). At the receiving end the checksum can be recalculated from the payload, and if it does not equal the checksum that was received, a transmission error is detected. The checksum is also used to check for typing errors when a Bitcoin key or address is entered by the user.

The process of converting a bit string of arbitrary length, the payload, to Base58-Check format is shown below:

1. A 1-byte version number is prepended to the payload. If the payload represents a Bitcoin address, the version is 00_{16}. Other versions indicate Testnet[4] address, private key, or Bitcoin extension keys/addresses.

2. The hash function SHA-256 is applied to the payload, and the output is used as input again to SHA-256. This is called a double-hash.

3. Only the first four bytes of the double-hash are used. These four bytes constitute the checksum, and are then appended to the payload.

4. This result, consisting of version, payload, and checksum is then encoded with Base58 encoding to produce the final Base58-Checksum result.

A diagram of this process is shown in Figure 4.7 in which the size of the data payload to be encoded is `n` bits. The checksum is a double hash of the version+payload. The final result is ASCII text representing a base 58 integer.

[4] A separate peer-to-peer network used to test modifications to the Bitcoin software

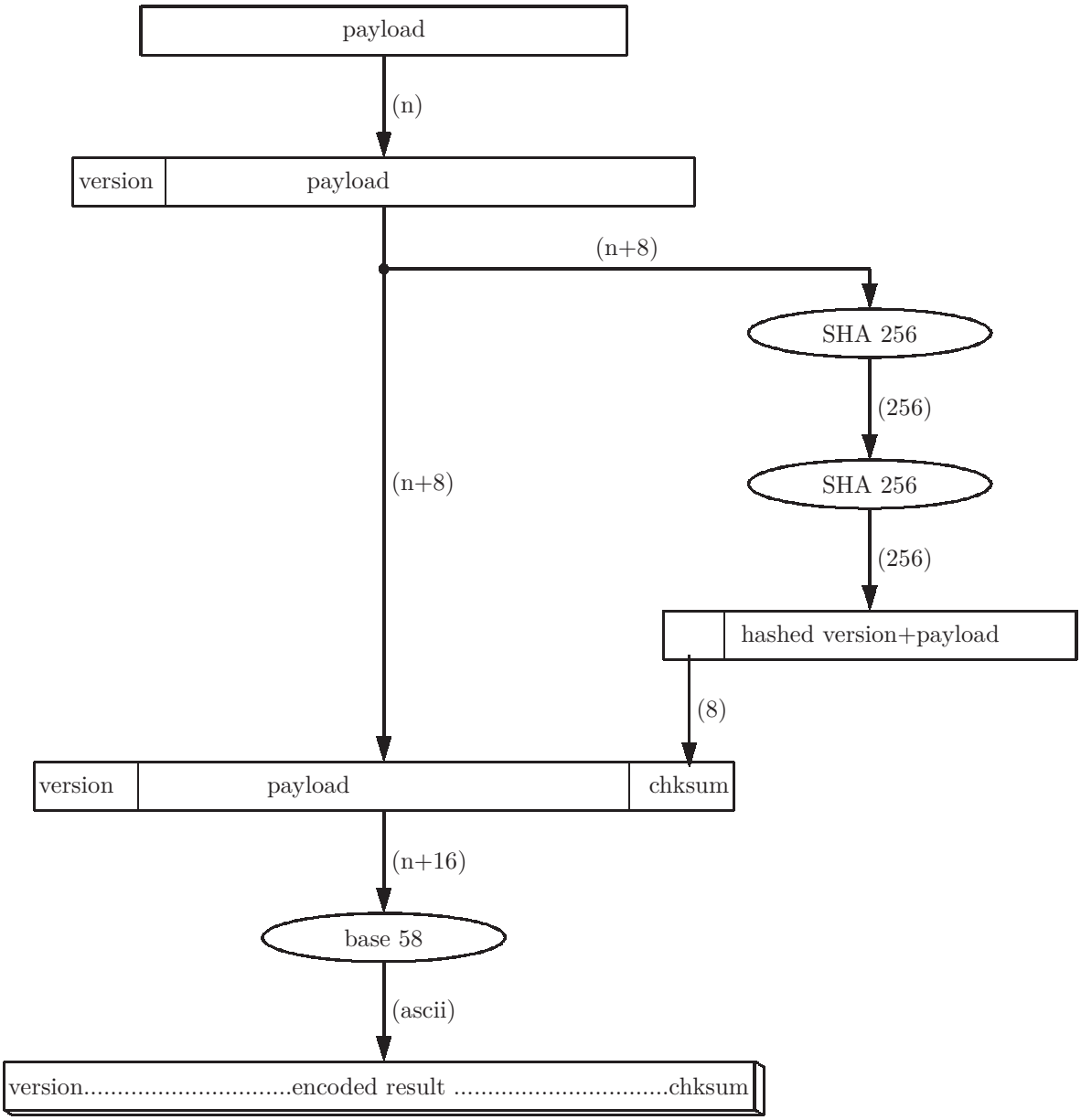


Figure 4.7: Diagram of the base 58 encoding process, with a checksum. The size of the payload to be encoded is n bits.

4.3.4 Exercises

1. Show each of the following integers in binary. Group the bits in groups of 4 or 6, for clarity .
   (a) 23_{16}
   (b) $4c0fd_{16}$
   (c) $3c_{64}$
   (d) $+0Bz_{64}$
2. A given binary field is `011011 101000`. Show this field in
   (a) Octal
   (b) Hexadecimal
   (c) Base-64
3. Show each of the following decimal integers with base-58 encoding.
   (a) 25
   (b) 116
   (c) 10000
4. Show each of the following base-58 values in decimal.
   (a) 411_{58}
   (b) BB_{58}
   (c) 1111_{58}

4.4 Factoring large integers

If two integers, `p` and `q` are multiplied, producing a product, `pq` we say that `p` and `q` are *factors* of `pq`. For example, 5 and 7 are factors of 35 because $5 \cdot 7 = 35$.

A one-way function that will be used to ensure the security of the RSA algorithm deals with factoring of large numbers. One way to factor a large number, `n`, is by brute force search: Divide the number by every integer less than `n` and look for a remainder of 0. If a list of prime numbers is available, divide by the prime numbers[5] instead of every integer less than n. Also, an integer which is not prime will have at least two prime factors. If one of these factors is greater than the square root of `n`, the other(s) must be less than the square root of `n`. Thus our search can stop when we reach that square root. Even with these speed-ups, the task of factoring a large integer (with hundreds of bits) can take too long to be viable. When we speak of "large" integers, we are thinking of integers even larger than the 64-bit integer (called `long` in Java and C++).[6]

[5] See section 4.7 for a discussion of prime numbers.
[6] See section 4.5 for a discussion of very large integers.

```
for (int i=0; i<n; i++)
   print (i);
```

Figure 4.8: The running time for a simple loop is $O(n)$.

```
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      print (i + j);
```

Figure 4.9: The running time for a nested loop is $O(n^2)$.

4.4.1 Why is factoring of large integers hard?

When we say that a given problem is *hard*, we mean that we cannot find an algorithm to solve the problem that takes a reasonable amount of time to execute. An algorithm which takes many years to find a solution is not a viable, or usable, algorithm.

Run time - big O

When comparing algorithms for speed, we say that the run time is $O(f(n))$ if the time it takes to complete is proportional to $f(n)$ as n gets large, where n represents the size of the input to the algorithm. A simple loop, as shown in Figure 4.8 is $O(n)$. A nested loop, as shown in Figure 4.9 is $O(n^2)$

If the input to an algorithm is a List or an array, then n represents the size, or length, of the List or array, as shown in Figure 4.10.

In the next section we further explore the problem of factoring large integers.

4.4.2 Exercises

1. Show all the positive factors of each of the following numbers (exclude 1 and the given number):

   (a) 75

   (b) 24

   (c) 97

```
// Given an array of integers, myArray, with length len
for (int i=0; i<len; i++)
   print (myArray[i]);
```

Figure 4.10: If the input to an algorithm is a list, n represents the size, or length, of the list. The running time is $O(n)$.

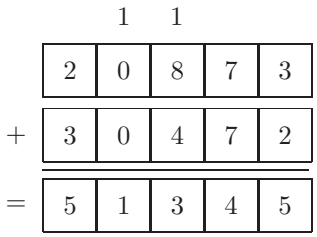


Figure 4.11: Diagram of the addition of two big integers, represented as an array, or list, of digits

(d) 128

2. What is the big O run time for the following Java method:

```
void method(List<Integer> numbers)
{   for (int i=0; i<numbers.size(); i++)
        System.out.println(numbers.get(i);
}
```

3. Write a computer program to find the factors of a given integer. Use a brute force search of every integer up to the given integer.

4.5 Big integers

In many cryptographic applications security is derived from the fact that certain computations take too much time, even on fast computers. These computations generally deal with large integers; i.e. integers so large that they exceed the limits of primitive types `int` (32 bits) and `long` (64 bits). Arithmetic with such large numbers can be accomplished by representing the number as an array (or list) of digits. Such a data type is called a *BigInteger* in the Java programming language.

A diagram showing the addition of two such integers is shown in Figure 4.11.

4.5.1 Arithmetic with big integers

Despite the availability of Big Integer software packages, we feel it is instructive for the student to see the internal workings of these packages. This will provide a better understanding of the performance measures, which are used to assure security in crypto systems. This will better enable us to answer questions, such as:
Why is factoring hard but multiplication is not hard?

In what follows we develop a Java class named BigNumber. In this class we have a field which stores a BigNumber as a list of decimal digits, in which the

low order digit is at position 0 of the list. We can create a BigNumber with the value 0:

```
List<Integer> digits = new LinkedList<Integer>();
digits.add(0);
```

Addition

To add two BigNumbers all we need is a loop to iterate through the two numbers to add corresponding digits, and adding in a *carry* if necessary.

```
Iterator<Integer> itThis = this.digits.iterator(),
                  itOther = other.digits.iterator();
int carry = 0;

while (itThis.hasNext() && itOther.hasNext())
{  sum = itThis.next() + itOther.next() + carry;
   result.digits.add(sum%10);
   carry = sum / 10;
}
```

It is possible that the two numbers will have different lengths, so two more loops are needed to finish the addition.

Subtraction

Implementing subtraction implies that we need to represent negative BigNumbers. We could use a sign-and-magnitude representation, but we find it better to use a *ten's complement* representation (similar to two's complement for binary numbers):

- If the high order digit is greater than 4, the number is negative.
- Some examples of 3-digit numbers:
  - 099 = +99
  - 999 = -1
  - 704 = -296
- To negate a number:
  1. Scan from low-order to high-order digit
  2. Copy zeros, until the first non-zero digit is encountered
  3. Copy the ten's complement of the non-zero digit[7]
  4. Copy the nine's complement of the remaining digits[8]

[7] The 10's complement of a digit, d, is 10-d.
[8] The 9's complement of a digit, d, is 9-d.

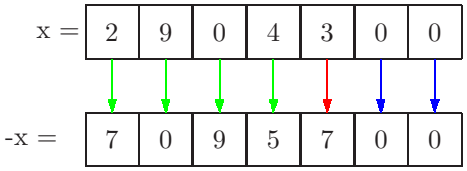


Figure 4.12: Diagram of a negation algorithm for big integers, represented as an array, or list, of digits

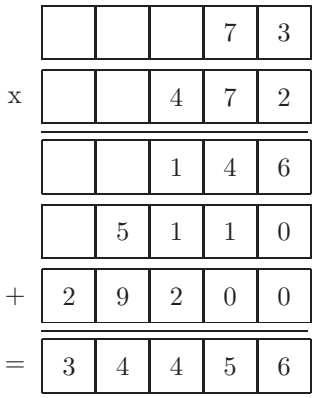


Figure 4.13: Diagram of the multiplication of two big integers, represented as an array, or list, of digits, using a shift-and-add algorithm

- A diagram depicting this negation algorithm is shown in Figure 4.12.[9] Now subtraction is easy:
  `A - B = A + (-B)`
  with only a few minor modifications to our addition algorithm.

Multiplication

Multiplication is easily implemented as repeated addition. However, it is more efficient to use a shift-and-add algorithm, which is similar to the algorithm we learned in elementary school, depicted in Figure 4.13.

We use the following Java algorithm for multiplication, in which each BigNumber stores a list of digits:

```
// Check for negative values, and complement if necessary
  Iterator<Integer> itThis = this.digits.iterator();
  while (itThis.hasNext())
     {  product = product.add(other.multByInt(itty.next());
        other.digits.add(0,0);           // shift left
     }
```

[9] Note that when negating a number such as -500, we'll need to add a digit to avoid overflow.

It uses another method which multiplies this BigNumber by a given scalar int, and then adding the result to the product being computed.

Division

Division will be a critical operation in cryptographic applications, because all arithmetic is done with respect to a given modulus, which is the remainder, or residue, after division. Division can be implemented as repeated subtraction.

We use the following Java algorithm for division. It creates two BigNumbers, the quotient and the remainder:

```
// Check for negative values, and complement if necessary
  BigNumber dividend = new BigNumber(this);           // copy dividend
  while (! dividend.isNegative())
     {  remainder = new BigNumber(dividend);    // copy dividend
        dividend = dividend.subtract(divisor);
        quotient = quotient.add(one);           // quotient++
     }
```

Division is much more efficient as a shift-and-subtract algorithm, which is similar to the algorithm we learned in elementary school. We leave this as an exercise for the student.

4.5.2 Run time performance of big integers

Students who have been exposed to analysis of algorithms are familiar with the 'big O' notation for run time performance. If the running time of an algorithm is $O(f(n))$, that means that the running time will be proportional to `f(n)`, as `n` gets large, and `n` represents the size of the input to the algorithm.

If the input to an algorithm is an integer, stored as a primitive value, then n represents the magnitude of that integer. However if the input is a BigNumber, which stores a list of digits, the input is actually a list, and n represents the size of the list. In what follows, n represents the size of the list of digits in a BigNumber, not the magnitude of the number.

add A simple loop is needed to add corresponding digits. The size of the list of digits of the larger number determines the number of times the loop repeats. The algorithm run time is $O(n)$, where `n` is the size of the list of digits in the larger number.

subtract `A-B = A + (-B)`
A simple loop is needed to negate the second operand, and a simple loop is needed for the add. $O(n) + O(n) = O(2n) = O(n)$

multiply With the shift-and-add algorithm described above, the loop is $O(n)$. In the body of the loop we have:

- multByInt(int): $O(n)$

- add(BigNumber): O(n)
- shift: O(1)

The result is:
$O(n \cdot (n + n + 1) = O(n^2)$

Factoring To find a factor of a given big integer, we can divide by smaller integers until we obtain a zero remainder. If $n \bmod x = 0$, then x is a factor of n. To find a factor of n, a sequential search can be used until a factor is found. Since a big integer is a list (or array) of digits, the analysis presumes the input is not an integer, but a list of (decimal or binary) integers. The big O run-time will be a function of the size of that list. If we add one (decimal) digit to the list, the time for the division will be multiplied by ten. Thus the run time for factoring is $O(10^n)$.[10] This is exponential, and consequently a factoring algorithm will require more time than we have. No one knows of a faster factoring algorithm, so we say that factoring of big integers is *hard*. Many public key cryptographic algorithms rely on this principle for security.

4.5.3 Exercises

1. Show each of the following numbers in 10's complement representation using exactly 4 digits, if possible.

   (a) 23

   (b) -2

   (c) -23

   (d) 500

   (e) 5000

   (f) -500

2. Show each of the following numbers in 10's complement representation using only as many digits as necessary.

   (a) 23

   (b) -4

   (c) -5

   (d) 500

   (e) 5000

   (f) -402300

[10] If the big integer is a list of binary values instead of decimal digits, we get a similar result, which is still exponential.

3. In the addition algorithm for BigNumbers, two additional loops are needed for the case where the numbers being added have different lengths. Show these two extra loops.
4. Revise the addition algorithm for BigNumbers to accommodate operands which could be negative.
5. Show an efficient division algorithm for BigNumbers, using a shift-and-subtract algorithm.
   (a) Implement your algorithm, and test it.
   (b) What is the running time performance of your algorithm?

4.6 Discrete multiplicative inverses

We have seen that it is possible for the discrete product of integers to be 1. For example:
$3 * 7 \equiv 1 \ (mod\ 10)$
The discrete multiplicative inverse of a given integer, x, modulo m, is that integer y such that $x * y \equiv 1 \ (mod\ m)$. Thus 7 is a multiplicative inverse of of 3 (mod 10), and 3 is a multiplicative inverse of 7 (mod m). The integers 3 and 7 are mutual multiplicative inverses (mod 10).

One way to find a discrete multiplicative inverse is by brute force search. For example, suppose you wish to find the inverse of 4 (mod 9). Try every possible integer from 2..8, looking for a product of 1:
$2 * 4 \equiv 8 \ (mod\ 9)$
$3 * 4 \equiv 3 \ (mod\ 9)$
$4 * 4 \equiv 7 \ (mod\ 9)$
$5 * 4 \equiv 2 \ (mod\ 9)$
$6 * 4 \equiv 6 \ (mod\ 9)$
$7 * 4 \equiv 1 \ (mod\ 9)$
Thus 7 and 4 are mutual inverses (mod 9). We use a superscript of -1 to indicate multiplicative inverse. Thus:
$4^{-1} \equiv 7 \ (mod\ 9)$
$7^{-1} \equiv 4 \ (mod\ 9)$

Two integers, a and b, are said to be *relatively prime* iff the only factor they have in common is 1. Thus 15 and 22 are relatively prime because the factors of 15 are {1,3,5} and the factors of 22 are {1,2,11}.

What is the inverse of 8 (mod 10)? Unfortunately, 8 has no inverse (mod 10). In general an integer, n, will have a discrete multiplicative inverse (mod m) iff n and m are relatively prime.

4.6.1 Euclid's algorithms

In crypto systems we will be using multiplicative inverses extensively, and we woulld like to be able to find these inverses quickly. A sequential search will

take too long when the numbers are large. There is a faster way of finding multiplicative inverses, but first we should discuss *greatest common divisor*, abbreviated gcd. Since divisor is another word for factor, the greatest common divisor of two integers is the largest factor that they share. For example, the greatest common divisor of 30 and 105 is 15 because the divisors of 30 are 1,2,3,5,6,10,15, and the divisors of 105 are 3,5,7,15,21,35.
gcd(30,105) = 15

Note that two integers, x and y, are relatively prime iff gcd(x,y)=1.

The Euclidean algorithm for gcd(x,y) is:

1. Find r = x MOD y
2. If r is 0, the result is y
3. If r is not 0, the result is gcd(y,r)

Though this algorithm can be stated with a loop, we have stated it recursively. The gcd algorithm invokes the gcd algorithm. We can do this because step 2, called the *base* case, gives a final result. Step 3 is the recursive case. As an example, we will find gcd(120,50):
r = 120 MOD 50 = 20
Since r is not 0, we need to find gcd(50,20):
r = 50 MOD 20 = 10
Since r is not 0, we need to find gcd(20,10):
r = 20 MOD 10 = 0
Since r is 0, the result is 10

Note that the Euclidean algorithm generates a sequence of remainders, called r.

4.6.2 Multiplicative inverse

We can use Euclid's gcd algorithm to find a multiplicative inverse quickly (assuming it exists). This is known as the *Extended Euclidean algorithm* for discrete multiplicative inverses.

Given an integer `a`, and a modulus, `m`, such that `a` and `m` are relatively prime, find the discrete multiplicative inverse of `a`, a^{-1} $(mod\ m)$. This algorithm will use an array of remainders, r, an array of quotients, q, as well as arrays called u and v. At each step we will see that $r_n = u_n \cdot a + v_n \cdot m$

1. Setup:

$$\begin{aligned} r_{-2} &= m \\ r_{-1} &= a \\ u_{-2} &= 0 \\ v_{-2} &= 1 \\ u_{-1} &= 1 \\ v_{-1} &= 0 \end{aligned}$$

n	q	r	u	v	$u \cdot a + v \cdot m$
-2		m	0	1	m
-1		a	1	0	a

Figure 4.14: Setup for the extended Euclidean algorithm, to find a^{-1} $(mod\ m)$

2. Generate a table of quotients and remainders.
   Also generate the arrays for u and v
   Repeat for n = 0,1,2,3,... until $r_n = 1$

   (a)

   $$if\ r_{n-1} = 1\ then$$
   $$terminate\ the\ loop$$

   (b)

   $$if\ r_{n-1} \neq 1\ then$$
   $$\begin{aligned} q_n &= r_{n-2}\ /\ r_{n-1} \\ r_n &= r_{n-2}\ MOD\ r_{n-1} \\ u_n &= u_{n-2} - q_n u_{n-1} \\ v_n &= v_{n-2} - q_n v_{n-1} \end{aligned}$$

3. Since $r_n = 1$ we have:
   $1 = u_n \cdot a + v_n \cdot m$
   Since $v_n \cdot m \equiv 0\ (mod\ m)$ we have $u_n \cdot a \equiv 1\ (mod\ m)$
   Thus $a^{-1} \equiv u_n\ (mod\ m)$

4. If u_n is negative, we can get the preferred member of the congruence class by adding the modulus, `m`. Choose either of the following:
   $a^{-1} = u_n$ or
   $a^{-1} = u_n + m$

Figure 4.14 shows how to setup the table for the extended Euclidean algorthm, and Figure 4.15 shows how to fill in succeeding lines in the table.

As an example we show how to find the inverse of 5, mod 39, in Figure 4.16. To begin, `a` is 5, and `m` is 39. The setup (step 1 in the algorithm) fills in the first two rows of the table. Then the loop (step 2 in the algorithm) fills in the next two rows of the table, terminating when r=1. Note that in each line the remainder, `r`, is always equal to the value of $u \cdot a + v \cdot m$. In the last row (n=1) because $r = u \cdot a + v \cdot m$ we have
$1 = 8 \cdot 5 + -1 \cdot 39$

n	q	r	u	v
i-2	q_{i-2}	r_{i-2}	u_{i-2}	v_{i-2}
i-1	q_{i-1}	r_{i-1}	u_{i-1}	v_{i-1}
i	$q_i = r_{i-2}/r_{i-1}$	$r_i = r_{i-2}\ MOD\ r_{i-1}$	$u_i = u_{i-2} - q_i \cdot u_{i-1}$	$v_i = v_{i-2} - q_i \cdot v_{i-1}$

Figure 4.15: Iterated step for the extended Euclidean algorithm, to find $a^{-1}\ (mod\ m)$ (Not shown: column for $u \cdot a + v \cdot m$)

a = 5
m = 39

n	q	r	u	v	$u \cdot a + v \cdot m$
-2		39	0	1	39
-1		5	1	0	5
0	7	4	-7	1	4
1	1	1	8	-1	1

$$\begin{aligned} r = &\quad u \cdot a + v \cdot m \\ 1 = &\quad 8 \cdot 5 + -1 \cdot 39 \\ 5^{-1} \equiv &\quad 8\ (mod\ 39) \end{aligned}$$

Figure 4.16: Calculation of the discrete multiplicative inverse of 5, mod 39, using the extended Euclidean algorithm. The result is 8.

Since $-1 \cdot 39 \equiv 0$, we are left with
$1 = 8 \cdot 5$
Thus 8 and 5 are mutual mulitiplicative inverses. We can check that by multiplying, $8 \cdot 5 = 40 \equiv 1\ (mod\ 39)$

One more example will demonstrate the case where the final value of u is negative. Here we are looking for the inverse of 17 (mod 75). Thus `a` = 17 and `m` = 75. The table generated by the extended Euclidean algorithm is shown in Figure 4.17.

Notice that the loop repeats only four times, quickly finding a solution. In this case the final value of u is -22. This is not incorrect, but we are usually interested in the smallest value in the congruence class which is not negative. To get that we simply add the modulus, 75, to arrive at 53 for the final result.
$-22 \equiv 53\ (mod\ 75)$
We can check the result by multiplying, $17 \cdot 53 = 901 \equiv 1\ (mod\ 75)$ The integers 17 and 53 are mutual inverses, mod 75.

a = 17
m = 75

n	q	r	u	v	$u \cdot a + v \cdot m$
-2		75	0	1	75
-1		17	1	0	17
0	4	7	-4	1	7
1	2	3	9	-2	3
2	2	1	-22	5	1

$$\begin{aligned} r = &\quad u \cdot a + v \cdot m \\ 1 = &\quad -22 \cdot 17 + 5 \cdot 75 \\ -22 \equiv &\quad 53 \ (mod\ 75) \\ 17^{-1} \equiv &\quad 53 \ (mod\ 75) \end{aligned}$$

Figure 4.17: Calculation of the discrete multiplicative inverse of 17, mod 75, using the extended Euclidean algorithm. The result is 53.

4.6.3 Exercises

1. Use a brute force search to find the following discrete multiplicative inverses, if an inverse exists:
   (a) $5^{-1}\ (mod\ 9)$
   (b) $3^{-1}\ (mod\ 9)$
   (c) $2^{-1}\ (mod\ 11)$
2. Use the Euclidean algorithm to calculate each of the following:
   (a) gcd(15,5)
   (b) gcd(24,42)
   (c) gcd(77,30)
   (d) gcd(120,77)
3. Use the extended Euclidean algorithm to find each of the following discrete multiplicative inverses, if an inverse exists:
   (a) $2^{-1}\ (mod\ 11)$
   (b) $30^{-1}\ (mod\ 77)$
   (c) $33^{-1}\ (mod\ 77)$
   (d) $797^{-1}\ (mod\ 1047)$

4.7 Prime numbers

A positive integer greater than 1 is said to be *prime* iff its only divisors are 1 and itself. The first several prime numbers are:
`2,3,5,7,11,13,17,19,23,29,...`
It has been shown that there are inifinitely many prime numbers. Prime numbers have been studied extensively and have many important applications. Prime numbers are used extensively in public key cryptography.

If the product of two positive integers, x and y, is the positive integer $z = x \cdot y$, we say that x and y are *factors* of z. It can be shown that any positive integer is the product of 1 or more prime numbers; these are called *prime factors*. For example, the prime factors of 308 are:
$308 = 2 \cdot 2 \cdot 7 \cdot 11$

4.7.1 Exercises

1. List the next 10 prime numbers after 29.
2. Are there any even prime numbers?
3. Show the factors of 424 which are prime numbers. These are called *prime factors*.
4. Some primes occur in pairs, in which their difference is 2. Examples are (3,5) (5,7) and (11,13). These prime numbers are known as *twin primes*. Show three more examples of pairs of twin primes..
5. Show the prime factors of each of the following integers
   (a) 33
   (b) 256
   (c) 286
   (d) 2047
   (e) 4095

4.8 The discrete log problem

In this section we deal with exponents, again using only integers, and a modulus:

$y \equiv b^e \ (mod\ m)$
In section 4.2 we saw an efficient way of calculating y, given b, e, and m.

Here we are working on the *inverse* problem: Given y, b, and m, find e. This is called the *discrete log problem*.[11] For example:
$5 \equiv 3^? \ (mod\ 7)$

[11] In continuous math the logarithm function is the inverse of the exponential function.

a	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}
2*	4	8	5	10	9	7	3	6	1
3	9	5	4	1	3	9	5	4	1
4	5	9	3	1	4	5	9	3	1
5	3	4	9	1	5	3	4	9	1
6*	3	7	9	10	5	8	4	2	1
7*	5	2	3	10	4	6	9	8	1
8*	9	6	3	10	3	2	5	7	1
9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1

Figure 4.18: Table showing the four generators (mod 11): 2,6,7,8

A brute force search of all possible values (mod 7) reveals that:

$5 \equiv 3^5 \ (mod\ 7)$

Note that there might be more than one integer in the range (0..m-1) which solves this problem. We know of no more efficient way of solving the discrete log problem. For large values of m, a search could take so long as to make this problem unsolvable.[12] For this reason the discrete log problem is used to provide security in certain cryptographic algorithms, such as encryption (El-Gamal) and key distribution (Diffie-Hellman).

4.8.1 Generators

A word of caution is needed here. There are cases where the discrete log problem may not be hard. In Figure 4.18 we show all the powers of a given base, a with a prime modulus, $m = 11$. Note that for some bases, a, the powers are comprised of all values from 1 through 10. These bases (2,6,7,8 in Figure 4.18) are called *generators* or *primitive roots*. But for other bases (3,4,5,9,10 in Figure 4.18) the powers repeat, meaning that some of the numbers less than m do not correspond to any power for that base.

Thus the choice of an appropriate base can make the discrete log problem more difficult. For example, if we are attempting to solve $9 \equiv 3^? \ (mod\ 11)$, it is likely that we will find a solution after a few attempts.[13] However, if we are attempting to solve $9 \equiv 6^? \ (mod\ 11)$, more attempts may be required before finding the solution. Since 6 is a generator, it is a better choice for the base if the goal is to make the discrete log problem hard.

We know of no algorithm to find the generators for a given prime modulus, other than trial and error.

[12] Even when working with 64-bit primitive long integers, the discrete log problem takes a lot of time.

[13] With a base of 3, there are two solutions to the problem: 2,7

4.8.2 Exercises

1. Solve the following discrete log problems by filling in the missing number:
   (a) $2^{?} \equiv 8 \pmod{10}$
   (b) $2^{?} \equiv 2 \pmod{10}$
   (c) $2^{?} \equiv 10 \pmod{11}$
   (d) $2^{?} \equiv 1 \pmod{7}$
   (e) $7^{?} \equiv 1 \pmod{97}$
2. Write a program to solve the discrete log problem. The parameters, or inputs, should be y, b, and m, and the result should be the value an array or list of values, e, such that
   $y \equiv b^e \pmod{m}$
3. Develop a spreadsheet to solve the discrete log problem. Allow the user to type in the values of y, b, and m. The spreadsheet should then calculate e.
   $y \equiv b^e \pmod{m}$
4. What are the generators for the prime modulus $m = 41$?

4.9 Theorems from Fermat and Euler

For cryptology, one of the most influential results from classical mathematics involves Fermat's little theorem,[14] circa 1640. This theorem is primarily used as the basis of the RSA public key algorithm. We find it remarkable that a fairly obscure seventeenth century result can be useful in keeping the internet alive today!

4.9.1 Fermat's little theorem

Fermat's little theorem simply stated:
If p is a prime number, then for any integer, a, $a^p - a$ is a multiple of p. We can state that more formally:
$a^p - a \equiv 0 \pmod{p}$
or multiply both sides by a^{-1}, to get:

$$\begin{aligned} a^p \cdot a^{-1} - a \cdot a^{-1} &\equiv 0 \pmod{p} \\ a^{p-1} - 1 &\equiv 0 \pmod{p} \\ a^{p-1} &\equiv 1 \pmod{p} \end{aligned}$$

[14] Not to be confused with Fermat's *last* theorem

As an example, this theorem tells us that $2^{11} \equiv 2 \pmod{11}$ and that $2^{10} \equiv 1 \pmod{11}$ because 11 is a prime number. There is no need to multiply at all.

Given a positive integer, a, and a prime number, p, which is not a factor of aa, he described the following derivation for some exponent, e:

$$\begin{aligned} a^e &\equiv a^{e-(p-1)+(p-1)} \pmod{p} \\ &\equiv a^{e-(p-1)} \cdot a^{p-1} \pmod{p} \\ &\equiv a^{e-(p-1)} \cdot 1 \pmod{p} \\ &\equiv a^{e-(p-1)} \pmod{p} \end{aligned}$$

Above we are adding and subtracting (p-1) to the exponent, and then simplifing using Fermat's little theorem.

We can repeat that process, again adding and subtracting (p-1) to the exponent::

$$\begin{aligned} a^e &\equiv a^{e-(p-1)} \pmod{p} \\ &\equiv a^{e-(p-1)-(p-1)+(p-1)} \pmod{p} \\ &\equiv a^{e-2\cdot(p-1)} \cdot a^{(p-1)} \pmod{p} \\ &\equiv a^{e-2\cdot(p-1)} \cdot 1 \pmod{p} \\ &\equiv a^{e-2\cdot(p-1)} \pmod{p} \end{aligned}$$

Notice that this process can be repeated as often as necessary, repeatedly subtracting (p-1) from the exponent, until we are left with the residue:
$e \bmod p-1$
We have derived the following from Fermat's little theorem:

$a^e \equiv a^{e \bmod p-1} \pmod{p}$

Informally our result is:

When working mod p, the exponent can be reduced, mod (p-1).

This result will be useful in calculating exponent expressions with a prime modulus.

Examples:
$3^{15} \equiv 3^3 \equiv 6 \pmod{7}$
$2^{2349393923} \equiv 2^3 \equiv 8 \pmod{11}$

4.9.2 Euler's generalization of Fermat's little theorem

The 18th century mathematician Euler derived a generalization of Fermat's little theorem, and it is actually that extension which is used in modern crypto systems.

n	Multiple of 3	Multiple of 5	Multiple of neither 3 nor 5
1	No	No	Yes
2	No	No	Yes
3	Yes	No	
4	No	No	Yes
5	No	Yes	
6	Yes	No	
7	No	No	Yes
8	No	No	Yes
9	Yes	No	
10	No	Yes	
11	No	No	Yes
12	Yes	No	
13	No	No	Yes
14	No	No	Yes
Total	4	2	8

Figure 4.19: Example of Euler's totient function for a product of two prime numbers, 3 and 5. The number of integers less than 15 relatively prime to 15 is 8.

Euler's totient function

Euler defined a function, $\varphi(n)$, or phi(n), to be the number of positive integers less than n which are relatively prime to n (including 1). This is known as Euler's *totient* function. Some examples:
$\varphi(5) = 4$ because 4,3,2,1 are all relatively prime to 5
$\varphi(11) = 11$ because 10,9,8,7,6,5,4,3,2,1 are all relatively prime to 11
If p is prime, $\varphi(p) = p - 1$
If p and q are primes, we will show that $\varphi(p \cdot q) = (p - 1) \cdot (q - 1)$

Figure 4.19 examines the case where p=3 and q=5. We need to count the number of integers less than 15 which are relatively prime to 15. We do that by subtracting the number of integers less than 15 which are *not* relatively prime to 15. In this chart we are working with the prime numbers, p=3 and q=5, pq=15. In the following explanation we work with this example, and generalize for any primes p and q.

```
Example:                                  General case:
Find phi(15)                              Find phi(n)
15 = 3 x 5                                n = pq, both are primes

There are 14 (positive) integers          Threre are n-1 (positive) integers
less than 15.                             less than n.

4 of them are divisible by 3              q-1 of them are divisible by p
```

```
because 5-1 = 4

2 of them are divisible by 5
because 3-1 = 2

Total number of integers less
than 15 which are relatively
prime to 15 is:
        14 - 4 - 2 = 8


phi(15) = 8
```

```
p-1 of them are divisible by q


Total number of integers less
than n which are relatively
prime to n is:
        (n-1) - (q-1) - (p-1)
             = (pq-1) - (q-1) - (p-1)
             = pq - q - p + 1
             = (p-1)(q-1)

phi(n) = (p-1)(q-1)
```

Euler's theorem

Euler proved a generalization of Fermat's little theorem:

$a^e \equiv a^{e \ MOD \ \varphi(m)} \pmod{m}$

In this text we will be interested in the case where the modulus is the product of two primes: $m = p \cdot q$. In this case Euler's result reduces to:

$a^e \equiv a^{e \ MOD \ (p-1)\cdot(q-1)} \pmod{m}$

because $\varphi(pq) = (p-1) \cdot (q-1)$.

Informally, **when working mod pq, work mod (p-1)(q-1) in the exponent.** As an example, we evaluate $5^{18} \pmod{15}$.

Since $\varphi(15) = (3-1) \cdot (5-1) = 8$, we will reduce the exponent, mod 8.

$$\begin{aligned} 5^{18} \pmod{15} &\equiv 5^{18 \ MOD \ 8} \pmod{15} \\ &\equiv 5^2 \pmod{15} \\ &\equiv 10 \pmod{15} \end{aligned}$$

4.9.3 Exercises

1. Use Fermat's little theorem to evaluate each of the following:
   (a) $3^{17} \pmod{17}$
   (b) $7^{52} \pmod{53}$
   (c) $2^{98} \pmod{97}$
2. Evaluate each of the following using results derived from Fermat's little theorem:
   (a) $3^{18} \pmod{17}$
   (b) $1023^8 \pmod{5}$

(c) $2^{3938284849892383} \pmod{11}$

3. Evaluate each of the following examples of Euler's totient function:

   (a) $\varphi(37)$

   (b) $\varphi(21)$

   (c) $\varphi(77)$

4. Use Euler's theorem to evaluate each of the following:

   (a) $3^{18} \pmod{15}$

   (b) $3^{26} \pmod{35}$

   (c) $2^{44} \pmod{55}$

4.10 Discrete elliptic curves

One of the most powerful and secure crypto systems is the one which is based on discrete elliptic curves. This is the method used by Bitcoin and other digital currencies, where the algorithms secure transactions worth millions of dollars - security is a top priority. It has been estimated that for a given key size, discrete elliptic curves are 10 times more secure than other crypto systems.

4.10.1 Continuous elliptic curves

In order to understand discrete elliptic curves we must first take a look at continuous (i.e. non-discrete, or ordinary) elliptic curves. When graphing an elliptic curve, the x and y axes are real number liness, and the graphs will be smooth curves rather than discrete points. The general equation for an elliptic curve is:
$y^2 = x^3 + ax + b$
where `a` and `b` are constants that determine the shape of the curve.

As an example, we set `a` to -4 and examine the curves for four values of b: 0, 2, 4, and 6. The graphs of $y^2 = x^3 - 4x + 0$ and $y^2 = x^3 - 4x + 2$ are shown in Figure 4.20. The graphs of $y^2 = x^3 - 4x + 4$ and $y^2 = x^3 - 4x + 6$ are shown in Figure 4.21.[15] We note that there is symmetry with respect to the x axis for all four of these curves. This is true of all elliptic curves because of the y^2 term.

Note that the two curves (`b` = 0 and `b` = 2) in Figure 4.20 intercept the x axis in three points and consist of two separate parts. The other two curves (`b` = 4 and `b` = 6), shown in Figure 4.21 intercept the x axis in only one point and consist of one part.

We define the *discriminant*, D, of an elliptic curve as:
$D = 4a^3 + 27b^2$. For our example with `a` = -4, we see that the values for the discriminant are shown below:

[15] This example is adapted from *Introduction to Cryptography* by Stanoyevitch

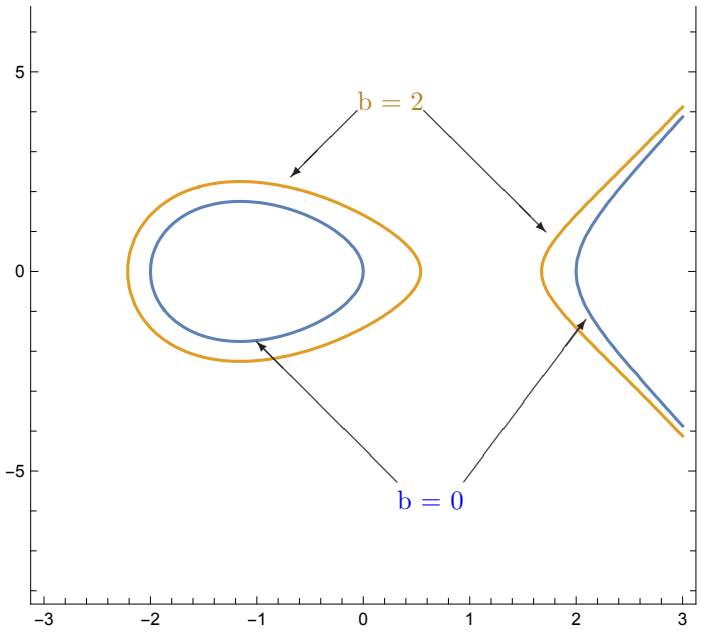


Figure 4.20: Graph of the elliptic curves $y^2 = x^3 - 4x + b$ for b=0 and b=2. The discriminant is negative.

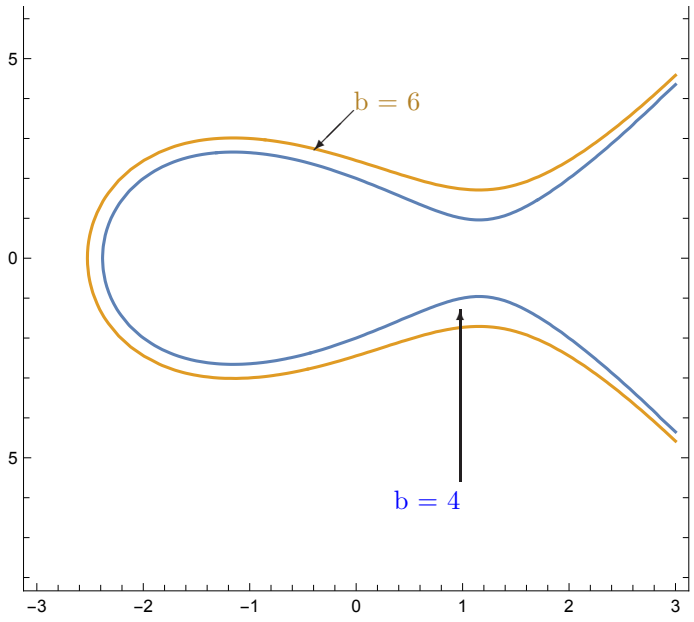


Figure 4.21: Graph of the elliptic curves $y^2 = x^3 - 4x + b$ for b=4 and b=6. The discriminant is positive.

a	b	Discriminant
-4	0	-256
-4	2	-148
-4	4	+176
-4	6	+696

An elliptic curve is *singular* iff its discriminant is zero. When the discriminant is negative, the elliptic curve consists of two pieces, with three x-intercepts. When the discriminant is positive, the elliptic curve consists of one piece, with only one x-intercept. When the discriminant is zero, the elliptic curve consists of one piece, with two x-intercepts: we will not be concerned with singular elliptic curves.

4.10.2 Addition of points on an elliptic curve

In this section we define an "addition" operation for points on an elliptic curve. This operation does not resemble the ordinary addition of numbers, but it is called "addition" because it shares several properties with ordinary addition, as described below. We are now treating points on the curve as values which can be added, not (x,y) pairs. We also define a point at infinity, ∞, to be a part of every elliptic curve. Think of ∞ as being infinitely far from the origin in any direction. We define addition of two points, $P3 = P_1 + P_2$, on a non-singular elliptic curve as follows, divided into three distinct cases:

1. $P_1 \neq P_2, P_1 \neq \infty, P_2 \neq \infty$
   This case is depicted in Figure 4.22.

   (a) Draw a line through the points P_1 and P_2. This line should intersect the curve at a third point, call it Q. If the line is vertical, $Q = \infty$.

   (b) Reflect the point Q with respect to the x axis, as shown in Figure 4.22 to obtain the sum, $P3$

2. $P_1 = P_2, P_1 \neq \infty, P_2 \neq \infty$
   In this case we are adding a point to itself.
   $P + P = 2 \cdot P$
   This case is depicted in Figure 4.23.

   (a) Use the tangent line at P, which should intersect the curve at a second point, call it Q. If the tangent is vertical, $Q = \infty$.

   (b) Reflect the point Q with respect to the x axis to obtain the sum, $2 \cdot P$, as done in the previous case.

3. $P_1 = \infty$ or $P_2 = \infty$
   $P + \infty = P$ for either P_1 or P_2.

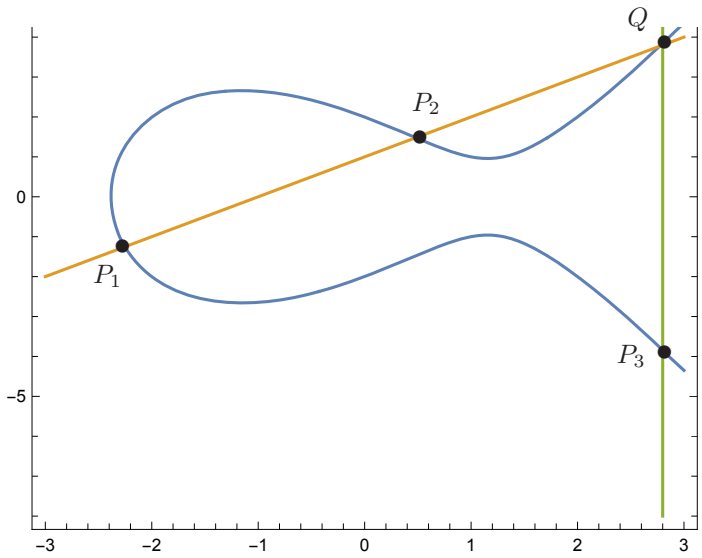


Figure 4.22: Addition of points on the elliptic curve $y^2 = x^3 - 4x + 4$.
$P_1 + P_2 = P_3$

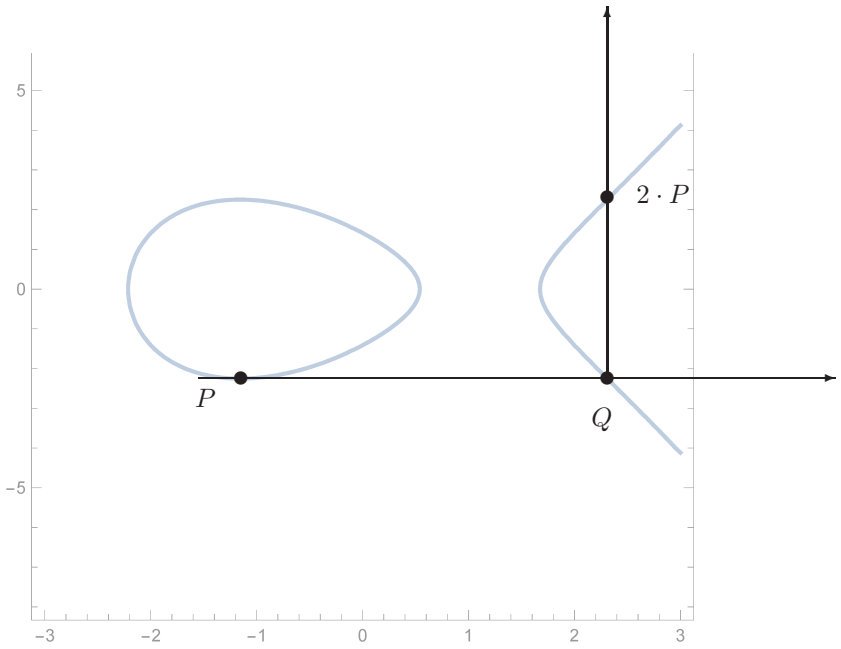


Figure 4.23: Addition of a point to itself on the elliptic curve $y^2 = x^3 - 4x + 2$. $P + P = 2P$

This addition operation is known as an *abelian group*, which is a commutative, associative, and closed operation with an identity and inverses:

$P_1 + P_2 = P2 + P1$	Commutative
$P_1 + (P_2 + P_3) = (P_1 + P_2) + P_3$	Associative
$P_1 + P_2$ is a member of the group	Closed
∞	Is the identity: $P + \infty = P$
$-P$ satisfies $P + -P = \infty$	Every point has an inverse
$-(x, y) = (x, -y)$	Find the inverse
$-\infty = \infty$	The identity is its own inverse

The addition algorithm described above does not explain how to calculate steps 1 and 2, so we clarify here:

We need to calculate the slope of the line in Figures 4.22 and 4.23, assuming it is not a vertical line. Call the slope `m`.

- If $P_1 \neq P_2$, $m = (y_2 - y_1)/(x_2 - x_1)$
- If $P_1 = P_2$, $m = (3 \cdot x_1^2 + a)/(2 \cdot y_1)$

We can now find the sum, $P_3 = (x_3, y_3)$:
$x_3 = m^2 - x_1 - x_2$
$y_3 = m \cdot (x_1 - x_3) - y_1$

As an example we take the elliptic curve:
$y^2 = x^3 - 4x + 4$
We will find the sum of two points:
$P_1 = (-2, -2)$
$P_2 = (0, 2)$

First we calculate the slope of the line through these points:
$m = (y_2 - y_1)/(x_2 - x_1) = (2 - (-2))/(0 - (-2)) = 4/2 = 2$

Next we calculate x_3:
$x_3 = m^2 - x_1 - x_2 = 4 - (-2) - 0 = 6$

Next we calculate y_3:
$y_3 = m \cdot (x_1 - x_3) - y_1 = 2 \cdot (-2 - 6) - (-2) = -14$

Thus the solution is:
$P_1 + P_2 = (6, -14)$

We can verify that this point is on the elliptic curve:
$x^3 - 4x + 4 = 6^3 - 4 \cdot 6 + 4 = 216 - 24 + 4 = 196$
$-14^2 = 196$

x	x^3	$x^3 \ (mod\ 11)$	$y^2 = x^3 + x + 1 \ (mod\ 11)$	y
0	0	0	1	1,10
1	1	1	3	5,6
2	8	8	0	0
3	27	5	9	3,8
4	64	9	3	5,6
5	125	4	10	
6	216	7	3	5,6
7	343	2	10	
8	512	6	4	2,9
9	729	3	2	
10	100	10	10	

Figure 4.24: Table showing all the points (in addition to ∞) on the discrete elliptic curve $y^2 = x^3 + x + 1 \ (mod\ 11)$.

4.10.3 Discrete elliptic curves

When working with discrete elliptic curves,[16] we use the same formula described earlier, but the variables `a`, `b`, `x`, `y` are now restricted to whole numbers, and we include a prime modulus, `p`, implying that there is a finite number of points on the curve.
$y^2 = x^3 + ax + b \ (mod\ p)$
The fact that we are using a modulus means that all values will be in the range $[0..p-1]$. The discriminant for discrete elliptic curves is defined the same as for continuous elliptic curves, except that there is a prime modulus:
$D = 4a^3 + 27b^2 \ mod(p)$.
If the discriminant is zero, the elliptic curve is singular, and will not be useful.

As an example, we use the discrete elliptic curve with $a = b = 1$ and $p = 11$:
$y^2 = x^3 + x + 1 \ (mod\ 11)$
The discriminant is :
$D = 4a^3 + 27b^2 \ (mod\ p) = 4 + 27 = 31 \neq 0$ and thus the curve is nonsingular.

A table of all the points on this elliptic curve is shown in Figure 4.24. Note that there are two ways to calculate $x^3 \ (mod\ 11)$. For example if x=7:

- $7^3 = 7 \cdot 7 \cdot 7 = 343$
  $343 \equiv 2 \ (mod\ 11)$

- $7^2 = 49 \equiv 5 \ (mod\ 11)$
  $5 \cdot 7 = 35 \equiv 2 \ (mod\ 11)$

We find the second method easier, certainly when doing these computations in our head, because it is easier to work with small numbers.

[16] Discrete elliptic curves are also known as *modular* elliptic curves.

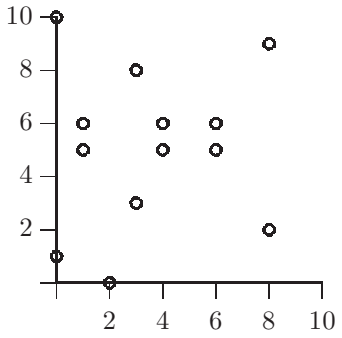


Figure 4.25: Plot of all 13 points on the discrete elliptic curve $y^2 = x^3 + x + 1 \ (mod\ 11)$

We also note that, as with continuous math, a number can have two square roots. They are actually additive complements:
If $y = x^2$, then $\sqrt{y} = \pm x$. In the discrete case the two solutions add to 0, with a modulus.
$sqrt(9) \equiv 3 \ (mod\ 11)$
$-3 \equiv 8 \ (mod\ 11)$
Thus the two square roots of 9 are $\{3, 8\}$ (mod 11). To verify, note that
$8 \cdot 8 = 64 \equiv 9 \ (mod\ 11)$

Unlike continuous math, some values have no square root. Working mod 11, the values 2, 6, 7, 8, and 10 have no square root.

Another example: $\sqrt{3} = \{5, 6\}$ because
$5 \cdot 5 = 25 \equiv 3 \ (mod\ 11)$ and
$6 \cdot 6 = 36 \equiv 3 \ (mod\ 11)$

We show a two-dimensional plot of all 13 of these points in Figure 4.25. It does not resemble the corresponding continuous elliptic curve in the slightest! For one thing there are an infinite number of points on the continuous elliptic curve, and here we have only 11 points. Also, because of the modulus the shape does not resemble the shape of the continuous elliptic curve.

4.10.4 Addition of points on a discrete elliptic curve

Addition of points on discrete elliptic curves is very much like addition on continuous elliptic curves; though we use the same formulas, modulo p, the graphical representation does not follow suit, as you might have guessed. The discrete elliptic curve given by $y^2 = x^3 + ax + b \ (mod\ p)$ where p is prime, must not be singular, i.e. the discriminant is not zero.

If $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ are two points on the curve, then we can find the sum $P_1 + P_2 = P_3 = (x_3, y_3)$ as follows:

- If $P_1 = \infty$, $P_3 = P_2$
- If $P_2 = \infty$, $P_3 = P_1$

- Find the slope:
  - If $P_1 \neq P_2$, $m = (y_2 - y_1) \cdot (x_2 - x_1)^{-1}$ $(mod\ p)$
  - If $P_1 = P_2$, $m = (3x_1^2 + a) \cdot (2y_1)^{-1}$ $(mod\ p)$
  - If either inverse above is undefined, the slope is undefined. The sum, $P_3 = \infty$.
- $x_3 = m^2 - x_1 - x_2$
  $y_3 = m(x_1 - x_3) - y_1$
- The sum is $P3 = (x_3, y_3)$

Note that in calculating the slope, instead of dividing we multiply by the multiplicative inverse (mod p). Thus we always satisfy the property that $(x \cdot y^{-1}) \cdot y \equiv x\ (mod\ p)$

Examples of addition on a discrete elliptic curve

Here we use the same curve described above:
$y^2 = x^3 + x + 1\ (mod\ 11)$
We'll work through three examples:

- (1,5) +(1,6)
- (3,8) + (6,5)
- (8,2) + (8,2)
- First we add two different points with the same x coordinate; they lie on a vertical line: (1,5) + (1,6)
  $P_1 = (1, 5)$
  $P_2 = (1, 6)$
  The slope m is undefined, so the result is:
  $P_1 + P_2 = \infty$
- Now we add two points with different x coordinates:
  (3,8) + (6,5)
  We'll need to calculate the slope, m, of the line connecting them
  $P_1 = (3, 8)$
  $P_2 = (6, 5)$

$$\begin{aligned} y^2 &= x^3 + ax + b\ (mod\ p) \\ y^2 &= x^3 + x + 1\ (mod\ 11) \\ m &= (y_2 - y_1) \cdot (x_2 - x_1)^{-1}\ (mod\ p) \\ m &= (5 - 8) \cdot (6 - 3)^{-1}\ (mod\ 11) \end{aligned}$$

$$
\begin{aligned}
&= -3 \cdot 3^{-1} \ (mod\ 11) \\
&\equiv 8 \cdot 4 \ (mod\ 11) \\
&\equiv 10 \ (mod\ 11) \\
x_3 &= m^2 - x_1 - x_2 \ (mod\ p) \\
&= 10^2 - 3 - 6 \ (mod\ 11) \\
&= 100 - 9 \ (mod\ 11) \\
&\equiv 1 - 9 \ (mod\ 11) \\
&= -8 \\
&\equiv 3 \ (mod\ 11) \\
y_3 &= m(x_1 - x_3) - y_1 \ (mod\ p) \\
y_3 &= 10 \cdot (3 - 3) - 8) \ (mod\ 11) \\
&= -8 \\
&\equiv 3 \ (mod\ 11)
\end{aligned}
$$

(3,8) + (6,5) = (3,3)
Note that (3,3) is a point on the elliptic curve (see Figure 4.24).

- Now we add a point to itself; we'll need to calculate the slope, m, of the tangent line: (8,2) + (8,2)
  $P_1 = P_2 = (8, 2)$

$$
\begin{aligned}
y^2 &= x^3 + ax + b \ (mod\ p) \\
y^2 &= x^3 + x + 1 \ (mod\ 11) \\
m &= (3x_1^2 + a) \cdot (2y_1)^{-1} \ (mod\ p) \\
m &= (3 \cdot 8^2 + 1) \cdot (2 \cdot 2)^{-1} \ (mod\ 11) \\
&= (3 \cdot 64 + 1) \cdot 4^{-1} \ (mod\ 11) \\
&\equiv (3 \cdot 9 + 1) \cdot 3 \ (mod\ 11) \\
&= 28 \cdot 3 \ (mod\ 11) \\
&\equiv 6 \cdot 3 \ (mod\ 11) \\
&= 7 \ (mod\ 11) \\
x_3 &= m^2 - x_1 - x_2 \ (mod\ p) \\
x_3 &= 7^2 - 8 - 8 \ (mod\ 11) \\
&= 49 - 8 - 8 \ (mod\ 11) \\
&\equiv 5 - 8 - 8 \ (mod\ 11) \\
&= (-3) - 8 \ (mod\ 11) \\
&\equiv 8 - 8 \ (mod\ 11) \\
&= 0 \ (mod\ 11) \\
y_3 &= m(x_1 - x_3) - y_1 \ (mod\ p)
\end{aligned}
$$

$$\begin{aligned} y_3 &= 7(8-0)-2 \ (mod\ 11) \\ &= 56-2 \ (mod\ 11) \\ &= 54 \ (mod\ 11) \\ &\equiv 10 \ (mod\ 11) \end{aligned}$$

(8,2) + (8,2) = (0,10)
Note that (0,10) is a point on the elliptic curve (see Figure 4.24).

Subtraction of points

Subtraction of points on a discrete elliptic curve can be described easily in terms of addition. We note that for any given x coordinate of a point on the elliptic curve, there are two possible y coordinates whose sum is congruent to the prime modulus. For example, in Figure 4.24 the two y values on each row add up to 11, which is the prime modulus. For the x value 2, the y value is 0, which is congruent to 11, mod 11. This means that $-(x,y) = (x,-y) \equiv (x, y\ MOD\ p)$

To subtract two points, $A = (A_x, A_y)$ and $B = (B_x, B_y)$
$A - B = A + -(B) = A + (B_x, -B_y)$

4.10.5 Multiplication by integers

To multiply a point on a discrete elliptic curve by an integer, we can use repeated addition. For example, if P is a point on a discrete elliptic curve, then:
$3 \cdot P \equiv P + P + P$
We can speed this up by using intermediate results. To multiply the point P by 17:
$P_2 = P + P$
$P_4 = P_2 + P_2$
$P_8 = P_4 + P_4$
$P_{16} = P_8 + P_8$
$17 \cdot P = P_{16} + P$
This requires only 5 additions, rather than 16 additions.

As an example, we use the same elliptic curve:
$y^2 = x^3 + x + 1 \ (mod\ 11)$
and we saw that (8,2) is a point on this elliptic curve. We wish to find the product $13 \cdot (8,2)$

First we find the powers of 2, for this point:

$$\begin{aligned} P_1 &= (8,2) \\ P_2 &= P_1 + P_1 = (8,2) + (8,2) = (0,10) \\ P_4 &= P_2 + P_2 = (0,10) + (0,10) = (3,8) \\ P_8 &= P_4 + P_4 = (3,8) + (3,8) = (6,6) \end{aligned}$$

Noting that $13 = 8 + 4 + 1$:

$$\begin{aligned} 13 \cdot (8,2) &= P_8 + P_4 + P_1 \\ &= (6,6) + (3,8) + (8,2) \\ &= (6,6) + (1,5) \\ &= (8,9) \end{aligned}$$

4.10.6 Discrete log problem for elliptic curves

One important aspect of discrete elliptic curves is the fact that the discrete log problem is hard. This technically means that for a large modulus it will take too long to find a solution. The discrete log problem is defined as:
Given points, P and Q, find the integer m such that $P = m \cdot Q$ if such an integer exists.

Note that since the additive inverse of a point, $P = (x, y)$ is defined as:
$-P = -(x, y) = (x, -y)$
we must allow for a negative solution to the discrete log problem. For example, if m is positive, and $P = -m \cdot -Q$, then $-m \equiv p - m \ (mod\ p)$ is also a solution to the discrete log problem because $-m \cdot -Q = m \cdot Q$.

We can solve this problem with a brute force (i.e. sequential) search, trying every possible value of m, from 2 to the modulus, p.[17] If that fails to find a solution, we must try the negative integers from -2 to -p. We know of no faster way to solve this problem, and this is why the discrete log problem for elliptic curves is considered *hard*.

To understand why this is called 'log', recall that the addition of points on a discrete elliptic curve is called 'addition' because it forms an algebraic group. We might have just as well called it 'multiplication' since multiplication of integers also forms an algebraic group. In that case we could have defined a *power* operation as:
P^m, where m is a positive integer, as repeated multiplication. For example,
$P^5 = P \cdot P \cdot P \cdot P \cdot P$
If we had done so, the logarithm terminology would be more apparent. Given two points, P and Q, find the integer m, such that $P = Q^m$, if it exists.

As an example we use the same discrete elliptic curve:
$y^2 = x^3 + x + 1 \ (mod\ 11)$
Two of the points on this curve are (8,9) and (4,5). We wish to find the log of the point (8,9) with base (4,5), i.e. we will find $log_{(4,5)}((8,9))$.

We will search for a positive integer, m, such that: $(8,9) = m \cdot (4,5)$
Our brute force search, plays out as follows:

$$2 \cdot (4,5) = (6,5)$$

[17] Actually we can stop at p-1 because $p \equiv 0 \ (mod\ p)$

$$\begin{aligned} 3 \cdot (4,5) &= (1,6) \\ 4 \cdot (4,5) &= (0,1) \\ 5 \cdot (4,5) &= (8,2) \\ 6 \cdot (4,5) &= (3,8) \\ 7 \cdot (4,5) &= (2,0) \\ 8 \cdot (4,5) &= (3,3) \\ 9 \cdot (4,5) &= (8,9) \end{aligned}$$

Note that we stop at m = 9, having found the solution. If that search had failed we would have searched for a negative integer, m, such that:
$(8,9) = m \cdot (4,-5)$

4.10.7 Fast square roots

In chapter 6 we will need to find the two y coordinates on a discrete elliptic curve, for a given an x coordinate, i.e. we will need to find the square roots of the right side of:
$y^2 = x^3 + a \cdot x + b \ (mod\ p)$
Generally this will require a sequential search of all integers less than p. However, there is a much faster way to do this, in the case where $p \equiv 3 \ (mod\ 4)$, which is the case for the following prime numbers:
3, 7, 11, 19, 23, 31, 43, ...
$\sqrt{x} \equiv \pm x^{(p+1)/4}$ iff x has square roots, and $p \equiv 3 \ (mod\ 4)$.

4.10.8 Exercises

1. For the discrete elliptic curve given by $y^2 = x^3 - 4x + 4 \ (mod\ 23)$
   (a) Determine whether this elliptic curve is singular.
   (b) Show a table, similar to Figure 4.24, of all the points on this elliptic curve.
   (c) Show a graph, similar to Figure 4.25, of this elliptic curve.
   (d) Calculate the following sums on this discrete elliptic curve:
      i. (4,11) + (4,12)
      ii. (2,21) + (11,7)
      iii. (6,9) + (6,9)
2. For the elliptic curve given by $y^2 = x^3 - 4x + 4 \ (mod\ 23)$ find the product $5 \cdot (6,9)$
3. For the elliptic curve given by $y^2 = x^3 - 4x + 4 \ (mod\ 23)$ find the integer, m, such that $(4,11) = m \cdot (11,16)$

4. Write a computer program to implement calculations on a discrete elliptic curve.

   (a) Allow the user to specify the elliptic curve coefficients, `a` and `b` and the prime modulus.

   (b) Determine whether the elliptic curve is singular; if so, get new values for the coefficients and modulus from the user.

   (c) Use a data structure, such as an array or list, to store all the points on the curve.

   (d) Implement addition of any two points on the curve.

   (e) Implement multiplication of a point by a positive integer.

   (f) Implement a brute force solution to the discrete log problem.

Chapter 5

Integrity: Hash Functions

[This chapter assumes the reader has seen sections 4.3 and 3.1]

5.1 Requirements and desirable features

In recent years one of the more important tools to have been developed is known as *hash function*. These are called functions, from the mathematical concept of a mapping from one set of values, called the *domain*, to another (or the same) set of values, called the *range*. Hash functions have many areas of application, including cryptography, databases, data structures, passwords, and random number generators. As with all mathematical functions, a hash function is a many-to-one mapping, meaning that a single value in the domain must map to a unique value in the range. Given a value, `x` in the domain, there is one and only one value, `y`, such that `hash(x) = y`.

The domain of a hash function can be larger than its range, as shown in Figure 5.1, and in cryptographic applications the domain is typically much larger than the domain.[1] Also, the domain is not limited to primitive types, such as 32-bit integers, but may be long bit strings, or even an unlimited stream of bits.

Because the domain is generally a larger set than the range, it follows that there must be cases where two different values from the domain map to the same value in the range:
`hash(x1) = hash(x2)`
This is called a hash *collision* and is depicted in Figure 5.2.

5.1.1 Requirements for hash functions

As mentioned above, the mapping of a hash function is many-to-one (not one-to-many). We could say that if $x_1 = x_2$ then $hash(x_1) = hash(x_2)$.

[1] Hash functions are often called *message digests*, abbreviated MD, because they can accept a huge input and produce a much smaller output (think of Reader's Digest).

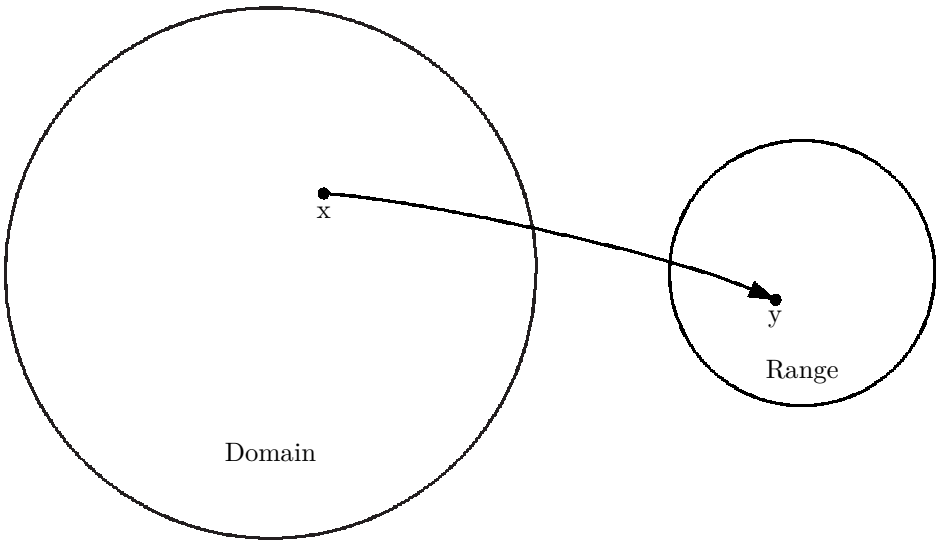


Figure 5.1: A hash function maps a value,x, from its domain to a value, y, in its range. y = hash(x)

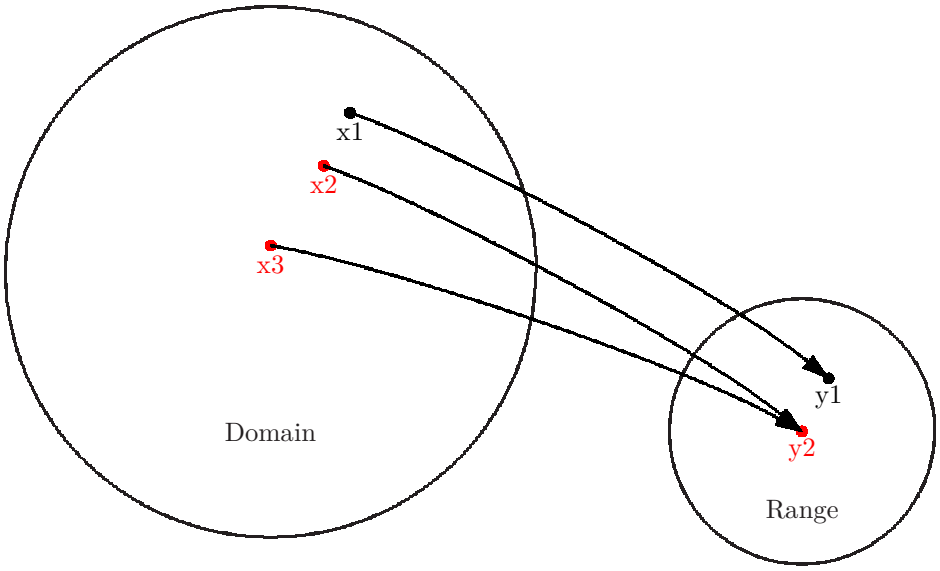


Figure 5.2: A hash function collision: hash(x2) = hash(x3)

In object-oriented programming languages, such as Java, hash functions are called *hashCode()* and always return a 32-bit integer. If using Java, this requirement can be expressed as follows:
When `obj1.equals(obj2)` is true, then `obj1.hashCode() == obj2.hashCode()` must also be true.[2]

Note that we can find a trivial hash function which obeys this requirement:
`hash(x) = 17;`
A hash function which always returns the same value satisfies the requirement (though it is not a good hash function).

5.1.2 Desirable properties for hash functions

We now describe the properties of a hash function which make it a good or effective hash function. These properties, for the most part, are desirable in cryptographic applications, but also apply to other applications. We will say that a hash function is *effective* if it satisfies the properties shown below.

These properties are all expressed in terms of problems which are computationally *hard*. A hard problem may be easy to program, but the run time of the program will be so large as to be unusable.[3] The names of the properties are `Preimage`, `Second preimage`, and `Collision`.

Preimage - Problem: Given a value from the range, y, find a value from the domain, x, such that hash(x) = y.

Second preimage - Problem: Given a value from the domain, x1, find a different value from the domain, x2, such that hash(x1) = hash(x2).

Collision - Problem: Find two values from the domain, x1 and x2, such that hash(x1) == hash(x2).

A good (i.e. effective) hash function will be designed in such a way that all three of these problems are computationally hard.

5.1.3 Exercises

1. For each of the following hash functions, determine whether it is a valid hash function, and, if valid, determine whether it is an effective hash function:

[2] Many students confuse this statement with its converse: When `obj1.hashCode() == obj2.hashCode()` is true, then `obj1.equals(obj2)` must also be true. The converse of a true statement is not always true.

[3] An example of a problem which is thought to be computationally hard is the *traveling salesman* problem: Given a graph, of cities, which a salesman needs to visit, with the distance or travel time between cities, find a shortest path which visits every city exactly once. All known solutions to this problem take so long to execute, for moderately large graphs, that they are unusable.

(a) Domain: integers
Range: integers
hash(i) = i / 2

(b) Domain: Double precision floating point numbers
Range: 32-bit integers
hash(x) = floor((31 * x * - 7) / 15.3) ($mod\ 2^{32}$)

(c) Domain: Strings of ASCII characters
Range: integers
hash(str) = length of str + ascii code of first character + ascii code of last character

(d) Domain: {-2,0,5,7}
Range: {-17,-2,0,17}

x	hash(x)
5	17
0	0
7	-17
-2	-2
5	-17

5.2 Applications

5.2.1 Hashtables

Hashtables are thought to be the original application to make use of hash functions. A hashtable is a data structure which is used to store a collection of values; it is designed to expedite searching the table for a given value. Hashtables may take different forms; one such form could be an array of lists, as shown in Figure 5.3.[4] To install a value in the hashtable, the value is used as the argument to a hash function, for which the result is an (apparently random) integer, call it a *hashCode.*. The hashCode can be used to select one of the lists in the hashtable: take the absolute value of the hashCode, and divide by the length of the array. The remainder can be used to select a list in the hashTable:
$ndx = abs(hashCode)\%array.length$
The value to be installed can then be added to the list at position `ndx` in the array.

When searching the hashtable for a given target value, it is not necessary to examine all values sequentially. The target is used as the argument to the hash function, and an index is computed as described above. The selected list is searched; if the target is in the hashTable, it must be in the selected list.

Hashtables are most efficient when the lists are approximately equal in length, minimizing the amount of searching necessary. There are algorithms to expand the table by increasing the size of the array and rebuilding the table.

[4] A hashtable can also be designed as a large array of values, with empty positions. When installing a value which hashes to an existing value, it *probes* for an empty position.

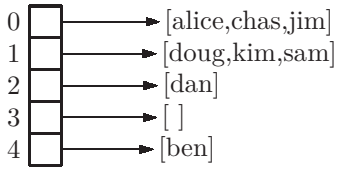


Figure 5.3: A hashtable as an array of lists. The values are strings which were entered in alphabetic order.

If the client were to somehow examine the hashtable, the values would appear to be in a somewhat random order, and not in the order in which they were installed. This is probably the origin of the term *hash* in this context; the values are shuffled, much like the ingredients in corned beef hash. In Figure 5.3 the values were installed in alphabetic order.[5]

5.2.2 Integrity/Verification

Secure communication requires more than confidentiality. If we are concerned that our enemy may be altering our communications, we need to find a way to ensure data *integrity*. There are cryptographic algorithms which will do this, primarily using hash functions.

Suppose that Alice sends a message through the internet to Bob: "The US consulate will be attacked at 6:00 a.m. tomorrow!".[6] Eve, working for the enemy, then does the following:

1. Eve intercepts the message.
2. She changes the text of the message to "All is clear at the consulate".
3. She sends the message to Bob, with the original sender id: Alice.
4. Bob receives the altered message, thinking it came directly from Alice. He never sees the original message.

This scenario constitutes a breach of data integrity. A public hash function can be used to foil the perpetrator.

We should be clear that hash functions will not prevent this kind of tampering with our communication. However, hash functions will make it possible for the receiver to determine whether tampering has taken place.

Here is how Bob and Alice can use a public hash function to assess the integrity of their communication:

1. Before sending the message, Alice uses it as the argument for a standard public hash function on which they have agreed. The function result is

[5] We have used the Java `hashCode()` method to install the strings.

[6] Alice does not encrypt the message because she is not concerned with confidentiality for this message.

`h1`:
`h1 = hash(msg)`

2. She sends `h1` to Bob.

3. After waiting a short time, she sends the message to Bob, in the clear.

4. Bob receives a message, `msg'`, ostensibly from Alice, and uses it as an argument to the hash function. Call the function result `h2`:
`h2 = hash(msg')`

5. Bob compares `h1` with `h2`. They should be equal; they are presumably the output of the hash function for the same message. If `h1` is not equal to `h2`, he is sure that there was tampering, and he disregards the message.

6. If `h1` is equal to `h2`, he is fairly certain that there was no tampering. It is possible that two different messages can produce the same hash function result (a collision), but if the hash function is a good one, this is very unlikely.

5.2.3 Message Authentication Codes (MAC)

However, there is a flaw in the algorithm given above to detect tampering. Suppose Eve does the following:

1. She intercepts the hash value, `h1`. She does not forward it to Bob.

2. She intercepts the message sent by Alice. She alters the message, and uses the altered message as the argument to the hash function.[7] Call the function result `h2`:
`h2 = hash(msg')`

3. She sends `h2` to Bob, with the original sender id: Alice.

4. After waiting a short time, Eve sends the altered message to Bob, also with the original sender id: Alice.

5. Bob receives `h2` and the altered message, `msg'`, thinking they were both sent directly from Alice.

6. Bob uses the altered message as the argument to the hash function. Call the result `h3`:
`h3 = hash(msg')`

7. Bob compares `h2` with `h3`. Since they are equal, he concludes that tampering is very unlikely.

[7] Recall that the hash function is public.

A better algorithm to ensure integrity is to use a *keyed* hash function, also known as a *Message Authentication Code*, or MAC. The hash function is still public, but it uses a private key, k, which Alice and Bill share. Eve does not know the private key.[8] We denote this function, using the private key, as $hash_k(msg)$. We are now using a *family* of hash functions, one for each possible key value. The hash function can employ a block cipher using the key, as part of its function computation. If Eve does not know the private key, it is very unlikely that she will be able to obtain a hash function result from the altered message, which matches the hash function result of the original message.

5.2.4 Encryption

Hash functions can be used for communications confidentiality, i.e. encryption and decryption. The plaintext is in fixed-size blocks, and a shared private key is used. As each block of plaintext is processed, the key is concatenated with the previous block of plaintext (if there is one) and then hashed, and the hash output is used to obtain a block of ciphertext using an exclusive OR with the plaintext. Algebraically:

Encryption

$$
\begin{aligned}
K_1 &= hash(K) \\
y_1 &= K_1 \oplus x_1 \qquad (5.1) \\
K_2 &= hash(K||x_1) \\
y_2 &= K_2 \oplus x_2 \qquad (5.2) \\
&\ldots \\
K_i &= hash(K||x_{i-1}) \\
y_i &= K_i \oplus x_i \qquad (5.3)
\end{aligned}
$$

Figure 5.4 shows a diagram for encryption using a hash function.

Decryption

We will now derive the decryption formulas for our hash function encryption algorithm, beginning with the encryption formulas. First we solve equation 5.1 for x_1 by forming the exclusive OR of both sides with K_1:

$$
\begin{aligned}
y_1 \oplus K_1 &= K_1 \oplus x_1 \oplus K_1 \\
y_1 \oplus K_1 &= x_1
\end{aligned}
$$

[8] See chapter 6 for a discussion of secure key distribution algorithms.

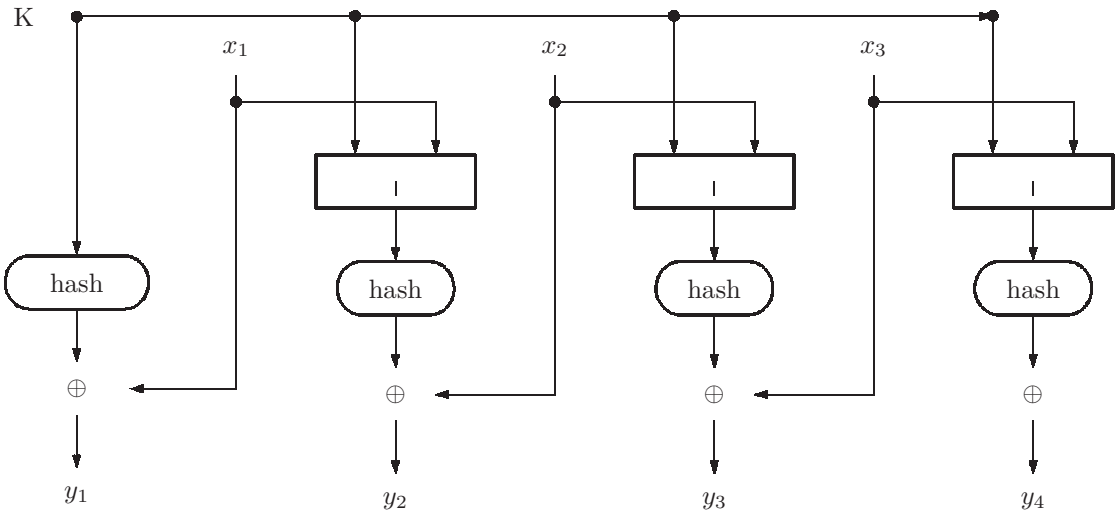


Figure 5.4: Diagram of encryption with a hash function. The input to the hash function can be a block or a double-block. The output is one block.

Next we solve equation 5.2 for x_2 by forming the exclusive OR of both sides with K_2:

$$\begin{aligned} y_2 \oplus K_2 &= K_2 \oplus x_2 \oplus K_2 \\ y_2 \oplus K_2 &= x_2 \end{aligned}$$

Next we solve equation 5.3 for x_i by forming the exclusive OR of both sides with K_i:

$$\begin{aligned} y_i \oplus K_i &= K_i \oplus x_i \oplus K_i \\ y_i \oplus K_i &= x_i \end{aligned}$$

The equations generating the keys for each block, K_1, K_2, ..., can be used as is, since they use the previous block's plaintext. The decryption formulas are:

$$\begin{aligned} K_1 &= hash(K) \\ x_1 &= y_1 \oplus K_1 \\ K_2 &= hash(K||x_1) \\ x_2 &= y_2 \oplus K_2 \end{aligned}$$

$$\begin{aligned} &\ldots \\ K_i &= hash(K||x_{i-1}) \\ x_i &= y_i \oplus K_i \end{aligned}$$

5.2.5 Other

There are other applications which make use of hash functions:

- Digital signatures, described in chapter 6, are used to authenticate the identity of people or other entities involved in communication.

- Digital certificates, also described in chapter 6, are used as digital ID cards, for authentication of individuals or other entities.

- System password files make extensive use of hash functions. If a multiuser system, or web site, has many users with passwords for security, the system does not store the passwords. If the passwords were stored, the risk of a security breach would be too great. A disgruntled system administrator could easily access the password file before leaving the company. Instead, when a password is first generated, it is used as the argument to a hash function, and the hash result is stored in the password file. Now a hacker would have to find a password which hashes to a user's stored password hash value, in order to break in. This is essentially the preimage problem stated above.

- Crypto-currencies such as Bitcoin, presented in chapter 12, make extensive use of hash functions. In this context the hash function is often called a *fingerprint*. Like a human fingerprint, it can be used to identify a data source because multiple data sources are extremely unlikely to have the same fingerprint. Bitcoin uses fingerprints to verify blocks of transactions in a chain of blocks known as the *blockchain*. Despite the billions of dollars represented by the blocks on Bitcoin's blockchain, no hackers have succeeded in altering a transaction for their own benefit. Bitcoin uses a secure hash function.

- Hash functions can be used to generate sequences of *random numbers*, which have many applications, including simulations, scientific research, computer networks, and lottery systems. By using the output of the hash function as the next input to that hash function iteratively, we can get a reproducible sequence of apparently random numbers. To obtain a different random number sequence we would begin with different *seed* or starting value. If the hash function is a good one, the sequence will not repeat itself in our lifetime.

5.2.6 Exercises

1. For Java programmers:
   Find the hashCode values for each of the Strings:
   `"alice", "ben", "chas ", "dan", "doug", "jim", "kim", "sam"`
   Show how these values are used to build the hashtable of Figure 5.3.

2. Secret agent Bond wishes to send a message to secret agent Smiley. Bond is not concerned with confidentiality, but he is afraid the enemy may tamper with the message. Agents Bond and Smiley have agreed on a secure, public hash function.

   (a) Agent Bond creates the message "Dr. No is on your tail". He uses this message as input to a hash function.

   (b) The hash result, in decimal, is 1032383.

   (c) Agent Bond sends the hash result, 1032383, to agent Smiley.

   (d) Agent Smiley receives the hash result, 1032383, apparently from agent Bond.

   (e) Agent Bond sends the message "Dr. No is on your tail" to agent Smiley.

   (f) Agent Smiley receives the message "Dr. No is on your tail" apparently from agent Bond.

   (g) Agent Smiley uses the message he received as input to the hash function. The hash result is 1032383.

   Agent Smiley can now conclude which one of the following:

   - He is sure that nobody tampered with the message
   - He is almost certain that nobody tampered with the message
   - He is sure that somebody tampered with the message
   - He is almost certain that somebody tampered with the message
   - None of the above

3. An example of a hash function which uses a permutation/selection vector is defined below:
   Domain: 8-bit integers
   Range: 4-bit integers
   $hash(x) = x[3, 2, 7, 0]$

   (a) If x=01101010, find hash(x)

   (b) Preimage: Find a value from the domain, x, such that hash(x) = 1100

   (c) Second preimage: x1 = 00110101. Find a different value from the domain, x2, such that hash(x1) = hash(x2).

(d) Collision: Find two values from the domain, x1 and x2, such that:

- hash(x1) = hash(x2)
- Neither x1 nor x2 is 00110101
- Neither hash(x1) nor hash(x2) is 1100

4. Refer to the hash function defined in the previous problem.

   (a) Show how that hash function can be used with key K = 1011 to encrypt the 12-bit message: 1001 0110 1111.

   Hint: Use a block size of 4 bits, prepending 0's to a block if necessary.

   (b) Using your solution to part (a) show how you can decrypt the ciphertext to obtain the original 12-bit message.

5. Show a diagram corresponding to Figure 5.4 for decryption.

6. An example of a MAC, with key K, is defined below:
   Domain: 8-bit integers
   Range: 4-bit integers
   $hash(x) = x[3, 2, 7, 0] \oplus K$
   If K=0111 and x=01101010, find hash(x)

5.3 A simple hash function

To further expose the concept of hash function, we present a simple example of a hash function which can take an input of unlimited size and produce a fixed size output. Think of the input as a stream of bits, to be processed one block at a time. The output will be an 8-bit integer.

The domain for this hash function will be an unlimited stream of bits. The range will be an 8-bit block. The hash function will process the input in blocks of 8 bits, and it will use an initial vector, IV = 0001 1010. The input is padded with 0's at the end, if necessary, to form a full block. As each block is read, it is used as an input to an exclusive OR operation. The other input to the exclusive OR is the hash result obtained thus far (for the first block this will be IV).

Algebraically:
$result \leftarrow IV \oplus x_1$
$result \leftarrow result \oplus x_2$
$result \leftarrow result \oplus x_3$
...
$result \leftarrow result \oplus x_n$

A diagram of this simple hash function is shown in Figure 5.5.

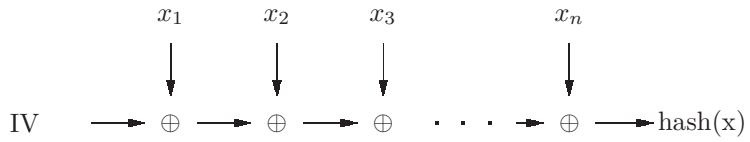


Figure 5.5: Diagram of a simple hash function with unlimited input and an initial vector, IV.

5.3.1 Exercises

1. Show the result when the 20-bit stream 0110 1110 1010 1111 0111 is used as input to the hash function of Figure 5.5. Assume that:
   - the initial vector is IV = 0001 1010
   - the block size is 8 bits
   - the input is padded with 0's at the end to form a full block
2. Find a collision of two input values each of which is at least 16 bits, for the hash function given in the previous problem.

5.4 SHA-1

There are many standard hash functions designed to be secure with respect to the three properties previously described: Preimage, Second preimage, and Collision. They are generally in the public domain (available to everyone) and many are open source. The names of these standard functions are usually of the form MDn or SHA-n, where n could represent a version number, or the size of the output, in bits. The MD stands for Message Digest,[9] and the SHA stands for Secure Hash Algorithm.

In this section we take a careful look at a standard hash function known as SHA-1. We begin by explaining some of the operations used in this algorithm.

5.4.1 Preliminary operations

Before getting into the SHA-1 algorithms we expose some of the operations and notation we will be using.[10]

- Rotation of bits: $x \hookleftarrow n$ is a *rotate* operation to the left. The bits of x are shifted to the left by n bits, high-order bits shifted off the left end

[9]The word *digest* is used when speaking of hash functions because they can take a large volume of input, and squeeze it to a much smaller output, as in the periodical Reader's Digest. This should not be confused with compression algorithms, which permit decompression to obtain the original data.

[10]See the open source textbook on *Computer Organization* for a more complete discussion of bit manipulation and boolean operations.

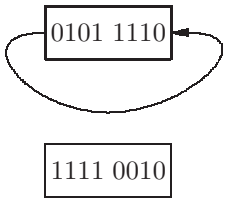


Figure 5.6: Diagram of a 3-bit rotation used in SHA-1

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

Figure 5.7: A table showing the 16 hexadecimal digits, and their corresponding decimal values

are shifted back in at the right end. For example, if x is the 8-bit value 0101 1110, then after $x \leftarrow 3$ has completed, the value of x is 1111 0010. dropping off the left end. A diagram of this rotation is shown in Figure 5.6.

A rotation to the *right* is described with an arrow pointing in the other direction: $x \hookrightarrow 4$, but we will not need this operation in SHA-1.[11]

- SHA-1 works with 32-bit *words*. Rather than write out all 32 bits of a word, we use a shorthand notation called *hexadecimal*, or base 16. In this notation there are 16 digits, including 0-9 and a,b,c,d,e,f, corresponding to the decimal values 10,11,12,13,14,15. This is summarized in Figure 5.7.

  Consequently, rather than writing out all 32 bits of a word, such as
  `0010 1010 0111 0010 1001 1111 0000 1000`
  we can write it in hexadecimal as

[11] A rotation should not be confused with a *shift* in which bits shifted off the end are discarded.

K		H	
K_0	5a82 7999	H_0	6745 2301
K_1	6ed9 eba1	H_1	efcd ab89
K_2	8f1b bcdc	H_2	98ba dcfe
K_3	ca62 c1d6	H_3	1032 5476
		H_4	c3d2 e1f0

Figure 5.8: A table showing the values of the 4 constants, K, and the initial values of the 5 variables, H, for SHA-1. All values are shown in hexadecimal.

$f_0(B,C,D) = B \cdot C + B' \cdot D$
$f_1(B,C,D) = B \oplus C \oplus D$
$f_2(B,C,D) = B \cdot C + BD + CD$
$f_3(B,C,D) = B \oplus C \oplus D$

Figure 5.9: Definition of the four boolean functions used in SHA-1

`2a72 9f08`.

- SHA-1 uses four 32-bit constants, called K, and it uses five 32-bit variables, called H. These constants and the initial values for H are shown in Figure 5.8.

- SHA-1 uses 4 boolean functions which perform bit-wise operations on 32-bit words. For example, $f_0(B,C,D) = B\dot{C} + B'D$. This means that this function will:

  1. Form the 32-bit bitwise AND of B with C,
  2. Negate each of the bits of B
  3. Form the bitwise AND of the result from step 2 with D
  4. Form the bitwise OR of the result from step 1 with the result from step 3. This is the final result of the function.

  For example:
  $B = 0403\ 1a22$
  $C = 0000\ ffff$
  $D = 1111\ 0123$k
  $B \cdot C = 0000\ 1a22$
  $B' \cdot D = fbfc\ e5dd \cdot 1111\ 0123 = 1110\ 0101$
  $0000\ 1a22 + 1110\ 0101 = 1110\ 1b23$
  Thus $f_0(B,C,D) = 1110\ 1b23$

  Figure 5.9 shows the definition of each of these boolean functions.

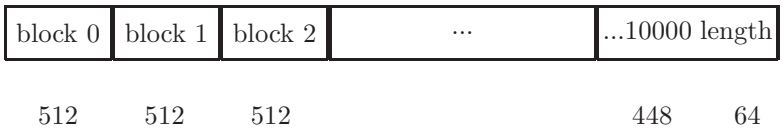


Figure 5.10: Diagram of the setup for SHA-1. The input consists of 512-bit blocks, followed by padding, followed by the length.

5.4.2 SHA-1 algorithm

We are now ready to begin our presentation of the SHA-1 algorithm. The input to SHA-1 must be of fixed size. That means the length must be known before the algorithm begins, though there is no given limit on the length.[12] The final result of SHA-1 will be 160 bits, but it works with 512-bit blocks.

As with most secure hash functions, there is much bit manipulation in SHA-1, including rotations, exclusive ORs, and other boolean operations. All this is designed to ensure security, i.e. the properties of Preimage, Second Preimage, and Collision are all hard. Reasoning, or justification, for the specific operations is beyond the scope of this text.

Initial setup

Begin by forming the input into a series of 512-bit blocks. It is necessary to store the length of the input, and padding, if necessary, to complete the last block. Use a 1 followed by several 0's, followed by the length(in bits) of the actual input as a 64-bit field. This is shown in Figure 5.10.

.

Processing

After the initial setup, the blocks are processed sequentially, beginning with the first block. In doing so, we use four 32-bit variables, A, B, C, D, and an array of 32-bit variables, W, in addition to the H variables and K constants already defined. For each block, the steps are:

1. Divide the block into 16 32-bit sub-blocks, $W_0, W_1, W_2, ..., W_{15}$
2. For t = 16 to 79 do the following:
   $W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \hookleftarrow 1$
3. $A = H_0, B = H_1, C = H_2, D = H_3, E = H_4$
4. For t = 0 to 79
   (a) $T = (A \hookleftarrow 5) + f_{t/20}(B, C, D) + E + W_t + K_{t/20}$ [13]

[12] A length field of 64 bits is included with the input, implying that the number of bits in the input is limited to 2^{64}, which is far greater than the storage capacity of today's largest computers.

[13] The plus symbols here represent addition, not logical OR. The division is integer division.

(b) $E = D$

(c) $D = C$

(d) $C = (B \hookleftarrow 30)$

(e) $B = A$

(f) $A = T$

5. Update the H variables:
   $H_0 = H_0 + A$
   $H_1 = H_1 + B$
   $H_2 = H_2 + C$
   $H_3 = H_3 + D$
   $H_4 = H_4 + E$

6. The final result is H (160 bits).

5.4.3 Exercises

1. Apply the SHA-1 algorithm to the input string:
   01234567_{16}
   Show the steps involved and the final 160-bit output in hexadecimal.

   Hint: Since the input is so short, there will be only one 512-bit block, which contains the given input, the padding, and the length of the input.

2. Write a computer program to implement the SHA-1 algorithm.

5.5 Other current hash algorithms

Over the years many hash algorithms have been developed. Each time a standard hash algorithm is published, there is a flurry of activity as researchers (and hackers) attempt to break the new algorithm. When a system is able to find a Preimage, Second preimage, or Collision in a reasonable amount of time, we say the hash algorithm has been broken.

Secure hash algorithms are vital to the security of the internet, digital currencies, and other communication technologies. Figure 5.11 shows some public hash algorithms which have been developed over the years. As mentioned earlier, SHA stands for Secure Hash Algorithm, and MD stands for Message Digest. As mentioned earlier, the numeric suffix represents either a version number or the size, in bits, of the output. Thus SHA-1 is version 1 of SHA, MD5 is version 5 of MD, and SHA-256 is SHA with a 256-bit output.

One way to attack a secure hash function is called a *brute force* attack, in which the attacker tries every possible input until a collision, preimage or second preimage is found. When the range of the hash function (determined by the number of bits in the output) is sufficiently large, a brute force attack is

Name	Date	Output size (bits)	Secure?	Notes
SHA-1	1993	160	No	
SHA-256	2002	256	Yes	For 32-bit computers
SHA-512	2002	512	Yes	For 64-bit computers Used in Bitcoin
SHA-2	2001	224 or 256 or 384 or 512	Yes	Includes SHA-256 and SHA-512
SHA-3	2015	224 or 256 or 384 or 512	Yes	Also known as Keccak
BLAKE	2008	224 or 256 or 384 or 512	Yes	
RIPEMD	1996	160	Yes	Used in Bitcoin
Whirlpool	2000	512	Yes	Based on AES
MD2	1989	128	No	Used in PKI
MD4	1990	128	No	Based on MD2
MD5	1991	128	No	Based on MD4

Figure 5.11: Some standard hash algorithms. Source: Wikipedia

not likely to succeed. Note that as computers evolve with faster, and parallel, processors, the possibility of a brute-force attack on a hash function becomes more likely. Thus hash functions developed later will have larger outputs, to forestall brute force attacks by faster computers.

5.5.1 Exercises

1. What do the acronyms MD and SHA represent?

2. With today's technology, how large (i.e. how many bits) should a hash output be in order to foil a brute-force attack on a hash function?

Chapter 6

Public Key Encryption Algorithms

[This chapter assumes the reader has seen sections 3.1, 4.1, 4.2, 4.4, 4.5, 4.6, 4.7, and 4.8]

One of the most serious problems with private key encryption is the *key distribution* problem. In this chapter we will take a very different view of encryption and decryption, which offers a potential solution to this problem. Here we note the main distinctions between private and public cryptosystems:

Private key cryptosystems

As described in chapter 3:

- Encryption and decryption are done with the same key
- Key distribution is problematic
- Also known as *symmetric* cryptosystem
- Used exclusively for confidentiality

Public key cryptosystems

- Keys are created in pairs. Each pair consists of a public key and a secret key
- The public key is used for encryption, in which case the *receiver's* public key is used
- The secret key, also known as a private key, [1] is used for decryption, in which case the *receiver's* secret key is used. A summary of key terminology is shown in Figure 6.1.

[1] We use the term *secret* to distinguish from the keys used for private key encryption, described in chapter 3.

Key type	Shared with	Used in	Purpose
Private	Friends	Private key algorithms DES, AES, SDES, etc.	Encryption, Decryption
Public	Everyone	Public key algorithms RSA, ElGamal, etc.	Encryption Verifying signatures
Secret	No one	Public key algorithms RSA, ElGamal, etc.	Decryption Creating signatures

Figure 6.1: Key terminology, as used in this textbook

- Key distribution is not a problem (as long as authenticity is ensured)
- Is also used for applications other than confidentiality: authenticity, digital signatures, and certificates.

As with private key cryptosystems:

- Keys are simply bit strings.[2]
- The message to be encrypted is also a bit string. Whether it is a string of ascii characters, a photograph, sound clip, video clip, etc, internally it is just a string of bits.[3] Consequently any digital information can be encrypted.

A simplified view of public key encryption is shown in Figure 6.2.

6.1 Encryption with public/secret key-pairs

We first introduce some notation for encryption and decryption with key-pairs. $Encr(msg, K_{pub})$ denotes the encryption of a plaintext messsage, msg, using a *public* key, K_{pub}. If the resulting ciphertext is denoted as c, then
$c = Encr(msg, K_{pub})$
$Decr(c, K_{sec})$ denotes the decryption of a ciphertext, c, using the *secret* key, K_{sec}:
$msg = Decr(c, K_{sec})$ denotes the decryption of a ciphertext, c, using the the corresponding *secret* key, K_{sec}, and producing the original plaintext message, msg. Note that K_{pub} and K_{sec} form a *key pair* and are mathematically related.

Suppose Alice wishes to send a confidential message to Bob, and they have agreed to use a public key crypto-system. They each have created a pair of keys, individually. The sequence of events shown below is diagrammed in Figure 6.3.

1. Alice encrypts the message, msg, using Bob's public key, K_{Bpub}, to produce the ciphertext, c:
   $c = Encr(msg, K_{Bpub})$

[2]Current public cryptosystems use keys on the order of 200 bits in length.
[3]These bit strings can be blocked into fixed size blocks, each of which is then treated as an integer by the encryption and decryption algorithms.

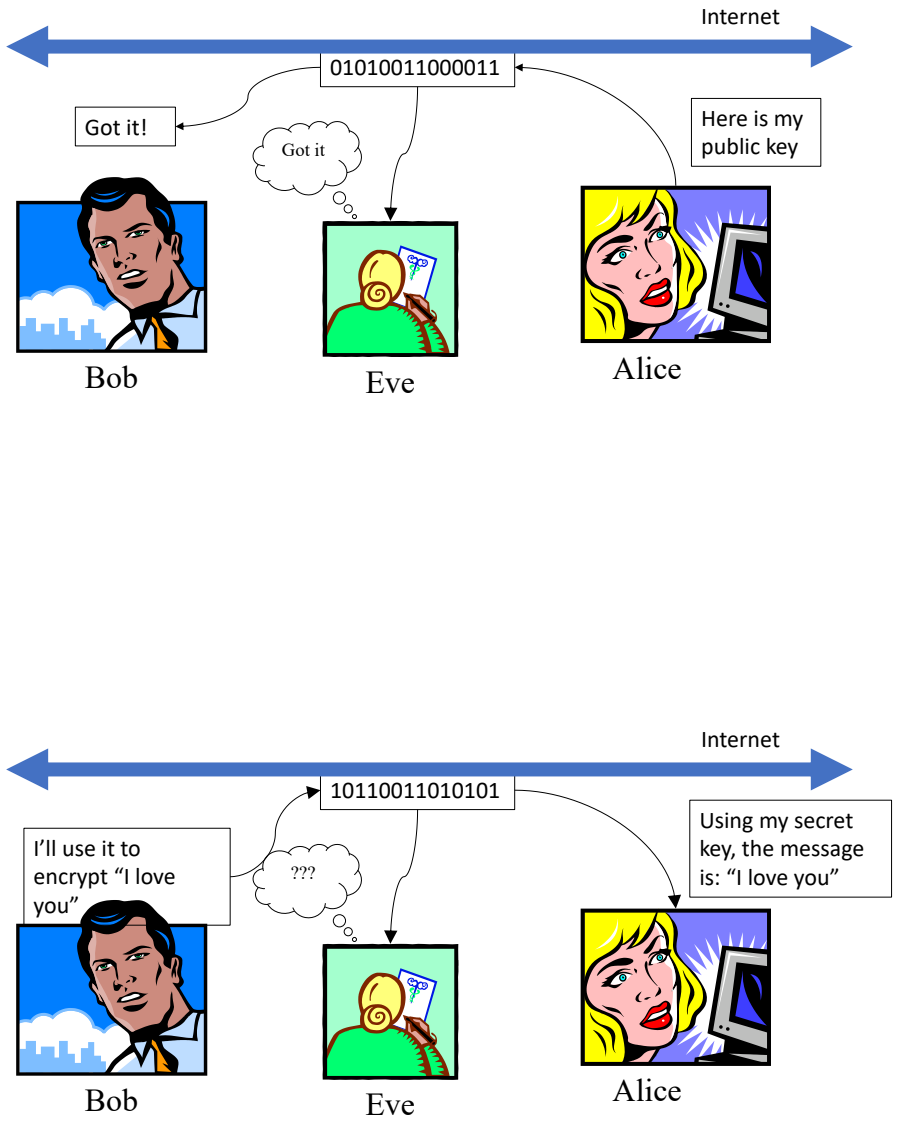


Figure 6.2: Alice sends her public key to Bob. Then Bob encrypts a message to Alice using her public key. Only Alice can decrypt the message; she uses her secret key.

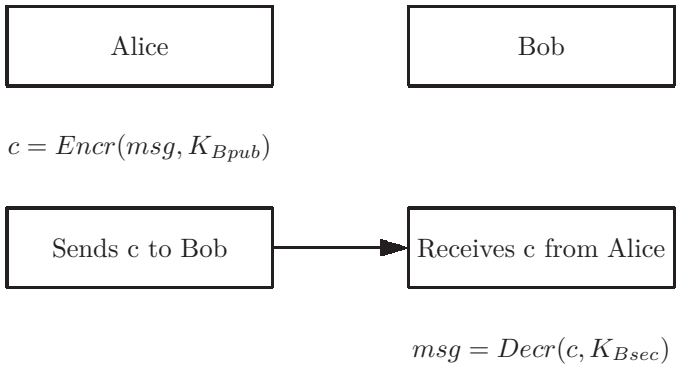


Figure 6.3: Alice encrypts the plaintext message, msg, with Bob's public key and sends the ciphertext, c, to Bob. Bob decrypts the ciphertext with his secret key to obtain the original message.

2. Alice sends the ciphertext, c, to Bob through an insecure channel, such as the internet. Anyone who intercepts c will not be able to decrypt it, because it can be decrypted only with Bob's secret key.
3. Bob receives the ciphertext, c.
4. Bob decrypts the ciphertext, c, using his own secret key, K_{Bsec} to produce the original message, msg:
   $msg = Decr(c, K_{Bsec})$

Note that this scheme appears to have solved the key distribution problem which we mentioned in chapter 3. Each person does not divulge his/her secret key to anyone; no one other than Bob needs to know Bob's secret key.

Each person or entity who wishes to use a public key cryptosystem will have their own key-pair(s):

- Alice's *public* key is denoted as K_{Apub}, and she broadcasts this public key to everyone; it is not secret. This public key can be used by anyone to encrypt messages to Alice (and only Alice).
- Alice's *secret* key is denoted as K_{Asec}, and she keeps this secret key to herself; nobody else needs to know this key. This secret key can be used to decrypt messages which were encrypted with Alice's public key.
- Alice's public and secret keys are mathematically related.
- Alice derived her secret key from her public key, but others are not able to do this.
- Bob's *public* key is denoted as K_{Bpub}, and he broadcasts this public key to everyone; it is not secret. This public key can be used by anyone to encrypt messages to Bob (and only Bob).

Key Pairs	
My Public Key	My Secret Key
Shared with everyone	Not shared with anyone
Used by others to encrypt messages for me	Used by me to decrypt messages encrypted with my public key
	Cannot be (easily) derived by others from my public key
My public key $\rightarrow$ *My secret key* Mathematically related	
	Only I can derive it from my public key

Figure 6.4: A summary of key pairs for a public key cryptosystem

- Bob's *secret* key is denoted as K_{Bsec}, and he keeps this secret key to himself; nobody else needs to know this key. This secret key can be used to decrypt messages which were encrypted with Bob's public key.
- Bob's public and secret keys are mathematically related.
- Bob derived his secret key from his public key, but others are not able to do this.

This description of key pairs is summarized in Figure 6.4. In the following sections we introduce some common public key algorithms.

We noted in chapter 3 that key distribution was a serious obstacle in establishing secure communication in a private key cryptosystem. Public key cryptosystems appear to solve the key distribution problem in a few ways:

- Since secret keys are used for decryption, and not encryption, there is no need to share these secret decryption keys with others.
- When people are communicating securely with a public key system, they can encrypt a key to be used with a private key system, and send the encrypted key to others. This encrypted key is often called a *session* key. Each time they start up a new communication session, a new shared session key can be distributed, using public key encryption.

The second bullet above is the mode which is used more frequently; one of the disadvantages of a public key cryptosystem is that encryption and decryption can take several minutes, whereas a private key system is much faster.

6.1.1 Exercises

1. Joe, Mary, and Pat are communicating using a public key cryptosystem. Their keys are shown in the table below:

	public key	secret key
Joe	K_{Jpub}	K_{Jsec}
Mary	K_{Mpub}	K_{Msec}
Pat	K_{Ppub}	K_{Psec}

For each of the following, show which one of the six keys would be used.

(a) Joe wants to encrypt a message being sent to Pat.

(b) Pat wants to encrypt a message being sent to Mary.

(c) Someone not shown in the table wants to encrypt a message being sent to Joe.

(d) Mary wants to decrypt a message that she received from Pat.

(e) Joe wants to decrypt a message that he received from Mary.

2. Which, zero or more, of the following are true of a public key cryptosystem involving several people?

(a) Each person has their own key pairs.

(b) Each person keeps their key pairs secret.

(c) If two people wish to communicate with confidentiality, they need to share secret (encryption/decryption) keys.

(d) They should share their secret keys with each other but not with anyone else.

(e) They should share their public keys with everyone.

(f) They should not share their secret keys with anyone.

(g) Each person will need to derive a secret key from their own public key.

6.2 RSA

The RSA public-key algorithm was developed by Rivest, Shamir, and Adleman in the late 1970's.[4] They shared the 2002 Turing Award for this work. RSA is one of the first public key algorithms developed, and it has become one of the most commonly used public key algorithms.

6.2.1 Derivation of the key pair

As described above, each user of this algorithm must derive a pair of mathematically related keys. In the RSA algorithm this is done with the following steps:

[4]Ron Rivest, Adi Shamir, and Leonard Adleman were all employed at MIT when they developed the algorithm now known as RSA.

- Choose two large prime numbers, p and q.
- Multiply the prime numbers; the product, $m = p \cdot q$, will be used as a public modulus. The number of bits in this modulus will determine the block size for the plaintext and ciphertext.
- Find the product $N = (p-1) \cdot (q-1)$
- Choose a number, e, which is relatively prime to N:
  $gcd(e, N) \equiv 1 \ (mod\ N)$
  This is the public encryption exponent.
- The pair of numbers, e and m, constitute the public key.
- Derive d, the discrete multiplicative inverse of e (mod N)
  $d \equiv e^{-1} \ (mod\ N)$
  This can be done with the extended Euclidean algorithm, described in chapter 4. The number d will be used as the secret key; it is the secret decryption exponent.

In the following example we derive an RSA pair of keys (for Bob), but we use small numbers to simplify the calculations.

- Choose a pair of prime numbers:
  p = 7, q = 13
- $m = p \cdot q = 91$ is the public modulus
- $N = (p-1) \cdot (q-1) = 6 \cdot 12 = 72$
- We choose $e = 5$ for the public encryption exponent. 5 is relatively prime to 72.
- We find the secret decryption exponent by finding the inverse of e (mod N).
  $d \equiv e^{-1} \pmod{m}$
  $d \equiv 5^{-1} \ (mod\ 72)$
  We do this with the extended Euclidean algorithm from section 4.6.

n	q	r	u	v	$u \cdot e + v \cdot m$
-2		72	0	1	72
-1		5	1	0	5
0	14	2	-14	1	2
1	2	1	29	-2	1

Thus from the bottom row we have $29 \cdot 5 + -2 \cdot 72 = 1$ We conclude that $d = 5^{-1} \equiv 29 \ (mod\ 72)$
and this is the secret decryption exponent. The public encryption key is the pair of numbers, e = 5, and m = 91. The secret decryption key (or exponent) is d = 29.

6.2.2 Encryption and decryption with RSA

In what follows, x represents the plaintext, and y represents the ciphertext. To encrypt a message we use the public key of the intended recipient. We use the public exponent, e on the plaintext as follows:
$y = Encr(x) = x^e \pmod m$
The recipient can then decrypt the ciphertext by using their secret exponent, d, on the ciphertext as follows:
$x = Decr(y) = y^d \pmod m$

We can explain why this works; we will need a few results from section 4.9:

- When evaluating an expression (mod m), we can work mod $\phi(m)$ in the exponent:
  $x^y \ (mod\ m) \equiv x^{y \ (mod\ \phi(m))} \ (mod\ m)$
- If m is the product of two primes, $m = p \cdot q$, then $\phi(m) = (p-1) \cdot (q-1)$
- $x^y \ (mod\ m) \equiv x^{y \ (mod\ (p-1)\cdot(q-1))} \ (mod\ m)$

To understand the RSA decryption algorithm, since the receiver of the message chose

$$\begin{aligned} d &\equiv e^{-1}(mod\ (p-1)\cdot(q-1)) \\ Decr(y) &= y^d \ (mod\ m) \\ &= (x^e)^d \\ &= x^{e\cdot d} \\ &\equiv x^{e\cdot d \ (mod\ (p-1)\cdot(q-1))} \ (mod\ m) \\ &\equiv x^1 \ (mod\ m) \end{aligned}$$

Why is the RSA encryption algorithm secure? Can an attacker simply find the factors of m, and then calculate $N = (p-1)\cdot(q-1)$? After calculating N, the attacker could use the extended Euclidean algorithm to find $d \equiv e^{-1} \ (mod\ N)$. The attacker can easily write a program to factor an integer, m, but if that integer is sufficiently large,[5]the program will take too much time to execute.[6]

6.2.3 Exercises

1. When generating a key-pair, an RSA user chooses two large prime numbers, p and q.

[5]By *large* we mean at least two hundred bits; a Java int is only 32 bits, and a long integer is 64 bits, so the BigInteger class would be used. A BigInteger uses an array of integers (bytes) to store a large integer.

[6]Factoring of large integers is known as a *hard* problem; an efficient algorithm has not been found. It is generally thought that there is no such efficient algorithm.

(a) Explain why the prime numbers need to be large.

(b) How large do the prime numbers have to be?

(c) What might happen if the prime numbers are not large enough?

(d) What happens if the prime numbers are too large?

(e) How can a Java programmer do arithmetic with very large integers (i.e. integers which exceed the capacity of `long` integers, 64 bits)?

2. When generating a key-pair, an RSA user chooses two large prime numbers, p and q. Then a public encryption exponent, e, is chosen, but it must be relatively prime to $N = (p-1) \cdot (q-1)$. Explain why e must be relatively prime to N.

3. In showing why the RSA decryption algorithm works, we made use of the algebraic identity:
$(x^y)^z = x^{yz}$
Give an example which illustrates this identity, i.e. choose appropriate values for x, y, and z.

6.3 RSA encryption/decryption example

Now suppose Alice wishes to encrypt the plaintext message x = 15 to be sent to Bob. She uses Bob's public key, {m = 91, e = 5}. With the modPower algorithm from section **??**:

$$\begin{aligned} y = Encr(x) &= x^e \ (mod\ m) \\ y = Encr(15) &= 15^5 \ (mod\ 91) \\ &= 15^4 \cdot 15 \ (mod\ 91) \\ &= 15^2 \cdot 15^2 \cdot 15 \\ &\equiv 43 \cdot 43 \cdot 15 \ (mod\ 91) \\ &\equiv 29 \cdot 15 \ (mod\ 91) \\ &\equiv 71 \ (mod\ 91) \end{aligned}$$

The ciphertext, y, is 71.

Bob receives the ciphertext and decrypts using his secret key, d = 29:

$$\begin{aligned} x = Decr(y) &\equiv y^d \ (mod\ m) \\ &\equiv 71^{29} \ (mod\ 91) \\ &= 71^{16} \cdot 71^8 \cdot 71^4 \cdot 71 \\ &\equiv 15 \ (mod\ 91) \end{aligned}$$

6.3.1 Degenerate RSA keys

We begin with another example. Bob chooses the prime numbers 5 and 11. His public modulus is $m = 5 \cdot 11 = 55$. To calculate his secrete decryption exponent he first calculates $N = \phi(m) = 4 \cdot 10 = 40$. He chooses a public encryption exponent $e = 21$, which should work, because 21 is relatively prime to 40, i.e. 21 has an inverse $\pmod{40}$.

Alice wishes to encrypt the message, $msg = 17$, for Bob.

$$\begin{aligned} cipher &= msg^e \pmod{m} \\ &= 17^{21} \pmod{55} \\ &= 17 \end{aligned}$$

The plaintext, $msg = 17$ is the same as the ciphertext, $cipher = 17$. The RSA algorithm has failed to encrypt the plaintext! Moreover, this RSA key will fail to encrypt every possible plaintext! Most would agree that this is not desirable.

It seems that certain RSA keys behave this way; these keys fail to encrypt all plaintexts. These keys are called *degenerate*[7] keys because of their inappropriate results.[8]

When choosing an RSA key, we certainly wish to avoid choosing a degenerate key. This can be done easily by using the following necessary and sufficient condition that an RSA key is degenerate:

An RSA key with

- Prime numbers p and q
- Public modulus $m = p \cdot q$
- Private $N = (p-1) \cdot (q-1)$
- Public encryption exponent, e, which is relatively prime to N

will fail to encrypt all plaintexts if and only if
$lcm(p-1, q-1) | (e-1)$
where lcm is *least common multiple* and $x|y$, read "x divides y", is true if y / x has no remainder, i.e. $y \equiv 0 \pmod{x}$.

In the example given here, $lcm(4, 10) = 20$, and the exponent, $e = 21$. Since $20|(21-1)$, the RSA key is degenerate and will fail to encrypt all plaintexts. Note that $e = 41$ would also be a degenerate key. Note that for both of these degenerate keys, if we had calculated the corresponding decryption keys, d, we would see that e = d, but this will not always be the case for degenerate keys. If we had chosen p=13 and q=29 for our primes, then the modulus, m, would be 377, and N would be (13-1)*(29-1) = 336. Then the encryption key, e =

[7] A more appropriate name for these keys, from algebra, could be *idempotent* keys.

[8] Most textbooks do not mention this problem with RSA, but see the book *Handbook of Applied Cryptography* by Menezes. Also see *SIGCSE Inroads*, Vol. 41, No. 2, June 2009.

85 would be degenerate because lcm(12,28) | 84. However, the decryption key would be $d = 85^{-1} \pmod{336} \equiv 253$.

In view of what we know about degenerate keys, why is RSA the most commonly used public key cryptosystem? We have been working with very small numbers. In an 'industrial strength' cryptosystem the selected prime numbers would be much larger, typically hundreds of bits, and all calculations would be done with an unlimited precision arithmetic package (such as Python, or Java's BigInteger class). In this case the chance of selecting a degenerate key is very small.

6.3.2 Exercises

1. Alice plans to receive messages encrypted with the RSA algorithm; she chooses two prime numbers: p = 17 and q = 19.
   (a) What is her public modulus?
   (b) In order to generate her key-pair, she will need to find $N = \phi(m)$, where m is her public modulus. What is the value of N, for Alice?
   (c) Which one of the following values is a valid choice for e, her public encryption exponent?
   148, 150, 151, 152, 153, 162, 213, 231
   (d) Having chosen a valid encryption exponent, e, what two values make up her public key?
   (e) Show how Alice derives her secret decryption exponent, d, from her public encryption exponent, e.
   (f) What is Alice's RSA key-pair?
   (g) What values does Alice post on her web site for everyone to see?
2. Bob wishes to encrypt a message intended for Alice (using the results from the previous exercise). Show how he would encrypt the message, msg = 256.
3. Show how Alice would decrypt the ciphertext, c, received from Bob.
4. Write a program to factor the following integers, each of which is the product of two prime numbers:
   (a) 62615533
   (b) 992654469589
   (c) 4153748674359113993
5. Which of the following RSA keys are degenerate?
   (a) $m = 77, e = 13$
   (b) $m = 91, e = 24$

(c) $m = 91, e = 25$

(d) $m = 323, e = 5$

(e) $m = 323, e = 144$

(f) $m = 323, e = 145$

6. Build a spreadsheet which will determine whether a given RSA key is degenerate. Columns for p, q, and e allow for the user to enter integers. Calculated columns would include:
$m = p \cdot q$, $lcm(p-1, q-1)$, $(e-1) \bmod lcm = 0$.

7. Write a program which will determine whether a given RSA key is degenerate. Assume 32-bit integers for all calculations (i.e. Java int).

6.4 ElGamal

Another public key cryptosystem was described in 1985 by Taher ElGamal.[9] The ElGamal system relies on the difficulty of the discrete log problem (see section 4.8) for security.

The algorithm is described as follows, in which Alice and Bob wish to communicate without distributing keys. In this description Alice will be encrypting a message intended for Bob.

1. Alice (or Bob, or even a third party) chooses a large prime number, p, and a base, s, in the range [2..p-2]. These are known publicly.

2. Alice chooses a secret number, `a`, in the range [2..p-2]. She calculates $\alpha = s^a \ (mod\ p)$; Alice makes α known publicly.

3. Bob chooses a secret number, `b`, in the range [2..p-2]. He calculates $\beta = s^b \ (mod\ p)$; Bob makes β known publicly.

4. Alice divides the message into blocks, such that each block is an integer in [2..p-1].

5. Alice chooses an integer, k, in the range [2..p-2]. This is known as a *secret session* key.

6. Alice encrypts the plain text, msg: $t = s^k \ (mod\ p)$ and
$y = \beta^k \cdot msg \ (mod\ p)$

7. Alice sends t and y to Bob. These two values, together, make up the ciphertext.

Bob (and only Bob) can then decrypt the ciphertext as follows:

1. Bob calculates $(t^b)^{-1} \cdot y \ (mod\ p) = msg$ to obtain the plaintext msg.

[9] A native of Egypt, ElGamal received his Ph.D. at Stanford University, with Martin Hellman as his advisor.

To see why this works, recall that $t \equiv s^k \ (mod\ p)$. Thus in Bob's decryption, we have

$$\begin{aligned} & (t^b)^{-1} \cdot y \ (mod\ p) \\ \equiv \quad & ((s^k)^b)^{-1} \cdot y \ (mod\ p) \\ \equiv \quad & (s^{k \cdot b})^{-1} \cdot y \ (mod\ p) \end{aligned}$$

And since $y = \beta^k \cdot msg \ (mod\ p)$, and $\beta \equiv s^b \ (mod\ p)$ we make these substitutions for y and β, to obtain the plaintext, msg:

$$\begin{aligned} \equiv \quad & (s^{k \cdot b})^{-1} \cdot \beta^k \cdot msg \ (mod\ p) \\ \equiv \quad & (s^{k \cdot b})^{-1} \cdot (s^b)^k \cdot msg \ (mod\ p) \\ \equiv \quad & (s^{k \cdot b})^{-1} \cdot s^{b \cdot k} \cdot msg \ (mod\ p) \\ \equiv \quad & msg \end{aligned}$$

Wherein lies the security of the ElGamal algorithm? The attacker knows all of the public information:

- The prime number, p
- The base, s
- Bob's value for β

In order to decrypt the message the attacker needs only Bob's value for b. He can calculate b if he can solve the following for b:
$\beta \equiv s^b \ (mod\ p)$
However, if p is sufficiently large, his program to find b will have a long running time. This is the *discrete log problem*, discussed in chapter 4.[10]

6.4.1 Example for ElGamal

Here we illustrate the ElGamal public key encryption and decryption algorithms. In this example, Bob and Alice have agreed, publicly, to use the prime number p = 97, and the base, s = 45. To generate his secret key, Bob chooses b = 15, and then calculates

$$\begin{aligned} \beta \quad \equiv \quad & s^b \ (mod\ p) \\ \equiv \quad & 45^{15} \ (mod\ 97) \\ \equiv \quad & 28 \end{aligned}$$

[10] With a sufficiently large modulus, the running time to solve the discrete log problem could be many years.

He shares $\beta = 28$ publicly, but keeps b = 15 secret.

Alice wishes to encrypt the plaintext, msg = 79, and send the resulting ciphertext to Bob, using the ElGamal algorithm. She chooses k = 20, for her secret session key. She calculates

$$\begin{aligned} t &\equiv s^k \ (mod\ p) \\ &\equiv 45^{20} \ (mod\ 97) \\ &\equiv 50 \\ y &\equiv \beta^k \cdot msg \ (mod\ p) \\ &\equiv 28^{20} \cdot 79 \ (mod\ 97) \\ &\equiv 33 \cdot 79 \ (mod\ 97) \\ &\equiv 85 \ (mod\ 97) \end{aligned}$$

She sends both of these values, t = 50, and y = 85, to Bob. This is the ciphertext.

Bob can then decrypt the ciphertext by calculating:

$$\begin{aligned} (t^b)^{-1} \cdot y \ (mod\ p) & \\ &\equiv (50^{15})^{-1} \cdot 85 \ (mod\ 97) \\ &\equiv 33^{-1} \cdot 85 \ (mod\ 97) \\ &\equiv 50 \cdot 85 \ (mod\ 97) \\ msg &\equiv 79 \end{aligned}$$

6.4.2 Exercises

1. In a cryptosystem using the ElGamal algorithm, the number of bits in the plaintext, msg, depends on the public prime, p. For each of the following values for p, show the number of bits in the plaintext, and the number of bits in the ciphertext.
   (a) p = 509
   (b) p = 32,009
   (c) Show the number of bits in the plaintext, and the number of bits in the ciphertext, as a function of p

2. Bill wishes to send an encrypted message to Ann. They agree to use the prime number p = 107, and the base s = 31. Bill chooses a secret exponent, $b = 7$, calculates $\beta = s^b \ (mod\ p)$, and makes the value of β known publicly. Ann chooses a secret exponent $a = 12$, calculates $\alpha = s^a \ (mod\ p)$, and makes the value of α known publicly.

(a) Show how Bill can encrypt the plaintext, msg = 75, to be sent to Ann. He will use the session key, k = 82.

(b) Show how Ann can use the ciphertext received from Bill to recover the original plaintext, msg.

6.5 Discrete elliptic curves

[This section assumes the reader has seen section 4.10]

A relative newcomer to the group of algorithms used in public key cryptography is the use of discrete elliptic curves. Elliptic curve algorithms may seem daunting at first, and they may take longer to execute, but the added security they provide is significant. Many cryptocurrencies, such as Bitcoin, use discrete elliptic curves to ensure security.

As we showed in chapter 4, a discrete elliptic curve is defined by the equation: $y^2 = x^3 + ax + b \ (mod\ p)$
where a and b are chosen constants, and p is a prime number. We defined the meaning of 'addition' of points on a discrete elliptic curve.[11] We were then able to define the multiplication of a point by a scalar by using repeated addition of the same point. For example, if P is a point on the discrete elliptic curve, then $5 \cdot P = P + P + P + P + P$

6.5.1 Encryption and decryption

Representing plaintext - Koblitz' algorithm

Before we get to encryption and decryption algorithms using discrete elliptic curves, we need to decide how the plaintext (i.e. digital information) can be represented on an elliptic curve. At this point, the curve consists of several discrete points; we cannot simply use the x coordinates, nor the y coordinates, nor a linear combination of x-y coordinates. A given plaintext block will be a binary value which may not be an x or y coordinate for any point on the curve.

A good solution to this problem was proposed by Neil Koblitz (circa 1985). It is a probabilistic algorithm, which can fail, but which succeeds in an arbitrarily large number of cases. This algorithm relies on the fact that roughly half of the integers, mod p, have square roots. Thus means that for a given plaintext, msg,[12] Koblitz' algorithm will use a given value, K, which is the number of different attempts at finding a square root of the right side.[13] At each attempt, the probability that there will be a square root will be 1/2. Thus after K attempts, the probability that a square root is found will be $1 - 1/2^K$. Thus, if we choose a sufficiently large K, it is likely we will find a square root. Then the

[11] The operation is called 'addition' because, like the addition of natural numbers, it forms an algebraic group.

[12] Think of the plaintext msg as a binary value, less than p, representing any digital information to be encrypted.

[13] The plaintext block size, and the constant K must be sufficiently small such that $(Msg + 1) \cdot K < p$, where Msg is the largest plaintext to be encrypted.

point representing the plain text will consist of a computed (x,y) pair which is on the elliptic curve.

In these attempts to find a point we will try several consecutive integers, beginning with $K \cdot msg$, as an x coordinate:
$x = K \cdot msg$
$x = K \cdot msg + 1$
$x = K \cdot msg + 2$
$x = K \cdot msg + 3$
...

Here is Koblitz' algorithm for representing a given plaintext, msg, as a point on a discrete elliptic curve (with $p \equiv 3 \ (mod\ 4)$), with a given probability of success, $1 - 1/2^K$:

1. Set $i = 0$
2. Set $x = K \cdot msg \ (mod\ p)$
3. Repeat as long as $i < K - 1$ and a solution has not been found:[14]
   (a) Set $right = x^3 + a \cdot x + b \ (mod\ p)$
   (b) Set $y = \sqrt{right} = right^{(p+1)/4} \ (mod\ p)$
   (c) Is $y^2 \equiv right \ (mod\ p)$?
      i. If so we have found a point on the elliptic curve. Terminate the algorithm, with a result of the point (x, y).
      ii. If not, increment $x = x + 1$ and increment $i = i + 1$
4. The algorithm failed to find a point on the elliptic curve. Terminate.

As an example we will work with the plaintext, $msg = 0110_2 = 6$. We wish to find a point representing this plaintext on the discrete elliptic curve:
$y^2 = x^3 + x + 1 \ (mod\ 11)$
The points on this curve are shown in Figure 4.24.

Using Koblitz' algorithm we will choose a value for K that determines how likely we will succeed. We will choose $K = 10$ so the chance of success is $1 - 2^{-10} \approx 99.9\%$. Following the steps in the algorithm:

1. $i = 0$
2. $x = K \cdot msg \ (mod\ p) = 10 \cdot 6 = 60 \ (mod\ 11) \equiv 5$
3. (a) $right = x^3 + x + 1 \ (mod\ 11) = 5^3 + 5 + 1 \ (mod\ 11) \equiv 10$
   (b) $y = right^{(11+1)/4} \ (mod\ 11) = 10^3 \ (mod\ 11) \equiv 10$
   (c) $y^2 \equiv 1 \ (mod\ 11) \neq right$

      i.

[14] See chapter 4 for a quick way to calculate discrete square roots.

ii. First attempt fails: (5,10) is not on the curve.
$x = x + 1 = 6$

(a) $right = x^3 + x + 1 \ (mod\ 11) = 6^3 + 6 + 1 \ (mod\ 11) \equiv 3$
(b) $y = right^{(11+1)/4} \ (mod\ 11) = 3^3 \ (mod\ 11) \equiv 5$
(c) $y^2 \equiv 3 = right$

i. Success: The point is (6,5). Terminate
ii.

We found a point after two iterations. This is not surprising, as we have a 75% chance of success after two iterations.

As we will see below, after a message has been represented as a point, it can be encrypted, the result of which is another point on the same curve. Thus at the receiving end, when this point is decrypted, the result will be the original point created by the sender. But the receiver will need to find the plaintext, msg. This can be done because the receiver knows the sequence of x values, shown above, which were searched by Koblitz' algorithm. The decrypted plaintext can then be found from the x coordinate of the decrypted point:
$msg = x/K$
where the division is integer division, producing an integer quotient.[15]

Encryption/Decryption with ElGamal

We now discuss the encryption/decryption process using discrete elliptic curves. The algorithm is analogous to the ElGamal encryption algorithm for integers, but we are now using points on a discrete elliptic curve instead of integers. The ElGamal algorithm (either case) relies on the discrete log problem for security. In the case of integers, we computed a public 'power', $s^e \ (mod\ p)$ using a secret exponent, e. With elliptic curves, we will compute a public product, $e \cdot s$, where s is a point on the elliptic curve, and e is a secret integer. The encryption algorithm is shown below:

1. Bob and Alice agree on some public values:
   - A large prime number, p
   - coefficients a and b defing a discrete elliptic curve, $y^2 = x^3 + a \cdot x + b$
   - A random point, s, on the curve (the 'base')
   - Koblitz' $K < p$, used to represent a plaintext as a point on the curve (typically $K \approx 1000$)
2. They each choose a secret 'exponent' between $\sqrt{p}$ and $p - \sqrt{p}$:
   - Alice chooses e_a between $\sqrt{p}$ and $p - \sqrt{p}$

[15] Here we assume that K is much smaller than the prime modulus, p. This will normally be the case, but is not so in our example which uses a very small prime modulus, 11.

- Bob chooses e_b between $\sqrt{p}$ and $p - \sqrt{p}$

3. They each compute a public 'power':
   (a) Alice computes $A = e_a \cdot s$
   (b) Bob computes $B = e_b \cdot s$
4. The sender (i.e. the person encrypting the msg) uses Koblitz' algorithm, and K, to find a point, P, on the elliptic curve which represents the msg being encrypted.
5. The sender calculates a cipher point, C, using P, an exponent, and a 'power':
   (a) If Alice is encrypting a message to Bob
   $C = P + e_a \cdot B$
   (b) If Bob is encrypting a message to Alice
   $C = P + e_b \cdot A$

   The ciphertext is the pair of points, the 'power' and C

Decryption

To decrypt the ciphertext, the recipient uses the 'power' and the point C to obtain the point representing the plaintext. Then the plaintext is obtained from that point:

1. (a) If Alice is the recipient, she calculates the plaintext point, P:
   $P = C - e_A \cdot B$
   (b) If Bob is the recipient, he calculates the plaintext point, P:
   $P = C - e_B \cdot A$
2. The plaintext is obtained from the x coordinate of the plaintext point and the Koblitz integer, K:
   $msg = P_x/K$
   The division produces an integer quotient.

Figure 6.5 shows a comparison of the ElGamal encryption/decryption algorithm using integers, which we described earlier, with this encryption/decryption algorithm using points on a discrete elliptic curve.

We use an example to illustrate this encryption/decryption algorithm. The calculations can be done using the software developed for the exercises in chapter 4. Alice wishes to encrypt the plaintext $msg = 6 = 0110_2$ for Bob. Alice and Bob prepare to communicate:

1. They agree on the public elliptic curve to be used:
   $y^2 = x^3 + x + 1$ $(mod\ 97)$
   $a = 1$, $b = 1$, $p = 97$

	Integers	Elliptic Curves
public setup	p = large prime base = s	p = large prime base = random point, s Elliptic curve coefficients, a, b Koblitz' K
secret exponent	Alice = a Bob = b	$\sqrt{p} < e_a < p - \sqrt{p}$ $\sqrt{p} < e_b < p - \sqrt{p}$
public power	Alice, $A = s^a \ (mod\ p)$ Bob, $B = s^b \ (mod\ p)$	Alice, $A = e_A \cdot s$ Bob, $B = e_B \cdot s$
Encryption Bob sends msg to Alice	ciphertext = (A,C) $C = A^b \cdot msg \ (mod\ p)$	msg is represented by point P ciphertext = (A,C) $C = P + e_b \cdot A$
Alice decrypts	$msg = B^{p-1-a} \cdot C \ (mod\ p)$	$P = C - e_a \cdot B$ $msg = P_x/K$

Figure 6.5: A comparison of ElGamal encryption with integers, and ElGamal encryption with discrete elliptic curves. Bob is encrypting a plaintext message, msg, for Alice.

2. They agree on a public point on this elliptic curve to be used as the base: $s = (3, 82)$

3. They agree on a public integer for Koblitz' algorithm: $K = 10$

4. Alice chooses the secret 'exponent' $e_a = 5$
   Bob chooses the secret 'exponent' $e_b = 7$

5. Alice calculates her public 'power', A, and Bob calculates his public 'power', B:

$$\begin{aligned} A &= e_a \cdot s \\ &= 5 \cdot (3, 82) \\ &= (68, 67) \\ B &= e_b \cdot s \\ &= 7 \cdot (3, 82) \\ &= (82, 54) \end{aligned}$$

6. Alice uses Koblitz' algorithm, $K = 10$, to represent the plaintext, $msg = 6$ as a point on the elliptic curve:
   $P = (62, 16)$

7. Alice calculates the ciphertext point, C :

$$C = P + e_a \cdot B$$

$$\begin{aligned} &= (62,16) + 5 \cdot (82,54) \\ &= (62,16) + (19,63) \\ &= (77,41) \end{aligned}$$

She sends $C = (77, 41)$ to Bob.

8. Bob decrypts the cyphertext point, C, to obtain the plaintext point, P:

$$\begin{aligned} P &= C - e_b \cdot A \\ &= (77,41) - 7 \cdot (68,67) \\ &= (77,41) + 7 \cdot (-(68,67)) \\ &= (77,41) + 7 \cdot (68,30) \\ &= (77,41) + (19,34) \\ &= (62,16) \end{aligned}$$

(See chapter 4 for a discussion of subtraction of points)

9. Bob uses Koblitz' formula to get the original plaintext msg from the x coordinate of the plaintext point, $P_x = 62$:
$msg = P_x/K = 62/10 = 6$

We conclude this section with a justification for this encryption/decryption algorithm. Begin with the decryption formula:
$C - e_b \cdot A$
Then derive the plaintext point, P, by substituting the encryption formula for $C = P + e_a \cdot B$:

$$\begin{aligned} C - e_b \cdot A &= (P + e_a \cdot B) - e_b \cdot A \\ &= P + (e_a \cdot B - e_b \cdot A) \\ &= P + (e_a \cdot e_b \cdot s - e_b \cdot e_a \cdot s) \\ &= P \end{aligned}$$

6.5.2 Exercises

1. Joe and Mary wish to communicate using the ElGamal algorithm with this discrete elliptic curve:
$y^2 = x^3 + -4x + 4 \ (mod\ 101)$
Joe wishes to encrypt the plaintext, $msg = 17$, to be sent to Mary.

   (a) Show how the plaintext $msg = 17$ can be represented by a point on the curve. Use a Koblitz constant, $K = 5$.

(b) Joe is using a secret 'exponent', $e_j = 32$
Mary is using a secret 'exponent', $e_m = 43$
They have agreed to use the point (29,21) as their public base, s. Show how Joe can encrypt the plaintext, $msg = 17$, which is to be sent to Mary.

(c) Show how Mary can decrypt to obtain the original plaintext, 17.

2. Develop software needed for encryption/decryption using the ElGamal algorithm with a discrete elliptic curve. If using an object-oriented programming language, such as Java, C++, or Python, consider the following class design:

- class Group has:
  - Integer coefficients, a and b, defining the elliptic curve
  - Prime modulus
  - Methods:
    - Represent an integer on the curve (Koblitz' algorithm)
    - Multiply a Point by an integer,
- class Point has:
  - x and y coordinates
  - A method to add this Point to another Point
- class ElGamal has:
  - A Group
  - A public base, s.
  - Private exponents for sender and receiver.
  - Methods to encrypt and decrypt a given plaintext.

6.6 Cryptography with an Obfuscating Compiler

6.6.1 Program obfuscation

Program *obfuscation* is the process of converting a source level program to an unintelligible source-level form. An *obfuscating compiler* is a compiler which transforms a program into unintelligible, but equivalent, form. It is importanbt that the original program and the obfuscated program be *equivalent*. This means that when executed they would exhibit the same input/output relation; they do exactly the same thing. However, the obfuscated version is virtually unreadable to people.

6.6.2 Some uses of an obfuscating compiler

An obfuscating compiler can be used for:

- **Protection of proprietary source code -** Proprietary software is code which is owned by the developers, and the source is generally not released to customers. Only the executable machine language version is released. Thus customers can use the sofware but they cannot make changes or edits to the software, which would require access to the source code.[16] If Microsoft customers had access to the Office source code, they could make enhancements, and compete with Microsoft in the office software market.[17]

  A decompiler is software which translates a machine language program to a high level programming language (i.e. source level). In recent years decompilers have been improving significantly and can now produce meaningful source level code, which can then be used to fork the project. To counter this development, owners (such as Microsoft) of proprietary projects use an obfuscating compiler to produce an unintelligible version of the software. They distribute either the source level or machine language version of the obfuscated software. The customer who attempts to make changes to the software, by decompiling, will be faced with unintelligible code.

  See the open source textbook, *Compiler Design: Theory, Tools, and Examples*, at `http://cs.rowan.edu/~bergmann` for more on obfuscating compilers.

- **Public key cryptography -** Recent developments have revealed ways that obfuscating compilers can be used in public key cryptosystems. Here we focus on one such usage - encryption to ensure confidentiality, and assume that program obfuscation is a *one-way* function. Alice can encrypt a secret message for Bob by obfuscating a program that puts out the message, but only if provided with Bob's secret key.[18]

6.6.3 An example of encryption

We elaborate with an example. Alice wishes to send a secret message to Bob, and she knows Bob's public key, K_{Bp}.

1. Alice writes the following program in the source file Encr.java:[19]

```
import java.security.*;
String encr(Key k, KeyPair pair)
{   if (pair.getPublic().equals(KBp)
```

[16] Source level code for proprietary software, such as Microsoft Office, is carefully, and legally, guarded by Microsoft. Employees who divulge source, even after leaving the company, would be subject to criminal and/or civil law suits.

[17] This is known as a project *fork*, yielding two different versions of a project.

[18] See *Communications of the ACM*, March 2024, for a more detailed description of this process, as well as theoretic results.

[19] Here we use Java, but any high level language works, including interpreted languages such as Python.

```
        && k.equals(pair.getPrivate())
        return "Attack at dawn";        // secret message
    return null;
```

2. Alice uses this as input to her obfuscating compiler, producing an obfuscated version of this program, calling it `Scramble.java`.

3. Alice sends `Scramble.java` to Bob.

4. Bob has his own key pair (mathematically related public and secret keys). He writes a driver for the method which he received from Alice, in which he calls the `encr(Key,KeyPair)` method, providing his secret key and his KeyPair as parameters. The method returns the String `"Attack at dawn"`.

5. Suppose Eve intercepts the message from Alice. When Eve executes the program, the call to `encr(Key,KeyPair)` returns `null` because Eve does not know Bob's secret key. If Eve is able to read and comprehend the obfuscated program, she will see the secret message, but if Alice's obfuscating compiler is a good one, Eve is foiled.

Fig 6.6 depicts the first part of this process in which Alice obfuscates the program into the file `Scramble.java`, and sends it to Bob.

Fig 6.7 depicts the second part of this process in which Bob is able to compile and execute the obfuscated program, providing his secret key and his `KeyPair`, to get the secret message, `"Attack at dawn"`. Eve tries to do the same thing but gets a null output because she does not know Bob's secret key (and therefore his `KeyPair`).

6.6.4 Exercises

1. Suppose Eve intercepts Alice's transmission to Bob. Eve decides to construct her own KeyPair, using Bob's public key. She will find a private key corresponding to that public key to form a KeyPair, KP_E. Then she calls the `encr(Key,KeyPair)` method:
   $encr(KP_E.getPrivate(), KP_E)$
   Explain why this tactic cannot work.

2. Suppose Eve intercepts Alice's transmission to Bob. She wants to remove the `import` statement from the program. Then she could define her own KeyPair and Key classes, which give her the capability of forming a KeyPair from a given public Key. She builds a KeyPair using Bob's public key and the integer 17 as the corresponding private key:
   `KeyPair kp = new KeyPair(17,`K_{BP}`);`
   Then she could call:
   `encr(`KP_E`.getPrivate(), `KP_E`)`
   which would show her the secret message. Explain why this does not work.

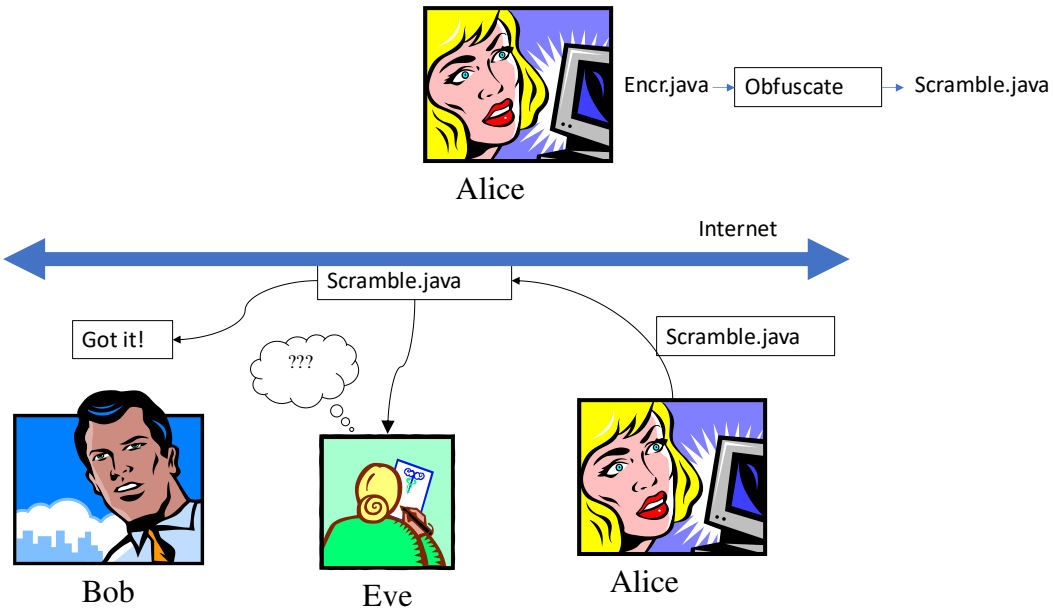


Figure 6.6: Alice obfuscates a program which puts out a secret message, but only when given Bob's private key and sends it to Bob

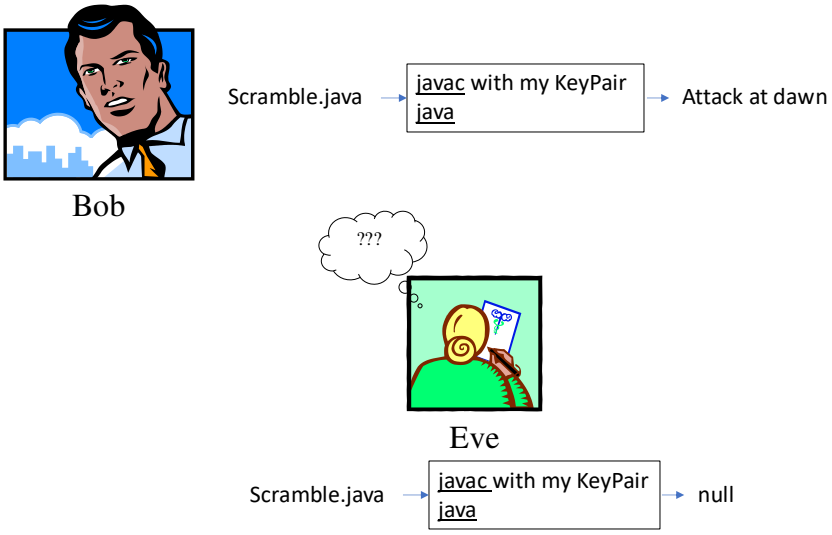


Figure 6.7: Bob compiles the obfuscated program, `Scramble.java`, and executes the program, using his KeyPair as input, and sees the secret message. Eve does the same thing and gets a null reference as output, because she does not know Bob's secret key (and therefore his KeyPair).

3. Define a Java method for encryption using obfuscation, similar to the one Alice defined in this section, but use only one parameter, a `KeyPair`.

Chapter 7

Key Distribution

As noted in chapter 3 one of the most difficult obstacles to secure communication is the problem of *key distribution*. If Bob wishes to use a private key algorithm, such as DES, to encrypt a message to Alice, he will use a private key to encrypt the message, producing a ciphertext which he sends to Alice over an insecure channel (e.g. the internet). In order to decrypt the ciphertext Alice will need Bob's private key. In general, when using private key algorithms, we need to share keys with our trusted friends, but not with our enemies (or others).

How can our private keys be shared? This is the essence of the key distribution problem. The internet is not a safe channel; traffic can be intercepted by others. We could use a courier or a package delivery service (FedEx, USPS, etc.) but these are generally too slow and expensive.

Of course we could use a public key algorithm to encrypt our messages, but in order to be secure a long public key is needed, which causes encryption and decryption to take too long (at least several seconds). We seek other means to share private keys. The algorithms we seek for key distribution:

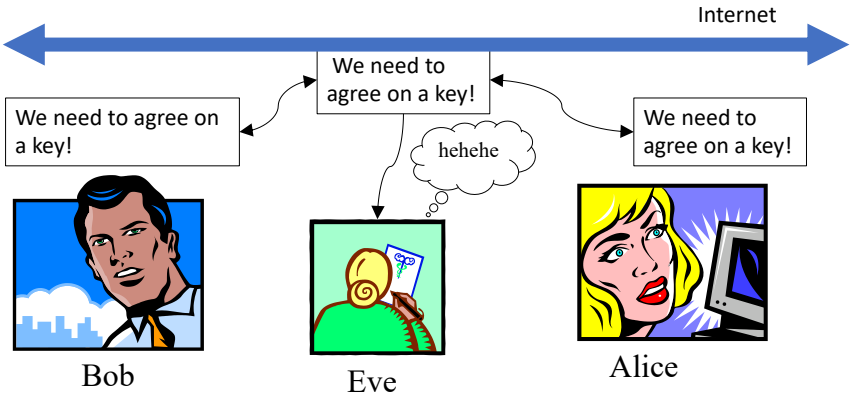
- will be public - as with encryption algorithms, our key distribution algorithms are known to everyone. Security lies in the secret information that is not shared.
- will generally follow this pattern - two parties wishing to share keys will share partial information publicly which, when combined with their own secret information, can be used by both parties to calculate a common key.

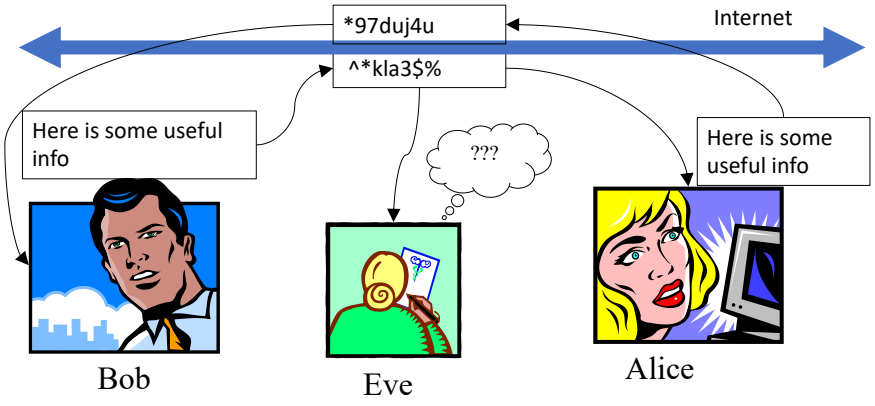
See Figure 7.1 for a simple description of key distribution.

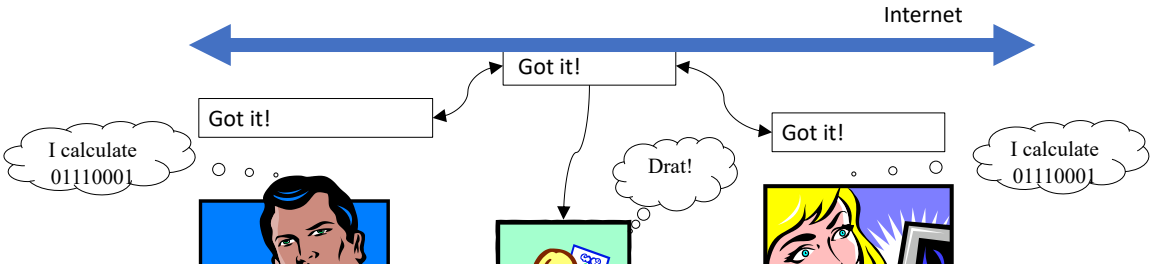
7.1 Encryption of keys

[This section assumes the reader has seen chapters 3 and 6]

The most commonly used solution to the key distribution problem involves a public key algorithm, such as RSA. RSA can be used to encrypt a private DES


Internet
We need to agree on a key!
We need to agree on a key!
We need to agree on a key!
hehehe
Bob
Eve
Alice


*97duj4u
Internet
^*kla3$%
Here is some useful info
???
Here is some useful info
Bob
Eve
Alice


Internet
Got it!
Got it!
Got it!
I calculate 01110001
Drat!
I calculate 01110001

key. This key is called a *session key*. Once the session key has been encrypted, it can be sent out; the receiver decrypts the session key, and it is used with a private key algorithm for two-way communication.

Here is an example showing how Bob and Alice can communicate, back and forth, using the private key algorithm, DES:

1. Bob and Alice create their own RSA key pairs:
   - Bob's public RSA (encryption) key is K_{Bpub}
   - Bob's secret RSA (decryption) key is K_{Bsec}
   - Alice's public RSA (encryption) key is K_{Apub}
   - Alice's secret RSA (decryption) key is K_{Asec}
2. Bob (or Alice) chooses a secret DES key, K_{DES}. This will be their session key.
3. He encrypts the session key, using Alice's public RSA key, to generate an encrypted session key, K'_{DES}:
   $K'_{DES} = Encr(K_{Apub}, K_{DES})$
4. He sends the encrypted session key, K'_{DES} to Alice.
5. Alice decrypts the session key using her secret RSA key.
   $K_{DES} = Decr(K_{Asec}, K'_{DES})$
6. Both Bob and Alice now have the session key, K_{DES}, and it was never sent out over the internet. They use this key to communicate securely with DES.
7. If Bob and Alice wish to communicate again, they would generate, and share, a new session key.

One advantage of this scheme is that we use a different key for each so-called session. Our adversaries will be attempting to break our codes using statistical techniques (covered in chapter 3). As they attempt to find our key, they will need lots of data (i.e. ciphertexts). However, if we keep our sessions fairly short, and use a different key for each session, statistical cryptanalysis is not likely to succeed.

7.1.1 Exercises

1. Marge and Jim wish to communicate with confidentiality using DES. They each have software to encrypt plaintext and decrypt ciphertext using DES. Marge has created a suitable key for use with DES: K_{DES}. The functions for encryption/decryption with DES are:
   - $EncrDES(msg, key)$ to encrypt a plaintext message using DES with a key

- $DecrDES(cipher, key)$ to decrypt a ciphertext using DES with a key

They also have RSA software for public key encryption/decryption:

- $EncrRSA(msg, m, e)$ to encrypt a plaintext using RSA with a modulus, m, and encryption exponent, e.
- $DecrRSA(cipher, m, d)$ to decrypt a ciphertext using RSA with a modulus, m, and decryption exponent, d.

They also have the following RSA keys:

- Marge's public RSA modulus is m_M
- Marge's public RSA exponent is e_M
- Marge's private RSA exponent is d_M
- Jim's public RSA modulus is m_J
- Jim's public RSA exponent is e_J
- Jim's private RSA exponent is d_J

(a) Show how Marge can encrypt the plaintext message, *How are you?*, with DES so that she can send it to Jim securely on the internet.

(b) Show how Jim can decrypt this message.
Hint: Marge will need to send the DES key to Jim.

7.2 Diffie-Hellman key exchange

In 1976 Whitfield Diffie and Martin Hellman, at Stanford University, developed (with contributions from Ralph Merkle, a grad student at UC Berkeley) what is considered to be a seminal work in public key cryptography. The algorithm, known today as Diffie-Hellman Key Exchange,[1] or Key Agreement, allows two people to agree on a secret key while communicating on an insecure channel. The algorithm relies on the fact that the Discrete Log Problem (covered in chapter 4) is hard.

Suppose Bob and Alice wish to agree on a key to be used with a private (i.e. symmetric) encryption/decryption algorithm such as DES, but there is no secure and efficient communication channel. Here is how it works:

1. Alice and Bob agree on a large prime number, p, publicly. This will be used as the modulus.

2. Alice and Bob agree on a public base, s, in the range $1 < s < p - 1$. For maximum security it is desirable that s is a generator for the prime p, as described in chapter 4.

3. Alice chooses a secret exponent, a, in the range $1 < s < p - 1$. She calculates the public power $\alpha = s^a \pmod{p}$ and sends α to Bob.

[1] Diffie-Hellman Key Exchange predates RSA by a few years

4. Bob chooses a secret exponent, b, in the range $1 < s < p-1$. He calculates the public power $\beta = s^b \pmod{p}$ and sends β to Alice.

5. Alice calculates the shared key:
   $K = \beta^a \pmod{p}$

6. Bob calculates the shared key
   $K = \alpha^b \pmod{p}$

They use the shared key, K, for secure communication with a private key algorithm such as DES.

To see why this works, we start with Alice's calculation of the key, and substitute Bob's calculation of β:

$$\begin{aligned} K &= \beta^a \\ &= (s^b)^a \\ &= s^{b \cdot a} \pmod{p} \end{aligned}$$

Then we start with Bob's calculation of the key, and substitute Alice's calculation of α:

$$\begin{aligned} K &= \alpha^b \\ &= (s^a)^b \\ &= s^{a \cdot b} \pmod{()p} \end{aligned}$$

Since $s^{b \cdot a} = s^{a \cdot b}$, they have calculated the same key without sending it through an insecure channel.

How can their adversary, Eve, find their secret shared key? She knows α, β, s, and p. If she can solve for a in $\alpha = s^a$ or if she can solve for b in $\beta = s^b$ she can then find the secret key the same way that Alice and Bob calculated it:
$K = \alpha^b \pmod{p}$
or
$K = \beta^a \pmod{p}$
but this would require her to solve the discrete log problem, which we have seen to be hard in section 4.8. If the numbers are sufficiently large it will take years for Eve's computer to find a or b.

7.2.1 Example for Diffie-Hellman key exchange

Here we show how Alice and Bob can share a secret key without sending it out over the internet. We use the Diffie-Hellman key exchange algorithm. We use small numbers to simplify the calculations; in a secure system the numbers would be much larger.

1. Alice chooses the public prime modulus, $p = 79$. She chooses the public base, $s = 3$, which is a generator for 79.

2. Alice sends $p = 79$ and $s = 3$ to Bob.

3. Alice chooses a secret exponent $a = 17$.
   Bob chooses a secret exponent $b = 5$.

4. Alice calculates $\alpha = s^a \ (mod\ p) = 3^{17} \equiv 48 \pmod{79}$.
   Bob calculates $\beta = s^b \ (mod\ p) = 3^5 \equiv 6 \pmod{79}$.

5. Alice sends $\alpha = 48$ to Bob.
   Bob sends $\beta = 6$ to Alice.

6. Alice calculates the shared key $K = \beta^a = 6^{17} \equiv 54 \pmod{79}$
   Bob calculates the shared key $K = \alpha^b = 48^5 \equiv 54 \pmod{79}$

7. Alice and Bob have agreed on a secret key, $K = 54$, which can be used to encrypt and decrypt confidential messages.

7.2.2 Exercises

1. What positive integers less than 17 are generators for 17?

2. Marge and John wish to use the Diffie-Hellman key exchange algorithm to agree on a key.

   - Marge chooses the public prime modulus, $p = 53$, and the public base, $s = 10$. She sends both of these to John.
   - Marge chooses the secret exponent $m = 3$, and John chooses the secret exponent $j = 9$

   (a) Show the public information which Marge then sends to John.
   (b) Show the public information which John then sends to Marge.
   (c) Show how Marge calculates the shared key.
   (d) Show how John calculates the shared key.
   (e) Show the shared key.

3. Write a program to implement the Diffie-Hellman key exchange algorithm using integers. Assume the prime modulus is fewer than 8 decimal digits. If using an object-oriented language, you could use a class, `Person`. A `Person` has a prime modulus, a base, an exponent, and a power.
   $power = base^{exponent} \pmod{p}$.

   A `Person` Can calculate a shared key, with an other Person:

```
/** Calculate the key shared with another Person */
   void setKey(Person other)
```

7.3 Diffie-Hellman key exchange with discrete elliptic curves

The Diffie-Hellman key exchange algorithm can also be implemented with discrete elliptic curves (see section 4.10). The parameters for the elliptic curve will be public, as will a point to be used as the base. Each participant will choose a secret integer, and compute a public point on the curve by multiplying the base by his/her secret integer. Each participant will independently calculate the key point by multiplying his/her secret integer by the other participant's public point. The shared key can be extracted from the key point by taking its x coordinate. The algorithm is detailed below:

1. Alice and Bob agree on a large prime, p, and coefficients for a non-singular discrete elliptic curve, a and b:
   $y^2 = x^3 + a \cdot x + b$
   such that $4 \cdot a^3 + 27 \cdot b^2 \not\equiv 0 \pmod{p}$

2. Alice and Bob choose a public base point on the elliptic curve, G.

3. Alice chooses a secret integer, n_A, such that $\sqrt{p} < n_A < p - \sqrt{p}$

4. Bob chooses a secret integer, n_B, such that $\sqrt{p} < n_B < p - \sqrt{p}$

5. Alice calculates a public point, $A = n_A \cdot G$ and sends it to Bob.

6. Bob calculates a public point, $B = n_B \cdot G$ and sends it to Alice.

7. Alice calculates the key point, $K = n_A \cdot B$, and chooses the x coordinate of K as the shared key.

8. Bob calculates the key point, $K = n_B \cdot A$, and chooses the x coordinate of K as the shared key.

We note how this version of the Diffie-Hellman key exchange algorithm differs from the version using integers from the previous section. The public *power* in the previous section was the result of the *base* raised to an *exponent*:
$power = base^{exponent} \pmod{p}$

Here we use multiplication of a point by an integer, n, instead of exponentiation, and the base is a point, G on the discrete elliptic curve:
$point = n \cdot G$

The security of this algorithm lies in the fact that the discrete 'log' problem for discrete elliptic curves is hard:
Given points G and pt, and the prime modulus, p, it is hard to find an integer, n, such that:
$pt = n \cdot G \pmod{p}$

We can see how this version of the Diffie-Hellman key exchange algorithm enables two participants to arrive at the same key. We begin with Alice's computation of the key point, K, and substitute Bob's public point for B:

$$\begin{aligned} K &= n_A \cdot B \\ &= n_A \cdot (n_B \cdot G) \\ &= (n_A \cdot n_B) \cdot G \end{aligned}$$

Next we begin with Bob's computation of the key point, K, and substitute Alice's public point for A:

$$\begin{aligned} K &= n_B \cdot A \\ &= n_B \cdot (n_A \cdot G) \\ &= (n_B \cdot n_A) \cdot G \end{aligned}$$

Since $n_A \cdot n_B = n_B \cdot n_A$, Alice and Bob have calculated the same point. They use the x coordinate of that point as their secret encryption key.

7.3.1 Example using a discrete elliptic curve

For an example of Diffie-Hellman key exchange using discrete elliptic curves, we use the same curve described in chapter 4:
$y^2 = x^3 + x + 1 \pmod{11}$

1. Alice and Bob agree on a public base point, $G = (8, 2)$
2. Alice chooses the secret integer, $n_A = 5$
3. Bob chooses the secret integer, $n_B = 6$
4. Alice calculates the public point $A = n_A \cdot G = 5 \cdot (8, 2) = (1, 5)$ and sends it to Bob
5. Bob calculates the public point $B = n_B \cdot G = 6 \cdot (8, 2) = (6, 5)$ and sends it to Alice
6. Alice calculates the key point $K = n_A \cdot B = 5 \cdot (6, 5) = (0, 10)$. She uses the x coordintate, 0, as the shared key.
7. Bob calculates the key point $K = n_B \cdot A = 6 \cdot (1, 5) = (0, 10)$. He uses the x coordintate, 0, as the shared key.

In this example we get a shared key of 0; with larger numbers we would likely get a non-zero shared key.

7.3.2 Exercises

1. Using the discrete elliptic curve $y^2 = x^3 - 4x + 4 \pmod{23}$, show how Ann and Bill can share an integer key without sending it to each other.

   - Ann proposes that they use the shared base $G = (6, 14)$.
   - Ann uses the secret integer $n_A = 7$.
   - Bill uses the secret integer $n_B = 12$.

Chapter 8

Authenticity - Digital Signatures and Certificates

[This chapter assumes the reader has seen sections 4.1, 4.2, and 4.6]

We have seen how to ensure confidentiality, with secure encryption of plaintexts (chapter 3), and we have seen how to ensure integrity of our communications (chapter 5). However there is one very important aspect of security which is still missing: *authenticity*.

When a message is received, whether plaintext or ciphertext, how do we know that the person shown in the `from:` field is really who they claim to be. If you receive a message from `JoeSmith@gmail.com`, how can you be sure that:

- It is from your friend Joe Smith?
- It is actually sent from the email address `JoeSmith@gmail.com`?[1]

This raises the issue of authenticity, which is commonly seen in situations which may or may not involve the internet:

1. When you login to your bank's web site.
2. When you attempt to make a credit/debit card transaction.
3. When you wish to board an airplane.
4. When you wish to travel from one country to another.
5. When you wish to purchase restricted items such as cigarettes, CBD, or alcoholic beverages.

In each case some form of *identity verification*, or ID, is needed. This is more generally known as authentication.

[1] It is possible to put false information in any of the email fields, though we will not explain how this can be done.

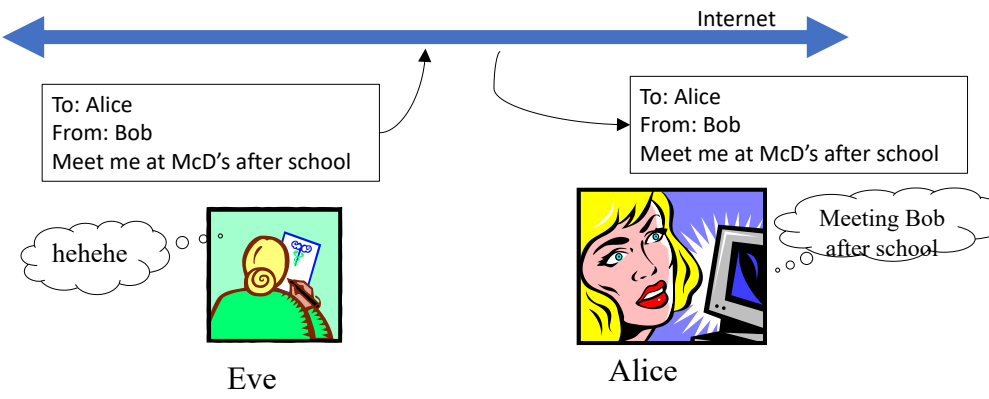


Figure 8.1: Eve enters false information in the `From:` field of a message to Alice. Authenticity has been violated.

In cryptologic applications authenticity may encompass integrity as well. In addition to verifying the authenticity of the sender, the software can also verify that the message (plaintext or ciphertext) has not been altered in transit.

Figure 8.1 shows how authenticity can be violated.

8.1 Vulnerability of public key algorithms

In chapter 6 we exposed the concept of public key encryption. By using a key pair (public and secret), one can encrypt plaintext using a person's public key; then only that person can decrypt it with their own secret key.

However, this fails if authenticity is not ensured. Imagine that Eve claims to be Alice and announces her public key. Then others who send encrypted confidential information to her, thinking it is going to Alice, will violate confidentiality. Eve will be able to decrypt the ciphertext.

In chapter 7 we exposed the concept of key distribution. We looked at a few ways of sharing keys without sending them over an insecure channel (the internet). Again, this fails if authenticity is not ensured. We saw a few ways of distributing keys over an insecure channel:

- We can share a (private) session key using a public key algorithm to encrypt a session key. But if authenticity is violated we could be sending an encrypted session key to our adversary. Imagine that Eve sends an encrypted session key to Bob, claiming that she is Alice. Bob will use the session key to send confidential information back to Eve, thinking she is Alice.

- The Diffie-Hellman key exchange algorithm was used: parties wishing to agree on a key sent partial information to each other, enabling each of them to calculate the same key. This fails if one of the parties is not who they claim to be. Eve, claiming to be Alice, sends partial information to Bob, and Bob replies to Eve, thinking she is Alice, with his partial information. Bob has unwittingly agreed on a key with Eve, thinking she was Alice.

8.1.1 Exercises

1. Explain why public key encryption algorithms are insecure.

2. Explain how public key distribution algorithms can be vulnerable to attack.

8.2 Digital signatures

Before the advent of computers identity was often ensured with handwritten signatures. Though forensic experts are capable of detecting fraudulent signatures, handwritten signatures were often fraudulent. Handwritten signatures are legally binding, but are not considered a sure method of authentication. A digital signature is NOT a traditional hand-written signature in digital form, as shown in Figure 8.2.

Cryptographic algorithms can ensure authenticity with *digital signatures*, which generally consist of binary data associated with a particular entity[2] and plaintext. Others will be able to use this binary data, or signature, to ensure the identity of the entity. A digital signature is the digital equivalent of a government-issued ID card.

The term *signature* used here can be misleading. Digital signatures are similar to conventional (handwritten) signatures in that they can be used to establish authenticity. However, digital signatures differ from conventional signatures as follows:

- A conventional signature is associated with an individual person. Bob's signature on a check looks just like his signature on his tax return (essentially).

[2]We use the word *entity* to apply to a person, an organization, a company, or any institution.

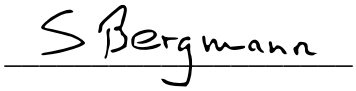


Figure 8.2: This is NOT a cryptographic digital signature. This is a handwritten signature which has been digitized for inclusion in this book.

- A digital signature is associated with a document, or plaintext, as well as the person signing. As we show below, a digital signature is applied to a particular plaintext (or ciphertext) and will be completely different if applied to a different plaintext (or ciphertext), even if both signatures were produced by the same person.

8.2.1 Signatures derived from public key cryptography

In chapter 6 we studied the use of key pairs to encrypt and decrypt to ensure confidentiality. Here we use the same key pairs to produce digital signatures.

Each entity has their own key pair (public and secret).
K_{public} is the public encryption key of the recipient.
K_{secret} is the secret decryption key of the recipient.
For confidentiality the sender uses the recipient's public key to encrypt a plaintext, msg:
$ciphertext = Encr(msg, K_{public})$
Then the recipient uses their secret key to decrypt ciphertext:
$msg = Decr(ciphertext, K_{secret})$

Verification of authenticity is depicted in Figure 8.3. Bob generates a digital signature by decrypting the plaintext. Alice verifies that it must be from Bob by encrypting the signature. Surprisingly, this is the *opposite* of what we have been doing for confidentiality. Instead of encrypting plaintext, we are decrypting

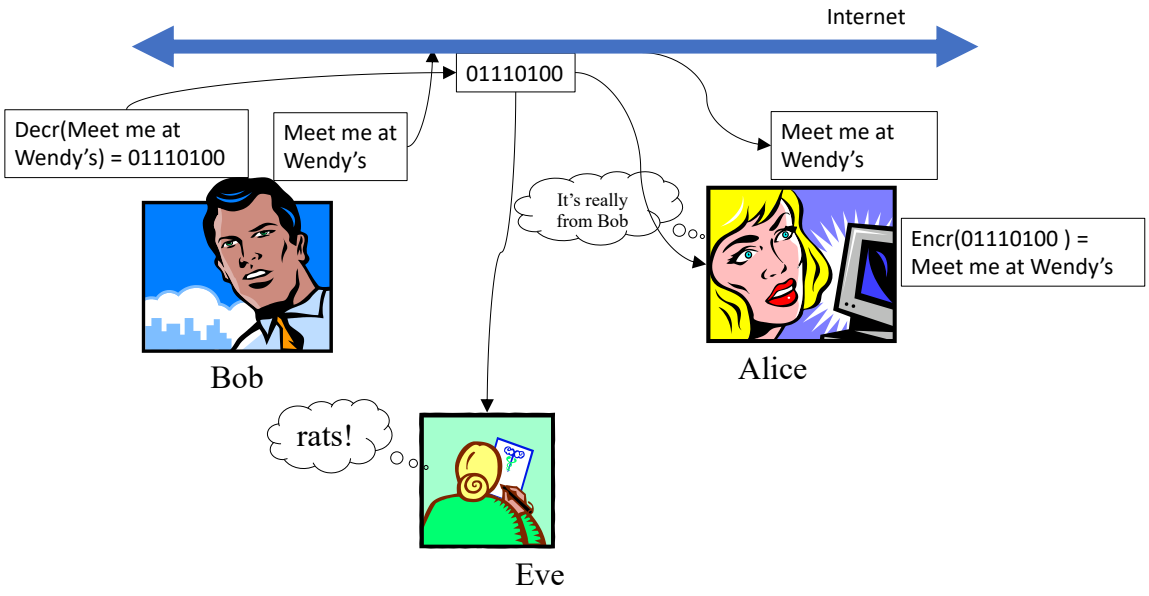


Figure 8.3: Bob sends a message to Alice, along with the signature, 01110100. Alice can verify that the message was truly from Bob, when she encrypts the signature.

plaintext. Instead of decrypting ciphertext, we are encrypting ciphertext. This discovery in the 1980's was a huge leap forward in verifying authenticity.

Signatures for non-confidential messages

Here we assume the message to be signed is not confidential but authenticity needs to be verified. The sender does not care who sees the message, but would like the recipient to be certain of the sender's identitiy, and of the integrity of the message. To create a digital signature the sender will use his/her own secret key in applying the decryption algorithm to the plaintext, msg, to produce a digital signature, sig:
$sig = Decr(msg, K_{secret})$
This signature is then sent, along with the message, in either encrypted or unencrypted form, to a recipient.

To authenticate, also known as verification, the recipient will use the sender's public key in applying the encryption algorithm to the signature, which should produce the original plaintext.
$msg' = Encr(sig, K_{public})$

If $msg \neq msg'$ the recipient does not trust the content of the message. One of the following has occurred:

- The message was not sent by the purported sender
- The message was altered in transit

This works because the encryption and decryption algorithms are *inverse functions*: $Encr(Decr(msg)) = msg$ and $Decr(Encr(msg)) = msg$

Since the signature was created with the sender's secret key, the sender is the only person who could have created that signature. Anyone can verify the authenticity because the sender's encryption key is public.

Signatures for confidential messages

Now we assume that the message needs to be confidential as well as authenticated. Here Bob and Alice each have their own key-pairs:

	public encryption key	secret decryption key
Alice	e_A	d_A
Bob	e_B	d_B

Bob wants to send a confidential message, msg, to Alice, and he wants her to be able to authenticate the message; i.e. he wants her to be assured that it has not been altered and that it is truly from him.

1. He first creates a 'pre-signature' by applying his decryption algorithm to the plaintext message:
   $pre = Decr(msg, d_B)$

2. He does not send this to Alice; if Eve intercepts this pre-signature, she can easily obtain the original message by applying Bob's public key to the pre-signature. Instead he encrypts the pre-signature with Alice's public key to produce the actual signature:
   $sig = Encr(pre, e_A)$

3. He then encrypts the plaintext message with Alice's public key to produce a ciphertext:
   $cipher = Encr(msg, e_A)$

4. He sends sig and $cipher$ to Alice.

5. Alice applies her own decryption algorithm to the signature, producing the pre-signature:
   $pre = Decr(sig, d_A) \pmod{m_A}$
   she then applies Bob's encryption algorithm to the pre-signature to obtain what is presumably the plaintext message:
   $msg_1 = Encr(pre, e_B)$

6. Alice then decrypts the ciphertext with her own decryption algorithm, again producing what is presumably the plaintext message:
   $msg_2 = Decr(cipher, d_A)$

7. Alice verifies: if $msg_1 = msg_2$, then Alice knows that this message is the actual message that was sent (it was not altered), and that it was in fact sent by Bob. if $msg_1 \neq msg_2$, she does not trust either message; someone has tampered with the message or it was not sent by Bob.

Digital signatures, as described here, may seem ironic, or it may seem to be an incorrect reversal of roles. To create a digital signature the sender is *decrypting* the plaintext. To authenticate, the recipient is *encrypting* the signature.

Digital signatures are a very significant by-product of public key cryptography.

8.2.2 Examples of digital signatures using RSA

Here we include two examples showing how a digital signature can be created using RSA, and used for authenticity. In the first example the plaintext is not confidential, and in the second example the plaintext is confidential. For both of these examples we use the key-pairs shown below:

	public encryption key	secret decryption key
Alice	$m = 77, e = 7$	$d = 43$
Bob	$m = 91, e = 5$	$d = 29$

m is the public modulus, e is the public encryption exponent, and d is the secret decryption exponent.

Alice has calculated her secret decryption key as described in chapter 6:
$m = 77 = 7 \cdot 11$
$N = (7-1) \cdot (11-1) = 60$
$d = e^{-1} \pmod{N} = 7^{-1} \pmod{60} = 43$

Bob has calculated his secret decryption key as described in chapter 6:
$m = 91 = 7 \cdot 13$
$N = (7-1) \cdot (13-1) = 72$
$d = e^{-1} \pmod{N} = 5^{-1} \pmod{72} = 29$

Non-confidential message

We now show how Alice can sign a non-confidential message for Bob. She applies the RSA decryption algorithm to the plaintext, $msg = 53$, to create the signature:
$sig = msg^{d_A} \pmod{m_A} = 53^{43} \pmod{77} = 25$
She sends the signature, $sig = 25$, and the message, $msg = 53$, to Bob.

When Bob receives the signature and the plaintext message, he applies the RSA encryption algorithm, using Alice's public key, to verify authenticity:
$sig^{e_A} \pmod{m_A} = 25^7 \pmod{77} = 53$

Since encryption of the signature with Alice's public key matches the plaintext, $msg = 53$, Bob can assume that the message is truly from Alice, and that there was no tampering with the message.

Confidential message

Here we include a similar example showing how a digital signature can be created using RSA, and used for authenticity, but in this case the plaintext is considered confidential. Alice wishes to sign a confidential message for Bob. She applies the RSA decryption algorithm to the plaintext, $msg = 53$, to create the pre-signature:
$pre = msg^{d_A} \pmod{m_A} = 53^{43} \pmod{77} = 25$
She then creates the signature by encrypting the pre-signature with Bob's public key:
$sig = pre^{e_B} \pmod{m_B} = 25^5 \pmod{91} = 51$
She then encrypts the message with Bob's public key, to ensure confidentiality, producing a ciphertext:
$cipher = msg^{e_B} \pmod{m_B} = 53^5 \pmod{91} = 79$
She sends the signature, $sig = 51$, and the ciphertext, $cipher = 79$, to Bob.

When Bob receives the signature and the ciphertext, he wishes to verify authenticity. He first applies the RSA decryption algorithm, using his own key, to the signature, to obtain the pre-signature:
$pre = sig^{d_B} \pmod{m_B} = 51^{29} \pmod{91} = 25$
Next he uses Alices public key to encrypt the pre-signature, to obtain what is presumably the message:
$msg_1 = pre^{e_A} \pmod{m_A} = 25^7 \pmod{77} = 53$
Then Bob decrypts the ciphertext, $cipher = 79$, using his own secret key:
$msg_2 = cipher^{d_B} \pmod{m_B} = 79^{29} \pmod{91} = 53$

Since encryption of the signature with Alice's public key matches the plaintext, $msg_1 = msg_2 = 53$, Bob can assume that the message is truly from Alice, and that there was no tampering with the message.

8.2.3 Efficiency - Hashed signatures

We recall from chapter 6 that public key algorithms are generally not used for confidentiality, particularly for long plaintext messages because private key algorithms are much faster. We face the same problem when generating digital signatures. If the plaintext being signed is very large, the signature algorithms that we have seen will require too much time.

Here we suggest a more efficient, and more commonly used, method for creating digital signatures. Instead of applying the decryption algorithm to the plaintext, we first use a hash function on the plaintext, msg, to produce a fixed size, and significantly smaller, *fingerprint* of the plaintext.

This fingerprint is then the input to the decryption, using the signer's secret key, producing the signature, sig.

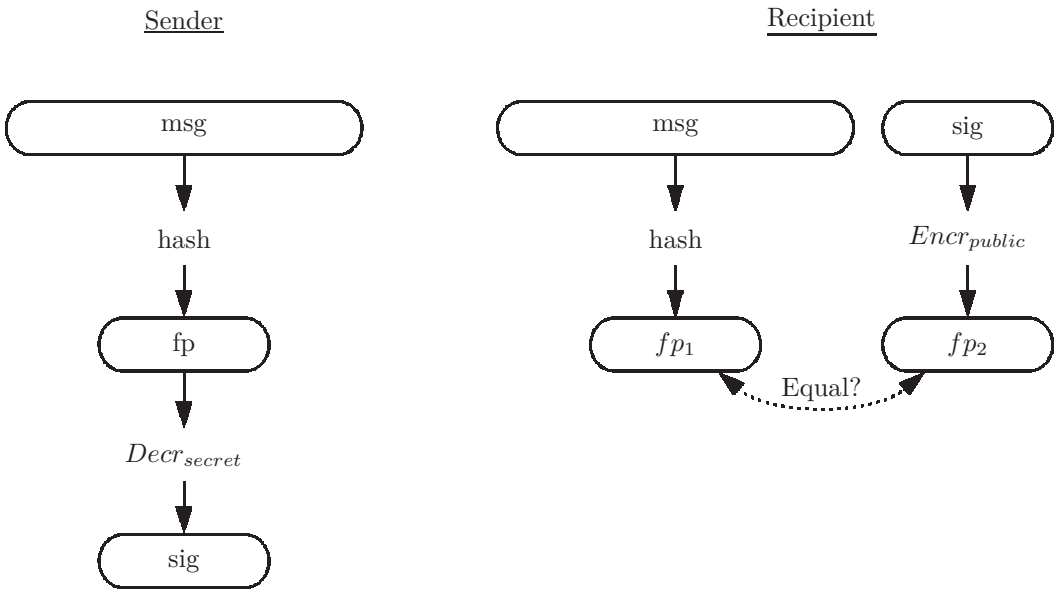


Figure 8.4: A digital signature is produced by hashing the plaintext and signing the hash output, or fingerprint (fp), for efficiency. The recipient verifies authenticity if the two fingerprints, fp_1 and fp_2, are equal.

$sig = Decr(hash(msg), sec)$
where $hash$ is a standard hash function such as SHA-1, and sec is the signer's secret decryption key. The sender sends the plaintext, msg, and the signature, sig to the recipient.[3]

At the receiving end the recipient generates two fingerprints,[4] fp_1 and fp_2, and compares them:

- $fp_1 = hash(msg)$
- $fp_2 = Encr(sig, pub)$

fp_1 is obtained by using the plaintext as input to the hash function, and fp_2 is obtained by using the signature as input to the encryption algorithm with the signer's public key.

The recipient compares the two fingerprints, and if they are equal, assumes that authenticity has been verified. Note that since the output of the hash function is of a fixed size, and typically much smaller than a large plaintext, all encryption and decryption will be fast.

A diagram of this process is shown in Figure 8.4.

[3]Here we are assuming that confidentiality is not necessary.

[4]Recall from chapter 5, that the output of a hash function is often called a *fingerprint*. If two different values are input to a hash function, their outputs are very likely to be different.

8.2.4 Exercises

1. Show, using appropriate substitutions, that verification of a digital signature should produce the original plaintext:

   (a) In the case that the plaintext is not confidential.

   (b) In the case that the plaintext is confidential.

2. You are given Jane's and Rob's public RSA keys:

	public encryption key	secret decryption key
Jane	$m = 77, e = 7$	$d =?$
Rob	$m = 91, e = 5$	$d =?$

   (a) Derive their secret decryption keys

   (b) Show how Rob can create a signature for the non-confidential plaintext, $msg = 24$

   (c) If Rob sends the plaintext and signature from the previous problem to Jane, show how Jane can verify its authenticity.

   (d) Show how Jane can create a signature and ciphertext for the confidential plaintext, $msg = 24$, which she intends to send to Rob.

   (e) If Jane sends the ciphertext and signature from the previous problem to Rob, show how he can verify its authenticity and decrypt the ciphertext.

3. In the section on efficiency we showed how digital signatures can be produced more efficiently by using a hash function to generate fingerprints. The recipient compares fp_1 and fp_2 to verify authenticity. Using substitutions, show that fp_1 and fp_2 should be equal.

4. In the section on efficiency we showed how digital signatures can be produced more efficiently by using a hash function to generate fingerprints. In that discussion we assumed that the sender and receiver are not concerned with confidentiality. Draw a diagram, similar to Figure 8.4, showing how this is done in the case that the sender and receiver wish the communication to be confidential. The plaintext, msg should be encrypted to ensure confidentiality. Assume the sender's public and secret keys are pub_s and sec_s, respectively. Assume the recipient's public and secret keys are pub_r and sec_r, respectively.

8.3 Public key infrastructure - Digital certificates

8.3.1 Man-in-the-middle attack

We have seen how public keys appear to have solved the problem of key distribution, and we have seen that digital signatures appear to solve the authenticity problem.

However, it is still possible for security to be compromised by what is known as a *man in the middle attack* which we describe below:

1. Bob is communicating with Amazon, using public key encryption.
2. Initially Eve intercepts all messages and relays them without interfering.
3. Bob clicks on the "check-out" button.
4. Eve does not relay this to Amazon. Instead she fabricates an Amazon page and sends it to Bob, requesting payment information (e.g. credit card number), and providing Eve's public key.
5. Bob sends the payment information, encrypted with Eve's public key.
6. Eve intercepts, decrypts, and now has Bob's credit card information.
7. Eve completes the transaction with Amazon as though she is Bob, using Amazon's public key to encrypt the credit card information.
8. The purchase is complete, but Eve has Bob's credit card information, and neither Bob nor Amazon suspects any foul play.

8.3.2 Public key authority

More generally, if people simply announce their public keys by sending them out on the internet, there is nothing to prevent one's adversary from masquerading with a false identity, and putting out a public key. For example, Eve could claim to be Bob and put out a public key, ostensibly Bob's public key, but actually Eve's public key.

This problem led to the establishment of *public key authorities*. People would send their public keys, along with identifying information and contact information (email or url), to a public key authority. The authority would make these available upon request. If the public key authority is used by many people, it eventually becomes known to be reliable, i.e. a *trusted* authority. To obtain an entity's (an individual, company, or organization) public key, instead of contacting that entity directly, one would request it from the public key authority, by sending the identifying information to the authority.

The public key authority would also have its own key-pair. Communication to the authority can be encrypted with its public key. Suppose Alice and Bob wish to obtain each other's public key from the authority, PKA. The sequence of events could be as shown below:

1. Alice sends a message to the PKA requesting Bob's public key; she also sends a time-stamp with this request.
2. The PKA responds to Alice with a message, and signature (signed with PKA's secret key). Included in this message are:
   (a) Bob's public key
   (b) Alice's original request, to verify this is coming from PKA
   (c) Alice's time-stamp, ensuring this is not a response to an earlier request
3. Alice uses Bob's public key to encrypt a message to him identifying herself
4. Bob obtains Alice's public key in the same way that she obtained his public key

8.3.3 Digital certificates and certificate authorities

The problem of sharing public keys eventually became so prevalent, that public key authorities were soon replaced by digital certificates and certificate authorities.

A *digital certificate* is similar in purpose to a government-issued id card or passport. It not only contains an entity's public key, but also contains information which uniquely identifies the entity. Digital certificates enable people to exchange public keys without contacting a public key authority. To share a public key, instead of sending out the key a person will normally send out their digital certificate, which contains the public key. The recipient verifies the certificate (signature) using the issuer's public key. A digital certificate for an entity may consist of:

- The entity's public key
- Information which should uniqely identify the entity:
  - Name
  - email address
  - location
- A digital signature of the above, provided by a *certificate authority*

A certificate authority (CA) is an entity which verifies certificates by signing them with their secret key. A certificate can then be verified by using the certificate authority's public key. As with public key authorities, a certificate authority which has issued many certificates comes to be trusted.

As of 2022 some common certificate authorities are:

- Let's Encrypt
- Comodo

- Digicert

These CAs do not use their own format for the certificates, rather a standard format has been provided by the Iternational Telecommunications Union (ITU). This standard is known as X.509, and consists of the following:

- Version number
- Serial number
- Signature algorithm ID (RSA, ElGamal, etc)
- Issuer name (Let's Encrypt, Comodo, Digicert, etc.)
- Validity period (with expiration date)
- Subject's name, organization, organizational unit, and country
- Subject's public key information
  - Public key algorithm (RSA, ElGamal, etc)
  - Public key
- Issuer's signature algorithm and signature

When a certificate expires (typically several months to a few years), the subject applies for a new certificate.

8.3.4 Public key exchange

We now consider the case where two people (or any entities) wish to share their public keys with each other. In the following scenario Alice and Bob each have their own key pairs. They wish to share their public keys with each other. They have agreed to use the certificate authority Comodo.

1. They independently obtain certificates from Comodo
   - Alice sends her public key, K_A, and other information which identifies her, ID_A, such as email address, location, her company or organization, to Comodo
   - Bob sends his public key, K_B, and other information which identifies him, ID_B, such as email address, location, his company or organization, to Comodo
2. Comodo sends signed certificates to both Alice and Bob
   - Comodo creates a certificate, $Cert_A$ for Alice, consisting of:
     (a) A timestamp, T_1
     (b) Alice's identification information, ID_A
     (c) Alice's public key, K_A

- Comodo creates a signature, Sig_A, for Alice's certificate using its secret decryption key
- Comodo sends the signature, Sig_A to Alice

- Comodo creates a certificate for Bob, consisting of:
  (a) A timestamp, T_2
  (b) Bob's identification information, ID_B
  (c) Bob's public key, K_B
- Comodo creates a signature, Sig_B, for Bob's certificate using its secret decryption key
- Comodo sends the signature, Sig_B to Bob

3. Alice obtains her certificate,$Cert_A$, from the signature

   (a) Alice applies Comodo's public key, K_C (available from Comodo's web site) to the signature, Sig_A

   (b) Alice now has her certificate, $Cert_A$. She checks the time stamp to ensure that the certificate was not forged from a previous certificate.

4. Bob obtains his certificate, $Cert_B$ from the signature

   (a) Bob applies Comodo's public key, K_C (available from Comodo's web site) to the signature, Sig_B

   (b) Bob now has his certificate, $Cert_B$. He checks the time stamp to ensure that the certificate was not forged from a previous certificate.

5. Alice and Bob can now share their certificates with each other

   - Alice sends her signed certificate, Sig_A, to Bob
   - Bob sends his signed certificate, Sig_B, to Alice

6. Alice and Bob can now obtain each other's public key

   - Alice extracts Bob's public key, K_B, from Bob's signed certificate, Sig_B, by applying Comodo's public key, K_C, to Sig_B
   - Bob extracts Alice's public key, K_A, from Alice's signed certificate, Sig_A, by applying Comodo's public key, K_C, to Sig_A

A diagram of steps 1 and 2 of this process is shown in Figure 8.5.
.
A diagram of steps 3 through 6 of this process is shown in Figure 8.6.
.

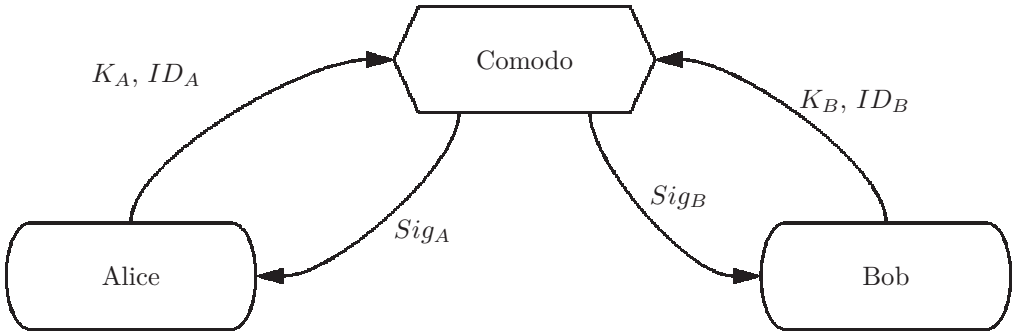


Figure 8.5: Alice and Bob each obtain a digital certificate from the certificate authority, Comodo. The signatures, Sig_A and Sig_B each contain a time stamp, ID information, and the public key for each individual.

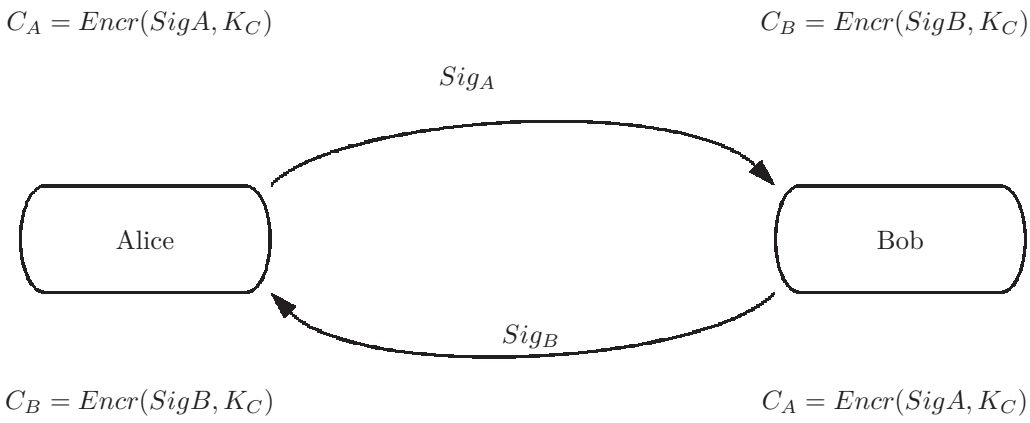


Figure 8.6: Alice and Bob extract their own certificates from the signatures received from the certificate authority Comodo. They then exchange the signatures with each other, and extract each others public certificates, which contain their public keys.

8.3.5 Exercises

1.

The steps involved in an exchange of public keys, using a certificate authority, are shown below, with a few minor (and erroneous) changes. Without peeking, identify three changes from the steps shown in the this section. Bob and Alice are using Comodo to share public keys:

1. They independently obtain certificates from Comodo
   - Alice sends her public key, K_A, and other information which identifies her, ID_A, such as email address, location, her company or organization, to Comodo
   - Bob sends his public key, K_B, and other information which identifies him, ID_B, such as email address, location, his company or organization, to Comodo
2. Comodo sends signed certificates to both Alice and Bob
   - Comodo creates a certificate, $Cert_A$ for Alice, consisting of:
     (a) A timestamp, T_1
     (b) Alice's identification information, ID_A
     (c) Alice's public key, K_A
   - Comodo creates a signature, Sig_A, for Alice's certificate using its secret decryption key
   - Comodo sends the signature, Sig_A to Alice
   - Comodo creates a certificate for Bob, consisting of:
     (a) A timestamp, T_2
     (b) Bob's identification information, ID_B
     (c) Bob's public key, K_B
   - Comodo creates a signature, Sig_B, for Bob's certificate using its secret decryption key
   - Comodo sends the signature, Sig_B to Bob
3. Alice obtains her certificate,$Cert_A$, from the signature
   (a) Alice applies Comodo's secret key, K_C (available from Comodo's web site) to the signature, Sig_A
   (b) Alice now has her certificate, $Cert_A$. She checks the time stamp to ensure that the certificate was not forged from a previous certificate.
4. Bob obtains his certificate, $Cert_B$ from the signature
   (a) Bob applies his public key, K_B to the signature, Sig_B

(b) Bob now has his certificate, $Cert_B$. He checks the time stamp to ensure that the certificate was not forged from a previous certificate.

5. Alice and Bob can now share their certificates with each other
   - Alice sends her signed certificate, Sig_A, to Bob
   - Bob sends his signed certificate, Sig_B, to Alice
6. Alice and Bob can now obtain each other's public key
   - Alice extracts Bob's public key, K_B, from Bob's signed certificate, Sig_B, by applying Comodo's public key, K_C, to Sig_B
   - Bob extracts Alice's public key, K_A, from Alice's signed certificate, Sig_A, by applying his public key, K_B, to Sig_A

Chapter 9

Packages: PGP

We have presented several algorithms designed to ensure confidentiality, integrity, and authenticity in our digital communications. These algorithms are effectively keeping the internet 'alive' and safe from attack.

As a practical matter we cannot expect every user to implemement these algorithms for their own protection. Even if they were capable of developing their own software reliably, the time and effort spent would be huge. Instead, there is packaged software ready to use. One such package is PGP, and its derivatives.

9.1 History

PGP, or Pretty Good Privacy, was developed by Phil Zimmerman in 1991. PGP is a software package designed to provide encryption, decryption (both symmetric and asymmetric), hash functions, and digital signatures with a simple user interface. It was later available free from MIT.

This sofware was considered so secure that in 1993 the FBI accused Zimmerman of exporting munitions.[1] Zimmerman was able to avoid prosecution by publishing a book, through MIT Press, containing the PGP source code (version 2) and distributing it across the world. This work was protected by the first amendment to the U.S. Constitution.

In 1996 the government dropped its case against Phil Zimmerman, who then formed the company PGP Corporation to market version 3 of PGP. This was eventually acquired by Symantec in 2010.

At about the same time open source versions of PGP were made freely available. OpenPGP is now distributed by GNU[2] as GnuPG, or GPG. It is

[1] There were, and still are, federal statutes which forbid the exporting of weapons such as howitzers, mortars, tanks, etc. This list of munitions also included strong encryption software, because the efforts of government agencies such as the NSA, DIA, and CIA would be compromised if our enemies had secure communications.

[2] The GNU Project, started in 1983 by Richard Stallman of MIT, devoted to the principle that software source code, including compilers and operating systems, should be freely avail-

available with a graphical user interface or command line interface from the GNU Software Project. `https://gnupg.org`

9.2 GPG

In this section we expose some of the principle GPG commands. If using a graphical user interface, the functionality is the same but the user would select from a window of buttons and text fields to accomplish the same task.

9.2.1 Unix/Linux command line

For those unfamiliar with the Unix/Linux command line we provide a brief introduction:

- Commands read from an input data file known as `stdin`, which is the keyboard by default.
- Commands write to an output data file known as `stdout`, which is the display by default.
- The default input file can be redirected to a data file using a less-than symbol:
  `cmd <inFilename`
- The default output file can be redirected to a data file using a greater-than symbol:
  `cmd >outFilename`
- Commands may have zero or more operands which follow the command name:
  `cmd operand operand operand ...`

9.2.2 GPG commands

When using the command line, the format of a GPG command is:
`gpg --cmd-name -operand -operand`
in which the `gpg` is a command to the operating system to invoke GPG, and the `--cmd-name` tells GPG which of its commands is to be invoked.

List keys

GPG will store your keys and key-pairs on a 'key ring'. To see the public keys on your key ring use the `list-keys` command:
`gpg --list-keys`
Initially you will have no public keys to be shown.

able, as a means to promote further development and collaboration. This is thought to be the impetus for open source software as we know it today. GNU is said to be a recursive acronym: *GNU's not Unix*

Generate keys

To generate a new key-pair use the `gen-keys` command:

```
gpg --gen-keys
```

GPG will then prompt you to enter information used to identify yourself, such as your name, an email address, and a passphrase for security. GPG will then generate your keys; in the process it will use a pseudo-random number generator to generate random bytes. You can improve the randomness by typing random keys on the keyboard while your keys are being generated.

Your key-pair has now been created. You can view your public key with the `list-keys` command:

```
gpg --list-keys
```

This will show a short identification number for the key (in hexadecimal). Note that the key itself, which is hundreds of bits, is not shown here.

As we saw in chapter 6 each key pair consists of a public key and a private (or secret) key. To view your secret key use the `list-secret-keys` command:

```
gpg --list-secret-keys
```

Your keys are stored in a *keyring* which is a hidden file[3] in your home directory.

Exporting and importing keys

Users of GPG often need to send their keys to others, or to save a backup copy elsewhere. To do this the keys must first be *exported* to a text file from the keyring. To export a public key to a text file named `filename`:

```
gpg -a --export NAME > FILENAME.gpg
```

in which NAME matches some part of the user's name stored in the key. Since the `export` command writes to the standard output file, `stdout`, we have redirected it to a specified file `FILENAME.gpg`, in the current directory. The `-a` option, means `armor` the output file: convert it to ASCII characters, in base-64 encoding (see chapter 4 for a description of base-64 encoding).

To export a secret key to a text file:

```
gpg -a --export-secret-keys NAME > FILENAME.gpg
```

Be sure to name your files appropriately so that you do not send out a secret key inadvertently.

The files you have created are plain text files, which can be viewed with a text editor, or the `cat` or `more` commands. The public keys can also be placed on a web page or emailed to others.

When you receive an armored public key from someone else, you can import it to your keyring:

```
gpg --import FILENAME.gpg
```

This will read the armored key in the given file, and add it to your keyring.

[3] A hidden file has a name that begins with a '.', and can be viewed on a Unix/Linux system with the -a option: `ls -a` to show all filenames.

You can refer to this key by specifying some portion of its NAME field, i.e. the person's identification information.

Deleting keys from the keyring

To remove a public key from your keyring:
`gpg --delete-keys NAME`
where NAME is some part of the identification information.

To remove a secret key from your keyring:
`gpg --delete-secret-key NAME`
where NAME is some part of the identification information.

Encryption/decryption

To encrypt a plaintext with a public key (i.e. asymmetric encryption):
`gpg -r NAME --output OUTFILE.gpg --encrypt INFILE`
This will read the given INFILE as the plaintext and write the ciphertext to the OUTFILE. It will use the public key of the given NAME.

To decrypt a ciphertext with your own secret key
`gpg -r NAME --output OUTFILE.gpg --decrypt INFILE`
This will read the given INFILE as the ciphertext and write the plaintext to the OUTFILE. It will use the secret key of the given NAME (presumably your own).

To encrypt a plaintext with your secret key (i.e. symmetric encryption):
`gpg -u NAME -c --output OUTFILE.gpg <INFILE`
In which the NAME identifies yourself. The command reads from `stdin` which is redirected here to INFILE, which stores the plaintext. The ciphertext is placed in OUTFILE.gpg.

To decrypt a ciphertext that was encrypted with a secret key (i.e. symmetric)
`gpg -u NAME -d <INFILE`
In which the NAME identifies yourself. The command reads the ciphertext from `stdin` which is redirected here to INFILE.

Digital signatures

To create a digital signature for a plaintext, use the `--sign command`:
`gpg --sign > SIG.gpg`
This command writes the signature to `stdout` which here is redirected to `SIG.gpg`. It reads the file to be signed from `stdin`. To read from a plaintext file, redirect `stdin`:
`gpg --sign < INFILE > SIG.gpg`

To verify a signature:
`gpg --verify < SIG.gpg`
This command writes the result of verification to `stdout`.

9.2.3 Exercises

1. Use GPG to
   (a) Generate a key-pair on your keyring.
   (b) Encrypt a message (stored in the plain text file `msg.txt`) to yourself using your public key. Store the ciphertext in a file named `cipher.txt`.
   (c) Decrypt the ciphertext with your secret key, sending the plaintext to the `stdout` file.
2. Use GPG to
   (a) Send your public key, as an ASCII text file, to a classmate or the instructor.
   (b) Request a public key from a classmate or the instructor.
   (c) Encrypt a plaintext question for a classmate or your instructor using their public key. Ask them to respond with an encrypted ciphertext.
   (d) Use your secret key to decrypt the response.
3. Repeat the previous problem, but send a signature to the recipient, in addition to the ciphertext. Ask the recipient to reply with a signature, and use the signature to verify the response for authenticity.
4. Use GPG to
   (a) Generate a secret session key to be used for GPG encryption (symmetric).
   (b) Encrypt the session key with a classmate's (or instructor's) public key.
   (c) Send the encrypted session key to the classmate or instructor.
   (d) Begin a session in which all communication is encrypted and decrypted with the session key.

Chapter 10

Cryptographic Protocols

10.1 Attacks

The internet is composed of many nodes, each of which is responsible for forwarding data packets to their respective destinations. Some of these nodes may be infected with malware that has access to data which is in transit to a destination. A web server responds to requests for data from web browsers. Both the request and the response can be vulnerable to attck. Here we outline some common attacks on web traffic on the internet. We describe below some protocols which are designed to thwart these attacks.

- Integrity attacks - An *integrity attack*is one in which the data in a packet is modified while in transit to its destination. An integrity attack can be implemented with a so-called 'Trojan Horse' attack in which a malware program disguises itself as a harmless program. Other forms of malware which modify data in transit are also considered integrity attacks. Another common implementation of an integrity attack is known as a 'buffer overflow' attack, in which a pointer or array subscript out of range is not caught by a compiler, allowing the malware to access sensitive memory areas.[1]

- Confidentiality Attacks - An attack on confidentiality results in the attacker gaining access to data which the sender and receiver consider confidential. This is also known as eavesdropping, as there is no modification of the data, which is sent on to its destination.

- Denial of Service - An attack which attempts to overload a server with so many requests that the server is no longer functional. The attack may have several sources, in which case it is known as a distributed denial of service (DDoS) attack. Examples of DDOS attacks occurred in 2017 against Google Cloud, and in 2020 against Amazon Web Services.

[1] A buffer overflow attack can also be used to implement a confidentiality attack.

- Authentication - An attack in which the originator's identity is altered. Integrity attacks are often included here as authentication attacks because ensurance of authenticity usually includes ensurance of integrity.

10.1.1 Exercises

1. What are the four kinds of attacks that may occur on the internet?
2. Identify the name of each of the following attacks on the internet:
   (a) An eavesdropping attack
   (b) The sender's identity is changed
   (c) Data is changed while in transit to a destination
   (d) A server is flooded with so many requests that it is unable to function

10.2 SSL and TLS

One of the first security protocols developed is known as the Secure Socket Layer protocol (SSL). This software was developed by Netscape, one of the first modern web browsers, in 1995.

After several security flaws were identified, version 2 of SSL was released in open source, and was known as OpenSSL. A buffer overflow security flaw, known as 'heartbleed', was discovered in 2014. This bug was corrected in version 3 of SSL which is also known as TLS (Transport Layer Security). In what follows we describe version 3 of SSL (TLS).

The TLS protocol consists of two phases; the first phase, known as *handshake*, establishes the identities of the sender and receiver, as well as the encryption and authentication algorithms to be used. In the handshake phase the parties agree on a session key for confidentiality. The second phase, known as *record*, transfers the encrypted data, and signature, from sender to receiver.

TLS handshake phase

In the handshake phase of TLS parameters for a transaction are established. Also authenticity of the sender and receiver is verified.

In this example, we assume Alice wishes to pay Target for an online purchase. Since credit card information is involved here, security is important.

1. Alice sends to Target:
   (a) The highest version of TLS that she supports
   (b) A 4-byte time-stamp and a 28-byte random number: r_A
   (c) A cipher suite, ordered by preference, to establish the algorithms to be used:

i. Public key algorithms that she supports, e.g. RSA, ElGamal,...
ii. Private key algorithms (symmetric) that she supports, e.g. DES, AES, ...
iii. Hash functions, e.g. MD5, SHA-1, ...
iv. Data compression algorithms, e.g. PKZIP

2. Target responds with:
   (a) A 4-byte time-stamp and a 28-byte random number: r_T
   (b) Algorithms to be used (which Alice supports), e.g. RSA, AES, SHA-1, PKZip
   (c) Target's X.509 certificate, containing Target's public key
3. Alice responds with a 48-byte 'pre-master secret', s_{pm}, encrypted with Target's public key
4. Both Alice and Target now have:
   - r_A, which is Alice's time-stamp and random number
   - r_T, which is Target's time-stamp and random number
   - s_{pm}, which is their 48-byte pre-master secret
5. Both Alice and Target can now calculate the master secret, s_m, independently. s_m is the concatenation of the following;
   (a) MD5(s_{pm} || SHA-1("A" || s_{pm} || r_A || r_T))
   (b) MD5(s_{pm} || SHA-1("BB" || s_{pm} || r_A || r_T))
   (c) MD5(s_{pm} || SHA-1("CCC" || s_{pm} || r_A || r_T))

   in which the "A", "BB", and "CCC" are included to ensure that the hash results are not at all similar.[2] The || symbol means concatenation of bit strings.
6. Since MD5 produces a 16-byte result, the master secret,s_m, will consist of 48 bytes. Three 16-byte keys are then generated from s_m using the same process that generated the master secret from the pre-master secret: The three keys are:
   (a) A secret session key for the block cipher (e.g. AES):
   Ksession = MD5(s_m || SHA-1("A" || s_m || r_A || r_T))
   (b) A Message Authentication Key (MAK):
   MAK = MD5(s_m || SHA-1("BB" || s_m || r_A || r_T))
   (c) An initial vector for the CBC blocking mode (IV):
   IV = MD5(s_m || SHA-1("CCC" || s_m || r_A || r_T))

[2]Recall that a good hash algorithm will produce very different results if only a few bits of the input are changed.

The strings "A", "BB", "CCC" are included, here again, and ensure that the three keys are very different. A diagram of the TLS handshake phase is shown in Figure 10.1.

TLS record phase

The second phase of TLS, known as *record*, uses the three keys generated in the handshake phase to encrypt the plaintext purchase information and send the encrypted data, with a signature, from sender to receiver. Continuing with the transaction in which Alice is making a purchase from Target, the details of the record phase are shown below:

1. Alice compresses the plaintext, msg, containing the purchase information, including her credit card information, using the agreed upon compression algorithm, which in this case is PKZIP.
   $msg_{zip} = PKZIP(msg)$

2. Alice hashes the compressed message with the Authentication Key, MAK, to produce a Message Authentication Code, MAC:
   $MAC = MD5(msg_{zip}||MAK)$

3. Alice encrypts the purchase information, along with the message authentication key, MAK, including credit card information, using the block cipher, AES in this case, with session key $Ksession$, blocking mode CBC, and the initial vector, IV, created in the handshake phase:
   $cipher = AES_{Ksession}(msg_{zip}||MAC)$
   The AES encryption uses CBC blocking mode with the initial vector, IV, generated in the handshake phase. She sends the resulting ciphertext, *cipher*, to Target.

4. Target should be able to decrypt the ciphertext because Target also has the session key.

A diagram of the TLS record phase is shown in Figure 10.2.

10.2.1 Exercises

1. Bob wishes to make an online purchase from Amazon. In which TLS phase does each of the following occur?

   (a) Bob identifies the algorithms that he supports.

   (b) Amazon sends Bob a certificate containing Amazon's public key.

   (c) Bob encrypts the purchase information and sends it to Amazon.

2. True or False: In the handshake phase three keys are generated. These keys are almost the same because they differ only with the strings "A", "BB", and "CCC".

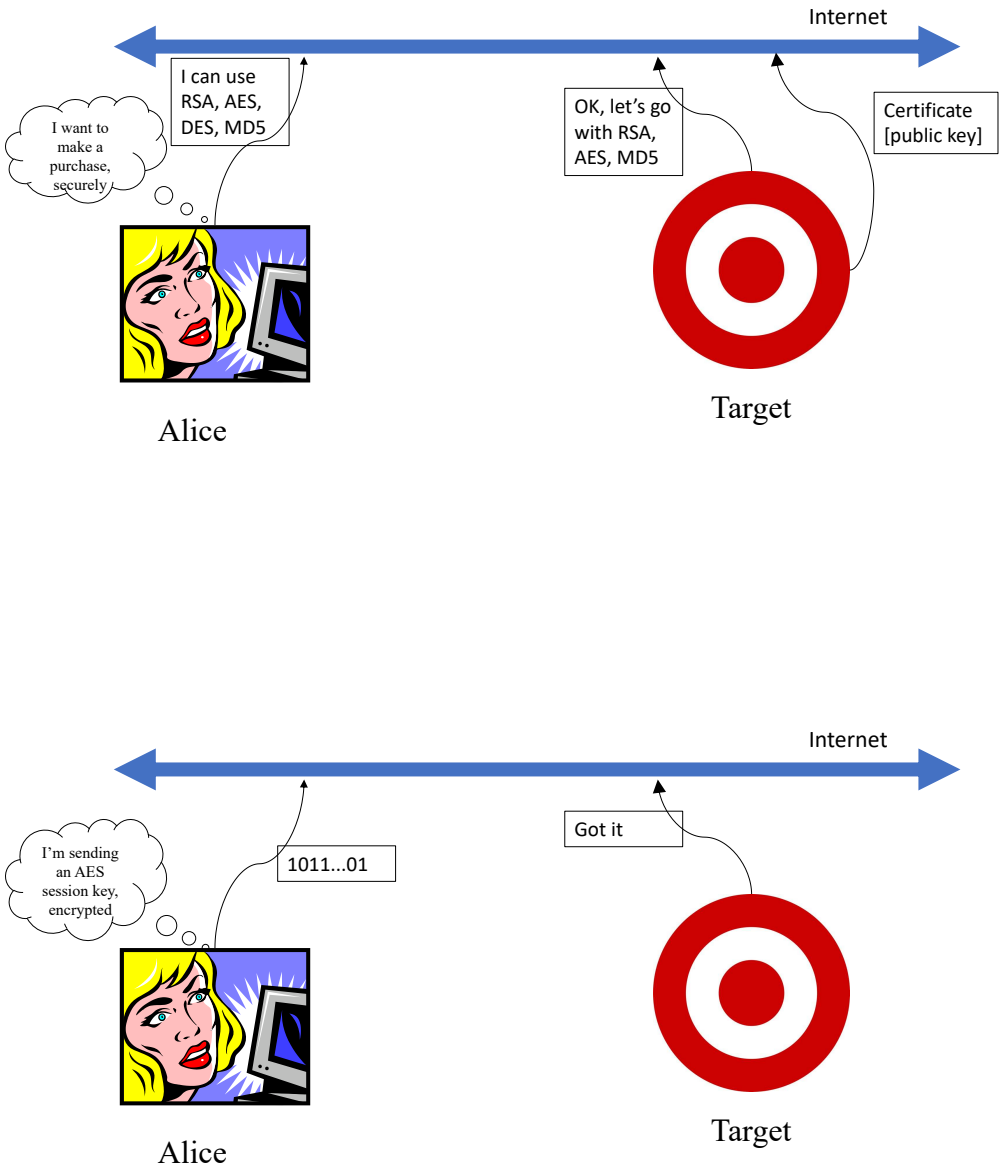


Figure 10.1: Alice wishes to make a secure purchase from Target. In the TLS handshake phase they agree on algorithms and keys to be used for authenticity and confidentiality.

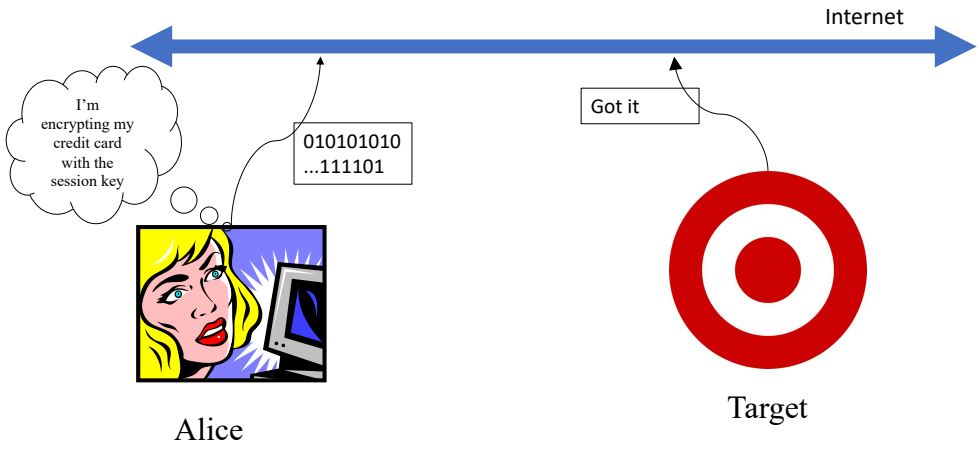


Figure 10.2: The TLS record phase: Alice encrypts her credit card and purchase information with the shared session key, and sends it to Target.

Chapter 11

Java Services

We have seen how security, in the form of confidentiality, integrity, and authenticity, can be ensured with several algorithms.

Most developers who need to ensure security use pre-packaged software such as GPG. Another option is to use packages which are part of the extensive Java class library.[1] The first such package is `java.security`. It was later augmented with another package known as `javax.crypto`. Services provided in these packages include:

- Encryption and Decryption
  - Symmetric, i.e. private key
  - Asymmetric, i.e. public key
- Hashing and message digests
- Authentication
- Digital Signatures
- Digital Certificates

In this chapter we describe how to use these Java security services.

11.1 Private key encryption and decryption

In this section we are looking at the services for private key encryption and decryption, i.e. symmetric encryption/decryption.

[1] Similar packages are also available for C++ and Python, but we prefer to use Java for its portability and security features.

11.1.1 Class Cipher

For encryption and decryption, services in the Java class library are in the Cipher class, which is in the package `javax.crypto`. A Cipher object is instantiated with a static *factory* method[2] named `getInstance`:

```
import java.security.*;
import javax.crypto.*;
/** @return an instance of a Cipher.
 *  @param transformation specifies the algorithm, blocking mode,
 *  and padding, separated by slashes for the Cipher being created.
 */
public static Cipher getInstance(String transformation)
```

The Cipher created below will use the DES algorithm, with Electronic Codebook (ECB) blocking mode, and PKCS5 padding:

```
Cipher myCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
```

In order to use this Cipher we must first generate a key. This is done with a KeyGenerator object, which is also produced with a factory method, `getInstance` in this case:

```
/** @return a new key to be used with the specified algorithm
 *  @param algorithm is the name of a private key algorithm
 */
public static KeyGenerator getInstance(String algorithm)
```

As an example we produce a KeyGenerator object named `keyGen` to be used with the DES algorithm:

```
KeyGenerator keyGen = KeyGenerator.getInstance("DES");
```

However, we have not yet produced a key. We must use the KeyGenerator object to produce a key. The key will be a `SecretKey`, meaning that it is to be used with a private key algorithm, as opposed to a public key algorithm. To produce the key we use an instance method in the KeyGenerator class:

```
/** @return a new SecretKey for the selected algorithm
 */
public SecretKey generateKey()
```

For example:

```
SecretKey key = keyGen.generateKey()
```

Encryption

Since we now have a key we can start encrypting plaintext. First we need to initialize the cipher by telling it that we will be using it for encryption (and not decryption) and by providing it with the key. This is done with an `init` instance method in the Cipher class:

[2] A factory method or class is a Java design pattern which produces a new object.

```
/** Initialize this Cipher with a mode and a Key
 */
public void init (int mode, Key key)
```

The mode can be a class constant in the Cipher class, such as ENCRYPT_MODE or DECRYPT_MODE. Continuing with our example:
`cipher.init(Cipher.ENCRYPT_MODE, key);`

We are now ready to encrypt a plaintext; the cipher expects the plaintext as an array of bytes,[3] rather than a String. We will use the getBytes() method in the String class to convert the String to an array of byte values. The ciphertext will also be produced as an array of bytes.[4]

```
String msg = "Let's have lunch";
byte[] plaintext = msg.getBytes();
byte[] ciphertext;
```

To do the encryption we can use the `update(byte[ ])` instance method as frequently as needed, or for short plaintext, we can use the `doFinal(byte[ ])` method.

```
/**
 *  @param bytes Either plaintext or ciphertext
 *  @return the ciphertext if encrypting, or the
 *  plaintext if decrypting.
public byte[ ] doFinal (byte[ ] bytes)
```

In our example:
`ciphertext = cipher.doFinal(plaintext);`
The integer values in the ciphertext are not necessarily the codes of ASCII characters; they are seemingly random integers in the range [-128..127]. We cannot view these values with a String constructor; instead we could write a simple method to display the integer values.[5]

Decryption

To decrypt a given ciphertext we could create and initialize a new Cipher object (with the same key), or we could use the same Cipher object and the same key that we used for encryption. To do that we will need to initialize it for decryption:
`cipher.init(Cipher.DECRYPT_MODE,key);`
Then we can decrypt the plaintext, putting the recovered plaintext into an array of bytes:

```
byte[ ] recoveredPlainttext;
recoveredPlaintext = cipher.doFinal(ciphertext);
```

[3] A byte is simply an 8-bit integer

[4] The fact that these methods work with an array of bytes, rather than a String of characters, means we can encrypt any digital information, and not just plain text.

[5] There is no automatic `toString` method for an array.

As with encryption, for long ciphertexts one should call the update method repeatedly, and call the doFinal method for the last block. Fortunately there is an easy way to convert this array of bytes to a String:

```
String recoveredMsg = new String(recoveredPlaintext);
```

11.1.2 Exercises

1. Write a Java method to display an array of bytes in readable form.

2. (a) Create a Java class, `Symmetric` for secret key encryption/decryption. It should have fields for a `Cipher` object, a `SecretKey` object, and a `KeyGenerator` object.

   i. In the constructor initialize the `Cipher` object and the `KeyGenerator` object. Use the `KeyGenerator` object to initialize the `SecretKey` field.

   ii. Include a method to encrypt a given msg with the `SecretKey`.

   ```
   /** @param msg is a String to be encrypted
    *  @return the ciphertext as an array of bytes.
    */
   public byte[] encrypt (String msg)
   ```

   Write a driver to call this method, and display the ciphertext in readable form.

   iii. Include a method to decrypt a given ciphertext.

   ```
   /** @param ciphertext is an encrypted String
    *  @return The plaintext as an array of bytes.
    */
    public byte[] decrypt (byte[] ciphertext)
   ```

   In your driver test the decryption, and display the plaintext as a readable String. It should match the original msg.

   (b) Decrypt the ciphertext and display the recovered message as a String.

11.2 Public key encryption

In this section we look at the Java services for public key encryption/decryption, i.e. asymmetric encryption/decryption. This will be similar to the section on symmetric encryption/decryption;

11.2.1 Class Cipher

The Cipher class is in the package `javax.crypto`. We will use a static factory method to produce an instance of Cipher. We must give it an algorithm, a blocking mode, and a padding algorithm. Alternatively, if we do not provide a blocking mode or padding algorithm, it will use defaults:

```
\** @return a Cipher object for the given transformation.
 *  @param transformation specifies the algorithm, blocking
 *  mode, and padding algorithm. Default values may be
 *  used for the blocking mode and padding algorithm.
 */
public static Cipher getInstance(String transformation)
```

For example:
`Cipher cipher = Cipher.getInstance("RSA");`

We now need to generate a pair of keys; recall that with asymmetric encryption there is a public key (for encryption) and a private key (for decryption), and they are mathematically related. In order to generate the key pair, we will need a `KeyPairGenerator`. Again this is produced with a static factory method:

```
/** @return a KeyPairGenerator for the given algorithm
 *  @param algorithm is a public key algorithm, such as
 *  RSA or ElGamal
 */
public static KeyPairGenerator getInstance (String algorithm)
```

In our example, we will produce a `KeyPairGenerator`:

```
KeyPairGenerator keyPairGen;
keyPairGen = KeyPairGenerator.getInstance("RSA");
```

We can now use the KeyPairGenerator to generate a pair of keys with an instance method:

```
/** @return a pair of keys which can be used for RSA
 *  encryption/decryption
 */
public KeyPair generateKeyPair()
```

For our example:

```
KeyPair keyPair;
keyPair = keyPairGen.generateKeyPair();
```

We now have a pair of keys for RSA encryption/decryption. Others will need the public key from that pair to encrypt confidential plaintexts for us. This should be handled by creating a digital certificate, and sending out, or publishing the certificate, because the certificate contains our public key.

Encryption

For our purposes, we can obtain the public key from the keyPair:
`Key publicKey = keyPair.getPublic();`
The public key is used for encryption. We need to initialize the cipher object for encryption:
`cipher.init(ENCRYPT_MODE,publicKey);`
Before we can encrypt the plaintext message, it must be an array of bytes. Suppose the message is a String:

```
String msg = "Let's have lunch";
byte [ ] plaintext = msg.getBytes();
byte [ ] ciphertext;
```

Then we call the `update` method as often as needed to encrypt the entire plaintext. If the plaintext is short, a call to `doFinal` is sufficient.
`ciphertext = cipher.doFinal(plaintext);`

Decryption

Upon receiving the ciphertext, the recipient will need to decrypt with the private key.
`Key privateKey = keyPair.getPrivate();`
Then the cipher needs to be initialized for decryption:
`cipher.init(DECRYPT_MODE,privateKey);`
Decryption is similar to encryption, in that the `update` method can be called repeatedly to decrypt successive blocks of plaintext. Ultimately, the `doFinal` method is called:

```
byte[ ] plaintext;
plaintext = cipher.doFinal(ciphertext);
String msg = new String(plaintext);
```

Here we assume the plaintext is a String.

11.2.2 Exercises

1. (a) Create a Java class, `Asymmetric` for public key encryption/decryption. It should have fields for a `Cipher` object, a `KeyPairGenerator` object, and a `KeyPair` object.

   i. In the constructor initialize the three fields (specify the RSA algorithm).

   ii. Include a method to encrypt a given msg with the `PublicKey`.

   ```
   /** @param msg is a String to be encrypted
    *  @return the ciphertext as an array of bytes.
   ```

```
 */
public byte[] encrypt (String msg)
```

Write a driver to call this method, and display the ciphertext in readable form.

iii. Include a method to decrypt a given ciphertext with the SecretKey which is in your KeyPair..

```
/** @param ciphertext is an encrypted String
 *  @return The plaintext as an array of bytes.
 */
 public byte[] decrypt (byte[] ciphertext)
```

In your driver test the decryption, and display the plaintext as a readable String. It should match the original msg.

11.3 Hashing, Message Digest, MAC

The concept of hash functions was introduced in chapter 5. A hash function may accept an input of unlimited size and produce a fixed-size output. A good hash function will minimize collisions, where two different inputs produce the same output. Another name for hash is 'message digest'.

Hash functions can be used to ensure integrity; by sending a hash of the message along with the message the recipient can hash the message and compare the output with the hash output received - if they are not equal, the recipient does not trust the message.

Hash functions can also be used to improve run time efficiency. When generating a digital signature for a long plaintext, we can hash the plaintext first, and then sign the hash output instead of signing the plaintext. The Java class is called `MessageDigest`, and is in the package `java.security`.

11.3.1 Hashing with MessageDigest

To produce an instance of `MessageDigest`, use the static factory method:

```
/** @param algorithm is the name of the hash or message digest
 *  algorithm, e.g. MD5 or SHA-1
 *  @return a MessageDigest object.
 */
public static MessageDigest getInstance(String algorithm)
```

For example:

```
MessageDigest md;
md = MessageDigest.getInstance("MD5");
```

We are now ready to run the algorithm. There are two relevant methods, `update` and `digest`:

```
/** @param bytes is an array of bytes to be hashed.
 *  Intermediate results are stored in the MessageDigest
 *  object.
 */
public void update (byte [ ] bytes)

/** @return the hash output as an array of bytes.
 */
public byte[ ] digest()
```

The `update` method can be called several times, each with its own input, and folding the outputs into one final digest. The result is then produced by a call to `digest()`. Continuing with our example:

```
md.update("Good morning".getBytes());
md.update("Good afternoon".getBytes());
md.update("Good evening".getBytes());
Byte [ ] result;
result = md.digest();
```

11.3.2 Message Authentication Code - MAC

A message authentication code is essentially a hash function with a password, for extra security. The Java class is `Mac`, and it is in the package `javax.crypto`.

To use a `Mac`, first we will need a secret key. As with the Cipher class, the secret key is produced by a `KeyGenerator`. Both the `KeyGenerator` and the secret key are produced by factory methods:

```
SecretKey key;
KeyGenerator keyGen;
// MD5 with Mac is the algorithm
keyGen = KeyGenerator.getInstance("HmacMD5");
key = keyGen.generateKey();
```

We now have a secret key which can be used with `MD5`. First we need to produce an instance of the `Mac` with a static factory method and initialize it:

```
Mac mac;
mac = Mac.getInstance("HmacMD5");
mac.init(key); // initialized to work with our secret key
```

We are now ready to produce the message digest with the instance method `update(byte [ ])`. To end the process we use the instance method `doFinal()`. We assume the plaintext is in an array of bytes:

```
mac.update(bytes);
byte [ ] md;            // result of message digest
md = mac.doFinal();
```

The result stored in `md` will be an array of 16 bytes, since `MD5` produces a fixed size output of 128 bits.

11.3.3 Exercises

1. Create a class named `MD` to create message digests. It should have:

   - A field which is a MessageDigest
   - A constructor which instantiates the field as an `MD5`.
   - A main method which creates a very long string, and calls a `hash` method to produce a digest of 16 bytes from a given String.
   - The `hash` method which returns the 16 byte digest:

   ```
   /** Use the msgDigest to produce an array of 16 bytes from
    *  a given String of any length.
    *  @param msg A String of characters of any length
    *  @return Array of 16 bytes, satisfying the usual characteristics
    *  of a good hash function.
    */
   public byte [] hash (String msg)
   ```

2. Repeat the previous exercise, but call the class `MAC` instead of `MD`. It should use a `MAC` instead of a `MessageDigest`. The fields should be:

   - A `MAC`
   - A `SecretKey`
   - A `KeyGenerator`

   The constructor should instantiate the fields, and initialize the `MAC`. Include a `hash` method which returns the 16 byte digest:

   ```
   /** Use the msgDigest to produce an array of 16 bytes from
    *  a given String of any length.
    *  @param msg A String of characters of any length
    *  @return Array of 16 bytes, satisfying the usual characteristics
    *  of a good hash function.
    */
   public byte [] hash (String msg)
   ```

11.4 Digital signatures

Digital signatures are used to authenticate communications between users and were covered in chapter 6. Signatures make use of public key algorithms such as RSA or ElGamal.

A signature is created using the private key from a public/private pair of keys. It is authenticated at the receiving end using the sender's public key.

11.4.1 Generating a signature

A digital signature for a given plaintext is generated with a secret key. Then the recipient can authenticate by using the corresponding public key to verify the signature. For efficiency we first hash the plaintext, then apply the *decryption* algorithm to the hash output.

In Java the `Signature` class is in the package `java.security`. We use a static factory method to generate a `Signature` object:

```
/** @param algorithm is a String representing the hash
 *  and public key algorithms.
 *  @return a Signature object
 */
 public static Signature getInstance (String algorithm)
```

The parameter specifies both the Hash algorithm and the public key algorithm, separated by "with". For example, to generate a signatura that uses the MD5 hash function and the RSA public key algorithm:

```
Signature sig;
sig = Signature.getInstance("MD5withRSA");
```

In order to sign a plaintext we will need the private key of a public/private key pair. We can use the key pair that we created in the section on public key encryption, above. All we need to do is extract the private key from that pair:

```
Key privateKey;
privateKey = keyPair.getPrivate();
```

Next we will initialize our `Signature` object for signing:

```
sig.init(privateKey);
```

To generate the signature, the plaintext must be in an array of bytes. Then we call the `update(byte [ ])` method to process the entire plaintext, after which we call the `sign()` method to return the signature as an array of bytes. To sign the message "Good Morning":

```
byte [ ] plaintext = "Good Morning".getBytes();
byte [ ] signature;
sig.update(plaintext);
signature = sig.sign();
```

This signature can be published or sent out to others, possibly as a String of numerics in the range [-128..127]:

11.4.2 Verifying a signed message

At the receiving end the recipient will wish to *verify* the true identity of the sender and the integrity of the message (verifying that the message has not been altered). This is done with a `Signature` object.

The recipient must first generate a `Signature` object as shown above. Then the that object must be initialized for verification. This is done using the sender's public key (obtained from a certificate):

11.4.3 Exercises

1. Define a class named `Sig`. It should have the following fields:
   - Two `Signature` fields, one for signing, and one for verifying.
   - A `KeyPair`
   - A `KeyPairGenerator`
   - A `PrivateKey`
   - A `PublicKey`

   (a) Use a constructor or a method to instantiate the fields.

   (b) Define a method to produce a signature for a given msg:

   ```
   /** @param msg A plaintext to be signed
    *  @return a digital signature as an array of bytes.
    */
   public byte [] signature (String msg)
   ```

2. Define a method to verify the signature:

   ```
   /** @param msg A plaintext received
     * @param sig  A signature for that plaintext
    *  @return true iff the plaintext is authentic
    */
   public boolean verify (byte [] sig, String msg)
   ```

3. Write a driver to test your signatures.

11.5 Certificates

As explained in chapter 8, digital certificates serve as a means of authentication, or identification of an individual. A certificate is like a government-issued photo-id, or driver's license; it is proof that you are who you claim to be. More generally, a digital certificate can apply to an organization, company, team, etc. In the Java documentation pages the general term for the subject of a certificate is *principal*.

In the `java.security.cert` package there is an abstract class, `Certificate` and a subclass, `X509Certificate` which is designed to work with X.509 certificates, a standard format established early this century. However, we cannot create objects of this class; we cannot even generate a fictitious certificate for educational purposes. In order to work with a certificate we must request an actual certificate from a certificate authority, such as Comodo.[6]

Once an actual certificate has been obtained from a certificate authority, the Java software allows us to extract information, such as a public key, from that certificate. This author has obtained an actual X.509 certificate, which is what we work with here. It is important to remember that in order to share a public key with others, a Certificate containing that public key must be used.

In order to instantiate a certificate we need to use a `CertificateFactory` which is instantiated with a static factory method, specifying the kind of certificate we want:

```
CertificateFactory factory = CertificateFactory.getInstance("X509");
```

Let's assume the certificate is in a file named `"seth.cer"`, which is in the same directory as our Java project. In order to open that file the certificate factory will need a `FileInputStream`:

```
FileInputStream inFile;
inFile = new FileInputStream("seth.cer");
```

Now we can instantiate the Certificate object:

```
Certificate cert;
cert = (X509Certicate) factory.generateCertificate (inFile);
```

The cast to `(X509Certificate)` is needed because the factory method returns a `Certificate` which is an abstract class.

Now that we have a `Certificate` object, we can extract information from it. The most important information is probably the public key:

```
Key pubKey;
pubKey = cert.getPublicKey();
```

Once we have the public key we can use it for encryption or signature verification as described earlier in this chapter.

Other uses of the certificate are:

- A method to obtain the type of certificate:
  ```
  String type = cert.getType();
  ```
  In our example the type would be `"X509"`.

- A method to return the entire certificate as a String:
  ```
  String prtCert = cert.toString();
  ```

[6] This makes sense; would it make sense to allow people to generate their own driver's licenses?

- A few methods to verify that the signature is authentic, i.e. that it was actually provided by a certificate authority. One such method is:
  `cert.verify(comodoKey, "Comodo");`
  This is a void method which uses Comodo's public key to verify the certificate; the certificate was signed with Comodo's private key. If the verify fails, an exception is thrown (`SignatureException`).

11.5.1 Exercises

1. (a) Obtain an actual certificate from a certificate authority, and store it in the same directory as your Java project.
   (b) Define a Java class, `Cert`, with two fields: A `CertificateFactory` and a `Certificate`.
   (c) Define a method to instantiate the certificate and print it to `stdout`.

Chapter 12

Cryptocurrency - Bitcoin

12.1 Introduction to digital currencies

[This chapter assumes the reader has seen section 4.3]

Perhaps we should begin by describing what is *not* a digital currency. You often use a credit card, debit card, or mobile phone to make a purchase, and those transactions make use of digital information on the cards, as well as the internet and other digital devices. However, the currency involved is the US$, or currency of some nation. This is not what we mean by a digital currency.

A digital currency:

- May have its own unit of value which may or may not be independent of some nation's legal currency.
- Should permit transactions among users of that particular digital currency.
- Is one of the following:
  - Centralized - works with a centralized server, such as a bank or private company. All transactions go through the server, and are verified by the server which is responsible for keeping a record of all clients and their holdings.
  - Distributed (decentralized) - works with no centralized server. Instead the clients are given incentive to verify transactions and prevent fraud. Bitcoin is an example of a distributed digital curency.
  - Serves as an intermediary for transactions. Facilitates transactions between clients, vendors, and banks. Examples are Apple-Pay, PayPal, and Venmo which is a subsidiary of PayPal.

Most viable currencies rely on the concept of *trust*. Currency originally took the form of something which had actual value, such as rare metals. As early as 1100 BC these metals were shaped into coins which were accepted and trusted. Paper money, also known as banknotes, or promissory notes, were issued by

banks or merchants as early as 600 BC. Later, governments issued these notes; they were trusted because they were backed by rare metals, such as gold or silver. From 1878 through 1964 the U.S. government issued *silver certificates*, paper money backed by actual silver bars. One could exchange a silver certificate for silver ingots. As the economy grew, the government's supply of silver was insufficient to back all the silver certificates which it had issued. Thus in 1964 the U.S. engraving office discontinued silver certificates and replaced them with *Federal Reserve Notes*, and that is what you see printed on paper money today. This paper money is trusted because it is backed by the federal government's own bank - the Federal Reserve Bank. Digital currencies such as Bitcoin are viable and useful because they also have come to be trusted.

In this chapter we will focus on Bitcoin which uses a distributed peer-to-peer network with no centralized server. In the Bitcoin peer-to-peer network each Bitcoin user, known as a *node*, is directly connected to a few (about 3-5) other nodes. These neighboring nodes need not be geographically close (and are usually not so). A diagram of a peer-to-peer network is shown as an undirected graph in Figure 12.1, in which the geographic locations are shown in brackets. This geographic information is not stored as part of the network. Note that joe (in chicago) and kai (in tokyo) are directly connected on the network, but are geographically far away from each other. Conversely, jill (in nyc) and mary (in bronx) are geographically close but are not directly connected on the network.

12.1.1 Bitcoin history

A technical paper titled *Bitcoin: A Peer-to-Peer Electronic Cash System* was distributed online to a group of cryptography researchers in October of 2008. The author of the paper was Satoshi Nakamoto. This paper described the workings and implementation of a non-centralized digital currency. The following January the software was released in open-source form. Today the source software can be found in a huge repository known as *Github*. Anyone who wishes to use the software can download it free from Github.

At this time Nakamoto created the first Bitcoin block, which is a block of transactions kept on the Bitcoin network (actually on several users' computers on the Internet). This block is known as the *genesis* block, the first block of transactions in a sequence of blocks known as the *blockchain*.

The software included a *mining* operation, enabling users to create more blocks of transactions, and add them to the blockchain.

Acccording to Wikipedia, the first known commercial transaction involving Bitcoin occurred in 2010 when Laszlo Hanyecz bought two Papa John's pizzas for 10,000 Bitcoins:

> ฿ 10,000

The most fascinating and mysterious part of this history is that no one seems to know who Satoshi Nakamoti is.[1] No one has ever met Satoshi Nakamoto in

[1]The New Yorker magazine dispatched an investigative reporter to track down Satoshi

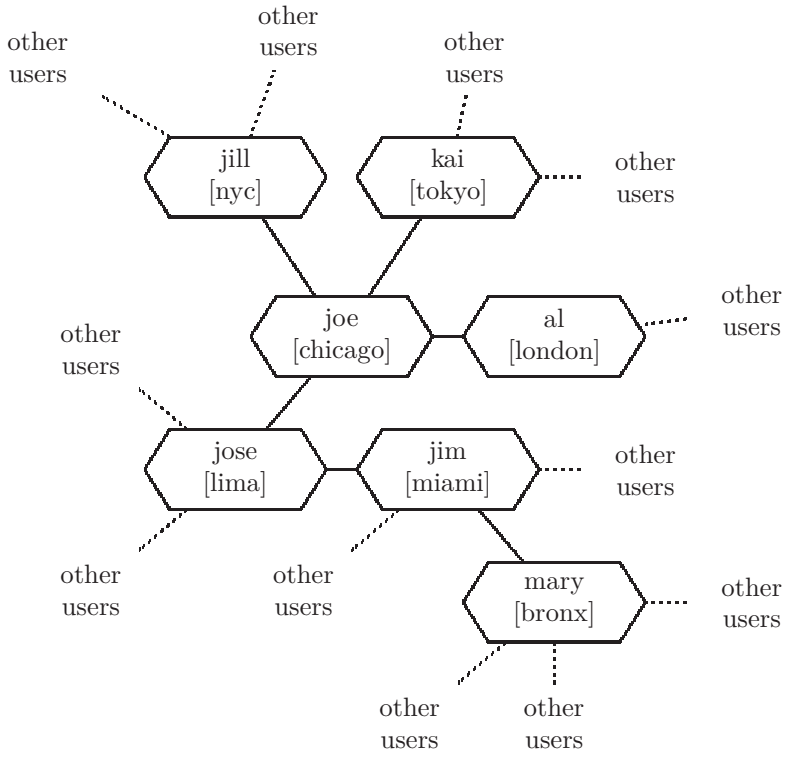


Figure 12.1: Diagram of users on a peer-to-peer distributed network. Users which are neighbors on the network may be geographically distant.

person. Satashi Nakamoto could be a pseudonym for some actual person, or it could be a pseudonym for a group of people. We may never know. But whomever he/she/they were, they produced some extremely interesting and reliable software. In 2011 Nakamoto abruptly left the Bitcoin development project. At this time Gavin Andresen took over as lead developer and established the *Bitcoin Foundation*, a consortium of developers for future improvements to Bitcoin.[2]

Today one Bitcoin is worth about US $91,000.[3]. Bitcoin can be bought and sold online at a crypto currency exchange, such as Coinbase. Many people speculate in digital currencies, which sometimes drives up the value. Speculation is the act of investing in a product in the hopes that others will buy it from them at a higher price. Speculators are often disappointed to learn that no one will pay a higher price, and they ultimately lose money on the investment. Cryptology consultant Bruce Schneier has publicly spoken against investment in cryptocurrencies, including Bitcoin, as an unwise investment.[4]

If the price of a Bitcoin fluctuates up and down, how are the currency exchanges, such as Coinbase, able to profit? The answer could lie in the fact that a Bitcoin owner must carefully protect the key(s) associated with their wallet. Often the keys are written on paper and stored in a safe place to guard against loss in the case of a computer failure. Sometimes the keys are misplaced or permanently lost. The owner can never recover the loss, and the currency exchange will never have to relinqish the money which was used to purchase the lost Bitcoin. Some fraction of the exchange's holdings can be invested for profit, without risk that the client(s) will cash in their crypto holdings.[5]

12.1.2 Exercises

1. What is the current, approximate, value of a Bitcoin in U.S. dollars?

2. (a) Name at least three Bitcoin exchanges where Bitcoin can be purchased.

   (b) Do they charge a fee?

   (c) If so, how much is the fee?

3. Explain the usage of the word *trust* with respect to digital currencies such as Bitcoin.

4. Explain what is meant by *speculation* with respect to Bitcoin, or investment in general.

Nakamoto. The reporter traveled the world and found several people who falsely claimed to be Satoshi Nakamote, but the reporter was not able to locate the actual creator of Bitcoin.

[2]Since it is open source, anyone can offer fixes and enhancements to Bitcoin. A *BIP*, or Bitcoin Improvement Proposal, if truly an enhancement, can be adopted by all Bitcoin users.

[3]As of January 2025

[4]See `https://www.schneier.com/cryptogram`.

[5]This is similar to the financial strategy employed by a bank or savings and loan company. The percentage of their holdings which must remain liquid is carefully regulated by the federal government.

12.2 Keys, addresses, wallets, and transactions

12.2.1 Types of nodes

When a new user decides to download the Bitcoin software, they must decide which features they require. The features available are:

- The capability of spending and receiving Bitcoin, i.e. transaction processing with software known as a *wallet*
- The capability of verifying transactions
- The capability of forwarding information to neighboring users (i.e. peers). All nodes have this function.
- The capability of storing the ledger of all Bitcoin transactions known as the *blockchain*
- The capability of adding new blocks to the blockchain, and creating new Bitcoins, known as *mining*

We can therefore classify Bitcoin nodes as follows:

- Full nodes - Have all the features shown above. They store the entire blockchain and can assist the network in reaching consensus (described below).
- Standard non-mining nodes - Can initiate and verify transactions, can forward transactions to its neighbors, and store the entire blockchain. They can assist the network in reaching consensus (described below), but they cannot mine new Bitcoins.
- Mining nodes - Can verify transactions and mine new Bitcoins, but do not have a wallet, which is needed to spend or receive Bitcoin. They store the entire blockchain and can assist the network in reaching consensus (described below).
- Lightweight nodes - Are typically installed on a mobile device, such as a phone. A Lightweight node can forward information to neighboring nodes, and can process transactions. It does not store the blockchain.

The possible functions in a Bitcoin node are shown in Figure 12.2.

12.2.2 Bitcoin keys

Bitcoin relies on public key cryptography for security. It uses the discrete elliptic curve form of public keys, as described in chapter 4.[6] Elliptic curve cryptography is probably the most secure form of public key cryptography; it is ensuring the

[6] The details of elliptic curve cryptography are not needed to understand Bitcoin. It should suffice if the reader understands the general concepts of public key encryption and signatures.

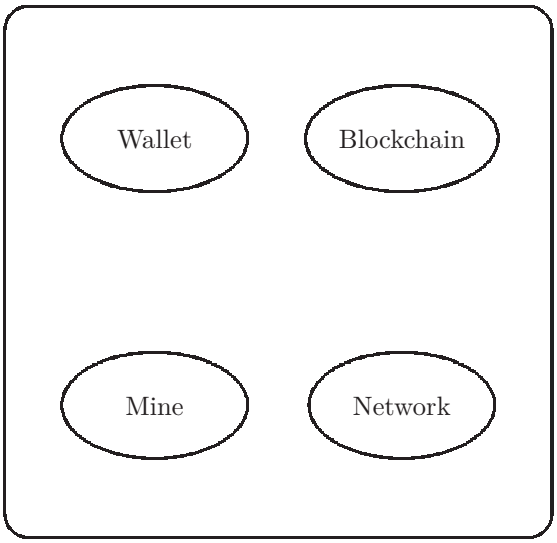


Figure 12.2: Possible functions in a Bitcoin node

security of billions of dollars worth of Bitcoin. When a new user downloads the Bitcoin software, they are assigned a unique public/secret key-pair. The public and secret keys in the pair are mathematically related, as described in chapter 6: The public key can be used by others to encyrpt information for the new user, and to verify authenticity of received information. The secret key can be used to decrypt information received from others, and to sign information for authenticity.

When a new user downloads the Bitcoin software, they are assigned a unique secret key. The elliptic curve software can then be used to find the unique public key associated with that secret key. This key pair will then be used to verify transactions involving the user, as described in the section on digital signatures in chapter 8.

Generating a public key

When a new user downloads the Bitcoin software, the software performs an initialization protocol:[7][8]

1. A 256-bit random key, K_{sec} is automatically assigned to that user. This is the user's *secret key*.

2. A corresponding public key, K_{pub}, is derived from the secret key using discrete elliptic curve cryptography:

[7] This section assumes an understanding of discrete elliptic curves, as described in chapter 4, and may be omitted without loss of continuity.

[8] The algorithm used by Bitcoin is a standard known as `secp256K1`, part of a collection of standard algorithms from `secg` an international organization for cryptographic standards.

(a) The elliptic curve is given by the parameters:
$y^2 = x^3 + ax + b \pmod p$
$a = 0$
$b = 7$
p is the 256-bit two's complement representation of -977
$p = fffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f_{16}$

(b) A fixed point on the elliptic curve, G, called the *generator*, with x coordinate:
$G_x =$ 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
and a prefix byte to indicate the sign of the y coordinate, 02 = positive, 03 = negative.

(c) The user's public key, K_{pub}, is calculated as
$K_{pub} = K_{sec} * G$
where the * represents multiplication on the discrete elliptic curve.[9]

It is important to note that this multiplication can be done in polynomial time, but the inverse operation, finding the secret key from the public key, requires *exponential* time, and therein lies the security of Bitcoin.

12.2.3 Bitcoin addresses

Once the new user has a key pair, the public key, K_{pub}, can be used to generate an address for that user. Bitcoin addresses are not the same as other kinds of addresses; a memory address is used by the computer's CPU to access one specific memory location. A Bitcoin address is information which is useful only to the owner of the address. When Alice sends information, such as a transaction encrypted with Bob's public key, to Bob, she puts the encrypted transaction, along with Bob's address out on the network, where it propagates to all Bitcoin users currently on the network. Only Bob is capable of using that encrypted transaction, because he is the only user holding his private key. All other users disregard the transaction.

The address is obtained by providing the public key as input to a hash function.[10] The address is then the output of the hash function. Bitcoin uses a composition of two standard hash functions: RIPEMD160[11] and SHA256, to produce a 160-bit address as shown below:

[9] Recall from chapter 4 that multiplication of a point on an elliptic curve is implemented as repeated additions of the multiplicand, G in this case.

[10] See chapter 5 for a description of hash functions.

[11] RIPEMD is the only 3-level acronym known to this author. MD is Message Digest, and
RIPE is RACE Integrity Primitives Evaluation, and
RACE is RDMA-Conscious Extendible hashing, and
RDMA is Remote Direct Memory Access.

$address = RIPEMD160(SHA256(K_{pub}))$[12]

The address will thus always be 160 bits, and may be displayed in base-58 encoding format. No two users will have the same address, but an individual may have more than one address because a user may request more than one secret key. A diagram showing the generation of a Bitcoin address is shown in Figure 12.3.

12.2.4 Wallets

In order to spend Bitcoin, or receive Bitcoin as compensation, a user node must have software known as a *wallet*. The wallet is essentially a user interface for Bitcoin transactions. The wallet does not store Bitcoins; instead it stores keys, each of which represents a transaction output, and each transaction output is for a specific quantity of Bitcoins (actually Satoshis). The user can use the wallet to:

- Determine how much Bitcoin is available to be spent. This is the sum of all the transaction outputs, represented by keys, in the wallet.
- Create a transaction of some amount to another user address.
- Receive Bitcoin from a transaction created by another user, adding the transaction output to this wallet.

There are at least two kinds of wallets, the original *nondeterministic* wallet, and *determistic* wallets were added later.

Nondeterministic wallets

Nondeterministic wallets also known as *random* wallets were included in the original version of Bitcoin, known as *Bitcoin core*. Each wallet contains a collection of randomly generated key-pairs. The wallet software generates 100 random secret keys, calculates their corresponding public keys, producing 100 random key-pairs. A key-pair is used for a transaction and then discarded. New random key-pairs are generated automatically as needed. Security is ensured by the fact that keys are not re-used in a multitude of transactions. One disadvantage of nondeterministic wallets is that is difficult to store backup copies of so many keys.

Deterministic wallets

Deterministic wallets are also knownd as *seeded* wallets. A deterministic wallet may contain several secret keys, but they are all derived from a single integer known as the *seed*. The seed is a random bit string, which is used as input, along with other information, to a hash function. The result of the hash function is a derived secret key.

[12] MD stands for Message Digest and SHA256 is Standard Hash Algorithm, with a 256-bit result.

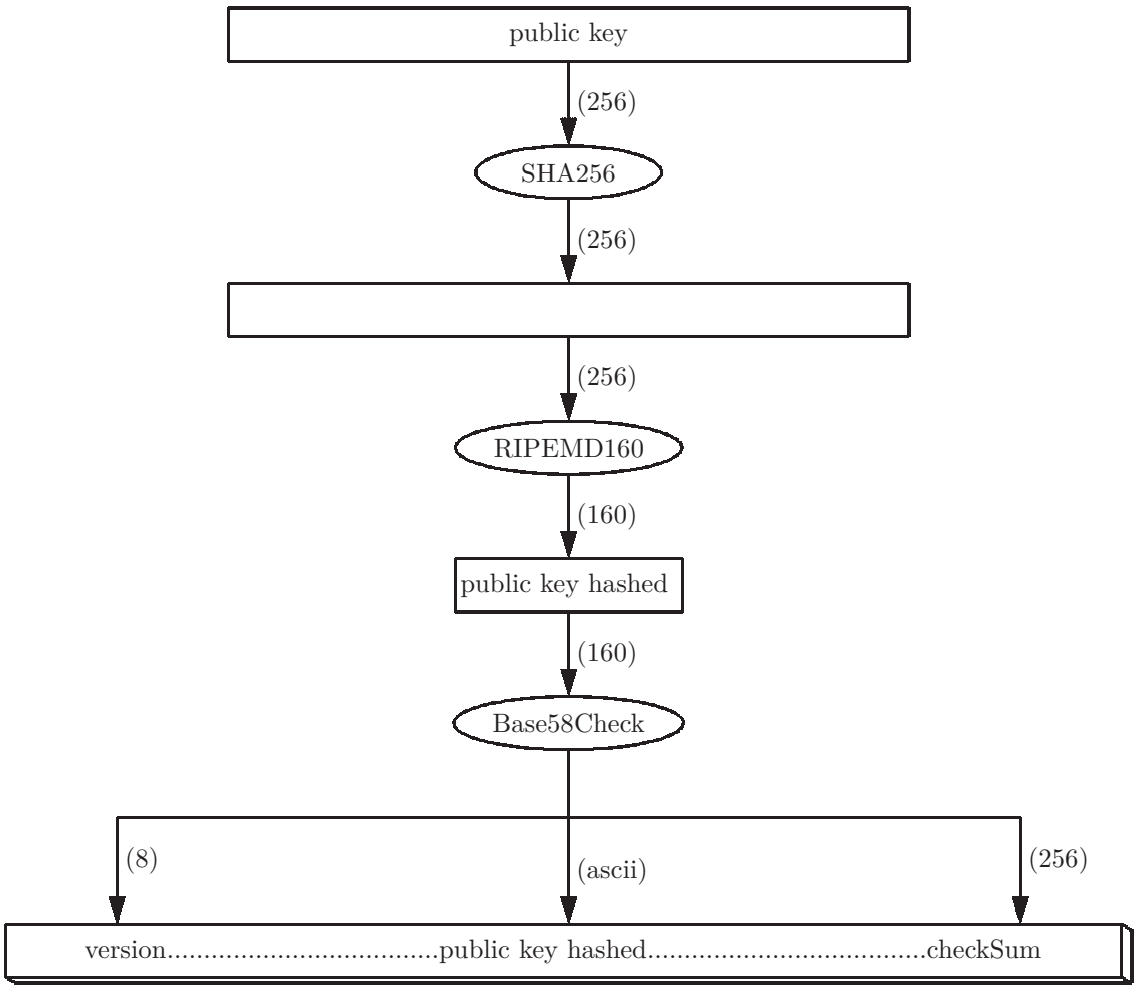


Figure 12.3: Diagram of Bitcoin address generation from a private key of any size. The final result is shown in base 58 encoding, with version and checksum.

A tree structure of keys is then derived from the seed as follows:

1. The seed is used as input to a hash function (SHA-512) to derive a secret key, K_0
2. A function to derive a child key from a parent key, CKD,[13] added to the Bitcoin software with BIP-32,[14] is used to produce a child key from the parent key. This function also produces a chain code integer which is used when the child generates a child, i.e. a grandchild. The arguments to this function are the parent key, a *chain code* integer, and an index counter. This function uses SHA-512 to produce a child key and a new chain code value.
   (a) The function can be applied several times, while incrementing the counter, to produce several child keys:
   $CKD(K_0, chain_0, 0) \rightarrow \{K_{1,0}, chain_{1,0}\}$
   $CKD(K_0, chain_0, 1) \rightarrow \{K_{1,1}, chain_{1,1}\}$
   $CKD(K_0, chain_0, 2) \rightarrow \{K_{1,2}, chain_{1,2}\}$
   (b) The same function can then be used to generate keys from the children, i.e. grandchildren:
   $CKD(K_{1,0}, chain_{1,0}, 0) \rightarrow \{K_{2,0,0}, chain_{2,0,0}\}$
   $CKD(K_{1,1}, chain_{1,1}, 0) \rightarrow \{K_{2,1,0}, chain_{2,1,0}\}$

   New keys can be produced to any number of generations. Recall from chapter 5 that a small change in the input to a good hash function will produce a radically different output, and this is why the index counter is used. In the formulas shown above, a key may have several subscripts. The first subscript shows the number of generations the key is from the original seed. The other subscripts show a path to the key. A tree diagram of the keys is shown in Figure 12.4.

   The main advantage of deterministic wallets is that to backup a wallet the user needs to save only the seed, because all the other keys can be generated from the seed.

12.2.5 Transactions

A Bitcoin *transaction* is data which specifies the transferral of some quantity of Bitcoin from one user (address) to another user (address). When this data is put onto the network, all nodes can see it and forward it to their neighbors. Standard nodes can verify the transaction using the initiator's public key, in addition to forwarding it to neighbors. Only the receiving node will be able to use its secret key to add the transaction output to its wallet.

[13] Child Key Derivation

[14] BIP is Bitcoin Improvement Proposal. These BIPs allow Bitcoin to evolve to a more secure, or more usable, package.

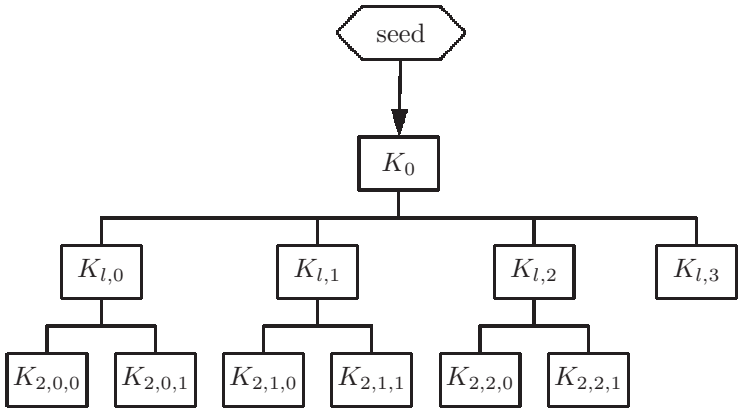


Figure 12.4: Diagram of the keys derived from a seed in a deterministic wallet

When a transaction is created, an entire output from a previous transaction must be utilized, or spent. The output can then be distributed to multiple user addresses. A transaction output is often written as `UTXO`, or UnspentTransactionOutput. We describe a few different kinds of transactions with diagrams:

- The simplest transaction is called a *basic* transaction. It involves using a single output from a previous transaction, to be sent to a single user. If the output from the previous transaction is more than sufficient, output from the current transaction is split into a pair of outputs, one of which is sent back to the user generating the transaction. This is like producing "change" of $2.75 for a 10 dollar bill, when making a purchase for $7.25.[15]

  A diagram of a basic transaction is shown in Figure 12.5 in which Alice wishes to pay Bob 10 Satoshis. The unspent transaction outputs in her wallet are 5, 30, and 40. Her wallet decides to use the 30, sending 10 to Bob and the remaining 20 back to her own wallet as "change".

- An *aggregating* transaction is one in which multiple outputs from previous transactions are combined into one large amount in the current transaction. This is useful when none of the unspent transaction amounts in the wallet are sufficient to cover the amount being spent.

  A diagram of an aggregating transaction is shown in Figure 12.6 in which Alice wishes to pay Bob 45 Satoshis. The unspent transaction outputs in her wallet are 12, 20, and 40. None of these are sufficient, so the wallet chooses to aggregate the 12 and 40 outputs, for a total of 52 Satoshis. Of these, 45 are included in the transaction output to Bob, and the remaining 7 are sent back to Alice, as "change".

- A third form of transaction is the *distributing* transaction, in which the output is distributed among two or more recipients.

[15] This is not completely accurate. Each transaction must include a small fee, in addition to the outputs sent to the recipient.

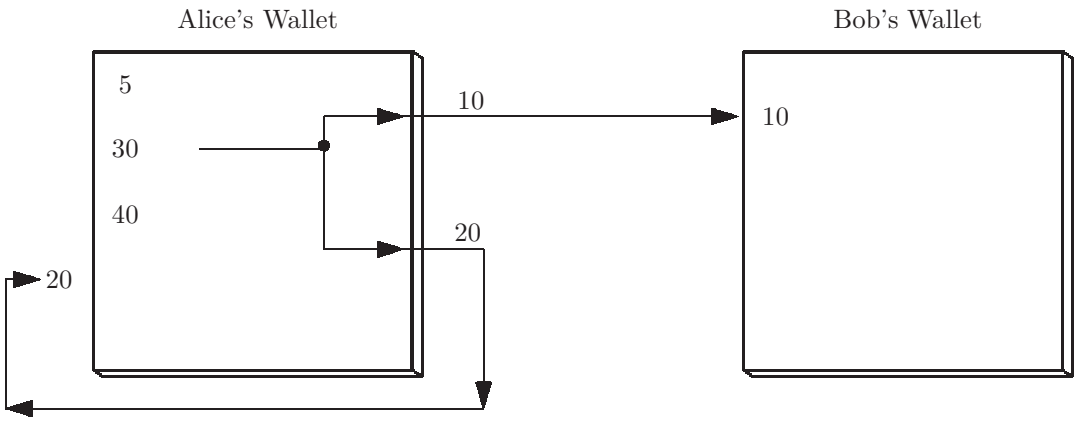


Figure 12.5: Diagram of a basic Bitcoin transaction. Alice needs to pay Bob 10 satoshis

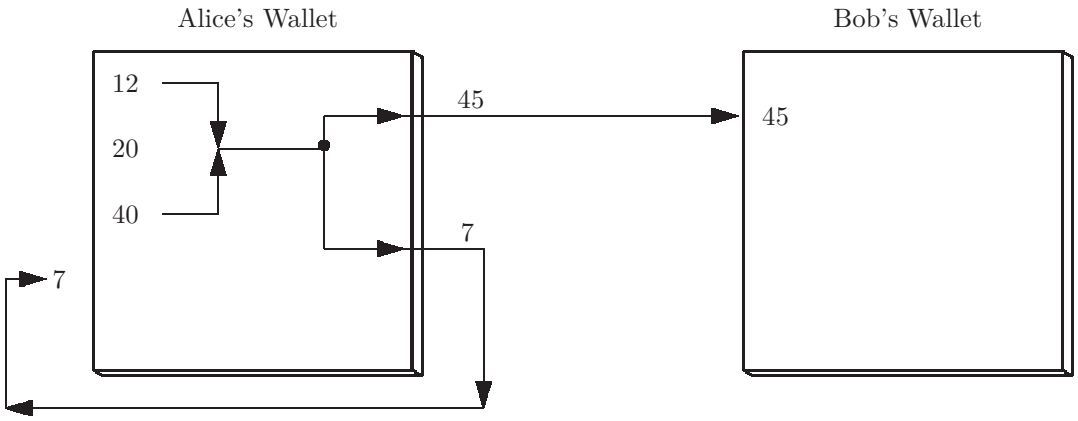


Figure 12.6: Diagram of an aggregating Bitcoin transaction. Alice needs to pay Bob 45 satoshis. None of the unspent outputs in her wallet are sufficient.

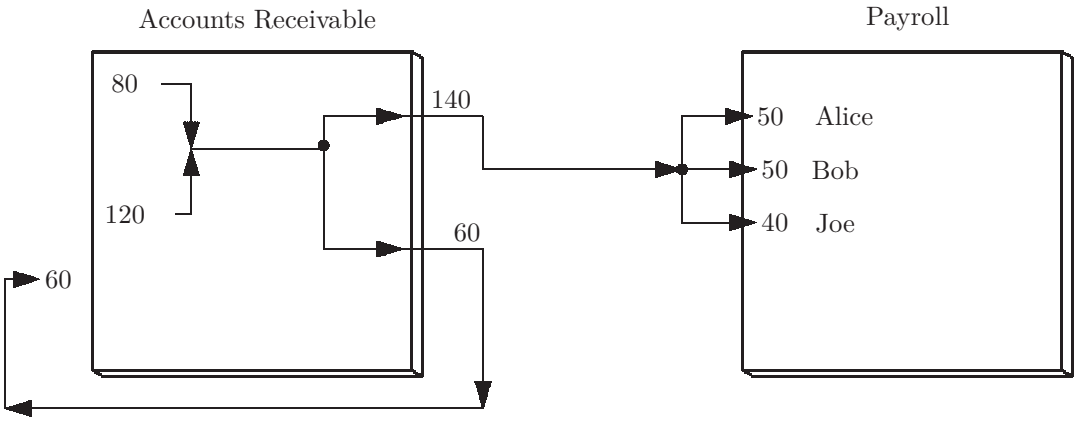


Figure 12.7: Diagram of a distributing Bitcoin transaction. Accounts Receivable needs to pay Alice (50), Bob (50), and Joe (40 satoshis).

A diagram of a distributing transaction is shown in Figure 12.7 in which the accounts receivable department wishes to pay three employees - Alice (50), Bob(50), and Joe (40 Satoshis). The accounts receivable department combines the unspent outputs of 80 and 120, for a total of 200 Satoshis. This is then distributed among the three recipients, with 60 Satoshis sent back to the accounting department as "change".

Transaction fees

Included with each transaction is a small, optional, transaction fee (not shown in Figures 12.5 through 12.7). The transaction fee has two purposes:

- The fee is awarded to the miner who includes the transaction in a new block of transactions, in addition to the normal mining award (see mining below). Thus miners have incentive to include the transaction in a block, which will verify the transaction, and finalize it in the blockchain.

- The transaction fees make it economically infeasible for an attacker to flood the network with false or fraudulent transactions.

The Bitcoin user's wallet has options which can be set to raise or lower transaction fees. In cases where a user wishes a transaction to be processed quickly, the fee can be increased. There are also third-party fee estimation services which can calculate a fee for the user.

The fee itself is not stored as part of the transaction; rather, when a new block of transactions is mined, the award is calculated as the sum of the outputs subtracted from the sum of the inputs over all transactions in the block:

$FeeAward = sum(inputs) - sum(outputs)$

12.2.6 Exercises

1. Name the four Bitcoin node classifications.
2. Which of the three public key algorithms discussed in chapter 6 is used in Bitcoin?
3. How many values must be saved to retrieve all the keys in a deterministic Bitcoin wallet?
4. Mary's wallet has three UTXO's in the amounts of 20, 35, and 50 Satoshis. She wishes to pay Mike 60 Satoshis. Show a diagram similar to Figure 12.6 for this transaction.
5. What are the two purposes of a Bitcoin transaction fee?

12.3 Transaction verification

Bitcoin transactions can be verified for authenticity in a few different ways.

- Any Bitcoin node with a wallet can verify transactions before relaying them through the Bitcoin network.
- A mining node can verify all the transactions in a block when forming a new block for the blockchain.

Here we cover only the verification process which takes place in a wallet. Verification in mining nodes is covered in the section on mining.

Scripting language

Transactions are verified using a stack-based *scripting* language. Briefyly, the scripting language is capable of:

- Pushing a given data value onto the top of a *stack*, which is a last-in, first-out list
- Removing the top value from the stack, usually called a *pop* operation
- Performing an operation on zero or more data values which are on top of the stack

Example scripting language

As an example of a scripting language operating on a stack, we will make use of two operations:

- OP_ADD - Pop two values from the top of the stack, add them, and push the result on top of the stack.

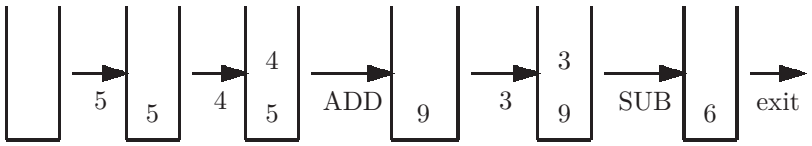


Figure 12.8: The postfix expression 5 4 + 3 - is evaluated with a stack.

- OP_SUB - Pop two values from the top of the stack, subtract the upper value from the lower value, and push the result on top of the stack.

An example of a script could be:
`5 4 ADD 3 SUB`[16]
As this script is scanned from left to right, the numbers are pushed onto the stack. As each operation is encountered it is executed as described above, and shown in Figure 12.8.

Bitcoin scripting language

The Bitcoin scripting language contains the following operations, (and others not shown here):

- OP_DUP - Duplicate the top value on the stack. This makes a copy of the top value and pushes it onto the stack.
- OP_HASH160 - This is actually a composition of two standard hash algorithms (see chapter 5):
  `HASH160(x) = RIPEMD(SHA256(x))`
  This stack operation will pop the top value from the stack, apply HASH160 to it, and push the result (a string of 160 bits) onto the stack.
- OP_EQUALVERIFY - This is actually two stack operations:
  - OP_EQUAL - Pop two values from the stack. Push the value TRUE if they are equal; otherwise push the value FALSE.
  - OP_VERIFY - If the top value on the stack is FALSE, verification fails, and the script terminates. Otherwise it pops the TRUE value from the stack.
- OP_CHECKSIG - Pop the public key and the signature (of a UTXO) from the stack. Apply the public key to the signature; the result should be equal to the UTXO being verified. If so, push True onto the stack (the transaction is verified and passed on to neighboring nodes); else push False (the transaction is not verified). The signature verification process was described in chapter 6.

[16] Computer science students understand that this postfix expression is equivalent to the infix expression (5+4)-3.

Scripts to verify a Bitcoin transaction

When a transaction is sent out on the Bitcoin network, each node with a wallet can verify it, and pass it on to neighboring nodes if verification succeeds. This is done with a pair of scripts, a *locking script* and an *unlocking script*. Recall from chapter 6 that a signature is formed (by the sender) by applying the decryption algorithm to the data being signed. It is then verified (by the receiver) by applying the encryption algorithm to the signature and comparing the result with the received data. In the case of Bitcoin, the verification process is done with discrete elliptic curve algorithms, as described in chapters 4 and 6.

This is best explained with an example. Here we assume that Alice's wallet is generating a transaction which will send (part of) one of her unspent transaction outputs (UTXOs) to Bob. The locking script for this transaction would be:

```
OP_DUP OP_HASH160 <Hash of Bob's public key> OP_EQUALVERIFY OP_CHECKSIG
```

The unlocking script would be:

```
<signature of the UTXO> <Bob's public key>
```

The unlocking script is concatenated to the locking script, producing a single script for verification:

```
   <signature of the UTXO> <Bob's public key>
OP_DUP OP_HASH160 <Hash of Bob's public key> OP_EQUALVERIFY OP_CHECKSIG
```

When the script terminates the transaction is verified, and relayed to neighboring nodes, if and only if the value TRUE is the only value on the stack.[17]

Figure 12.9 shows how this script works with a stack, with the following abbreviations, for a more concise diagram:

- B_K is Bob's public key
- $h(x)$ is RIPEMD(SHA256(x))
- HASH is the stack operation OP_HASH160 described above
- $sig_p(x)$ is the digital signature of x formed by user p
- DUP is the stack OP_DUP operation described above
- $= VERIFY$ is the stack operation OP_EQUALVERIFY described above
- CHK is the stack operation OP_CHECKSIG described above

12.3.1 Exercises

1. Name two ways that a Bitcoin transaction is verified for authenticity.

[17] To be more accurate, it is the hash of selected fields in the UTXO which is being signed.

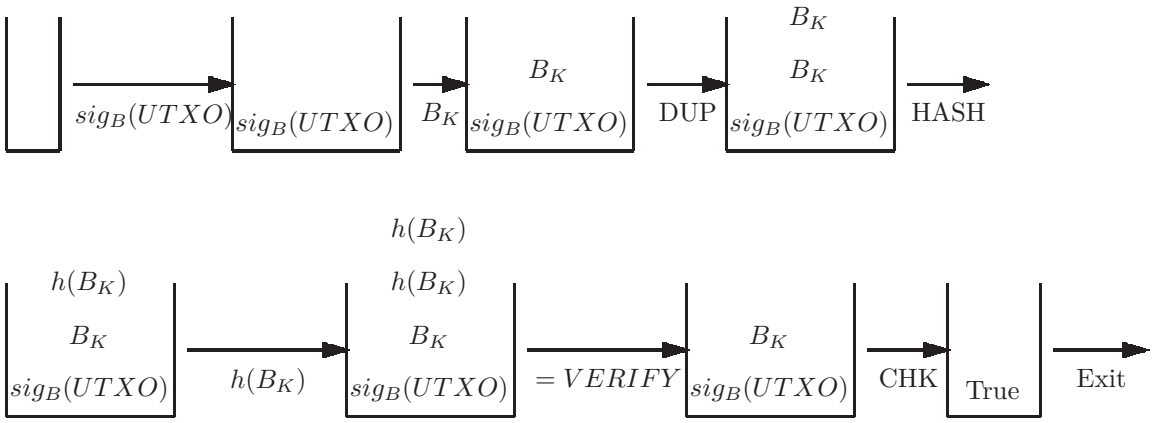


Figure 12.9: Verification of a transaction which is sending an UnspentTransactionOutput (UTXO) to Bob.

12.4 Blocks of transactions on the Bitcoin network

Completed transactions are kept in a "pool" of transactions on the Bitcoin network; each node (except for lightweight nodes) stores these transactions. These transactions are collected into a structure known as a *block* by mining nodes. In the next section we describe how a mining node adds a block of transactions to previous blocks, in a *blockchain* structure. Here we describe the details of the block.

The blockchain is usually visualized as a vertical stack of blocks. The most recently added block is at the top, and the very first block created, block 0 (also called the *genesis* block) is at the bottom. Each block stores a *hash pointer* to the previous block in the blockchain. The previous block is also called the *parent* block as shown in Figure 12.10. A hash pointer is a pointer (i.e. memory address) combined with a hash value of the structure to which it points.

Each block contains fields defining the block's size, a *header* field, described below, the number of transactions in the block, and the transactions. The number of transactions in a block can vary significantly, but is typically about 2,000. The block structure is shown in Figure 12.11.

The block header contains fields defining the Bitcoin version number which was used when the block was assembled, a hash of the previous, or parent block, a Merkle root, described below, a timestamp, a difficulty target, described in the section on mining, and a *nonce* field, also described in the section on mining. The header structure is shown in Figure 12.12.

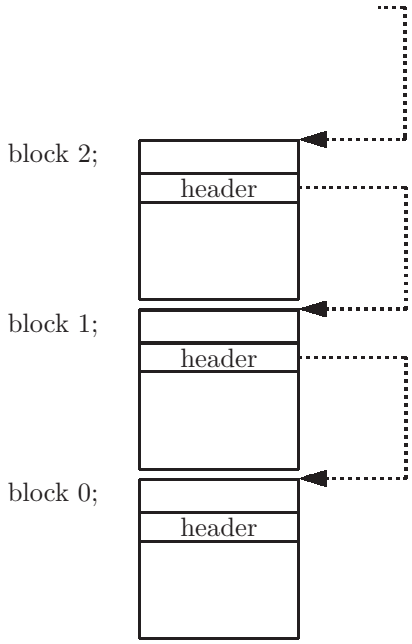


Figure 12.10: Diagram of the Bitcoin blockchain. The header of each block stores a fingerprint of its parent's block.

Size	Field	Description
4 bytes	Block size	Size of this block, in bytes
80 bytes	Block header	See Figure12.12
1-9 (var)	Transaction counter	Number of transactions in this block
Variable	Transactions	Transactions in contiguous memory

Figure 12.11: Contents of a Bitcoin block

Size	Field	Description
4 bytes	Version	Release version of Bitcoin
32 bytes	Parent hash	Hash of the previous block header
32 bytes	Merkle root	Hash of the root of the Merkle tree of this block's transactions
4 bytes	Timestamp	Creation time of this block (in seconds)
4 bytes	Difficulty	Proof of work difficulty level for a successful miner
4 bytes	Nonce	Random bits, used by the miner

Figure 12.12: Contents of a Bitcoin block header

12.4.1 Exercises

1. What are the four fields in a Bitcoin block?
2. What are the six fields in a Bitcoin block header?
3. What is the purpose of the nonce field in a block header?

12.5 Mining, consensus, and the blockchain

As we mentioned earlier, new Bitcoins are created by nodes which have the capability of *mining* a new Bitcoin. Each mining node attempts to accomplish this by assembling transactions into a *block* of transactions. The mining nodes then compete with each other to solve a computational problem; the node that wins the race for a solution is rewarded with 3.125 Bitcoins[18] and the sum of the transaction fees included in the block. The successful miner adds an extra transaction to the block called the *coinbase* transaction, which is the miner's reward (3.125 Bitcoins). The mined block is then added to the blockchain, which is a complete ledger of all transactions ever recorded, and distributed throughout the peer-to-peer network. All other mining nodes disband the blocks they had been attempting to mine, returning all transactions to the *transaction pool*, and must then reassemble new blocks and then race to solve another computational problem.

12.5.1 Proof of work

What is the computational problem that miners race to solve? It involves a standard hash algorithm, currently SHA-256. Recall from Figure 12.12 that one of the fields in the block is named *nonce*. The word 'nonce' denotes something which is nonsense, or has no real meaning. The nonce field may contain any random string of 32 bits. After each block is successfully mined, the Bitcoin software broadcasts a 256-bit *difficulty level* for the next block. Each miner fills in the nonce field with 32 bits, then uses the entire block as input to SHA-256. If the hash output is smaller than a max value representing the difficulty level, the miner succeeds. If not, the miner repeats the process with another nonce value. A miner may need to repeat this billions of times before succeeding; it is referred to as a brute force search. Recall from chapter 5 that changing even 1 bit of the input to a good hash function causes drastically different output. Figure 12.13 shows a simplified example in which the max value is $0000\ 0000\ 1000\ 0000_{16}$. After 12 attempts the miner succeeds in finding a nonce field, $309b\ 6010_{16}$, which produces a hash output, $0000\ 0000\ 0f31\ e298_{16}$, which is smaller than the max value. In this example the hash output is only 64 bits (16 hex digits). In Bitcoin, the SHA-256 algorithm actually produces a 256-bit result.

[18] This reward was originally 50 Bitcoins, but it is halved every four years. In July, 2029, the reward becomes 1.5625 Bitcoins.

nonce	hash(block)	
$24e3\ c069_{16}$	$90e2\ ba31\ 0ff3\ 2cb0_{16}$	failure
$00f3\ 50fd_{16}$	$e7b3\ 5921\ ba33\ a084_{16}$	failure
$ff31\ 7483_{16}$	$ab89\ 1a3a\ 0397\ 89d3_{16}$	failure
$5b36\ 0037_{16}$	$000b\ 9843\ 77a3\ 7872_{16}$	failure
$4099\ 9f32_{16}$	$e400\ 8720\ b01c\ 9003_{16}$	failure
$3074\ 9b32_{16}$	$88f1\ 7bb2\ 3937\ 3a59_{16}$	failure
$3074\ 9b32_{16}$	$0000\ 0000\ 1000\ 0001_{16}$	failure
$2105\ 3846_{16}$	$4baf\ 5012\ 4de0\ 6810_{16}$	failure
$b892\ 9037_{16}$	$402c\ 0937\ 0b43\ fe9c_{16}$	failure
$309b\ 6010_{16}$	$0000\ 0000\ 0f31\ e298_{16}$	success!

Figure 12.13: A simple example of proof of work. The max value is $0000\ 0000\ 1000\ 0000_{16}$. After 10 attempts the miner succeeds in producing a hash output smaller than the max value.

In order to avoid hashing duplicate nonce values, most miners will simply take a starting nonce value, and increment by 1 after each failed attempt. However, the nonce field in the block header is only 32 bits, and it is not unusual for a miner to try all 2^{32} possible nonce values without succeeding in finding a hash value smaller than the difficulty level. In this case, the miner can adjust a nonce field in the coinbase transaction and continue hashing until succeeding.

The Bitcoin software is designed to allow for a successful mining once every 10 minutes. If a miner succeeds in less than 10 minutes, the difficulty level is increased (i.e. the max value is decreased). If a miner succeeds after more than 10 minutes, the diffuculty level is decreased (i.e. the max value is increased). In this way the Bitcoin software ensures that miners succeed in mining new Bitcoins once every 10 minutes over the long term.[19]

The phrase usually used to describe this process is *proof of work*. Note that other nodes can verify that the miner is not cheating by simply hashing the block and noting that the hash output is indeed smaller than the max value.

In recent years the value of a Bitcoin has increased remarkably. Consequently there is keen competition among miners to win the race for a new Bitcoin. Many miners use large arrays of GPU's[20] to be first to solve the mining problem. These processing units consume much electric power, and produce substantial heat energy; [21] consequently miners often need to invest in cooling machinery as well.[22]

Other crypto-currencies may use different mining algorithms, such as proof-

[19]To be more accurate, the difficulty level is adjusted after every 2,016 blocks have been mined, which is exactly 2 weeks if there is a Bitcoin mined every 10 minutes.

[20]A GPU is a Graphics Processing Unit, which is a special purpose CPU, originally developed for complex graphics displays. They make extensive use of parallelism to attain very high speeds.

[21]Most semiconductor technology will fail at high temperatures.

[22]Many miners are located in cool climates, such as Canada and Scandanavia, for this reason.

Figure 12.14: Diagram of two miners on the Bitcoin network, A and B. Each miner mines a new block, b1 and b2, at about the same time, creating two different versions of the blockchain. Each miner shares its blockchain with its neighbor(s). The * represents all prior blocks in the blockchain, beginning with block 0, the genesis block.

of-stake.[23] Some authors refer to the various mining algorithms as 'consensus', giving us two slightly different meanings for the word 'consensus'.

Mining pools

If an individual with only a few CPUs attempts to mine a Bitcoin, he/she will not likely succeed, because a wealthy miner with a large array of GPUs will surely solve the problem first. However, many miners can cooperate in the mining process, forming a so-called *mining pool*. In a mining pool each miner is assigned a range of nonce values to attempt to solve the problem. When any member of the pool is successful, the rewards are distributed among the miners in the pool, on a pro-rated basis. If a miner has tried 1% of the total nonce value attempts, that miner receives 1% of the rewards.

12.5.2 Blockchain consensus on the network

When a new block is created, the successful miner adds it to the blockchain and broadcasts the updated blockchain to its neighbors. They, in turn, broadcast the new blockchain to their neighbors, and so on, until the updated blockchain has reached all nodes on the network.

It normally takes a few seconds for all the nodes on the network to receive the updated blockchain (they discard the old version in favor of the new version). However, it is possible that two miners successfully mine a new block at about the same time, each of them broadcasting their version of the new blockchain before receiving notice of the other miner's new blockchain. This is known as a blockchain *fork*, and it typically occurs about once per week. We cannot have two different versions of the blockchain coexisting on the network, so the network needs to choose one of these versions of the blockchain and discard

[23] See chapter 13

the other. This decision process is known as blockchain *consensus*. The two different versions of the blockchain will eventually reach a mining node which is (temporarily) storing both versions of the blockchain, and which mines a third block. It then adds the third block to one of its blockchains and discards the other blockchain. It broadcasts this larger blockchain, and when other nodes receive it they all adopt it as the valid version because it is longer, and they discard the shorter versions.

We describe the consensus process with a simple example. In this example the nodes labeled A, B, and C are all mining nodes. The unlabeled nodes are nodes which also store the blockchain. The following sequential events describe the consensus process:

1. Miners A and B succeed in mining a new block at about the same time. Miner A adds the block b1 to its version of the blockchain, and Miner B adds the block b2 to its version of the blockchain. They each forward their own version of the blockchain to their neighbors on the network. There are now two different versions of the blockchain, with equal size. This is shown in Figure 12.14, in which the asterisk represents all the preexisting blocks on the blockchain.

2. Eventually every node will be storing both versions of the blockchain, indicating that a fork has taken place. Mining node C then succeeds in mining a new block (b3 in Figure 12.15) It adds the new block to one of the blockchains (on top of block b1), discarding the other blockchain (storing block b2). Note that the blockchain with block b3 at the top is larger than the blockchain with block b2 at the top. This is shown in Figure 12.15

3. Mining node C then broadcasts this larger version of the blockchain to the network. When other nodes see multiple versions of the blockchan with different sizes, they also discard the smaller versions in favor of the larger version (with blocks b1 and b3). The network has now reached consensus. All nodes are storing identical copies of the blockchain. This is shown in Figure 12.16

Note that the transactions in the discarded block(s) are returned to the pool of transactions so that they can be included in a block by some miner in the future. Also, mining nodes A and C are rewarded for successfully mining a block, and node B is not rewarded.

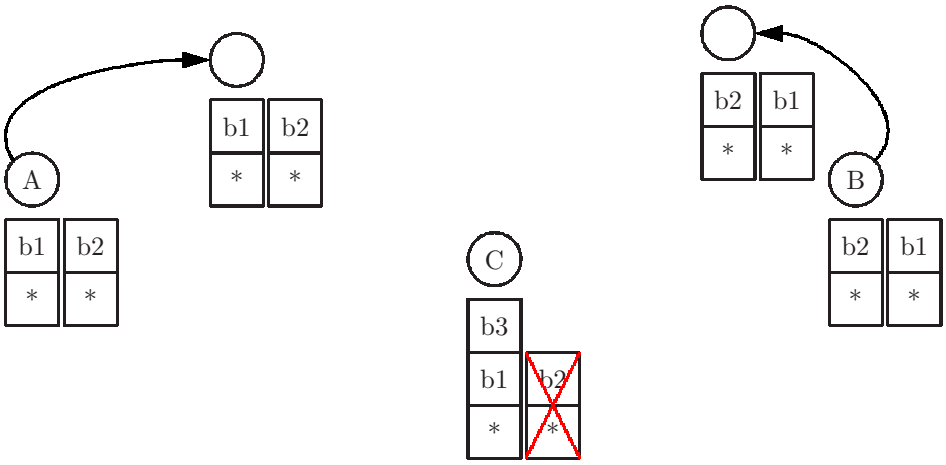


Figure 12.15: A third miner, C, successfully mines a new block, b3, and adds it to one of the blockchains it is storing. Miner C then discards the smaller blockchain.

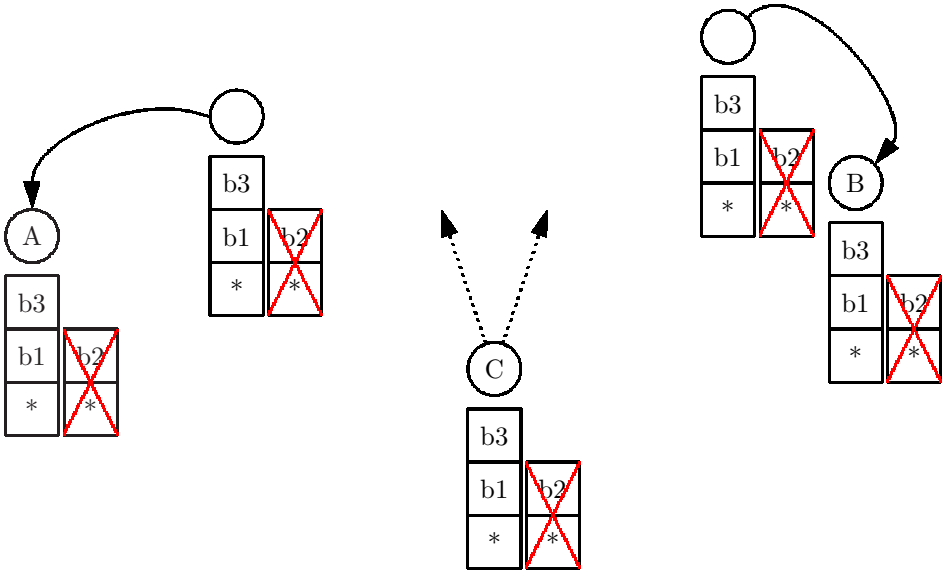


Figure 12.16: The larger blockchain propagates through the network, and all nodes discard the smaller blockchain. The network has reached consensus.

12.5.3 Block verification with Merkle trees

Hash pointers

A pointer[24] is simply a memory address. Pointers are used to form linked data structures, such as linked lists, trees, graphs, etc. A *hash pointer* is a structure which includes an ordinary pointer as well as a hash value, presumably the result of hashing the structure to which the pointer points.

Generation of Merkle trees

A Bitcoin block contains a collection (perhaps thousands) of transactions, as described above. Can a Bitcoin user tamper with these transactions? All users have access to the blockchain, they can access the most recent (top) block, for example, and change a transaction output address to their own address, thus stealing Bitcoin from the intended recipient.

For security, it is essential that if a user tampers with any of these transactions, then the tampering can be detected, in which case that version of the blockchain is discarded (i.e. not forwarded to the network). This is done by checking the hash value of the block, and comparing with the hash value on other nodes. If a user has tampered with the block, the hash values will not match, and the thief is thwarted.

It should be apparent that this verification process will be executed frequently in nodes across the network, and for that reason efficiency is of paramount importance. Bitcoin uses an efficient data structure known as a *Merkle tree* for this verification process.[25]

Bitcoin uses SHA-256 to form a Merkle tree as follows:

1. A hash pointer is created for each transaction.
2. Neighboring pairs of hash pointers are concatenated, and used as input to the same hash function.
3. This process is repeated on neighboring pairs of hash pointers, forming a tree structure, until the final result is a hash pointer to the root of the tree, as shown in Figure 12.17.

In the example of Figure 12.17 the letters $T_A, T_B, T_C, ...T_H$ refer to transactions in the block. Every neighboring pair of transactions is used as input to a hash function (SHA-256). Thus, for example, H_{AB} is the result produced when the transactions T_A and T_B are concatenated and used as input to the hash function. Likewise H_{ABCD} is the result produced when the two hash values, H_{AB} and H_{CD} are concatenated and used as input to the hash function.

The Merkle root, the hash pointer at the top of the tree, is stored in the block header, as described above.

[24] A pointer is also known as a reference in some programming languages, such as Java. The only difference between references and pointers is that the programmer can perform arithmetic, such as addition and subtraction, with pointers.

[25] Here the trees are always *binary* trees since each node may have no more than two children.

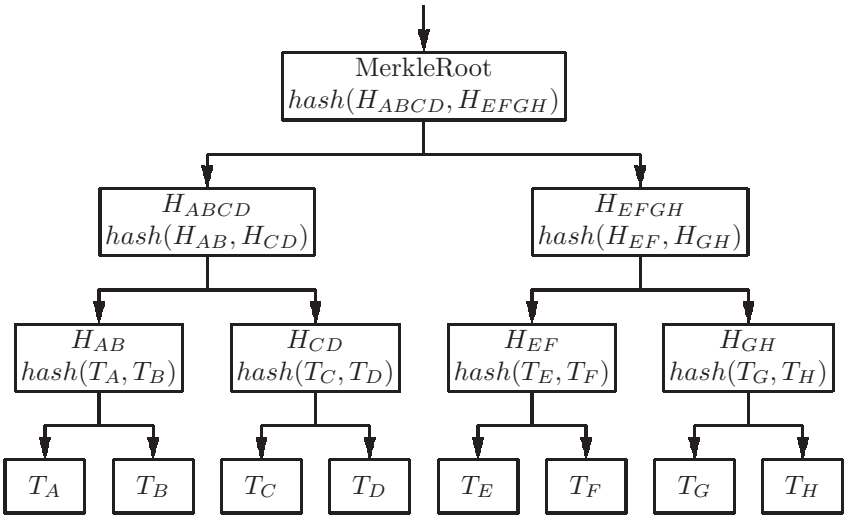


Figure 12.17: Diagram of a Merkle tree of hash values of transactions

Verification with Merkle trees

The purpose of the Merkle tree is to verify the transactions in a block, i.e. to ensure they have not been altered by an attacker. The Merkle tree provides an efficient process for verification. Note that a copy of the Merkle root is stored in the block header.

To verify one of the transactions a user needs to provide the transaction to be verified, and the Merkle hash values that lie on the path to that transaction in the Merkle tree. If there has been tampering with that transaction, there will be a contradiction somewhere in the Merkle tree, possibly at the Merkle root.

As an example, suppose Eve makes a change to transaction T_C in Figure 12.17. In order to avoid detection she will need to recompute H_{CD}. She may or may not do that. If she does recompute H_{CD}, she will need to recompute H_{ABCD} to avoid detection. She may or may not do that. If she does recompute H_{ABCD}, she will need to recompute the Merkle root to avoid detection. If she does that, a contradiction is found, since a copy of the original Merkle root is stored in the block header. If she tampers with the block header, a contradiction is found since there is a hash pointer to the block, which is now invalid, and the transaction can be removed from the block before it can be mined by a miner.

If a user wishes to verify a transaction in the block, they provide the Merkle hash values lying on a path to the desired transaction in the Merkle tree. The user will recompute the hash values on that path until a contradiction is found, as shown in Figure 12.18. If no contradictions are found, the transaction is verified.

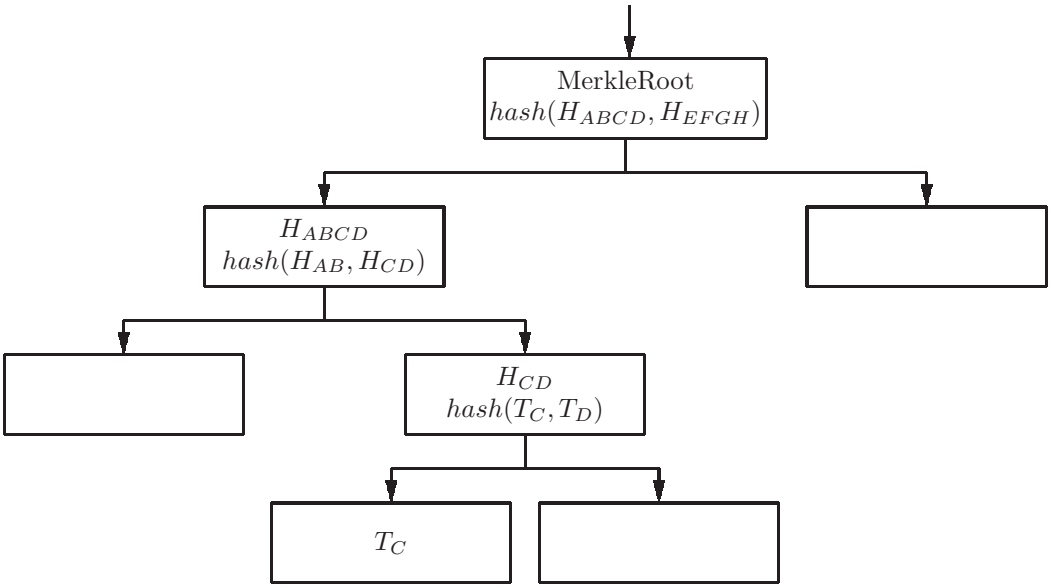


Figure 12.18: Diagram of a path to the transcaction T_C on the Merkle tree of Figure 12.17

.

12.5.4 Exercises

1. Identify the two ways a successful miner is rewarded.
2. How frequently is a new block mined.
3. What is a mining pool?
4. Explain what is meant by 'consensus' on the Bitcoin network.
5. In a block with 123 transactions, how many levels would its Merkle tree have (including the transactions and the Merkle root)?

Chapter 13

Blockchain Applications

13.1 Review of Bitcoin blockchain

In chapter 12 we exposed the Bitcoin cryptocurrency. We saw that Bitcoin transactions can be verified, and double-spending of a Bitcoin can be detected and prevented. Bitcoin transactions can be verified by Bitcoin nodes on the peer-to-peer digital network, without need for a central authority.

Bitcoin mining nodes assemble transactions into a *block* and (again) verify all the transactions in the block. Using a *proof-of-work* algorithm a miner who successfully mines a block adds that block to a collection of all previous blocks, known as the *blockchain*. The miner then distributes this new, larger, blockchain throughout the network. The blockchain is a complete ledger of all transactions ever completed using Bitcoin.

We also learned how the Bitcoin network reaches *consensus* in cases where two mining nodes succeed at the same time, producing two different versions of the blockchain.[1]

13.1.1 Exercises

1. What is the name of the structure which stores a complete ledger of all transactions involving Bitcoin?

13.2 Alternate cryptocurrencies

Bitcoin, though most popular, is not the only cryptocurrency. Other cryptocurrencies have used blockchain technology similar to Bitcoin, with varying degrees of decentraliztion.

[1] Some of the content in this chapter has been adapted from *50+ Real World Applications of Blockchain* self published by Gupta and Bansal.

13.2.1 Ethereum

Perhaps the most popular cryptocurrency, after Bitcoin, is Ethereum. However, Ethereum serves a more general purpose. Ethereum is actually a framework which can be used by independent developers to build their own blockchains, for financial transactions, other transactions, and database technology. The Ethereum blockchain has been used to host online games, social media platforms, internet service providers, etc.

From its inception in 2013 until 2022 Ethereum used a proof-of-work strategy, with miners, similar to that of Bitcoin. In 2022 it switched to a *proof-of-stake* strategy, with *validators*. Here the validators offer an 'ante', like a stake in a poker game, and validate the transactions. Then the validators are rewarded in proportion to their ante, and they lose their ante if they try to defraud the network. From the Ethereum web site:[2]

> To participate as a validator, a user must deposit 32 ETH [Ether units] into the deposit contract and run three separate pieces of software:
>
> - An execution client
> - A consensus client
> - A validator client
>
> On depositing their ETH, the user joins an activation queue that limits the rate of new validators joining the network.
>
> Whereas under proof-of-work, the timing of blocks is determined by the mining difficulty, in proof-of-stake, the tempo is fixed. Time in proof-of-stake Ethereum is divided into slots (12 seconds) and epochs (32 slots). One validator is randomly selected to be a block proposer in every slot. This validator is responsible for creating a new block and sending it out to other nodes on the network. This is known as *forging* a new block[3]Also in every slot, a committee of validators is randomly chosen, whose votes are used to determine the validity of the block being proposed. Dividing the validator set up into committees is important for keeping the network load manageable. Committees divide up the validator set so that every active validator attests in every epoch, but not in every slot.

A diagram of the proof-of-stake process is shown in Figure 13.1 and is described below:

(a) Several users, Sam, Sally, and Mary wish to become a validator, or 'forger' of a new block on the blockchain. They each offer an ante, or stake.

(b) The user offering the largest ante, Sally, is selected as the validator, and she 'forges' a new block of transactions.

[2] `https://ethereum.org`

[3] Forging is an odd term here because the block being validated may or may not be valid.

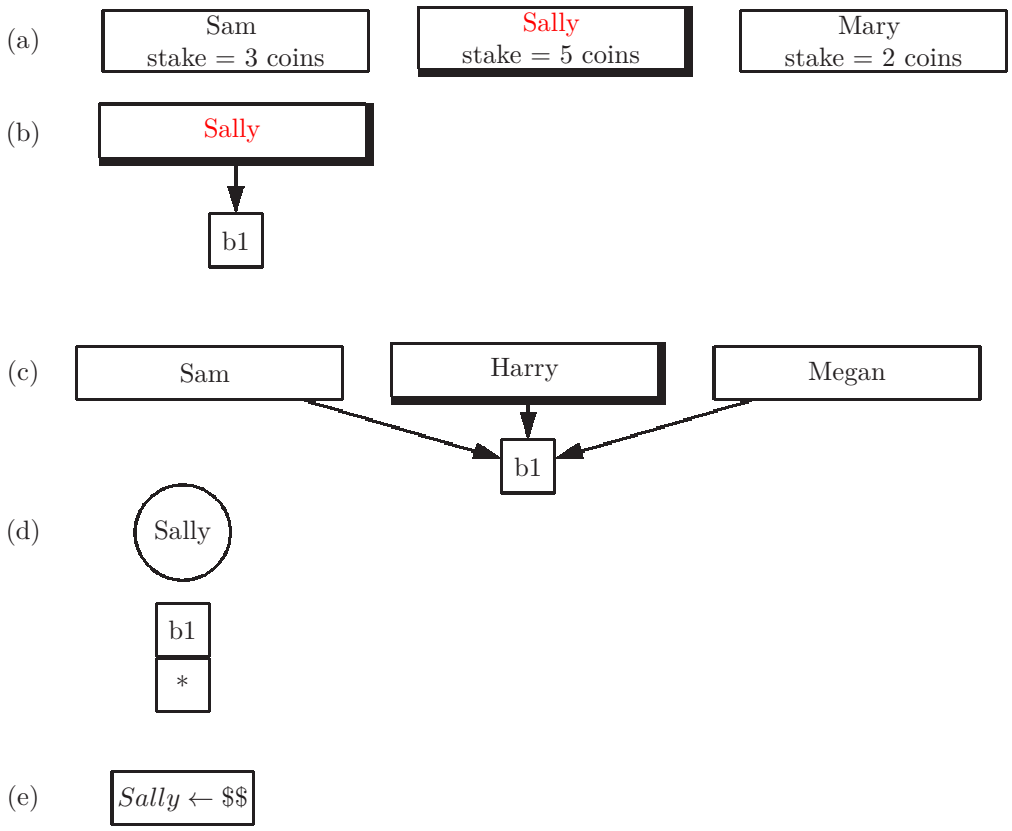


Figure 13.1: Proof-of-Stake process;
(a) a node with sufficient wealth is selected
(b) a new block is 'forged'
(c) validators verify the new block
(d) the new block is added to the blockchain
(e) the 'forger' is rewarded with transaction fees

(c) Other users, Sam, Harry, and Megan act to verify the new block

(d) If verified, the validator, Sally, adds the new block to her version if the blockchain, and puts it out on the distributed network.

(e) The validator, Sally, is rewarded with the transaction fees from all the transactions in the new block.

The proof-of-stake approach exhibits at least two advantages over proof-of-work:

- Energy consumption is reduced - Bitcoin miners form a huge drain on electrical energy.
- Scalability and efficiency are improved - Bitcoin transactions are verified in a new block every 10 minutes, whereas Ethereum blocks are verified every twelve seconds.

Similar to Bitcoin, Ether uses a decentralized distributed network with a blockchain of transactions. Its blockchain is known as Ethereum. The Ether software is open source.

The Ethereum blockchain was developed primarily by Vitalik Buterin, a Canadian, in 2015. Both Bitcoin and Ethereum permit other blockchain applications to make use of their network.

Unlike Bitcoin, Ethereum permits users to use holdings as collateral and to create and exchange NFTs.[4]

13.2.2 Dash [Xcoin,Darkcoin]

Originally named `Xcoin` as a fork of Bitcoin in 2014, it was renamed `Darkcoin`, and renamed again as `Dash` (Digital Cash) in 2015. Dash became one of the top 12 cryptocurrencies by 2018. Decisions on changes to Dash, known as *governance* are decentralized and made on a blockchain. This kind of decentralized governance is known as *Decentralized Autonomous Organization* (DAO). New coins are mined with proof-of-work, as with Bitcoin, but it uses an `X11` hash function with 11 rounds of hashing. The average time to mine a coin is only 2 minutes and 30 seconds, significantly less than the 10 minutes required by Bitcoin.

13.2.3 Dogecoin

Originally started as a joke by two employees of IBM and Adobe, the logo for Dogecoin, launched in late 2013, is an image of a large D superimposed on a dog. Amazingly, it reached a peak market value of US$8 billion by 2021.[5]

[4] An NFT, or non-fungible token is data which is associated with ownership of some holding, such as a digital image, that proves ownership of that holding. NFT's are not tied to any real currencies, such as the US dollar.

[5] Dogecoin bears no relation to the U.S. Department of Government Efficiency, DOGE, which originated 12 years later.

Dogecoin uses a *scrypt* technology in its proof-of-work algorithm. Pronounced "ess crypt", scrypt is a password-based key derivation algorithm. Instead of using a hash algorithm such as SHA-256, hardware known as FPGA[6] is used to mine new coins.

In 2021 the aerospace company SpaceX announced a rideshare mission to the moon, completely funded by Dogecoin, which was the first space mission funded by a cryptocurrency. SpaceX owner Elon Musk has been accused of insider trading, as posts on X (formerly Twitter) caused a steep rise in the market value of Dogecoin. Musk was never formally charged due to the lack of regulations pertaining to cryptocurrencies.

This event was part of a series occurring in early 2021, and NBA fans of the Dallas Mavericks were permitted to purchase tickets with Dogecoin. Other notable users of Dogecoin as a means of raising funds include the 2014 Jamaican Olympic Bobsled team, Charity: Water for well-water construction in Kenya, and NASCAR.

13.2.4 Monero

Monero, first developed in 2014, is a cryptocurrency which, similar to Bitcoin, uses a distributed blockchain, with proof-of-work, to maintain a record of transactions. However, Monero places a greater emphasis on privacy by relying not only on private keys, but also obfuscation[7] of transactions.

Because of these unusual privacy features, Monero has seen increased usage for unethical or illegal transactions, such as money laundering and ransomware. The US Internal Revenue Service has offered rewards for those who can develop Monero-tracing technology.

13.2.5 Litecoin

Litecoin is a fork of Bitcoin, starting in 2011. It differs from Bitcoin in that it entails lower transaction fees, faster verification of transactions, and more frequent updating of proof-of-work difficulty level. It also added optional privacy features in 2022.

Litecoin was designed to discourage the use of expensive hardware, such as arrays of GPUs[8], FPGAs[9], and ASIC[10] chips, for mining new coins.

13.2.6 $Trump

$Trump is a *meme* coin launched in January 17, 2025 by U.S. President Donald Trump, shortly before his second inauguration. This coin's value immediately soared to US $68 before plummeting to $3.70 in March of that year. Trump's

[6] Field Programmable Gate Array
[7] To *obfuscate* is to make confusing, or illegible. It is not quite the same as encryption.
[8] Graphics Processing Unit
[9] Field Programmable Gate Array
[10] Application Specific Integrated Circuit

political opponents accused him of unethical practices for using the office of president for his own financial gain.

13.2.7 Exercises

1. Name two advantages of proof-of-stake as compared with proof-of-work.
2. What are three advantages of smart contracts, with respect to traditional contracts.
3. Which cryptocurrency was used to finance a trip to the moon?

13.3 Other existing blockchain applications

13.3.1 Golem

Golem is a decentralized network which uses a blockchain to share computing resources. Users on the network can offer computational services (CPU time, data storage, algorithms, etc) when their systems are idle. In return they can receive services from other users as needed.

13.3.2 Ripple

Ripple, released in 2012, is technically not a cryptocurrency, but rather a means of making financial transactions, or settlements, among its users. The transactions themselves involve any fiat currency, such as the US dollar, British pound, or Chinese yuan. Ripple has been described as a real-time gross settlement system. It also supports the exchange of other financial items such as frequent flyer miles and mobile phone minutes. Ripple is noteworthy for its small energy consumption in the verification process.

Using a peer-to-peer distributed blockchain, Ripple claims to support "secure, instantly and nearly free global financial transactions of any size with no chargebacks". The settlements are made using a Ripple-defined token called *XRP*. In 2020 the US Securities and Exchange Commission (SEC) accused Ripple of selling unregistered securities in the form of XRP, but the lawsuit was rejected by a federal district court.

13.3.3 Project Bletchley

Launched by Microsoft in late 2023, Project Bletchley[11] is a blockchain based service available on Microsoft's cloud-based service named *Azure*. It is an example of *Blockchain-as-a-Service* (BaaS). Project Bletchley is not a specific blockchain, but rather a framework, or platform, which can be used to create

[11] The word *Bletchley* is clearly a reference to the World War II British code-breaking site described in chapter 2.

one's own blockchain applications. It can be used by private enterprise, government agencies, public health consortiums, and individuals. Some important aspects of Project Bletchley are:

- The platform is open and open source
- User authentication, key management, privacy, security are assured
- Scalability, efficiency, and stability
- Consortium blockchains can be restricted to members-only

13.3.4 Hyperledger

Similar to Project Bletchley, Hyperledger is another platform which supports the development of new blockchain applications. It was introduced by the Linux Foundation in 2015. Since then IBM, Intel, and SAP heve contributed to this open source project. This framework allows for collaboration among several entities developing a common, shareable blockchain. It allows the user to establish consensus and authenticity protocols. Examples of blockchains developed with Hyperledger are:

- Besu - An Ethereum codebase
- Fabric - A blockchain infrastructure with a modular architecture involving smart contracts, consensus, and other services
- Sawtooth - Developed by Intel in 2016, Sawtooth is notable for dynamic consensus, allowing for customized consensus algorithms, including *proof-of-elapsed-time*
- Aries - A toolkit for the development of authenticity or identity software
- Caliper - A benchmark tool which allows users to measure the performance of various blockchain applications
- Cello - Hosted by the Linux Foundation, Cello provides an *on-demannd* feature, allowing runtime development of blockchains
- Composer - A set of tools allowing for the collaborative development of new new blockchain applications for business applications and rapid prototyping
- Explorer - Hosted by the Linux Foundation, Explorer is designed to work with user-friendly web sites, which allows for easy viewing and searching transactions in a blockchain
- Quilt - Allows for interoperability among blockchains with an InterLedger Protocol (ILP)

13.3.5 Exercises

1. Ripple is not a cryptocurrency; what is it used for?
2. What is a BaaS?
3. Name two examples of BaaS applications.

13.4 Smart contracts

A legal *contract* is an agreement between two or more parties, stating necessary conditions for goods, services, or monetary compensation to be disbursed. For example, when you buy a new car, you and the dealer would both sign a contract stating:

- You will give the dealer the total cost of the car
- The dealer will give you the car (specifying make, model, features, colors, Vehicle Identification Number, etc.)
- If the vehicle fails during a certain period of time after the purchase, the dealer assumes responsibility for repairs or a new car.
- If the buyer does not maintain the car properly with oil/fluid changes, repairs to worn brakes, etc, the above item is not valid.

With blockchain technology, *smart contracts* can be used instead of written contracts. A smart contract consists of several validated transactions, stored in a block, or separate blocks, of a blockchain. The transactions making up a smart contract often take the form of a conditional statement, which could be a statement of the form:
`if <condition> then <agreement1> else <agreement2>`.

13.4.1 The phases of a smart contract

There are four phases in the lifetime of a smart contract:

1. **Creation** - The terms of the contract are established. These terms generally take the form of conditional statements. Often a third party such as an attorney or an expert in the field would be consulted to make sure the terms are clear, unambiguous, and appropriate. Each conditional statement will become a transaction when the contract is deployed.
2. **Deployment** - The parties involved sign the contract by applying their digital signatures to the transactions (see chapter 6). If there is later disagreement on whether a party has violated the contract, the exact terms can be extracted from the blockchain, as transactions, to resolve the dispute.

3. **Execution** - The transactions are put on the network, where 'mining' nodes will verify them and assemble them into one or more blocks. If a transaction involves a monetary payment, the verification process ensures that payment is valid, and is not double spending an unspent transaction output (UTXO) as described in chapter 12.

4. **Completion** - When the transactional conditions have been satisfied (e.g. the buyer has received the car), the consequences are implemented (the funds are transferred to the dealer's wallet).

Smart contracts have a few advantages, with respect to traditional contracts:

- **Trust** - After the creation phase, no third party is needed, including attorneys and courts, to resolve disputes. In a sense *trust* is ensured by the blockchain.

- **Transparency** - Assuming the transactions are appropriately worded, there should be no dispute as to the terms of the contract.

- **Immutability** - Once all the terms of the contract have been assembled into blocks, those blocks become a permanent part of the blockchain. None of the transactions can be rescinded or changed.

13.4.2 Smart contracts on Ethereum

In addition to being a blockchain-based cryptocurrency, Ethereum is also designed as a platform for hosting smart contracts. Every node on the Ethereum network has software known as the Ethereum Virtual Machine (EVM). The EVM is used to implement smart contracts.

Computations used in verifying smart contracts are measured in units of *gas*. An amount of gas can be used to verify a transaction in a smart contract. For every transaction submitted on the Ethereum blockchain, the user must pay Gas with Ethereum's cryptocurrency (Ether, which has a unit - ETH). The basic unit of gas is $1gwei = 10^{-9}$ ETH. More complex transactions require more gas for verification than simple contracts.

13.4.3 Examples of smart contracts

Insurance

There are many aspects of the insurance industry which could be implemented with smart contracts on a blockchain. The best example is probably claim processing for auto insurance. If several conditions apply, payment is made to the insurance client:

- Premiums for mechanical breakdown insurance are up to date

- Vehicle has broken cylinder spring, causing extensive engine damage

- Proof of periodic oil changes and preventative maintenance have been submitted
- Vehicle odometer is less than 100,000 miles

If the above conditions, each of which is a transaction on the blockchain, is satisfied the client's claim is automatically approved, and the client could be paid in ETH.

International supply chain

Today we deal with an international supply chain which is increasing in complexity and scope. For example, an automobile is constructed with:

- Steel from China, India, Russia, Japan, and the U.S.A.
- Electronics from Singapore
- Plastic and synthetic rubber parts from China, North America, and Europe

International entities can enter into smart contracts involving their own contributions, and expected gains, in this global partnership. They would not necesarily require government oversight.

13.4.4 Sharding

One of the main criticisms of the Bitcoin blockchain is that it does not scale well, meaning that performance degrades as the network grows very large. As millions of transactions are processed, the blockchain can grow very large, perhaps too large to be stored on some nodes. To alleviate this problem, some blockchain applications separate the blockchain into several sub-chains. This is known as *sharding*.

When a transaction has been verified it needs to be added to a block. At this point it is added to a *shard*[12] which is a sub-chain of the complete ledger. The shard can be selected using the low order bits of the transaction's hash value. For example, assume there are to be 16 shards. If the transaction is input to a hash function, and the hash result is $39a03dc657_{16}$, take the low order digit, 7. The transaction is assigned to shard 7, meaning that it can be included on a block to be added to the sub-chain for shard 7.

Sharding not only saves space, but searching for a transaction is more efficient. Simply hash the transaction, and the low order digit tells you in which shard its block is residing. This is the blockchain equivalent of well-known hashtable data structures. A diagram of a blockchain consisting of four shards is shown in Fig 13.2. In this case we use only the low order 2 bits of the transaction hash to determine the shard, since there are only 4 shards. Suppose transaction T1 hashes to $42bd3f98752_{16}$; the low order two bits are 10_2, so transaction T1

[12]The term 'shard' is taken from database technology.

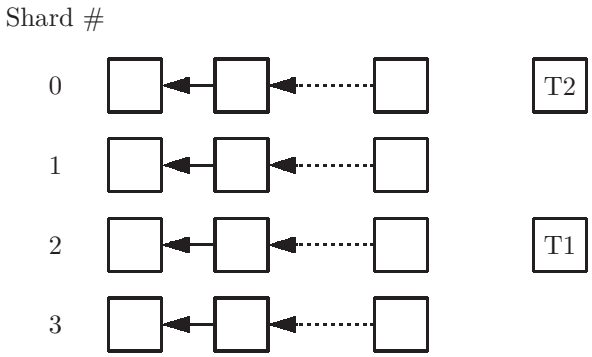


Figure 13.2: Diagram of two transactions being added to a ledger consisting of four sub-chains, known as shards

will be in shard 2. Suppose transaction T2 hashes to $66a890bf338_{16}$; the low order two bits are 00_2, so transaction T2 will be in shard 0. Both of the new blocks can be added to the ledger at the same time.

13.4.5 Internet of Things

The Internet of Things (IoT) is the name given to all inanimate devices which are connected through the internet, to 'cloud' storage and processing. These devices are often sensors of something in their surroundings, but may also be device controllers. For example the so-called *smart home* technology allows for remote control or sensing of heating/air conditioning, security surveilance, remote pet care, fridge/freezer contents, etc.

There are a few problems associated with using the internet in this way:

- The data needs to be secured. We do not want others to have access to our smart home. This can be assured by using blockchain technology. Each signal to or from a device in the smart home can be considered a transaction. These signals are acknowledged only if verified on the blockchain.
- Each smart home would require a *gateway* to interface with the internet. If there were to be a failure, or a faulty signal, there would be no way to recover from this single point of failure. With a blockchain there would be decentralized copies of the signals on the blockchain, enabling recovery from a single failure.

13.4.6 Supply chain

The phrase *supply chain* generally refers to the sequence and path through which the components of a product are obtained. Those components may also be thought of as products, having components to be obtained, and so on, until the components are raw materials. This network of components is often complex and international in scale.

For example, a bicycle is constructed from components as shown below:

- frame
  - aluminum or light weight alloy
  - paint
- wheels
  - tires
    - synthetic rubber
    - white paint
  - tubes
    - synthetic rubber
    - valve stems
  - rim
    - steel or light weight alloy
  - spokes
    - steel or light weight alloy
  - disc
    - steel or light weight alloy
- chain
  - steel or light weight alloy
- gear cassette(s)
  - steel or light weight alloy
- pedals
  - steel or light weight alloy
  - plastic reflectors
- cables
  - steel or light weight alloy
- crank
  - steel or light weight alloy
- brake assemblies
  - steel or light weight alloy

We note that some raw materials, such as a light weight alloy, may be used in several components. Also, if one raw material, such as synthetic rubber, is not available, production of a complete bicycle is not possible. In the Spring of 2020, with the advent of the Covid coronavirus, the world economy was paralyzed resulting from the halting of shipped materials; the supply chain was nullified.

Some challenges to a successful global supply chain are:

- Some items (food, drugs, livestock) are sensitive to temperature extremes.
- Contraband (illegal drugs, weapons) can be illegally inserted into the supply chain.
- The source of contaminated food may be unavailable.
- Excessive tariffs can have a detrimental impact on the global supply chain.

By using blockchain technology, many of these issues can be avoided. A transaction would consist of an item tag, its source, its destination, and necessary constraints, such as valid temperature range. If a food is found to be contaminated, its source can be found by finding its transaction on the blockchain ledger.

13.4.7 Healthcare

With the passage of the Affordable Care Act in 2010, the healthcare industry has grown huge. There are many potential applications of blockchain technology in the healthcare industry.

Medical records

Prior to 1990, doctors and nurses used paper charts to track a patient's care. Today virtually all medical records are in digital form. Every diagnosis, prescription, measurement, and outcome can be recorded as a transaction on a medical blockchain. When a patient sees various medical professionals the appropriate transactions would be made available to the doctor or nurse. All participating hospitals and medical professionals would have access to a shared blockchain. The patient could restrict access to his/her transactions to certain caregivers.

Medical research

Medical research leads to improved healthcare. However, practitioners need to refer to appropriate journals or conferences to discover new practices resulting from the research.

If each research result was a blockchain transaction, storing diagnoses, treatments, results of clinical trials, etc, a practitioner could access this information quickly.

Digital devices

Today there are many kinds of digital devices used in the healthcare industry. These devices include wearable, internal, and external devices. Some of these devices are:

- Pulsimeter (heart rate monitor)
- Pulse oxymetry (blood oxygen level)
- Blood pressure
- Blood sugar
- Pacemaker
- Body mass index (weight/height)
- Exercise data recorder
- Diet recorder (manual entry of data)

Each of these devices can record transactions for a patient on a blockchain, providing the appropriate caregiver with important information immediately.

Blood donation

When blood is donated to a blood bank, care must be taken to record the source of the donation, and the particular type of blood donated. Each donation would be recorded as a transaction on a blockchain of blood donations. In the case of a contamination, the individual can be identified to prevent future contaminations. When there is a strong need for blood of a certain type, the appropriate people can be contacted.

Clinical trials

Much medical research takes the form of clinical trials. These usually involve a large number of people (usually volunteers) and are conducted over a long term - several years, or decades. There are at least two categories of clinical trials:

- Non-therapeutic - In this study information is gathered from the participants, but no treatments, drugs, nor other therapy is part of the study. The study could collect information such as diet, exercise patterns, substance use and abuse, occupation, place of residence and work, and health of close relatives. The study would also track health problems, doctor visits, and survival rate, including cause of death.
- Therapeutic - This type of clinical trial generally focusses on a particular health issue, such as heart disease. The participants are randomly divided

into two[13] groups: a treated group and a control group. The treated group would receive a treatment, such as a new drug being tested, and the control group would generally receive a placebo[14] instead of the new drug. If the treated group shows significantly greater response to the treatment, the research could point to the efficacy of the new drug, irrespective of the placebo effect.[15]

Results of the study would then become transactions on a blockchain. Future studies could make use of this information to design more detailed, or related, clinical trials.

13.4.8 Finance

Finance involves the purchase or exchange of stocks, securities, bonds, precious metals, and other commodities. Financial institutions include banks, credit unions, investment companies, brokerage companies, and insurance companies. These companies all work with a centralized database of customers, but could benefit from a shared distributed network using blockchain technology.

Client authentication

First, and foremost, every financial institution must implement a secure strategy for client *authentication*. This means that the institution must be able to ensure that the client is really the person/entity that they claim to be. Impostors must not be granted access to financial information. Client authentication is also known as Know Your Customer (KYC).

How can blockchain technology be useful for client authentication? A potential client typically establishes proof of identity using documents such as government-issued photo ID (such as driver's license), passport, social security card, copies of IRS tax returns, utility bills, pay stubs, etc. After this has been done once for a financial institution, this information can be added as a transaction on a public blockchain. When other institutions need to authenticate the client, they can access the transaction on the blockchain to verify authenticity. The client's information would typically be hashed, and only the hash values stored on the blockchain for purposes of confidentiality.

Mortgages and loans

Banks and credit unions can profit by loaning funds, with the requirement that the returned amount be more than the inital loan (the additional amount repaid is known as *interest* on the loan). When the loan is used for the purchase of a house or real estate, the loan is called a *mortgage*. In addition to authenticating

[13] In some studies there would be a third group of participants who would have the option to use the new drug, if they wished.

[14] A placebo would be a pill that does not contain any active incredient.

[15] The case where participants taking a placebo show a reponse rate similar to the response rate for the actual drug.

the borrower's identity, the lending institution must establish the borrower's likely ability to repay the loan. Lending institutions will often require *collateral* property which would be forfeited to the lender in the case where the borrower is unable to repay the loan.

These requirements lead to a lengthy, and strictly regulated, process in the acqusition of a loan or mortgage. Blockchain technology could be used to streamline the process:

- Lending institutions could put their current interest rates, and requirements (collateral, employment records, savings, etc) on a public blockchain.
- Those seeking a loan would access the blockchain to determine whether they qualify for a loan.
- Those seeking a loan would access the blockchain to find a lending institution with the lowest interest rate.
- Lending institutions could use the blockchain to learn of potential borrowers that meet their criteria.

Crowdfunding

When launching a new project or company, there are many ways to obtain financing to get started. There are other situations where quick funding is needed, such as medical emergencies not covered by health insurance. Funding sources normally include self-funding, bank loans, and venture-capital funding. For all but the smallest projects, self-funding is not feasible. Bank loans and venture capital may have extensive requirements, including collateral, detailed business plans, borrowing history, etc.

Small companies, recent entrepreneurs, and those with immediate financial needs, may instead rely on *crowdfunding*, which is the process of discovering investors on the internet. Crowdfunding may be:

- Donation-based: Investors expect no compensation; this strategy may be used for individuals with large medical expenses or other needs.
- Rewards-based: Investors are promised financial reward if and when the project is succesful.
- Equity-based: Investors are promised a share in the profits earned by the project if succesful. This is similar to stockholders' dividends for publicly traded companies.

All three of these modes are subject to fraud. The project or financial need could be totally fictitious. Even if no financial reward is promised, this is clearly not ethical, if legal. On the other side of the coin, investors can use the rewards or equity-based crowdfunding projects for purposes of money laundering, since investors are typically anonymous.

Blockchain technology can be used to forestall these fraudulent efforts. Each crowdfunding project could consist of transactions on a public blockchain. Verification of the transactions would ensure that they are legitimate. The transactions would also reveal the source of the funds, to counteract money laundering.

13.4.9 Government

Voting

In recent years there has never been more interest in the establishment of secure, accurate, convenient, and efficient means for the casting of ballots and the tabulating of election results. In the 2000 U.S. presidential election the vote was so close, the final decision was left to the Supreme Court, declaring George W Bush the winner, after several recounts in the 'swing' state of Florida.[16] As a result many have been demanding better use of technology in our elections.

But if the internet, or any network, were to be used, protection from fraud would be a major issue. Andrew Appel, a computer science professor at Princeton University, has given this problem much attention, and has concluded that the internet should not be used to tabulate votes.[17] Appel, and others, have pointed out the importance of a *paper trail* when electronic machines are used for voting, to be used when the results are challenged by a canidate.

With blockchain technology these concerns can be addressed:

- Voters can be authenticated as described above, ensuring that each voter's identity is securely established.
- An independent organization would establish a network of validating nodes, to verify transactions on the blockchain.
- On election day a vote can be cast as a transaction on the national blockchain. The vote can be verified by other nodes on the network, ensuring that each voter casts exactly one ballot.
- The votes can be tallied quickly and accurately on the blockchain by the validating nodes, using a *Proof of Authority* (PoA) mechanism.
- Votes from members of the military (and others) stationed all over the world can be tabulated quickly.

Land Registration, assessment, and taxes

When real estate property is purchased or exchanged, there should be a public record of the transaction. This is necessary to ensure that the owner complies

[16] With so-called 'hanging chads' on paper ballots, receiving much exposure in the news media

[17] Appel was a member of a committee of the National Academies of Science, Engineering, and Medicine, which issued the report titled *Securing the Vote: Protecting American Democracy* in 2018.

with state and local regulations, and is subject to real estate taxes. In the past these records were initiated as paper records, possibly digitized for storage.

However, in addition to being inefficient, paper records are subject to fraud. With blockchain technology a real estate transaction can be a smart contract on a public blockchain. With an authenticated owner, disputes over ownership can be resolved easily. Also, property assessment and tax billing can be automated using the public ledger of real estate transactions, allowing for efficiency and cost savings at the local government level.

Vehicle Registration

When purchasing a new or used vehicle, there is a fairly extensive series of regulations with which the buyer and seller must comply. This lengthy process can be streamlined and protected from fraud or theft with a public blockchain.

1. In the case where a new vehicle is being purchased it must first be transferred from the manufacturer to the dealer. All manufacturers and dealers would have verified identities on the blockchain. Each vehicle also has a unique Vehicle Identification Number (VIN). A smart contract, i.e. transaction on the blockchain, would store a record indicating the vehicle, dealer, price, date, etc.

2. When the dealer sells the vehicle to a customer, a similar transaction is added to the blockchain. In this case the customer's identity is verified.

3. The customer must register the vehicle with the state's motor vehicle agency, but, in most states, must have proof of insurance to do so. The customer cannot buy insurance for an unregistered vehicle. Therefore registration and purchase of insurance are handled at the same time, with separate smart contracts on the blockchain.

4. If selling a used vehicle, it must pass a state inspection. This would be recorded as a transaction on the blockchain.

5. Questions and/or disputes as to a vehicle's owner, registration status, and insurance coverage can all be satisfied by inspection of the blockchain.

13.4.10 Miscellaneous technologies

E-commerce

When a purchase is made with a check, credit/debit card, Venmo, or other centralized system, customers' data is at risk if the central agent (bank) suffers a cyber attack. With a blockchain-based purchase, such as Bitcoin or Ethereum, the users' data is securely encrypted, and is decentralized. If one copy of the blockchain fails, there are several other copies of the blockchain on the distributed network.

If a transaction is disputed, it can be resolved by checking the blockchain which stores securely verified transactions.

As retailers sell stocked items, it is important that they restock appropriately, and avoid overstocking the inventory. A complete ledger of all transactions could be used to reorder items, study trends in purchaisng, and maintain an appropriate level of stocked inventory.

Swarm robotics

With the advent of intelligent and inexpensive drones, applications are being developed for the military, agriculture, weather forecasting, etc. It is important that the movement of the drones be coordinated to avoid collisions, and that they work together for a common purpose without duplicating each other's tasks unnecessarily. The tasks, movements, communications, images, etc. which are involved can be stored as transactions on a blockchain. Each drone would store its own copy of the system's blockchain.

Mobile communications

The smartphone revolution has been amazing; people all over the world now rely on this technology, not only for communcation, but also to gather various kinds of information, photography, financial transactions, etc. With this power comes risks involving the security services of a centralized service. There have been many cases where providers have been hacked, often yielding related information known as *meta data*[calling number, called number, time and date of the call, origin location, etc], if not also the digitized content of the communication as well. Text messages (SMS) are particularly vulnerable to eavesdroppers.

A decentralized system using blockchain would be more robust and secure. Each phone call or text message would be a secure and verified transaction on the blockchain shared by all users.

Big Data

The quantity of data available on the internet, normally known as the 'cloud', is enormous and increasing every day. Much of this data is derived from social media, but is also derived from commerce and communications. How much reliable information can be gleaned from this huge amount of data? Data stored on a blockchain would consist of verified transactions, providing an assurance of reliability, and disregarding false information.

Artificial Intelligence

Though artificially intelligent systems have existed for decades, the term Artificial Intelligence (AI) today generally refers to systems that look for common threads among a huge amount of data, and usually are able to arrive at solutions to problems or answers to questions, which we would normally expect to have required human intelligence. Because AI systems use statistical methods and

Large Language Models (LLM) they are only as good as the data which they use; they are not perfect.

Using a blockchain to store verified aspects of the LLM could improve the accuracy and power of AI.

13.4.11 Non-fungible tokens

A non-fungible[18] token (NFT) is a blockchain transaction which is used to verify ownership and authenticity of digital or actual objects. NFTs are most often used to verify ownership of valuable works of art or ancient artifacts. Proponents of NFTs claim that they provide proof of ownership, though there is not uniform agreement on this in the courts.

The 2021 total value of NFTs was US$17 billion. Much of this resulted from speculative trading of NFTs, and the bubble burst toward the end of 2023. Ethereum is one of the primary platforms for NFTs.

13.4.12 Exercises

1. What are the 4 phases of a smart contract?
2. What is meant by *sharding*?
3. Which of the following can be secured with blockchain technology?
   (a) Internet of things
   (b) Supply chain
   (c) Blood donation
   (d) Mortgages
   (e) Voting
4. What is an NFT?

[18]The word *fungible* pertains to something which is expressed, or affixed to, an actual or crypto currency, such as the US dollar or Bitcoin.

Glossary

AES - Advanced Enryption Standard

Affine cipher - A substitution cipher in which each letter of the ciphertext is a linear function of the corresponding letter of plaintext

Aggregating transaction - A Bitcoin transaction which makes use of outputs from two or more prior transactions

AND - The boolean operation in which the result is true if and only if both operands are true

Associative property - When an operation is performed two or more times, the result does not depend on which is done first:
`(x op y) op z = x op (y op z)`

Authenticity - The assurance that the identity of an individual, corporation, or other entity is known correctly

Base 58 encoding - Representation for numbers similar to base 64 encoding, omitting 0, O, 1, l, +, /

Base 64 encoding - Base 64 representation for numbers in which each digit represents 6 bits

BigInteger - A java data type allowing for unlimited precision arithmetic

Binary mod power - An algorithm to compute $x^p \pmod{m}$ efficiently

Bitcoin - A cryptocurrency based on a distributed peer-to-peer network which uses blockchain technology

Bletchley Park - Principle site of Britain's cryptanalysis efforts during World War II

Block - A collection of transactions and related information, for cryptocurrencies or other services using blockchain technology

Blockchain - A list of blocks forming a complete historical ledger of transactions, often used for cryptocurrency applications

Blocking mode - A grouping of bits into fixed size groups, with possible interaction between blocks during encryption and decryption

Boolean algebra - Algebra over a set of size 2, based on logic principles

Certificate - Digital information usually used to authenticate or identify an entity, often containing public key(s)

Certificate - A class in the java.security.cert package which offers cryptographic certificate services

Certificate authority - A trusted entity which can issue certificates for other entities

Cipher - An algorithm which can be used for encryption and decryption

Cipher Feedback (CFB) - A blocking mode in which the exclusive OR of each block of plaintext with a double block from the previous stage yields a block of ciphertext

Cipher Block Chaining (CBC) - A blocking mode in which the exclusive OR of each block of ciphertext with the following block of plaintext is used as the input to the following stage

Collision - A desirable property for a good hash function: Find $x1$ and $x2$ such that $hash(x1) = hash(x2)$ is hard

Combinatorics - Then mathematical study of counting the number of ways items can be selected from a given set

Confidentiality - The assurance that certain information is not publicly available

Congruence class - A set of integers all of which are congruent with respect to a given modulus

Congruent - The mathematical property assuring equivalence, but not necessarily equality, of structures

Consensus - Agreement, as done by Bitcoin users when choosing one of two different blockchains on the network

Crowdfunding - The process of discovering investors or donors, often facilitated with blockchain technology

Cryptanalysis - The area of study involving the discovery of the cryptography of others, i.e. breaking the enemy's codes

Cryptocurrency - A digital cash implementation, generally not associated with any national currency

Cryptography - The area of study involving the securing of information and communication

Cryptology - Cryptography and cryptanalysis

Cipher - An algorithm generally used for confidentiality

Cipher - A java class in the javax.crypto package which provides services for confidentiality

Ciphertext - Encrypted information, the result of the encryption process

Dash - Digital cash, mined with proof of work

DES - Data Encryption Standard

Decryption - The process of transforming encrypted information back to its original unencrypted state, by means intended by the encryptor; the inverse of encryption

Degenerate RSA key - An RSA key which fails to encrypt all possible plaintexts

Denial of service - An attack which renders a service unusable by overwhelming the service with a large volume of service requests

Diffie-Hellman - A key distribution algorithm

Discrete elliptic curve - Graph resulting from $y^2 = x^3 + ax + b \pmod{p}$ for integers x, y, a, b and prime p

Discrete log problem - The problem of finding the exponent, e, given x, b, and m where $x = b^e \pmod{m}$

Discriminant, elliptic curve - For the elliptic curve $y^2 = x^3 + ax + b$, $4a^3 + 27b^2$

Distributing transaction - A Bitcoin transaction in which the output is allotted to two or more users

Dogecoin - Crypto currency using FPGAs to mine new coins

Domain - The set of values which may be used as argument to a function

El Gamal - A public key algorithm which relies on the discrete log problem for security

Electronic Codebook (ECB) - A blocking mode with no interaction between blocks

Elliptic curve - Graph resulting from $y^2 = x^3 + ax + b$

Encryption - The process of making alterations to information such that the original can be retrieved by authorized entities, to ensure confidentiality; the inverse of decryption

Enigma - A cipher machine used by Germany prior to, and during, World War II

ETH - A unit of Ethereum currency

Ether - A framework for blockchain applications, such as crypto currency

Ethereum - An Ether blockchain

Euclidean algorithm - An algorithm to find the greatest common divisor of two integers quickly, extended to find the discrete multiplicative inverse of an integer

Euler's theorem - A generalization of Fermat's little theorem
$a^e \equiv a^{e \bmod \phi(m)} \pmod{m}$

Exclusive OR - The boolean operation in which the result is true if and only if the two operands are different

Exponent - The number of times a number is multiplied by itself

Feistel cipher [or network] - A cipher in which encryption and decryption are similar and consist of several rounds

Fermat's little theorem - If p is a prime number, then for any integer a, $a^p - a$ is a multiple of p.

Fingerprint - The result of a hash function

Fork - The split of a structure into two parts, as in a Bitcoin blockchain fork when two different versions of the blockchain are on the network

Frequency distribution - A count of the number of occurrences of each distinct value in a given list of values

Friedman test - A probabilistic algorithm used to find the length of the keyword, in the cryptanalysis of the Vigenere cipher

Full node - Bitcoin user which has all available features

Generator - The smallest non-negative member of a congruence class for a given modulus

Generator - An integer for which all values can be obtained by exponentiation with respect to a given modulus

Genesis block - The first block of the (Bitcoin) blockchain

GnuPG (GPG) - An open source version of PGP

Golem - A blockchain application which allows users to share computing resources

Handshake - The first phase of SSL or TLS in which communication parameters are established

Hash function - A function for which the domain is a string of bits of unlimited length and the range is a fixed-length string, such that pre-images are hard to find

Hash pointer - A pointer with a hash value, presumably of the structure to which it points

HASH160 - A composition if RIPEMD with SHA256 used in the Bitcoin scripting language

hashCode() - A java method which produces an integer value for any object

Hashtable - A data structure providing quick access via a hash function

Header - The initial part of a Bitcoin block storing its version number, hash of the previous block's header, timestamp, Merkle root, etc.

Hexadecimal - Base 16 representation for numbers in which each digit represents 4 bits

Hyperledger - An open source blockchain-as-a-service (Baas) developed by the Linux Foundation

Index of coincidence - The probability of selecting two matching letters from a given text, used in the cryptanalysis of the Vigenere cipher

Integrity - The assurance that a communication is received unaltered and intact

Internet of things - Digital devices which collectively and autonomously share information using the internet

Inverse - The opposite of a given operation or function, producing the original source value

Inverse permutation vector - A permutation vector which, when applied to an inverse permutation, produces the original list of values

java.security - A java package which provides cryptographic services

javax.crypto - An augmentation of the java.securty package

Key - Information which can be used to encrypt, decrypt, authenticate, or otherwise ensure security of information

Key distribution - The problem of informing allies of a secret key, while hiding it from enemies, using an insecure channel

Key pair - Mathematically related keys, a public key and a secret key

KeyGenerator - A java class in the javax.crypto package which provides factory methods to create cryptographic keys

KeyPairGenerator - A java class in the java.security package which provides factory methods to create cryptographic key pairs

Koblitz' algorithm - An algorithfm which is used to represent any integer as a point on a discrete elliptic curve

Lightweight node - Bitcoin user which can process transactions

Litecoin - A fork of Bitcoin with lower transaction fees and faster verification of transactions

Mac - A java class in the javax.crypto package which offers message authentication services

Man-in-the-middle attack - A cryptanalysis technique involving the interception of messages, and resending with the attacker's public key

Merkle root - The top hash value in a Merkle tree, used in Bitcoin

Merkle tree - A hash of two transactions or two hash values in a Bitcoin block, forming a tree structure

Message Authentication Code (MAC) - A hash function with a private key

Message digest (MD) - The output of a hash function applied to a given message

MessageDigest - A java class in the java.security package which offers hash algorithm services

Mining node - Bitcoin user which can add new blocks to the blockchain and create new Bitcoins

Mining pool - A collaborative group of (Bitcoin) miners attempting to mine a new coin

Mod - In discrete mathematics the operation which produces the remainder upon divison of integers

Mod power - An algorithm to compute $x^p \pmod{m}$

Modular arithmetic - In discrete mathematics arithmetic in which all values are reduced with respect to a given integer modulus

Monero - A cryptocurrency developed in 2014, often used for illicit purposes due to its emphasis on privacy

Multiplicative inverse - The relation of two integers such that their product is one

Nakamoto, Satashi - Name of the author of the original technical paper and software for Bitcoin; the name may be a pseudonym

Non-fungible token (NFT) - A blockchain transaction which is used to verify ownership or authenticity of digital or physical objects

Nonce - Meaningless or irrelevant information, as in a Bitcoin block header field used to demonstrate proof of work

NOT - A boolean operation with one operand in which the result is the logical complement of the operand

Obfuscating compiler - A compiler which produces an equivalent, but unintelligible version of a source program

One-time pad - A polyalphabetic cipher in which the length of the key exceeds the sum of the lengths of all plaintexts

One-way function - A function that has an inverse which is hard to derive

OP_CHECKSIG - An operation in the Bitcoin scripting language which will pop a public key and a signed UTXO from the stack which are then used to verify the transaction output

OP_DUP - An operation in the Bitcoin scripting language which will duplicate the top value on the stack

OP_EQUAL - An operation in the Bitcoin scripting language which will pop two values from the stack, compare them for equality, and push the result onto the stack

OP_EQUALVERIFY - An operation in the Bitcoin scripting language which consists of two stack operations: OP_EQUAL and OP_VERIFY

OP_HASH160 - An operation in the Bitcoin scripting language which will pop the top value from stack, use it as input to HASH160, and push the result onto the stack

OP_VERIFY - An operation in the Bitcoin scripting language which examines the top value on the stack; if FALSE, the script terminates; if TRUE, the stack is popped

OR - The boolean operation in which the result is true if and only if either operand is true

Output feedback (OFB) - A blocking mode in which the exclusive OR of each block of plaintext with an encrypted double block from the previous stage forms each block of ciphertext

Padding - Appending of extra bits at the end of a plaintext to form a full block

Permutation - A reordering of a given list of values

Permutation vector - A list of integers specifying a permutation for a given list of values

Plaintext - Unencrypted information, generally the source for the encryption process

Pointer - A memory address

Polyalphabetic cipher - A cipher in which each letter of the ciphertext is a function of the corresponding letter of the plaintext and its position in the plaintext

Preimage - A desirable property for a good hash function: Given y, finding a value, x, such that $y = hash(x)$ is hard

Pretty Good Privacy (PGP) - A package of cryptographic software developed by Phil Zimmerman

Private key - A cryptographic key which is shared with allies, but hidden from enemies

Prime - An integer which has no divisors other than one and itself

Project Bletchley - Microsoft's blockchain-as-a-service (BaaS)

Proof of stake - A scheme used to ascertain that a significant amount of cash is available, as in the mining of a new ETH

Proof of work - A scheme used to ascertain that a significant amount of computation has been expended, as in Bitcoin mining

Protocol - A sequence of steps which can be used to establish communication

Public key - A cryptographic key which is shared with all, but is paired with a secret key

Public key authority - A trusted entity which can distribute public keys for other entities

Range - The set of values which may result from a function

Record - The second phase of SSL or TLS in which communication takes place

RIPEMD - A secure hash algorithm

Ripple - A real-time gross settlement system using a blockchain

RSA - Public key algorithms used for encryption, integrity, and authentication, developed by Rivest, Shamir, and Adleman

S-box - A boolean function with an n-bit input and an m-bit output, used in many private key ciphers

Satashi - A unit of Bitcoin currency equal to 10^{-8} Bitcoin

Scrawl - Frequency distribution of the letters in a given text, sorted alphabetically

SDES - Simplified Data Encryption Standard

Second preimage - A desirable property for a good hash function: Given $x1$, finding a value, $x2$, such that $hash(x1) = hash(x2)$ is hard

Secret key - A cryptographic key which is not shared with anyone, but is paired with a public key

Secure hash algorithm (SHA) - A hash function which has the three properties: Preimage, second preimage, and collision

Secure Socket Layer (SSL) - A protocol to establish secure communication over an insecure channel

Session key - A private key used for symmetric encryption and decryption in a single communication session

Sharding - The separation of a blockchain into two or more blockchains to improve performance

Shift cipher - A classical cipher in which each letter of the plain text is replaced by another letter, chosen at a fixed distance in the alphabet

Signature - A bit string associated with a plaintext, or other bit string, which can be used to verify authenticity and/or integrity

Signature - A java class in the java.security package offering authentication services

Signature - Frequency distribution of the letters in a given text, sorted by frequency

Singular elliptic curve - An elliptic curve for which the discriminant is 0

Smart contract - An agreement stating conditions for goods or services to be disbursed, verifiable as a blockchain application

Supply chain - The sequence and path through which components of a product are obtained

Substitution cipher - A classical cipher in which each letter of the plain text is replaced by some other letter of the alphabet

Totient function - Defined by Euler, $\phi(n)$ is the number of positive integers less than n which are relatively prime to n.

Transaction - An exchange of goods, services, or information, such as a purchase made with Bitcoin

Transport Layer Security (TLS) - An improved version of SSL

Unspent Transaction Output (UTXO) - Bitcoin transferred to a user's wallet

Validator - Ethereum user who offers proof of stake to validate a transaction

Verification - The process of determining the authenticity and/or integrity of information

Vigenere cipher - A substitution cipher using a fixed-length key word

Wallet - Bitcoin software which can process and verify transactions

Bibliography

[1] Andreas M. Antonopoulos. *Mastering Bitcoin.* O'Reilly, 2017.

[2] Thomas H. Barr. *Invitation to Cryptology.* Prentice Hall, 2002.

[3] Abhijit Das. *Computational Number Theory.* CRC Press, 2013.

[4] Neil Daswani, Christoph Kern, and Anita Kesavan. *Foundations of Security.* Apress, 2017.

[5] Hans Delfs and Helmut Knebl. *Introduction to Cryptography.* Springer, 2006.

[6] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering.* Wiley, 2010.

[7] Behrouz A. Forouzan. *Cryptography and Network Security.* McGraw Hill, 2008.

[8] Nick Galbreath. *Cryptography for Internet and Database Applications.* Wiley, 2002.

[9] Aman Gupta and Roohi Bansal. *50+ Real World Applications of Blockchain.* self published, 2022.

[10] Darel W. Hardy, Fred Richman, and Carol L. Walker. *Applied Algebra: Codes, Ciphers, and Discrete Algorithms.* CRC Press, 2009.

[11] English Heritage. *Bletchley Park: Home of the Codebreakers.* Bletchley Park Guidebook, 2010.

[12] M. Jason Hinek. *Cryptanalysis of RSA and its Variants.* CRC Press, 2010.

[13] David Hook. *Beginning Cryptography with Java.* Wiley, 2005.

[14] Douglas Jacobson. *Introduction to Network Security.* CRC Press, 2009.

[15] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* CRC Press, 2015.

[16] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security*. Prentice Hall, 2002.

[17] Kevin Kenan. *Cryptography in the Database*. Addison Wesley, 2006.

[18] Philip Klein. *A Cryptotraphy Primer*. Cambridge, 2014.

[19] Richard Klima and Neil Sigmon. *Cryptology*. CRC Press, 2012.

[20] Heiko Knospe. *A Course in Cryptography*. American Mathematical Society, 2019.

[21] Jonathan Knudsen. *Java Cryptography*. O'Reilly, 1998.

[22] Susan Loepp and William Wooters. *Protecting Information*. Cambridge, 2006.

[23] Wenbo Mao. *Modern Cryptography*. Prentice Hall, 2004.

[24] Alfred J. Menezes et al. *Handbook of Applied Cryptography*. CRC Press, 1996.

[25] Richard Mollin. *An Introduction to Cryptography*. Chapman & Hall, 2007.

[26] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies*. Princeton University Press, 2016.

[27] Scott Oaks. *Java Security*. O'Reilly, 2001.

[28] Bruce Schneier. *Applied Cryptography*. Wiley, 1996.

[29] Nigel Smart. *Cryptography Made Simple*. Springer, 2016.

[30] William Stallings. *Cryptography and Network Security*. Pearson, 2017.

[31] Alexander Stanoyevitch. *Introduction to Cryptography*. CRC Press, 2011.

[32] Douglas Stinson and Maura Paterson. *Cryptography Theory and Practice*. CRC Press, 2019.

[33] Melanie Swan. *Blockchain*. O'Reilly, 2015.

[34] Peter Thorsteinson and G. Gnana Arun Ganesh. *.Net Security and Cryptography*. Prentice Hall, 2004.

[35] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. Pearson, 2020.

[36] Jason Weiss. *Java Cryptography Extensions*. Morgan Kaufman, 2004.

Index