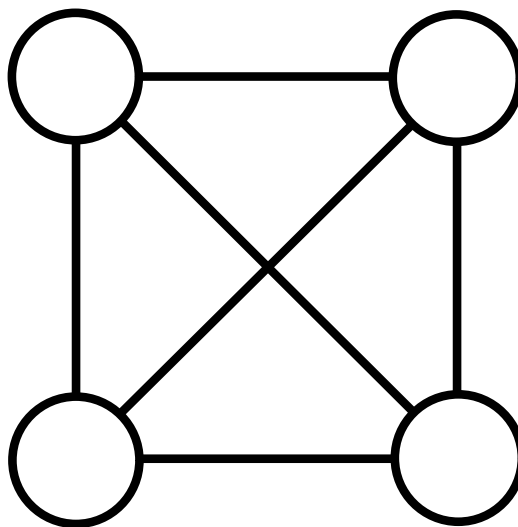


An Introduction to Cryptography

Ivan Damgård



August 10, 2020

Contents

<i>Preface</i>	2
1 Introduction	3
1.1 Model of algorithms and adversaries	3
2 Basic Facts from Probability Theory	5
2.1 Introduction	5
2.2 Conditional Probabilities	6
2.3 Waiting for an event	6
2.4 Jensen's inequality	7
3 Basic Number Theory and Algebra	8
3.1 Modular Arithmetic	8
3.2 Definition of Groups	10
3.3 Further examples of groups	12
3.4 Finite Fields	13
3.5 More about Groups: Lagrange's Theorem	17
3.6 More Number Theory	19
4 Symmetric Cryptosystems and Their History	23
4.1 Symmetric Cryptosystems	23
4.2 An Example: the Shift Cipher	24
4.3 Attacks on Symmetric Cryptosystems	24
4.4 More Historic Examples	26
4.5 Exercises	31
5 Unconditional Security and Information Theory	33
5.1 Introduction	33
5.2 Perfect security	34
5.3 Information theory and cryptography	37
5.4 When the adversary has incomplete information	48
5.5 Exercises	51
6 Modern Symmetric Cryptosystems	54
6.1 The DES Blockcipher	54
6.2 The AES Blockcipher	58
6.3 Differential and Linear Cryptanalysis	61

6.4	Defining Security of Symmetric Encryption	66
6.5	Good Symmetric Encryption from Pseudorandom Functions.	70
6.6	Exercises	74
7	The RSA 1-way trapdoor function	78
7.1	Introduction	78
7.2	RSA	80
7.3	RSA decryption works	80
7.4	Implementation of RSA	81
7.5	Optimizations of RSA	86
7.6	Security of RSA	87
7.7	Exercises	91
8	The Theory of Secure Public-Key Encryption	94
8.1	Public-Key Cryptosystems	94
8.2	Security of Public Key Encryption	95
8.3	Exercises	103
9	Public-Key Encryption Based on Discrete Logarithms	106
9.1	Introduction	106
9.2	Preliminaries	106
9.3	Three computational problems	107
9.4	The El Gamal Cryptosystem	109
9.5	Some Example Groups	111
9.6	Algorithms for solving discrete log	118
10	Public-Key Encryption Based on Learning With Error (LWE)	121
11	Symmetric Authentication Systems	124
11.1	Introduction	124
11.2	Hash Functions	126
11.3	Definition of Message Authentication Codes (MACs)	134
11.4	Existence of good MAC Schemes	135
12	Digital Signatures	138
12.1	Definition of Digital Signature Schemes	138
12.2	Signature schemes based on RSA	139
12.3	Signature schemes based on discrete logarithms	141
12.4	Combining Signatures and Hashing in general	145
12.5	Existence of good Signature Schemes	147
12.6	Dealing with replay attacks	149
	<i>List of Exercises</i>	151
	<i>References</i>	154

Preface

This is the second edition of a complete set of course notes for the introductory course in cryptography. It will be expanded and edited as we go.

There will almost certainly be plenty of typos, and the reader is encouraged to report any errors and omissions to the author.

Introduction

Contents

1.1 Model of algorithms and adversaries

3

This manuscript will introduce you to the basics of Cryptography. We will concentrate on the most classic use of crypto, namely to secure communication between two parties who trust each other, against attacks by an external adversary. Thus we will cover cryptosystems and authentication systems, how they are defined, how to construct them and how to define and prove their security.

On the way, we will introduce the tools that are necessary for this purpose, including basic number theory and algebra, and auxiliary concepts such as pseudorandom functions, one-way trapdoor functions, hash functions etc.

1.1 Model of algorithms and adversaries

Throughout this note, when we talk about adversaries or algorithms that do encryption or decryption, what we mean more formally is they are *Probabilistic Turing Machines*. Such a machine is a Turing machine with an extra read-only tape, a so-called random tape that is filled with uniformly random bits before the machine runs. At each step, the machine can choose to read from the random tape and in this way make a randomized decision about what to do next. This means that even if we fix the input, the output is not determined, instead there will be some probability distribution on the output, for each possible input.

The running time of such a machine is as usual the number of steps taken until it halts. When we talk about upper bounds on the running time, we always mean a bound that holds with probability 1, i.e., saying that the machine runs in time at most t means that it *always* halts in at most t steps (despite the fact that it is probabilistic). In the following, a probabilistic polynomial time algorithm (a PPT algorithm) is a probabilistic machine that always halts in time $p(n)$ where p is a polynomial and n is the length of the input.

We will often talk about giving an algorithm access to an oracle O . What this means formally is that the machine has two special tapes, for sending to the oracle and to receive from it. When it writes some string x on the send tape, the oracle will compute some (possibly randomized) function $O(x)$ and write this on

the receive tape. Such an oracle call usually counts as one step when computing the running time of the machine.

Basic Facts from Probability Theory

Contents

2.1	Introduction	5
2.2	Conditional Probabilities	6
2.3	Waiting for an event	6
2.4	Jensen's inequality	7

2.1 Introduction

This chapter contains a brief introduction to some basic notions and results from the theory of probabilities. If you took a course in this subject before, you will be familiar with most of what is here, and will not really need to read this chapter. Even if you did not take such a course, the best strategy is to skip this chapter on a first reading and return to it whenever you encounter something you need to have explained.

Probabilities come into play when we do a random experiment such as throwing a die a number of times. The experiment has some number of outcomes, in the example an outcome would be a list of the results we obtained from throwing the die.

Events are defined as sets of outcomes. For instance “at least one 6 was obtained” implicitly defines such a subset of the possible outcomes, namely all lists that contain at least one 6. For an event A , we use $P[A]$ to denote the probability of A , which can be computed by adding up the probabilities for all outcomes that are in A .

A random variable X can be defined as a function from the set of outcomes to the real numbers (or the integers, or the natural numbers, as the case may be). For instance, in the die example we could define X to be 1 if at least one 6 was obtained and 0 otherwise. A random variable takes on a certain value when the experiment is done, and the probability of a certain value can be computed by adding up the probabilities of all outcomes leading to the value in question.

Say X is a random variable taking values x_1, \dots, x_n with probabilities p_1, \dots, p_n . The expected value of X is written $E(X)$ and is defined to be the weighted

average of the outcomes:

$$E(X) = \sum_{i=1}^n p_i x_i$$

2.2 Conditional Probabilities

If we assume that an event B occurred, we can ask what the probability then is of another event A . Naturally, if we are given the information that B occurred, this may change the likelihood that A occurred.

We will let $P[A, B]$ denote the probability that both A and B occur, and $P[A|B]$ the probability that A occurs given that B occurs. Using this notation, we have

Theorem 2.1 (Baye's formula) *If $P[B] > 0$, we have*

$$P[A|B] = \frac{P[A, B]}{P[B]}.$$

2.3 Waiting for an event

Consider some random experiment and an event A . Suppose we do an arbitrary number of independent repetitions of the experiment. The question now is: how many repetitions do we have to do before A occurs? The following theorem provides an answer, in terms of the expected number of iterations:

Theorem 2.2 (Expected number of repetitions) *Given a random experiment and an event A , with $P[A] = p > 0$. Assume we repeat the experiment independently and indefinitely. The expected number of repetitions until A occurs is $1/p$.*

PROOF The probability that A occurs for the first time in the i 'th iteration is $(1-p)^{i-1}p$, namely for this to happen A must not occur the first $i-1$ times, but should occur the i 'th time. We can think of the index i where A happens for the first time as a random variable and compute its expectation:

$$\sum_{i=1}^{\infty} i(1-p)^{i-1}p = p \sum_{i=0}^{\infty} (i+1)(1-p)^i$$

The sum $\sum_{i=0}^{\infty} (i+1)(1-p)^i$ can be expanded as

$$\begin{array}{lll} i = 0 & i = 1 & i = 2 \dots \\ (1-p)^0 + & (1-p)^1 + & (1-p)^2 + \dots \\ & (1-p)^1 + & (1-p)^2 + \dots \\ & & (1-p)^2 + \dots \end{array}$$

The original expression tells us to compute the sum column by column, but we can instead sum the rows first and then add all these “horizontal” sums. This is fine, as all involved series converge. Now, the first horizontal sum is

$$\sum_{i=0}^{\infty} (1-p)^i = \frac{1}{1-(1-p)} = \frac{1}{p}$$

by a well known standard formula. Moreover, each horizontal sum is $(1-p)$ times the previous one, so we get

$$\sum_{i=0}^{\infty} (i+1)(1-p)^i = \sum_{j=0}^{\infty} (1-p)^j \frac{1}{p} = \frac{1}{p} \sum_{j=0}^{\infty} (1-p)^j = \frac{1}{p^2},$$

implying that the expectation is $p \cdot 1/p^2 = 1/p$. \square

2.4 Jensen’s inequality

Assume we are given a function f that maps the positive real numbers into real numbers (the restriction to positive real numbers is not really needed, but it fits in with the examples of functions that we are interested in). f is said to be *concave* if for any input numbers a, b , it holds that

$$\frac{f(a) + f(b)}{2} \leq f\left(\frac{a+b}{2}\right). \quad (2.1)$$

In other words, if we take the average of the function values, we get a smaller value than if we first take the average of the inputs and then apply the function. Concretely, if f is continuous, this means that if we draw the graph of f , we get a curve that is concave, or equivalently, the second derivative is always less than zero.

f is said to be *strictly concave* if we have $\frac{f(a)+f(b)}{2} < f(\frac{a+b}{2})$ for all $a \neq b$. This basically means that graph of the function always curves and is never linear. A logarithm function is an example of this.

Jensen’s inequality now basically says that the inequality from (2.1) also holds if more than two input values are involved, and if we take not only the standard average, but a weighted average using any probability distribution:

Theorem 2.3 *Let p_1, \dots, p_n be a probability distribution, that is, $\sum_i p_i = 1$ and $0 \leq p_i \leq 1$. Then for any x_1, \dots, x_n , we have*

$$\sum_{i=1}^n p_i f(x_i) \leq f\left(\sum_{i=1}^n p_i x_i\right)$$

Furthermore, if f is strictly concave, equality holds if and only if all the x_i ’s are equal.

Basic Number Theory and Algebra

Contents

3.1	Modular Arithmetic	8
3.2	Definition of Groups	10
3.3	Further examples of groups	12
3.4	Finite Fields	13
3.5	More about Groups: Lagrange's Theorem	17
3.6	More Number Theory	19
3.6.1	The Chinese Remainder Theorem	19
3.6.2	The structure of \mathbb{Z}_p^*	21

This chapter gives a basic introduction to modular arithmetic, finite groups and finite fields, with a minimum of general theory. If you are already familiar with these concepts, it is not necessary to read this chapter. If you are not, the best strategy is to not read this chapter on a first reading, but start from the next chapter, and refer back to here when you encounter concepts that you need to have explained.

3.1 Modular Arithmetic

Most people know, or can quickly figure out, that adding two even numbers, or two odd numbers, results in an even number. And similarly, that adding an even and an odd number produces an odd result. So in other words

$$even + even = odd + odd = even$$

$$odd + even = even + odd = odd$$

Another way of expressing exactly the same thing is to say: we can divide any number a by 2 and find the residue. This residue will be 0 (if a is even) or 1 (if a is odd). Now, if we add two numbers that have residue 0, or two that have residue 1, we get a number with residue 0. If we add two number that have residues 0 and 1, we get one that has residue 1. Summarizing:

$$0 + 0 = 1 + 1 = 0$$

$$1 + 0 = 0 + 1 = 1$$

These rules for adding 0's and 1's is known as addition modulo 2. In order not to confuse this with standard addition, where of course $1+1$ is not 0, this is usually written like this

$$0 + 0 \bmod 2 = 1 + 1 \bmod 2 = 0$$

$$1 + 0 \bmod 2 = 0 + 1 \bmod 2 = 1$$

What this expresses is something about properties of large classes of integers, and what happens if we add a number from one class to a number from another.

However, there is nothing special about 2 here. We can replace 2 by any natural number n and do exactly the same thing.

So let n be any natural number. Then Z_n will denote the set $\{0, 1, 2, \dots, n-1\}$, that is, all possible residues we can get when dividing by n .

Any integer x can be mapped to Z_n by dividing by n and taking the residue, this is called *reduction modulo n* , and the resulting residue is called $x \bmod n$. So any $x \in Z$ corresponds to an element in Z_n via reduction mod n . For instance, if $n = 3$, then the numbers $\dots -3, 0, 3, 6, \dots$ all correspond to 0 in Z_3 , while $\dots, -2, 1, 4, \dots$ all correspond to 1, etc. For any n , we can split the integers into classes, according to which element in Z_n they correspond to. There will exactly n classes. Numbers in the same class are said to be *congruent modulo n* . So 1 and 4 are congruent mod 3, this is written as $1 \equiv 4 \bmod 3$, while 0 and 5 are not.

We can now define addition and multiplication of numbers in Z_n in a special way, so we get results that are also in Z_n . This goes in the same way as we added even and odd before. Namely, from numbers $a, b \in Z_n$, we can compute a new number called $a + b \bmod n$ which is also in Z_n , as follows: compute $a + b$, divide by n and let $a + b \bmod n$ be the residue. Since we divide by n , the residue will always be in Z_n . Multiplication can be done in a similar way, $ab \bmod n$ is by definition the residue we get when dividing ab by n .

For instance, $1 + 4 \bmod 5 = 0$, $2 \cdot 3 \bmod 5 = 1$.

In fact, we can add or multiply arbitrary integers modulo n - the method is the same as before: we do the addition or multiplication, divide by n and let the residue be the result. For instance, $6 + 9 \bmod 5 = 0$, $7 \cdot 8 \bmod 5 = 1$.

It is important to understand that an equation like $2 \cdot 3 \bmod 5 = 1$ makes a very general statement. It says "take any two numbers where the first has residue 2 when divided by 5 and the second has residue 3. Multiply them, and you are guaranteed to get a result that has residue 1".

This leads to another important point, namely that from the point of view of arithmetic modulo n , numbers that are congruent modulo n are completely equivalent. An example: suppose someone asks you: "what will the time be 25 hours from now?". If you think for a few seconds before answering, you may notice that the answer of course is "one hour more than the time now. What this exploits is that $25 \bmod 24 = 1$, and hence adding 25 modulo 24 is the same as adding 1. Exactly the same thing happens for multiplication, although it may not seem as obvious: perhaps multiplication by 1748564 modulo 1748563 looks

like a complicated operation. But in fact, since $1748564 \bmod 1748563 = 1$, it is the same as multiplying by 1, i.e., doing nothing!

Lets argue that this will always work: suppose $a \bmod n = i$, then we want to show that for any $x \in \mathbb{Z}_n$, we have $ax \bmod n = ix \bmod n$. The assumption that $a \bmod n = i$ means that a can be written as $a = qn + i$, this just says that when we divide a by n , we get quotient q and residue i . So hence $ax = qxn + ix$. Clearly, when we divide this by n , since qxn is divisible by n , the residue we get is the residue we get when dividing ix by n . That is, it will be $ix \bmod n$ which is what we wanted to show.

For this reason, we can conclude that in any calculation modulo n , we can reduce any term modulo n , or we can choose not to. The final result will always be the same. For instance, look at $7 \cdot 8 \bmod 5$. We can calculate the result by first seeing that $7 \bmod 5 = 2$ and $8 \bmod 5 = 3$, and then calculating $2 \cdot 3 \bmod 5 = 1$. But we could also directly multiply 8 by 7 to get 56 and then divide by 5 to get the same residue 1.

A slightly more advanced example: can you calculate $1772 \cdot 1771 \bmod 1773$ in your head? – looks difficult? not at all! the trick is to notice that 1772 and 1771 are congruent to -1 and -2, respectively, modulo 1773. Now, $-1 \cdot -2 = 2$, and $2 \bmod 1773 = 2$ so the result is 2.

Modular arithmetic has been used in cryptography from the very beginning, for instance in the Caesar substitution and many of its variants. This will be described in more detail in the following chapters.

3.2 Definition of Groups

When we compute with ordinary integer or real numbers, there are several properties we are used to assuming. For instance, for addition of real numbers, the following should come as no surprise:

- Parentheses can be moved around freely (as long as only addition is involved), concretely for any $a, b, c \in R$, we have

$$(a + b) + c = a + (b + c)$$

we say that addition is *associative*.

- There is a number, namely 0, such that when we add it to something else, nothing happens. Concretely, for any $a \in R$

$$a + 0 = 0 + a = a$$

We say that 0 is the *neutral* element for addition.

- If we add a number to another, we can always add minus that number, and we are back where we started, that is, we have effectively added 0. Concretely, for any $a \in R$, there exists a number, namely $-a$ such that

$$a + (-a) = 0$$

We say that any number $a \in R$ has an *inverse* with respect to addition, namely $-a$.

A very similar list of properties can be stated about multiplication. Here, however, we shall have to restrict ourselves to the real numbers except 0, which is written R^* . That is, $R^* = R \setminus \{0\}$. Then we have

- Parentheses can be moved around freely, concretely for any $a, b, c \in R^*$, we have

$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

we say that multiplication is *associative*.

- There is a number, namely 1, such that when we multiply it with something else, nothing happens. Concretely, for any $a \in R^*$

$$a \cdot 1 = 1 \cdot a = a$$

We say that 1 is the *neutral* element for multiplication.

- If we multiply by a number a , we can always multiply $1/a$, and we are back where we started, that is, we have effectively multiplied by 1. Concretely, for any $a \in R^*$ there exists another number, namely a^{-1} such that

$$a \cdot a^{-1} = 1$$

We say that any number $a \in R^*$ has an *inverse* with respect to multiplication, namely a^{-1} .

The last property is of course the reason why we could not state all three properties for all real numbers: 0 does not have an inverse w.r.t. multiplication. Having multiplied by 0, there is no way to go back where we came from by multiplying by another number.

The two lists of properties look very similar, and it turns out that there are many other cases where such a list of properties can be stated. Moreover, there are many results that turn out to only depend on the type of properties we have stated. Therefore, it pays off to make one definition that talks only about the essential behavior of the operation and abstracts away unnecessary details.

So let us assume we have a set G , and a so-called binary operation \diamond on G . The operation \diamond can be thought of as a machine, where we can stick in two elements from G , and out pops a new element from G . If we stick in elements a, b , the result that pops out is called $a \diamond b$. Concretely, G could be R or R^* , and \diamond could be $+$ or \cdot , but there are also other possibilities, as we shall see.

We now have

Definition 3.1 The pair (G, \diamond) is said to be a *group*, if the following are satisfied

- \diamond is associative, i.e.

$$(a \diamond b) \diamond c = a \diamond (b \diamond c)$$

- There exists a neutral element for \diamond in G , that is, there exists an element $1_G \in G$ such that for all $a \in G$:

$$a \diamond 1_G = 1_G \diamond a = a$$

- For any $a \in G$, there exists an inverse $a^{-1} \in G$, such that

$$a \diamond a^{-1} = 1_G$$

So it should be obvious now that $(R, +)$ and (R^*, \cdot) are both groups. Note that it is important to specify both the set and the operation to see whether you have a group or not. For instance, (R, \cdot) is NOT a group, because 0 has no inverse element with respect to multiplication.

What about integers? well $(Z, +)$ is fine, this is indeed a group, but (Z, \cdot) doesn't work at all! can you see why? What about $(\{-1, 1\}, \cdot)$?

3.3 Further examples of groups

Modular arithmetic provides us with an infinite range of examples of sets and arithmetic operations on them. One may well wonder whether they are groups or not. The first result in this direction is:

Theorem 3.2 *For any natural number n , $(Z_n, +)$ is a group, where $+$ means addition modulo n .*

This is quite straightforward to verify, try it!

Multiplication is more involved. First of all, (Z_n, \cdot) is not a group, because 0 does not have an inverse. Whatever we try to multiply 0 with, we always get 0 and never the neutral element 1. So maybe it works if we just kick out 0? well, not quite! look at Z_6 , for instance. If you take the element 2, and multiply it with any possible number in Z_6 you will get as results 0, 2, 4, 0, 2, 4. So there is no inverse of 2 modulo 6.

It turns out that what is “wrong” with 2 (with respect to 6) is that the greatest common divisor of 2 and 6 is greater than 1. In fact $\gcd(2, 6) = 2$, and this is the reason why all the results we got were 0 or divisible by 2.

So the solution to making multiplication mod n be a group operation is to get rid of all numbers with bad gcds. We define Z_n^* to be all numbers in Z_n that have gcd 1 with n , i.e.

$$Z_n^* = \{a \in Z_n \mid \gcd(a, n) = 1\}$$

We then have

Theorem 3.3 *For any natural number n , (Z_n^*, \cdot) is a group, where \cdot means multiplication modulo n .*

PROOF Associativity is clear, and it is also clear that 1 is the neutral element w.r.t. multiplication. What needs to be shown is that every element has an inverse. For this, let $a \in Z_n^*$, and define a function f_a that goes from Z_n to Z_n and is defined by $f_a(x) = ax \bmod n$. We claim that this function is injective. To see this suppose we have x, y such that $f_a(x) = f_a(y)$. This means $ax \bmod n = ay \bmod n$, and so $ax - ay \bmod n = a(x - y) \bmod n = 0$. In other words, n divides $a(x - y)$. But since a has no prime factors in common with n (recall $\gcd(a, n) = 1$), it must be the case that n divides $x - y$, i.e., $x - y \bmod n = 0$, and this means that $x = y$.

Since the function f_a sends n elements to n elements and is injective, it must also be surjective, i.e. every element in Z_n is hit by f_a , including 1. In other words, there must exist a' such that $f_a(a') = 1$, i.e. $a \cdot a' \bmod n = 1$ so that a' is the inverse of a we are looking for. It is also clear that a' must itself be in Z_n^* : if not, then there must be a prime p that divides both a' and n . But then it divides aa' which can be written as $aa' = 1 + tn$ for some t . Then we would have that p divides both tn and $tn + 1$ which cannot be the case. \square

For instance, if we take $n = 5$, $Z_5^* = \{1, 2, 3, 4\}$, and the inverse elements w.r.t. multiplication are $1^{-1} = 1, 2^{-1} = 3, 3^{-1} = 2, 4^{-1} = 4$.

Why are we so worried about whether numbers have multiplicative inverses? Mathematicians have all kinds of good theoretical reasons for being interested in this, but in cryptography there are very concrete reasons: suppose, for instance, that we represent the letters in the English alphabet by the numbers $0, 1, \dots, 25$, that is, using the numbers in Z_{26} . Now, does it make sense to encrypt letter nr. i by replacing it by $ik \bmod 26$, for some secret k ?

The answer is that it does, but only if $k \in Z_{26}^*$, that is, if it has a multiplicative inverse. The reason is that this is necessary to be able to decrypt: given $ik \bmod n$ we can find the original i by multiplying by $k^{-1} \bmod 26$. If this inverse does not exist, there is NO way to reconstruct the original data. Of course, simply multiplying by a single element is hardly a secure way to encrypt by itself, but still, this sort of operation is an important ingredient in many good ciphers.

3.4 Finite Fields

Our old friend, the real numbers, has two basic calculation operations: we can add and multiply (subtraction and division can be thought of as the inverse of the basic operations). Here are some important properties that characterize the way these operations behave: we have special elements $0, 1$ that are neutral w.r.t. addition and multiplication, respectively. The real numbers form a group w.r.t. addition: after we add a number a , we can always subtract a and get back where we started. Moreover, the real numbers except 0 , form a group w.r.t. multiplication: if we multiply by a non-zero number a , we can multiply by a^{-1} and get back where we started.

It turns out that many other sets have an addition and a multiplication operation that behave in this way, and therefore in these domains, we can do lots of the things we can do with numbers, and this turns out to be important in cryptography, and in many other areas of research. It is therefore very useful to do the same type of abstraction that we did for groups:

Definition 3.4 Consider a set \mathbb{F} with two operations $+$ and \cdot . We say that $(\mathbb{F}, +, \cdot)$ is a field, if

- $(\mathbb{F}, +)$ is a commutative group, that is, we have all the properties from Definition 3.1, and in addition $a + b = b + a$ for all $a, b \in \mathbb{F}$. The neutral element for addition will be called 0 .

- Let $\mathbb{F}^* = \mathbb{F} \setminus \{0\}$. (\mathbb{F}^*, \cdot) is a commutative group, that is, we have all the properties from Definition 3.1, and in addition $a \cdot b = b \cdot a$ for all $a, b \in \mathbb{F}$. The neutral element for addition will be called 1.
- Multiplication distributes over addition, that is, for any $a, b, c \in \mathbb{F}$ we have $a \cdot (b + c) = a \cdot b + a \cdot c$

Here is a very important class of examples:

Theorem 3.5 *For any prime p , \mathbb{Z}_p with addition and multiplication modulo p as operations, is a field.*

This is very easy to verify using what we already said about \mathbb{Z}_p and \mathbb{Z}_p^* in the previous section. Try it!

Sometimes, when we want to emphasize the “field-ness” of \mathbb{Z}_p , we call it \mathbb{F}_p instead, but it is of course the same thing. This is an example of a finite field - since the underlying set is finite.

Finite fields are used all over the place in cryptography. For instance, many attacks on encryption schemes have been done by solving systems of linear equations over the field \mathbb{F}_2 . This is not hard to do, once you know how to do it with ordinary numbers. Suppose you know how to solve 2 equations with 2 unknowns, for instance, and you have 2 such equations over \mathbb{F}_2 . Then you simply manipulate the equations in exactly the same way you normally would, when working with real numbers. All you have to do is replace addition and multiplication of numbers with addition and multiplication in \mathbb{F}_2 . This is guaranteed to work because you are working in a field: to argue that linear equations can be solved easily, it turns out that you only need to assume the basic properties that characterize a field, as in the above definition.

Another notable example of application of fields in cryptography is that the Advanced Encryption Standard (AES) uses a finite field with 2^8 elements in a very essential way. Now, of course 2^8 is not a prime, but prime power. It turns out that for any prime power $q = p^k$, there is a finite field \mathbb{F}_q with q elements. This is called an extension field because you start from the field \mathbb{F}_p and then build \mathbb{F}_q by extending the arithmetic in \mathbb{F}_p .

We now explain how this works for the case of \mathbb{F}_{2^8} . The general case of \mathbb{F}_{p^k} can be found in any basic text book in algebra, but is also quite easy to extrapolate from what we say here.

In \mathbb{F}_2 , we add and multiply modulo 2. Since the elements in \mathbb{F}_2 are bits, 0 and 1, addition is the same as XOR and multiplication is the same as an AND operation. Now, \mathbb{F}_{2^8} will be the set of bytes, or 8-bit strings, of which there are indeed 2^8 . We want to extend the operations on \mathbb{F}_2 to this larger set. Addition is very simple: say we have bytes $A = a_7, a_6, \dots, a_0$, $B = b_7, b_6, \dots, b_0$, where $a_i, b_i \in \{0, 1\}$. Then we define addition as bit-wise XOR:

$$A + B = a_7 \oplus b_7, \dots, a_0 \oplus b_0$$

It is easy to see that \mathbb{F}_{2^8} with this addition is a group: the byte $O = (00000000)$ is the neutral element, and every byte is its own inverse.

Now to multiplication. For this, we will interpret a byte A as a polynomial $A(x) = a_7x^7 + a_6x^6 + \dots + a_0$. Then, just from standard multiplication of polynomials, we get that

$$A(x) \cdot B(x) = a_7b_7x^{14} + (a_7b_6 \oplus a_6b_7)x^{13} + \dots + (a_0b_1 \oplus a_1b_0)x + a_0b_0$$

Note that the coefficients are from \mathbb{F}_2 , and so we use the operations from \mathbb{F}_2 to compute on them. To emphasize this, we write $a_7b_6 \oplus a_6b_7$ and not $a_7b_6 + a_6b_7$.

This way, we have managed to make a new bitstring from A, B using operations from \mathbb{F}_2 that is in some sense the product of the two, namely the coefficients of $A(x) \cdot B(x)$. But it contains 15 bits and is therefore too long - the product of two elements in \mathbb{F}_{2^8} should, of course, return something in the same set.

So we need to get the length back down to 8 bits. To understand how to solve this, note that the issue is analogous to what happens when we multiply modulo 17, for instance (we choose a prime as example because arithmetic modulo a prime leads to a field). If we just multiply 8 and 7, for instance, we get 56 which is not immediately in \mathbb{Z}_{17} , it's too large. So we have to reduce modulo 17 to get $56 \bmod 17 = 5$. We do this by subtracting a multiple of 17 from 56, so that we get something that is smaller than 17, in this case 3 times 17 works.

So we now “translate” these ideas from numbers to polynomials: instead of reducing modulo a prime number, we will reduce modulo an irreducible polynomial, namely a polynomial that cannot be written as a product of polynomials of smaller degree. Note that this is similar to prime numbers: primes cannot be written as a product of smaller numbers. In our case, where we have polynomials with coefficients in \mathbb{F}_2 , it turns out that $F(x) = x^8 + x^4 + x^3 + x + 1$ is irreducible. The reason why we want a polynomial of degree 8 will become clear in a moment.

So now, similarly to standard modular reduction, the goal is to take our product $A(x)B(x)$ and subtract a multiple of $F(x)$ such that we get a “small enough” polynomial, which in our case means small enough degree. Recall that $A(x)B(x)$ has degree (at most) 14 with leading coefficient a_7b_7 . So therefore

$$A(x)B(x) - (a_7b_7)x^5F(x)$$

has degree at most 13. Note that we multiply by x^5 to shift up $F(x)$ so we kill the leading coefficient of $A(x)B(x)$

If we insert the actual expressions for $A(x), B(x), F(x)$, we see that new polynomial we have after the subtraction has leading coefficient $(a_7b_6 \oplus a_6b_7)$, so therefore

$$A(x)B(x) - (a_7b_7)x^5F(x) - (a_7b_6 \oplus a_6b_7)x^4F(x)$$

has degree at most 12.

We can continue this process to get smaller and smaller degree, until we have subtracted some coefficient times $F(x)$ itself, that is, without shifting up. At this point we have something of degree at most 7 and the process stops. The polynomial we have at the end is by definition $A(x)B(x) \bmod F(x)$ ¹.

¹ If you think this process seems a very similar to integer division, that's because it is. What we have

Now we are almost done. Note that a polynomial of degree 7, such as

$$C(x) = A(x)B(x) \bmod F(x)$$

can be described by exactly 8 coefficients:

$$C(x) = c_7x^7 + \dots + c_0,$$

so we can now define that the product in \mathbb{F}_{2^8} of A and B is $A \cdot B = C$ where C is a byte with bits c_7, c_6, \dots, c_0 .

So this is the reason why we choose to reduce modulo a polynomial of degree 8: this means we will have exactly 8 bits coming out of the modular reduction process.

Now, is it true that $(\mathbb{F}_{2^8}^*, \cdot)$ is a group, where $\mathbb{F}_{2^8}^* = \mathbb{F}_{2^8} \setminus \{0\}$? The neutral element is easy: Consider the byte $I = (00000001)$. Clearly $I(x) = 1$, the constant polynomial 1. So when we compute $I \cdot A$, we will have $I(x)A(x) \bmod F(x) = A(x) \bmod F(x) = A(x)$ since when you reduce something that already has degree at most 7, nothing happens, as is easy to see. Then we also need that every non-zero byte has a multiplicative inverse. This turns out to be true, exactly because we reduce modulo an irreducible polynomial. We will not give the proof here, but it is very similar to the proof that modulo a prime, every non-zero element has a multiplicative inverse.

Exercise 3.1 (Distributive law in \mathbb{F}_{2^8}) Show that for any $A, B, C \in \mathbb{F}_{2^8}$ we have $A(B + C) = AB + AC$.

This concludes the description of arithmetic in \mathbb{F}_{2^8} . Other finite extension field can be built in exactly the same way: to get \mathbb{F}_{p^k} for a prime p , we consider polynomials over \mathbb{F}_p of degree at most k . Addition is just addition of polynomials and multiplication is modular multiplication, modulo an irreducible polynomial of degree k .

For completeness, we include the theorem stating that the polynomial division we used above works in general. It can be shown by simply following the procedure we sketched above.

Theorem 3.6 (Polynomial Division) *Let $a(x), b(x)$ be polynomials over a field \mathbb{F} . Then we can divide $a(x)$ by $b(x)$ to get a quotient and a remainder, that is, there exist polynomials $q(x), r(x)$, such that $a(x) = q(x)b(x) + r(x)$, where the degree of $r(x)$ is less than the degree of $b(x)$*

From this follows easily a fundamental fact we will use several times:

Theorem 3.7 (Number of roots of a polynomial) *Let $f(x)$ be a polynomial over a field \mathbb{F} . A root of $f(x)$ is an element $\gamma \in \mathbb{F}$ such that $f(\gamma) = 0$. The number of distinct roots of f is at most the degree of $f(x)$.*

PROOF If γ_1 a root of $f(x)$, then the polynomial $x - \gamma_1$ divides $f(x)$: we can

described here is special case of dividing one polynomial by another to get a quotient and a remainder.

write $f(x) = q(x)(x - \gamma_1) + r(x)$, and the residue $r(x)$ has degree less than 1 so is it is a constant α . But if we insert a in the equation, we see that $\alpha = 0$. So $f(x) = q(x)(x - \gamma_1)$. Now, if $f(x)$ has another root γ_2 , then we have

$$0 = f(\gamma_2) = q(\gamma_2)(\gamma_1 - \gamma_2).$$

Since $(\gamma_1 - \gamma_2) \neq 0$ and we work in a field, we can multiply both sides by $(\gamma_1 - \gamma_2)^{-1}$ and we see that γ_2 is a root of $q(x)$. We can continue this process with all the distinct roots of $f(x)$, and we will get that $f(x) = q'(x) \prod_{i=1}^u (x - \gamma_i)$ where u is the number of distinct roots and $q'(x)$ is whatever remains when we have divided by all the $(x - \gamma_i)$'s. Now, the degree of $\prod_{i=1}^u (x - \gamma_i)$ is u , and when we multiply by $q'(x)$, the degree can only increase, so u is at most the degree of $f(x)$. \square

3.5 More about Groups: Lagrange's Theorem

For a finite group G , let $|G|$ be the order of G , that is, the number of elements in G .

As an example, consider \mathbb{Z}_7^* . Since 7 is a prime, all non-zero elements have inverses, so $|\mathbb{Z}_7^*| = 6$. In general, $|\mathbb{Z}_p^*| = p - 1$ for any prime p . For any m , $|\mathbb{Z}_m^*|$ is also written as $\phi(m)$, this is known as Euler's ϕ -function. We note without proof that if the prime factorization of m is $m = \prod_{i=1}^s p_i^{t_i}$ then

$$\phi(m) = \prod_{i=1}^s (p_i - 1)p_i^{(t_i-1)}.$$

Now, back to a general finite group G . Consider any $g \in G$, and look at the list of elements g, g^2, g^3, \dots . We claim that the neutral element 1 must be somewhere on this list. To see this, note that since G is finite, the list must at some point start to repeat itself. In other words, there must be some g^i , that equals some g^j that appears earlier on the list. In other words $g^i = g^j$, where $i > j$. This means that $g^{i-j} = 1$. But since $i - j > 0$, g^{i-j} is also one of the elements on the list, so we conclude that, as claimed, the neutral element 1 must be on the list. Therefore, we can define:

Definition 3.8 The order of $g \in G$ is the smallest natural number m for which $g^m = 1$. The order of g is denoted $|g|$.

Note that the elements $g, g^2, \dots, g^{|g|} = 1$ is a complete list of different powers of g in G . Going to higher powers generates nothing new, as $g^{|g|+1} = g^{|g|} \cdot g = g$. Even all negative powers of g are in this set. This is so because $g^{|g|-1}$ is on the list and $g^{|g|-1} \cdot g = g^{|g|} = 1$, so in fact $g^{|g|-1} = g^{-1}$. This means we can write all powers of g^{-1} as positive powers of g .

We write $\langle g \rangle = \{g, g^2, \dots, g^{|g|}\}$ and call it the (sub)group generated by g . The motivation for this name is that $\langle g \rangle$ is itself a group: the neutral element is in there, and it is easy to see that all the properties need for a group are satisfied.

As an example, consider $G = \mathbb{Z}_7^*$ and $g = 2$. The list of powers of 2 is

$$2, 2^2 \bmod 7 = 4, 2^3 \bmod 7 = 1.$$

So the order of 2 in \mathbb{Z}_7^* is 3. If we take $g = 6$ instead, we get $6, 6^2 \bmod 7 = 1$, so the order is 2. Similarly one can check that the order of 3 in \mathbb{Z}_7^* is 6. So we see that all the orders of single elements that we found divide the order of the whole group, namely 6. This is no coincidence, as shown by the following theorem:

Theorem 3.9 (Lagrange) *For any finite group G and any $g \in G$, it holds that $|g|$ divides $|G|$.*

PROOF We have, of course that $\langle g \rangle \subseteq G$. If $\langle g \rangle = G$, then $|g| = |G|$ and the conclusion of the theorem is of course true. So assume $\langle g \rangle \neq G$. This means we can choose some element $h_1 \in G \setminus \langle g \rangle$. Consider then the set $h_1 \cdot \langle g \rangle$, which is shorthand for $\{h_1 \cdot g, h_1 \cdot g^2, \dots, h_1 \cdot g^{|g|}\}$.

Now we make two important observations: First, $h_1 \cdot \langle g \rangle$ has the same number of elements as $\langle g \rangle$, simply because multiplying by h_1 is an injective mapping: if $g^i \neq g^j$, then $h_1 g^i \neq h_1 g^j$. Namely, if $h_1 g^i = h_1 g^j$ we could multiply by h_1^{-1} on both sides and get $g^i = g^j$, a contradiction. Second, $\langle g \rangle \cap h_1 \cdot \langle g \rangle = \emptyset$. If this was not the case, there would have to exist i, j such that $g^i = h_1 g^j$, namely an element in the intersection would have to be of both forms. But this would imply that $h_1 = g^{i-j}$ and therefore that $h_1 \in \langle g \rangle$, but we chose h_1 outside $\langle g \rangle$ so this is a contradiction.

These two observations imply that the number of elements in the join of the sets, $\langle g \rangle \cup h_1 \cdot \langle g \rangle$, is $2|g|$.

Now, it might be that $\langle g \rangle \cup h_1 \cdot \langle g \rangle = G$, in which case we are done, since $|G| = 2|g|$. If not, we can choose $h_2 \in G \setminus (\langle g \rangle \cup h_1 \cdot \langle g \rangle)$. and consider $h_2 \cdot \langle g \rangle$. By the same arguments as above, $\langle g \rangle \cup h_1 \cdot \langle g \rangle \cup h_2 \cdot \langle g \rangle$ has $3|g|$ elements. We can continue adding subsets this way, but the process must stop at some point because G is finite. When it stops, we can conclude that $|G| = t|g|$ for some natural number t . \square

The main take-home message in this section now is contained in the two following easy consequences of this theorem:

Corollary 3.10 *For any finite group G and any $g \in G$, it holds that $g^{|G|} = 1$*

PROOF From the above theorem, $|G| = t|g|$ for some t and hence $g^{|G|} = g^{t|g|} = (g^{|g|})^t = 1^t = 1$. \square

Corollary 3.11 *For any finite group G any $g \in G$ and any integer i , it holds that $g^i = g^{i \bmod |G|} = g^{i \bmod |g|}$.*

PROOF Since $i = i \bmod |G| + s|G|$ for some s , we get

$$g^i = g^{i \bmod |G| + s|G|} = g^{i \bmod |G|} g^{s|G|} = g^{i \bmod |G|} (g^{|G|})^s = g^{i \bmod |G|},$$

where the last step follows from the previous corollary. The fact that $g^i = g^{i \bmod |g|}$ follows by a similar argument. \square

It is helpful to collect what these corollaries say in a couple of “rules of thumb” for groups. The first corollary says;

In a group G , when you raise to power $|G|$, you always get 1.

The second corollary says that, when working in a group G , you can always reduce modulo $|G|$ in the exponent and nothing will change. Put differently:

In a group G , arithmetic in the exponent happens modulo $|G|$.

3.6 More Number Theory

3.6.1 The Chinese Remainder Theorem

Consider a number $n = pq$ where $\gcd(p, q) = 1$. This will be the case, for instance, if p and q are distinct primes. There is a very natural function that takes numbers in \mathbb{Z}_n to pairs of numbers in \mathbb{Z}_p and \mathbb{Z}_q , respectively, in other words, to $\mathbb{Z}_p \times \mathbb{Z}_q$. This function f_n does the following:

$$f_n(x) = (x \bmod p, x \bmod q).$$

f_n has a number of very nice properties. First, it is injective, and hence it is also surjective because \mathbb{Z}_n and $\mathbb{Z}_p \times \mathbb{Z}_q$ have the same number of elements, namely $(p-1)(q-1)$. We will show it is injective below by demonstrating that there exists an easily computable inverse function f_n^{-1} . If we can always reconstruct x from $f_n(x)$, it cannot be that f_n sends different inputs to the same output.

The second property is that f is a *homomorphism*, with respect to both addition and multiplication. To state this more precisely we define some shorthand:

$$x +_n y = (x + y) \bmod n$$

for $x, y \in \mathbb{Z}_n$. And define, for $(a, b), (c, d) \in \mathbb{Z}_p \times \mathbb{Z}_q$:

$$(a, b) +_{p,q} (c, d) = ((a + c) \bmod p, (b + d) \bmod q).$$

We also define multiplication operations \cdot_n and $\cdot_{p,q}$ in exactly the same way, as multiplication modulo n and multiplication of pairs modulo p and q . The wonderful properties of f are:

$$f_n(x) +_{p,q} f_n(y) = f_n(x +_n y) \text{ and } f_n(x) \cdot_{p,q} f_n(y) = f_n(x \cdot_n y) \quad (3.1)$$

To see why (3.1) is true, we just need the basic fact that because p divides n , reducing modulo n and then modulo p gives the same result as just reducing modulo p : for any integer u we have

$$(u \bmod n) \bmod p = (u + tn) \bmod p = u \bmod p + tpq \bmod p = u \bmod p. \quad (3.2)$$

With this, we can calculate as follows:

$$\begin{aligned} f_n(x +_n y) &= f_n((x + y) \bmod n) \\ &= (((x + y) \bmod n) \bmod p, ((x + y) \bmod n) \bmod q) \\ &= ((x + y) \bmod p, (x + y) \bmod q) \\ &= ((x \bmod p + y \bmod p) \bmod p, (x \bmod q + y \bmod q) \bmod q) \\ &= (x \bmod p, x \bmod q) +_{p,q} (y \bmod p, y \bmod q) \\ &= f_n(x) +_{p,q} f_n(y) \end{aligned}$$

An exactly similar calculation shows the second part of (3.1), for multiplication.

An example: let's look at $n = 15$ so that $p = 3, q = 5$. And say we want to multiply 7 and 8 modulo 15. This is $56 \bmod 15 = 11$. On the other hand,

$$f_{15}(7) = (7 \bmod 3, 7 \bmod 5) = (1, 2) \text{ and } f_{15}(8) = (2, 3).$$

Now, it is easy to calculate that $(1, 2) \cdot_{3,5} (2, 3) = (2, 1)$. And, indeed, $f_{15}(11) = (11 \bmod 3, 11 \bmod 5) = (2, 1)$.

This already suggests one way in which the function f_n is useful: if we need to calculate some expression with multiplications and additions modulo n , we do not have to do it mod n directly. We can instead apply f_n to all the inputs and do the intended calculations modulo both p and q . Potentially, this is easier and faster because p and q are smaller than n . Finally, we use f_n^{-1} to get the result modulo n .

Now, the only missing link is to specify the inverse function f_n^{-1} . So we are given $(x_p, x_q) \in \mathbb{Z}_p \times \mathbb{Z}_q$ and we need to construct $x \in \mathbb{Z}_n$ such that $x \bmod p = x_p$ and $x \bmod q = x_q$.

To this end, note that since $\gcd(p, q) = 1$, both $p^{-1} \bmod q$ and $q^{-1} \bmod p$ exist. Now, the integer $a_q = p \cdot (p^{-1} \bmod q)$ has the nice properties that

$$a_q \bmod p = 0 \text{ and } a_q \bmod q = 1,$$

because it is a multiple of p and since modulo q , p and p^{-1} cancel. Likewise $a_p = q \cdot (q^{-1} \bmod p)$ has similar properties: $a_p \bmod q = 0$ and $a_p \bmod p = 1$. We can now define the inverse function:

$$f_n^{-1}(x_p, x_q) = (x_p a_p + x_q a_q) \bmod n.$$

To see that this is correct, we need to verify that $f_n(f_n^{-1}(x_p, x_q)) = (x_p, x_q)$. What this means more concretely is that if we reduce the claimed value for $f_n^{-1}(x_p, x_q)$ modulo p and q , we should get x_p and x_q . Modulo p , we get

$$\begin{aligned} f_n^{-1}(x_p, x_q) \bmod q &= ((x_p a_p + x_q a_q) \bmod n) \bmod q \\ &= (x_p a_p + x_q a_q) \bmod q \\ &= x_p a_p \bmod q + x_q a_q \bmod q \\ &= x_q \end{aligned}$$

and similarly modulo p .

If we consider again our example with $n = 15$, we can see that $3^{-1} \bmod 5 = 2$ and $5^{-1} \bmod 3 = 2$. This means that, using our notation above that

$$a_3 = 5 \cdot (5^{-1} \bmod 3) = 10 \text{ and } a_5 = 3 \cdot (3^{-1} \bmod 5) = 6.$$

Recall that our multiplication of 7 and 8 modulo 3 and 5 resulted in $(2, 1)$. Mapping this back to \mathbb{Z}_{15} we get

$$f_{15}^{-1}(2, 1) = (2 \cdot 10 + 1 \cdot 6) \bmod 15 = 26 \bmod 15 = 11,$$

as expected.

We summarize what we have seen so far: Because we have a surjective and

injective homomorphism from \mathbb{Z}_n to $\mathbb{Z}_p \times \mathbb{Z}_q$, we say they are *isomorphic*. and the homomorphism is said to an isomorphism. So we have:

Theorem 3.12 (The Chinese Remainder Theorem, special case) *Let $n = pq$ where $\gcd(p, q) = 1$. Then \mathbb{Z}_n is isomorphic to $\mathbb{Z}_p \times \mathbb{Z}_q$, where the isomorphism is given by*

$$f_n(x) = (x \bmod p, x \bmod q).$$

The inverse function is given by

$$f_n^{-1}(x_p, x_q) = (x_p a_p + x_q a_q) \bmod n,$$

where $a_q = p \cdot (p^{-1} \bmod q)$ and $a_p = q \cdot (q^{-1} \bmod p)$.

It will become clear later that this theorem is a very important key to understanding how RSA encryption works. If you get stuck with a problem or an exercise concerning RSA, most likely this is because you forgot to apply the Chinese Remainder Theorem!

It should be mentioned that the theorem is not limited to cases where we split n in 2 factors. We state the general case here for completeness:

Theorem 3.13 (The Chinese Remainder Theorem, general case) *Let $n = \prod_{i=1}^t p_i$ where $\gcd(p_i, p_j) = 1$ for $i \neq j$. Then \mathbb{Z}_n is isomorphic to $\mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \dots \times \mathbb{Z}_{p_t}$, where the isomorphism is given by*

$$f_n(x) = (x \bmod p_1, x \bmod p_2, \dots, x \bmod p_t).$$

One can derive a similar expression for the inverse function as in Theorem 3.12, we leave this to the reader

3.6.2 The structure of \mathbb{Z}_p^*

We will need the following results later, in order to analyze both RSA and discrete log based cryptosystems.

Theorem 3.14 *For any prime p , \mathbb{Z}_p^* is a cyclic group, that is, there exists a generator $g \in \mathbb{Z}_p^*$ such that $\mathbb{Z}_p^* = \langle g \rangle$. In other words, any number in \mathbb{Z}_p^* can be written as $g^i \bmod p$ for some i .*

PROOF Rephrasing, we want to show that we can find g such that its order $|g|$ is $p - 1$. Any element has some order, so we can at least choose g such that its order is maximal: no other element has larger order. Now, if $\langle g \rangle$ is all of \mathbb{Z}_p^* we are done. If not, we could choose $h \notin \langle g \rangle$. We claim that it must be the case that $|h|$ divides $|g|$. If not, consider the element hg . It is not hard to see that, in general, the order of such a product is always the least common multiple of $|g|$ and $|h|$. But if $|h|$ does not divide $|g|$, then $|hg| = \text{lcm}(|g|, |h|) > |g|$ contradicting the fact that g has maximal order. Consider now the equation $x^{|g|} - 1 = 0$. Any element in $\langle g \rangle$ is a solution to this equation, and so is any element in $\langle h \rangle$: we just showed that $|h| \mid |g|$. So the number of solutions is certainly more than $|g|$.

Now, recall that \mathbb{Z}_p is a field as we saw earlier. Hence, by Theorem 3.7 the equation $x^{|g|} - 1 = 0$ can have at most $|g|$ solutions. But we just found more than $|g|$ solutions, so we have a contradiction. It follows that an element of maximal order is a generator \square

Lemma 3.15 *Let g be a generator of \mathbb{Z}_p^* . Then $(g^i)^{(p-1)/2} \bmod p \equiv 1$ if i is even and $(g^i)^{(p-1)/2} \bmod p \equiv -1$ if i is odd.*

PROOF Clearly we have for any $\gamma \in \mathbb{Z}_p^*$ that $\gamma^{(p-1)/2} \bmod p$ is congruent 1 or -1 – this follows since $(\gamma^{(p-1)/2})^2 \bmod p = 1$ by Langrange, and because \mathbb{Z}_p is a field, the quadratic equation $x^2 = 1 \bmod p$ can have at most 2 solutions, namely 1 and -1 (see Theorem 3.7).

Therefore since g is a generator, it must be that $g^{(p-1)/2} \bmod p$ is congruent to -1: it cannot be 1 since then its order would be less than $p-1$.

We can now rewrite the result we want as follows:

$$(g^i)^{(p-1)/2} \bmod p = (g^{(p-1)/2})^i \bmod p = (-1)^i \bmod p,$$

which is clearly 1 if i is even and -1 otherwise. \square

Lemma 3.16 *For any $a \in \mathbb{Z}_p^*$, we have that $a^{(p-1)/2} \bmod p$ is congruent to 1 or -1 $\bmod p$. If we choose a uniformly at random in \mathbb{Z}_p^* , then*

$$P[a^{(p-1)/2} \bmod p = 1] = 1/2.$$

PROOF let g be a generator of \mathbb{Z}_p^* . Thus $a = g^i \bmod p$ for some i . The lemma now follows from the previous one, because half the relevant powers of g are even. \square

Symmetric Cryptosystems and Their History

Contents

4.1	Symmetric Cryptosystems	23
4.2	An Example: the Shift Cipher	24
4.3	Attacks on Symmetric Cryptosystems	24
4.3.1	Attacking the Shift Cipher	25
4.4	More Historic Examples	26
4.4.1	The Affine Cipher	27
4.4.2	General Substitution	27
4.4.3	The Vigenere Cipher	28
4.4.4	Homophonic Substitution	30
4.4.5	The Permutation Cipher	30
4.4.6	The insight of Shannon	31
4.5	Exercises	31

4.1 Symmetric Cryptosystems

A cryptosystem is a triple (G, E, D) of algorithms, for key generation, encryption and decryption. In this chapter, we look at *symmetric* cryptosystems where the information you need to encrypt a message is the same as what you need to decrypt. For a symmetric system, there are 3 finite sets given, namely the key space \mathcal{K} , the plaintext space \mathcal{P} and the ciphertext space \mathcal{C} .

G , algorithm for generating keys: This algorithm is probabilistic, takes no input and always outputs a key $K \in \mathcal{K}$, usually G simply outputs a key chosen uniformly from \mathcal{K} .

E , algorithm for encryption: this algorithm takes as input K and $x \in \mathcal{P}$ and produces as output $E_K(x) \in \mathcal{C}$. Note that E may be probabilistic, that is, even though we fix x and K , many different ciphertexts may be produced as output from E , as a result of random choices made during the encryption process. In other words, the ciphertext will have a probability distribution that is determined from x and K , typically uniform in some subset of the ciphertexts.

D , algorithm for decryption: this algorithm takes as input $K, y \in \mathcal{C}$ and pro-

duces as output $D_K(y) \in \mathcal{P}$. It is allowed to be probabilistic, but is in most cases deterministic.

For a cryptosystem, we always require that for any key K output by G , correct decryption is possible, i.e. it holds that for any $x \in \mathcal{P}$, $x = D_K(E_K(x))$.

This of course says nothing about security. According to this definition, a system that "encrypts" by sending x to itself is a cryptosystem. Indeed it is, but of course not a secure one!

4.2 An Example: the Shift Cipher

This section assumes you are familiar with the material in Section 3.1. The *Shift Cipher* is also known as Caesar substitution. It works on an alphabet containing n symbols that we number from 0 to $n - 1$, corresponding to the elements in \mathbb{Z}_n . For instance, for English we would have $n = 26$ corresponding to the 26 letters in the English alphabet, where a corresponds to 0, b to 1, and so on. We set $\mathcal{K} = \mathcal{P} = \mathcal{C} = \mathbb{Z}_n$ and the system works as follows:

Key Generation Output K chosen uniformly at random from \mathcal{K} .

Encryption Set $E_K(x) = (x + K) \bmod n$.

Decryption Set $D_K(y) = (y - K) \bmod n$.

It should be clear that decryption works: if we first add K and then subtract it, nothing has changed.

Historically, this cipher was used by encrypting each letter in a message using the same key. This clearly leads to a very insecure solution as we explain in more detail later. The legend says that the roman emperor Caesar used the cipher, always setting $K = 3$, of course even worse as far as security is concerned! The cipher has also been used on rune stones found in Scandinavia, of course here based on the rune alphabet.

4.3 Attacks on Symmetric Cryptosystems

An attacker against a system in real life may follow a number of different strategies. The easiest option usually is to eavesdrop encrypted communication. But more advanced attackers may be able to guess (or get hold of) part of the plaintext that is being sent and compare this to the ciphertext that is seen. An even more harmful strategy is if the attacker can fool the sender into sending a particular message (chosen by the attacker), for which the attacker can then later see the ciphertext. Conversely, the attacker may be able to fool the receiver into decrypting a particular ciphertext for him.

To model these forms of attacks in a formal definition, we think of the adversary as a probabilistic algorithm A . To model the data the adversary obtains during the attack, we give A access to an *oracle* O . At any time, A may call the oracle, and obtain an answer. Depending on which type of question we allow the oracle to answer, we can model the different forms of attack we sketched before:

Ciphertext Only Attack: Some plaintext distribution D is fixed and the algorithm A may depend on D . For example, it may be that the adversary knows the plaintext is written English. This fixes, although indirectly, a distribution: some strings such as “the” have a high probability, while strings of letters that are not meaningful English have probability 0. Each time A calls the oracle, it will return $E_K(x)$, where x is chosen according to D , and K is produced by G (but fixed for the duration of the attack).

Known Plaintext Attack: Some plaintext distribution D is fixed and the algorithm A may depend on D . Each time A calls the oracle, it will return $x, E_K(x)$, where x is chosen according to D , and K is produced by G (but fixed for the duration of the attack). This models cases where the adversary can guess with good probability what (part of) the plaintext is, for instance, the plaintext belongs in some system where messages always start with a certain string, say to obey some formatting rule.

Chosen Plaintext Attack: A can call the oracle giving it any $x \in \mathcal{P}$ as input. The oracle returns $E_K(x)$, where K is produced by G (but fixed for the duration of the attack). This models a case where the adversary can influence the sender into sending some message of the adversary’s choice. For instance, if the adversary sends an invoice to the sender, this may cause him to send an encrypted payment order to his bank, containing the amount that the adversary chose.

Chosen Ciphertext Attack: A can call the oracle giving it any $y \in \mathcal{C}$ as input. The oracle returns $D_K(y)$, where K is produced by G (but fixed for the duration of the attack). This models, for instance, a case where the adversary sends a ciphertext he constructed to the receiver. Now, by monitoring the receiver’s behavior after decryption, he may get information on the result of the decryption.

Note that for the chosen plaintext and ciphertext attacks, we do not fix any distribution of plaintexts. This is because the adversary chooses the inputs to the oracle himself, and he is free to do so, no matter what the plaintexts look like in the actual application.

When A stops, it outputs some result. In the best case for the adversary, this result is the secret key, but we can (and indeed we must) also consider adversaries with much less ambitious goals, such as computing some partial information on an unknown plaintext. More on this in the following sections.

4.3.1 Attacking the Shift Cipher

To get an initial understanding of the difference between types of attacks, consider the Shift Cipher, where the same key is being used to encrypt large pieces of text. Breaking this cipher under a ciphertext only attack is not hard, but requires a little work:

One possibility is to do what is known as exhaustive search for the key, namely try to decrypt using every key possible, one can reasonably hope that only the

letter	frequency	letter	frequency
a	.082	n	.067
b	.015	o	.075
c	.028	p	.019
d	.043	q	.001
e	.127	r	.060
f	.022	s	.063
g	.020	t	.091
h	.061	u	.028
i	.070	v	.010
j	.002	w	.023
k	.008	x	.001
l	.040	y	.020
m	.024	z	.001

Figure 4.1 Approximate frequencies of single letters in English

correct key will lead to a meaningful plaintext. This is certainly doable with real life alphabets such as the English one, but is a bit tedious when done by pencil and paper, as would have been the case historically.

A smarter way is to do frequency analysis, where we exploit the fact that the frequency of letters is very far from uniform. For instance, the letter *e* is much more common than any other letter in almost all European languages, see Figure 4.1. What we can therefore do is to count the number of times each letter occurs in the ciphertext. The most common letter is very likely the letter that stands for *e*. Taking English as an example, say we find that the most common letter in the ciphertext is *b*, corresponding to the number 1 modulo 26. Since *e* corresponds to the number 4, we have that the key *K* satisfies

$$1 = 4 + K \bmod 26$$

from which we conclude that $K = 1 - 4 = -3 = 23 \bmod 26$.

The same cipher is even easier to break under a known plaintext attack: just knowing a single plaintext letter and corresponding ciphertext letter immediately reveals the key, because it gives us exactly an equation determining *K* like the one we just solved. A chosen plaintext attack is in general stronger, but in this case, where the cipher is already very weak, it is not more useful to the attacker than a known plaintext attack.

4.4 More Historic Examples

In this section, we give more examples of historic cryptosystems. They are all insecure, but still serve well to illustrate the concepts we have seen, and importantly they also illustrate the problems that insecure ciphers have, which we must avoid when designing good encryption. This section assumes you are familiar with the material in Section 3.2.

4.4.1 The Affine Cipher

This is a variant of the shift cipher that was (incorrectly) thought to be more secure because it is more complicated. We set $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$, $\mathcal{K} = \mathbb{Z}_n^* \times \mathbb{Z}_n$ and the system works as follows:

Key Generation Output $K = (a, b)$ chosen uniformly at random from \mathcal{K} .

Encryption Set $E_K(x) = ax + b \bmod n$.

Decryption Set $D_k(y) = a^{-1}(y - b) \bmod n$.

To see that decryption works, consider that $D_{a,b}(E_{a,b}(x)) = a^{-1}((ax + b) - b) = a^{-1}ax = x \bmod n$.

This cipher is also easy to break under a ciphertext only attack, though a bit more work is needed. Say we do a frequency analysis on the ciphertext and find that the most common letter is y_1 , while the next-most common letter is y_2 . A reasonable guess would be that if x_1, x_2 are the most common letters in plaintext, then x_1 encrypts to y_1 and x_2 encrypts to y_2 . If this guess is correct, we have two equations:

$$y_1 = ax_1 + b \bmod n, \quad y_2 = ax_2 + b \bmod n$$

We can think of this as 2 equations with 2 unknowns, namely a, b , and try to solve them just like one would solve ordinary equations on, say, real numbers. For instance, we can use the first equation to get $b = y_1 - ax_1 \bmod n$ and insert for b in the other equation to get $y_2 - y_1 = a(x_2 - x_1) \bmod n$. Now we have to hope that $x_2 - x_1$ is invertible modulo n . If it is, we can conclude that

$$a = (y_2 - y_1)(x_2 - x_1)^{-1} \bmod n, \quad b = y_1 - ax_1 \bmod n.$$

At this point, there are several possibilities: if we failed to invert $x_2 - x_1$ we need to try another guess, say at the third-most common letter. If we were able to solve the equations, the resulting (a, b) may or may not be correct. We can test by decrypting some of the plaintext and check if it is meaningful. If the test fails, again we have to try a new guess.

4.4.2 General Substitution

This cipher is the most general attempt to encrypt by substituting a letter from the alphabet by another letter. We set $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$, $\mathcal{K} = \Sigma_n$ where Σ_n is the set of all permutations of n items. The system works as follows:

Key Generation Output $K = \pi$ chosen uniformly at random from \mathcal{K} .

Encryption Set $E_K(x) = \pi(x) \bmod n$.

Decryption Set $D_k(y) = \pi^{-1}(y) \bmod n$.

Breaking this cipher under ciphertext-only is even harder work, but usually possible. Using frequency analysis and assuming English plaintext, we can make reasonable guesses at the encryption of the most common plaintext letters. Say we guess that e is encrypted to U and t is encrypted to X , where we use capital letters to denote ciphertext.

Compared to the previous cryptosystems, the difficulty is that even if we were promised that the guess is correct, this tells us nothing about how less common letters encrypt, because the permutation used as key is completely random. Given our initial guess, the plaintext letter h , for instance, could be encrypted to anything – except of course U and X . Or could it? We can be more clever and use the fact that also pairs and triples of letters are far from uniformly distributed in natural languages. For instance, say we find a few occurrences of the string UAX in the ciphertext. Give our initial guess and the fact that *the* is a very common English word, a good guess is that h encrypts to A . There are many tricks of this type, for instance in English, q is always followed by u .

Using this type of reasoning, a general substitution can usually be broken even just using pencil and paper, as long as enough ciphertext is available. This is true despite the fact that general substitution has a very large number of keys, namely there are $n!$ possible permutations of an alphabet with n letters. For English this is $26! \approx 2^{88}$ which is large enough that even on a computer it is infeasible to try all possible keys.

This demonstrates an important fact one needs to be aware of: a large number of keys is necessary for security, but a large number of keys is no guarantee for security in itself.

4.4.3 The Vigenere Cipher

This cipher was invented around 1553 and was first described by Giovan Bellaso. It has later incorrectly been attributed to Blaise de Vigenere who lived much later.

It was invented in response to the fact that substitution ciphers are insecure, as we have seen, due to the fact that the frequency distribution of plaintext survives in the ciphertext. The idea therefore is to combine several substitution ciphers, as follows:

We set $\mathcal{P} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_n^t$, where t is a parameter called the *block size* which is secret. So, in this system keys, plaintexts and ciphertexts are t -tuples of letters.

Key Generation Output $K = (k_1, \dots, k_t)$ chosen uniformly at random from \mathcal{K} .

Encryption Set $E_{k_1, \dots, k_t}(x_1, \dots, x_t) = (x_1 + k_1) \bmod n, \dots, (x_t + k_t) \bmod n$.

Decryption Set $D_{k_1, \dots, k_t}(y_1, \dots, y_t) = (y_1 - k_1) \bmod n, \dots, (y_t - k_t) \bmod n$.

It should be clear that decryption is correct, for the same reason that the Shift cipher is correct. To encrypt a long text, the plaintext is split in t -letter blocks and each block is encrypted as above.

The big advantage of this cryptosystem is that if t is large enough, the frequency distribution of ciphertext will be almost completely flat, and so none of the attacks we have seen so far will work. The distribution is flat because frequent letters in plaintext are unlikely to always occur in the same position in every block and hence they will be encrypted using different values from the key and will be mapped to different ciphertext letters.

The Vigenere cipher was unbroken for centuries after its invention, and it was

only in 1863 that German military officer Kassiski published the first general attack. A bit later, Danish mathematician Julius Petersen independently broke Vigenere in 1875.

The crucial observation towards breaking Vigenere is that if we can figure out the block size, then the rest is easy: suppose we know t . Then we can split the ciphertext y_1, \dots, y_N in t groups, G_1, \dots, G_t . G_1 will contain $y_1, y_{t+1}, y_{2t+1}, \dots$, and in general G_i contains $y_i, y_{t+i}, y_{2t+i}, \dots$. The point is that all letters in G_i have been encrypted using the letter k_i from the key, and we can think of this as encrypting letters $x_i, x_{t+i}, x_{2t+i}, \dots$ from the plaintext using a Shift cipher. We can break this shift cipher using frequency analysis exactly as we have seen above.

This suggests the following simple algorithm:

1. Set $t = 1$.
2. Test the ciphertext to see if the block size is t .
3. If, yes, then break the cipher as above and exit. Else, set $t = t + 1$ and go to 2.

Of course, the question is how we do the test in step 2? The answer is that if, for any given value of t , we split the ciphertext in groups G_1, \dots, G_t as explained above, then if t is the correct block size, then the frequency distribution in each block will resemble that of plaintext, so for European languages for instance, there will be a big spike corresponding to plaintext letter e . But if t not correct, then each group will contain a mix of letters that were encrypted using different letters from the key, and so the frequency distribution will tend to be more flat. So we get the following more refined version of the algorithm:

1. Set $t = 1$.
2. Split the ciphertext in groups G_1, \dots, G_t . If every group has a non-flat frequency distribution, then conclude that t is the correct block size, else conclude it is incorrect.
3. If t is correct, then break the cipher as above and exit. Else, set $t = t + 1$ and go to 2.

For standard languages, one can do the flatness test easily by hand by simply looking for the spike corresponding to plaintext letter e . If it is there in all groups we can assume we have the right blocksize and the position of the spike also tells us what the key is (this is what Julius Petersen suggested in 1875). In the 1930-ties, Friedman suggested a more systematic “measure of flatness” called the *Index of Coincidence* (IC), that lends itself more easily to programming.

Say we have a probability distribution p_0, \dots, p_{n-1} on the letters of our alphabet, then we define

$$IC(p_0, \dots, p_{n-1}) = \sum_{j=0}^{n-1} p_j^2$$

The IC-value is the probability that two letters drawn independently from the distribution are the same. Namely it is the probability that we got the first letter twice (p_0^2) plus the probability we got the second letter twice (p_1^2) and so on.

Intuitively, one expects that if the distribution is highly concentrated on one

outcome, such as the letter e in English, then the IC will be large, namely the experiment will return 2 e 's a lot of the times. Whereas the IC should be smaller for a uniform distribution where $p_j = 1/n$ for all j .

Indeed, one can compute that for the English alphabet, IC for the uniform distribution is $IC(1/26, \dots, 1/26) = 0.038$, while IC for the English letter frequency distribution from Figure 4.1 is approximately 0.065.

We can therefore do the flatness test for a group of letters G_i in the algorithm by computing the frequencies f_0^i, \dots, f_{n-1}^i of ciphertext letters in the group. Then we compute $IC(G_i) = IC(f_0^i, \dots, f_{n-1}^i)$ using the formula above. Now, if all $IC(G_i)$ are closer to 0.065 than to 0.038, then we believe we have the correct block size.

4.4.4 Homophonic Substitution

This cipher is interesting mostly because of the story that relates to it, so we do not give many details. The idea is that \mathcal{P} is the ordinary alphabet Z_{26} , whereas \mathcal{C} is, for instance, Z_{100} , i.e., a much larger set than \mathcal{P} . The key then consists of information that splits \mathcal{C} into 26 subsets A_0, A_1, \dots, A_{25} , and to encrypt letter number i , we choose a random element in A_i . This is called homophonic substitution, and the original idea historically was to choose larger subsets for encryptions of more common letters, in order to make the frequency distribution of ciphertext letters more flat. In fact, by choosing the size of A_i to be inversely proportional to the probability that letter i occurs in plaintext, the ciphertext distribution can be made exactly flat.

The cipher can still be broken using the distribution of pairs and triples of letters, similarly to general substitution.

This cipher was used by the infamous “Zodiac” serial killer who haunted the Bay area in the US around 1970, targeting at least 7 people. He was never caught, and sent several messages to newspapers in the area demanding that they publish them. The messages were a mix of plaintext and encryptions, and at least one ciphertext (the only one that was broken) used a homophonic substitution.

4.4.5 The Permutation Cipher

This final historic cipher is also a very old idea, where one tries to reorder symbols instead of substituting with others. We set $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n^t$, where t is a parameter called the *block size* which is (sometimes but not always) secret. We also set $\mathcal{K} = \Sigma_n$. So, in this system keys, plaintexts and ciphertexts are t -tuples of letters.

Key Generation Output $K = \pi$ chosen uniformly at random from \mathcal{K} .

Encryption Set $E_\pi(x_1, \dots, x_t) = x_{\pi(1)}, \dots, x_{\pi(t)}$.

Decryption Set $D_\pi(y_1, \dots, y_t) = y_{\pi^{-1}(1)}, \dots, y_{\pi^{-1}(t)}$.

It should be clear that decryption is correct. To encrypt a long text, the plaintext is split in t -letter blocks and each block is encrypted as above.

If t is known and enough blocks are available, one break this cipher using the

uneven distribution of pairs and triples of letters: if, for instance, a Q and a U is found in the same ciphertext block on positions i and j , there is a good chance that they should be moved together to form a pair QU (in English a q is always followed by a u). If this is correct, it means that in all other blocks a similar permutation should apply, because the key is the same for all blocks, that is, they should also have letters number i and j moved together. We can then test if the pairs formed in this way in other blocks are likely. Using this approach, permutation ciphers with known block size are not very hard to break in practice.

Unknown block size is of course harder, but one can always try a similar principle as for Vigenere: we know we can break it for known t , whereas of course we will fail for incorrect t , so we simply try to break the cipher assuming $t = 1, 2, \dots$ until we succeed. This is feasible because the number of possible t -values is not overwhelming.

4.4.6 The insight of Shannon

In the late 40-ties, Claude Shannon introduced information theory and also laid the foundations of modern cryptosystems. What he suggested was basically that if both permutation and substitution are not enough on their own, we should do both to get a secure cipher! More concretely, he suggested what is in modern terms called a substitution-permutation network: first we substitute symbols in the plaintext input by other symbols, in a way controlled by the key. Then we permute the symbols. We continue in this way, substituting and permuting some number of times, using possibly new substitutions and permutations in each iteration. Shannon conjectured that if we do this enough times, we will get a secure cryptosystem.

Of course, saying how many iterations are enough is not easy, but nevertheless Shannon's insight was striking: As we shall see in Chapter 6, virtually all modern symmetric cryptosystems are based on his principle.

4.5 Exercises

Exercise 4.1 (Attacking Historic Ciphers) Find the plaintext corresponding to the following ciphertexts, encrypted using various historic ciphers. You should give a clear account of the statistical analysis and computations that you did.

1. Affine Cipher (the plaintext is in French, but you can still use English letter frequencies):

```
KQEREJEBCPPCJCRKIEACUZBKRVPKRBCIBQCAREBJCVFCUP
KRIOFKPACUZQEPBKRXPETIEABDKPBCPFCDCCAFIEABDKP
BCPFQPKAZBKRHAIBKAPCCIBURCCDKDCCJCIDFUIXPAFF
ERBICZDFKABICBBENEFUCUPJCVKABPCYDCCDPKBCOCPERK
IVKSCPICBRKIJPKABI
```

2. General substitution(hint: F decrypts to w):

EMGLOSUDCGDNCUSWYSFHNSFCYKDPUMLWGYICOXYSIPJCK
QPKUGKMGOLICGINCGACKSNISACYKZSCKXECJCKSHYSXCG
OIDPKZCNKSHICGIWYGKKGKGOLDSILKGOIUSIGLEDSPWZU
GFZCCNDGYYSFUSZCNXEOJNCGYEOWEUPXEZGACGNFGLKNS
ACIGOIYCKXCJUCIUZCFZCCNDGYYSFEUEKUZCSOCFZCCNC
IACZEJNCSEHFZEJZEGMXCYHCJUMGKUCY

3. Unspecified Cipher:

BNVNSIHHQCEELSSKKYERIFJKXUMBGYKAMQLJTYAVFBKVT
DVBPVVRJYYLAOKYMPQSCGDLFSRLLPROYGESEBUUALRWXM
MASAZLGLEDJBZAVVPXWICGJXASCBYEHOSNMULKCEAHTQ
OKMFLEBKFXLRRFDTZXCIWBJSICBGAWDVYDHAFFJXZIBKC
GJIWEAHTTOEWTUHKRQVVRGZBXYIREMMASCPBNLHJMBLR
FFJELHWEYLWISTFVVYFJCMHYUYRUFSGESIGRLWALSWM
NUHSIMYYITCCQPZSICEHBCCMZFEGVJYOCDEMMPGHVAAUM
ELCMOEHLVTIPSUYILVGFLMVWDVYDBTHFRAYISYSGKVSUU
HYHGGCKTMLRX

Exercise 4.2 (Involutory Keys) A key K for a symmetric cryptosystem (G, E, D) where $\mathcal{P} = \mathcal{C}$ is said to be involutory, if for this key, the encryption function is the same as the decryption function, that is, if $E_K(x) = D_K(x)$ for all $x \in \mathcal{P}$.

Suppose (a, b) is a key for the Affine cipher, where $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$. Show that this key is involutory if and only if $a = a^{-1} \bmod n$ and $b(a + 1) \bmod n = 0$.

Determine all the involutory keys when $n = 15$.

Exercise 4.3 (An “Improvement” of Vigenere) We describe a modification of the Vigenere Cipher. Given a keyword (k_1, \dots, k_m) of length m , we encrypt as follows: The first block of m letters is encrypted as in Vigenere. Then we replace each letter by its successor modulo 26, so we get the key $(k_1 + 1 \bmod 26, \dots, k_m + 1 \bmod 26)$. Then encrypt the next block with this new key. We continue this way, adding 1 to each key letter each time use the key.

Describe how the known methods for Vigenere can be modified to break this cipher. Test your method by cryptanalyzing the following ciphertext:

IYMYLONRNCQXQJEDSHBUIBCJUZFOLQYSCHATPEQQQ
JEJNGNXZWHHGWFSUKULJQACZKKJOAAHGKEMTAFGMKVRDO
PXNEHEKZNFKIFRQVHHOVXINPHMRTJPYWQG JWPVVVKFP
OAWPMRKKQZWLQDYAZDRMLPBKJOBWIWPSEPVVQMBCRYVC
RUZAAOUMBCHDAGDIEMSZFZHALIGKEMJFPCIWKRMLMPIN
AYOFIREAOLDTHITDVRMSE

Unconditional Security and Information Theory

Contents

5.1	Introduction	33
5.2	Perfect security	34
5.3	Information theory and cryptography	37
5.3.1	Entropy	37
5.3.2	Conditional entropy	39
5.3.3	Entropy of random variables in cryptography	42
5.3.4	Unicity distance	44
5.4	When the adversary has incomplete information	48
5.5	Exercises	51

5.1 Introduction

In this chapter, we will study whether it is possible to have completely unbreakable encryption and whether such cryptosystems can be practical. We will consider the following scenario:

First, we will assume an unbounded adversary, i.e., there is no bound on how much computing time or memory the adversary can use. This means, for instance, that the adversary can always search through all possible keys in our system, no matter how many there are. Security against such an adversary is sometimes called *unconditional* security because security is not conditioned on any limitation we assume on the adversary.

Second, we will consider a ciphertext-only attack. This is the weakest of the attacks we defined, but there is a good reason for this: it turns out we can show that unconditional security can only be achieved if the cryptosystem is inefficient (in terms of key size, as we shall see). If this is the case already for the weakest type of attack, unconditional security is ruled out in practice for most scenarios.

In the following, we first look at the strongest form of unconditional security, namely Shannon's notion of perfect security and prove results about limitations on how to achieve it. We then introduce some basic notions from information theory, such as entropy, that can be used to characterize what amount of unconditional security we can have in practice.

The results we obtain paint a rather pessimistic picture of the possibilities for

unconditional security of encryption in standard scenarios. However, at the end of the chapter, we look briefly at some more optimistic result indicating that in specialised settings, unconditional security can be more practical than one might expect.

We begin with some notation. Assume we have some cryptosystem defined by algorithms (G, E, D) . In line with the definition of ciphertext-only attack, we will assume that a distribution on plaintexts is given. So every plaintext $x \in \mathcal{P}$ has a probability $P[x]$ assigned. In practice, what we mean by this is often that it is known which language the plaintext is in. Indeed, if we know the plaintext is English, this implies that certain sequences of letters are more likely than others. Actually computing such probabilities exactly is usually infeasible in practice, but we assume the adversary is unbounded and has no problem doing so. Likewise, the key generation algorithm (being probabilistic) induces a distribution on the key space \mathcal{K} , namely $P[K]$ is the probability that it outputs the key K .

The adversary knows, of course, the cryptosystem and the plaintext distribution, and can therefore also compute the probability distribution on ciphertexts. Namely, ciphertext $y \in \mathcal{C}$ occurs exactly if plaintext x and key K occur for which $E_K(x) = y$. We always assume that key and plaintext are independent, so the probability of such a pair (x, K) is $P[x]P[K]$. We then get $P[y]$ by summing over all such pairs:

$$P[y] = \sum_{(x,K):E_K(x)=y} P[x]P[K]$$

The conclusion is that when we define what security should mean in this scenario, we must assume that the adversary knows exactly the joint distribution of plaintexts, keys and ciphertexts.

5.2 Perfect security

The notion of perfect security was defined by Shannon in the early 50-ties. He considered the setting we defined above and said the following:

Definition 5.1 A cryptosystem has perfect security if for all $x \in \mathcal{P}$ and $y \in \mathcal{C}$, it holds that $P[x|y] = P[x]$.

The definition simply says that seeing the ciphertext does not help the adversary at all: the distribution on the plaintext remains the same whether you are given the ciphertext or not.

If some ciphertext y has probability 0, we can delete it from the ciphertext space without loss of generality. So we can assume that $P[y] > 0$ for all y , and so, using Baye's formula (Theorem 2.1), we see that if we have perfect security, then

$$P[x] = P[x|y] = \frac{P[x, y]}{P[y]} = \frac{P[y|x]P[x]}{P[y]}$$

which implies that $P[y|x] = P[y]$. Conversely, if $P[y|x] = P[y]$, essentially the same computation implies $P[x|y] = P[x]$. So we have

Lemma 5.2 *A cryptosystem has perfect security if and only if $P[y|x] = P[y]$.*

Armed with this observation, we can show:

Theorem 5.3 *The Shift cipher has perfect security, if each key is used only once.*

PROOF We will compute both $P[y]$ and $P[y|x]$ and see that they equal. Recall that in this system $E_K(x) = (x + K) \bmod n$. First we see that

$$P[y|x] = P[K = (y - x) \bmod n] = 1/n,$$

namely if we know the plaintext is x , then the ciphertext can only be y if the key is the difference $y - x \bmod n$. We can then use this to compute as follows:

$$P[y] = \sum_{x \in \mathcal{P}} P[x, y] = \sum_{x \in \mathcal{P}} P[y|x]P[x] = 1/n \cdot \sum_{x \in \mathcal{P}} P[x] = 1/n,$$

using Baye's formula and the fact the probabilities sum to 1. \square

Of course, using each key only once is a serious limitation in practice. It means that we need as many key symbols as we have plaintext symbols, or in other words, we need as many possible keys as we have plaintexts. However, this is unavoidable if we want perfect security. To see this, first note that for any cryptosystem we must have $|\mathcal{C}| \geq |\mathcal{P}|$ just in order to be able to decrypt correctly. Otherwise, two plaintexts would have to be sent to the same ciphertext while encrypting. But then consider a fixed plaintext x and imagine we encrypt it under every possible key. To have perfect security, we must hit every possible ciphertext this way. Namely, if ciphertext y cannot come from plaintext x , and y is actually sent as ciphertext, then the adversary could conclude that x was not the plaintext and perfect security is broken. However, to hit every ciphertext from a single plaintext requires that we have at least one key for every ciphertext, so we conclude that $|\mathcal{K}| \geq |\mathcal{C}|$. In summary, we have shown:

Theorem 5.4 *For every cryptosystem with perfect security, it holds that $|\mathcal{K}| \geq |\mathcal{C}| \geq |\mathcal{P}|$.*

Therefore, the best possible case is if $|\mathcal{K}| = |\mathcal{C}| = |\mathcal{P}|$. Shannon showed a characterization of perfect secure cryptosystems with this property. It basically says that there is only one way to get perfect security with a minimal number of keys, namely to do essentially what the shift cipher does:

Theorem 5.5 *Let (G, E, D) be a cryptosystem with $|\mathcal{K}| = |\mathcal{C}| = |\mathcal{P}|$. Such a system is perfectly secure if and only if it is the case that keys are chosen uniformly and for every pair $(x, y) \in \mathcal{P} \times \mathcal{C}$, there is exactly one key K such that $E_K(x) = y$.*

PROOF Assume first that the cryptosystem is perfectly secure. Now, as we also observed above, if we consider a plaintext x then for every ciphertext y there must be a key K such that $E_K(x) = y$. But remember that the number of keys equals the number of ciphertexts so we must have exactly one key for each y . This shows the second property. To see that keys are uniformly chosen, fix instead a ciphertext y and let K_x be the unique key we know exists for which $E_{K_x}(x) = y$.

Now, by perfect security we have $P[y] = P[y|x]$ for every x . But $P[y|x] = P[K_x]$, namely if the plaintext is x , the ciphertext can only be y if the key K_x is chosen. So we see that $P[y] = P[K_x]$ for every x , i.e., the probability is the same for every key, implying that keys are indeed chosen uniformly.

For the other direction, recall the proof we gave earlier that the shift cipher is perfectly secure. Taking a closer look, one sees that the only properties we actually used from the shift cipher was exactly the two properties from the theorem: keys are uniform and there is exactly one key for each plaintext/ciphertext pair. Therefore that proof suffices. \square

The one-time pad

One cannot talk about perfect security without mentioning the *one-time pad* which was invented by Gilbert Vernam in 1917 for secure transmission between teletype machines. It is essentially the Shift cipher using $n = 2$, and following the rule that every key is used only once. Observing that addition and subtraction modulo 2 is the same operation, both equal to the xor-operation, we get the following:

For encrypting t -bit messages using the one-time pad, we set $\mathcal{K} = \mathcal{P} = \mathcal{C} = (\mathbb{Z}_2)^t$ and the system works as follows:

Key Generation Output $K = (k_1, \dots, k_t)$ chosen uniformly at random from \mathcal{K} .

Encryption Set $E_K(x_1, \dots, x_t) = x_1 \oplus k_1, \dots, x_t \oplus k_t$.

Decryption Set $D_k(y_1, \dots, y_t) = y_1 \oplus k_1, \dots, y_t \oplus k_t$.

The one-time pad is the most practical perfectly secure cryptosystem: it is very easy to implement and decryption is the same operation as encryption. But of course, it cannot get around the fundamental limitation we proved above: the key must be at least long as the message. Of course, perfect security only holds if the key is unknown the adversary, but nevertheless the key must be known to both sender and receiver.

One might be tempted to argue that this means the one-pad and its relatives are completely useless: if we have a way to transmit the key securely from sender to receiver and it is as long as the message, why don't we send the message this way instead?! However, this argument is not completely fair: there may an opportunity at some particular point in time where the key can be exchanged between the parties. If this opportunity exists just once, then the key can be used at any later point to send messages securely.

For example, during the cold war, there was a secure telephone line, the so called hot line, intended for communication between the US president and the general secretary of the USSR. The story goes that this phone line was encrypted with the one-time pad, and key material was transported by agents manually on tape between the two countries. Whether the details of the story are true or not, it demonstrates that the one-time pad can be useful if you insist on the best possible security and you don't care what the cost is.

5.3 Information theory and cryptography

Since perfect security is impractical for fundamental reasons, one could ask if perhaps we could get more practical systems if we settled for less than perfect: what if we are OK with the adversary learning a small amount of information about the message, rather than nothing? But this immediately leads to a new question: what does “a small amount of information” mean? how do you measure information?

This is exactly what information theory is about. So in this section, we will introduce the basic tools we need for this and then show that if you use short keys and hence have to reuse the key for several messages, this reuse will eventually erode your unconditional security, and at some point the adversary will have enough information to completely break the system.

5.3.1 Entropy

Suppose I flip a fair coin and want to tell you what the result was. How many bits do I have to send to do this? A rather trivial question, 1 bit is of course necessary and sufficient. Suppose instead I flip 2 fair coins, so that we now have 4 different outcomes that each occur with probability $1/4$. It is also quite clear that in this case 2 bits are needed to communicate the outcome.

Now for a not quite so trivial question: assume we do an experiment with three possible outcomes, A, B, C , with probabilities $P[A] = 1/2, P[B] = 1/4, P[C] = 1/4$. How many bits do we now need to communicate the result? At first, one might be tempted to say 2 bits, since 1 bit has only 2 possible values and so is insufficient. But taking a closer look, one might realize that, at least on average, we can save space by having a shorter message for the more likely outcome. Using this idea immediately leads to sending, for instance, 0 for the outcome A , 10 for B and 11 for C .

In this way, we will send 1 bit with probability $1/2$ and 2 bits with probability $1/4 + 1/4 = 1/2$. So this means that on average we send $1 \cdot 1/2 + 2 \cdot 1/2 = 1.5$ bits.

Shannon’s notion of Entropy gives us a way to assign a number to a random variable with a certain probability distribution. This number is called the uncertainty or the entropy of the variable, and it corresponds exactly to the numbers of bits we just discussed. That is, if we define, for instance, a random variable that takes values A, B and C with probabilities $1/2, 1/4$ and $1/4$, then the entropy of this variable will turn out to be exactly 1.5 bits.

Shannon derived the definition of entropy from the following convention:

“If an event A occurs with probability p and you are told that A occurred, then you have learned $\log_2(1/p)$ bits of information”.

At first, this may seem like a rather arbitrary postulate, but it actually makes a lot of sense. To see this, consider the case of a coinflip. Here, the probability of an outcome tails, for instance, is $p = 1/2$, so $\log_2(1/p) = \log_2(2) = 1$, and indeed

this corresponds to our experiment above where we needed to send 1 bit for each outcome of a coinflip.

In the same way, if we consider the experiment from before with 3 outcomes, Shannon's convention would say that if outcome A occurs, we learn 1 bit of information, and if B or C occurs, we learn $\log_2(4) = 2$ bits. The definition of entropy now says that you should compute this number of bits for each possible output and then take the weighted average of these values. For the $A/B/C$ experiment, this becomes

$$\frac{1}{2} \cdot \log_2\left(\frac{1}{1/2}\right) + \frac{1}{4} \cdot \log_2\left(\frac{1}{1/4}\right) + \frac{1}{4} \cdot \log_2\left(\frac{1}{1/4}\right) = 1.5$$

exactly as expected.

The general definition goes as follows:

Definition 5.6 Let X be a random variable that takes values x_1, \dots, x_n with probabilities p_1, \dots, p_n . Then the entropy of X , written $H(X)$, is defined to be

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

The definition allows that some of the probabilities are 0. Of course, it does not make sense to talk about $\log(1/0)$, but it turns out that $p \log(1/p)$ approaches 0 as p goes to 0, so we adopt the convention that $0 \cdot \log(1/0) = 0$.

The choice of base-2 logarithms in the definition is quite arbitrary, and simply corresponds to the choice of bits as our unit of information. In the following, all logarithms will be base-2 and we will write \log instead of \log_2 .

As we hinted before, the entropy $H(X)$ tells us how many bits we need to send on average to communicate the value of X . It can also be thought of as the amount of uncertainty you have about X before you are told what the value is. This makes sense too: if I need to be told $H(X)$ bits to become certain of the value of X , it is reasonable to claim that my uncertainty was $H(X)$ bits before I was told.

We have the following result on the possible range of values for $H(X)$:

Theorem 5.7 For a random variable X taking n possible values, it holds that $0 \leq H(X) \leq \log_2(n)$. Furthermore, $H(X) = 0$ if and only if one value X has probability 1 (and the others 0). $H(X) = \log(n)$ if and only if it is uniformly distributed, i.e., all probabilities are $1/n$.

PROOF Clearly, $H(X) \geq 0$ because it is the sum of non-negative terms. Therefore it can only be 0 if all summands are 0. The function p mapping to $p \log(1/p)$ is only 0 if $p = 1$ or $p = 0$, so this and the fact that probabilities must sum to 1 implies that $H(X) = 0$ if and only if one value has probability 1 and all others are 0.

Because $\log()$ is a concave function, Jensen's inequality, Theorem 2.3, gives us

that

$$H(X) = \sum_{i=1}^n p_i \log_2(1/p_i) \leq \log\left(\sum_{i=1}^n p_i \cdot 1/p_i\right) = \log(n)$$

And because $\log()$ is a strictly concave function, we have equality only if all p_i are the same, which means they all have to be $1/n$, so X is uniformly distributed. \square

It is important to emphasize that these results are exactly what we should intuitively expect from a reasonable definition of uncertainty. Had they not been satisfied for Shannon's definition of entropy, he would have had to go home to the drawing board and look for a better mathematical model of uncertainty!

Indeed, when do we have no uncertainty about a random variable? well, of course only if a certain outcome always occurs, and so has probability 1. And when do we have maximal uncertainty? well, if one outcome is more likely than all others, we can make an educated guess at what will happen, but this fails for a uniform distribution (and only then), so this is the obvious case where uncertainty is maximal.

As a further illustration of the intuition behind entropy, consider the following thought experiment: suppose you get access to an oracle that will magically tell you if you will live to be more than 110 years old. One would naturally expect that the probability p of getting "yes" as the answer is very small, while the probability $1 - p$ of "no" is close to 1.

Now, if you ask and the oracle says no, it feels like you did not learn much: you almost knew the answer would be no already. This corresponds to $\log(1/(1 - p))$ being close to 0, in the language of Shannon's postulate above. On the other hand if it says yes, this is really interesting news, corresponding to the fact that $\log(1/p)$ is large. In general, how much information does a person learn from the oracle? Not much in the cases where it says no, and a lot when it says yes, but the latter only occurs very seldom. So on average you get only a small amount of information from the oracle. This corresponds to the fact that the entropy of the answer is $p \log(1/p) + (1 - p) \log(1/(1 - p))$ which is indeed small when p is small. As an example, it is about 0.08 when $p = 0.01$.

5.3.2 Conditional entropy

Suppose we have two random variables X, Y , taking values x_1, \dots, x_n and y_1, \dots, y_m respectively. Let p_i be the probability that $X = x_i$ and q_j the probability that $Y = y_j$. Finally, let r_{ij} be the probability that $X = x_i$ and $Y = y_j$. Note that if X and Y are independent we have $r_{ij} = p_i q_j$, but this does not have to be the case in general.

We can define a joint entropy of X and Y in a natural way: we just consider them as one variable with values that are pairs (x_i, y_j) , such an outcome has probability r_{ij} in the notation above. Then we just take the entropy of this

distribution. This means that the joint entropy $H(X, Y)$ is

$$H(X, Y) = \sum_{i=1}^n \sum_{j=1}^m r_{ij} \log(1/r_{ij})$$

This can be thought of as the total amount of information you get if you are told the values of both X and Y .

Let's consider intuitively how $H(X, Y)$ should compare to $H(X) + H(Y)$. If X depends on Y in some way, that should mean that there is some of the information contained in X that is also contained in Y . Let's consider an extreme case of this, where Y is defined to be exactly the same as X . Then, of course, when you are told X , there is nothing new to say about Y , so it should be the case that $H(X, Y) = H(X)$. We also have $H(X) = H(Y)$, so that $H(X) + H(Y) = 2H(X)$ and so is larger than $H(X, Y)$. At the other extreme, what if X is independent of Y ? Then being told about X and Y are two completely separate stories, so it ought to be the case that $H(X, Y) = H(X) + H(Y)$. Therefore, common sense would seem to tell us to expect that $H(X, Y)$ is smaller than $H(X) + H(Y)$ with equality if and only if X is independent of Y . This is exactly what the next theorem says:

Theorem 5.8 *For any two random variables X, Y it holds that $H(X, Y) \leq H(X) + H(Y)$, with equality if and only if X and Y are independent.*

PROOF We will show that $H(X, Y) - H(X) - H(Y) \leq 0$ which of course implies the theorem. By definition, we have

$$H(X, Y) - H(X) - H(Y) = \sum_{i=1}^n \sum_{j=1}^m r_{ij} \log(1/r_{ij}) - \sum_{i=1}^n p_i \log(1/p_i) - \sum_{j=1}^m q_j \log(1/q_j)$$

Now observe that we can split the event that $X = x_i$ into disjoint events, namely $(X = x_i, Y = y_1), \dots, (X = x_i, Y = y_m)$, and therefore $p_i = \sum_j r_{ij}$. In the same way we get that $q_j = \sum_i r_{ij}$. Inserting this in the above, we get that the quantity we want is

$$\begin{aligned} & \sum_{i=1}^n \sum_{j=1}^m r_{ij} \log(1/r_{ij}) - \sum_{i=1}^n \sum_{j=1}^m r_{ij} \log(1/p_i) - \sum_{j=1}^m \sum_{i=1}^n r_{ij} \log(1/q_j) = \\ & \sum_{i=1}^n \sum_{j=1}^m r_{ij} \log\left(\frac{p_i q_j}{r_{ij}}\right) \leq \log\left(\sum_{i=1}^n \sum_{j=1}^m r_{ij} \frac{p_i q_j}{r_{ij}}\right) = \log(1) = 0, \end{aligned}$$

where we used Jensen's inequality and the fact that the log-function is concave. This shows that first part. Since log is even strictly concave, we get that we have equality above if and only if all the arguments to the log-function are equal. So this means that for all i, j , $\frac{p_i q_j}{r_{ij}} = c$ for some constant c . From this, we can use the fact that probabilities sum to 1 to conclude that

$$1 = \sum_{i,j} p_i q_j = \sum_{i,j} c r_{ij} = c \sum_{i,j} r_{ij} = c,$$

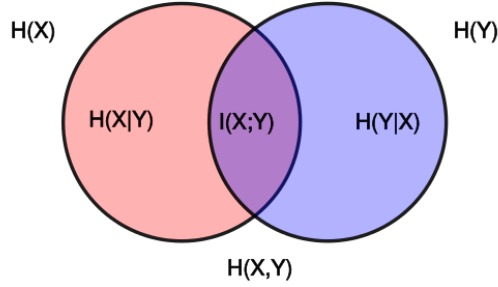


Figure 5.1 Diagram illustrating notions of entropy for 2 variables. The entire figure corresponds to $H(X, Y)$ while the left and right circles are $H(X)$, $H(Y)$, respectively. You get the entropy assigned to an area by adding the entropies for the subregions. For instance, one directly sees that $H(X, Y) = H(Y) + H(X|Y)$.

so since $c = 1$, we see that $P[X = x_i, Y = y_j] = r_{ij} = p_i q_j = P[X = x_i]P[Y = y_j]$, or in other words, X and Y are independent. \square

The difference $H(X) + H(Y) - H(X, Y)$ which we just showed is non-negative, has a name, it is called the mutual information between X and Y and is written $I(X; Y)$. It measures the amount of information that is present in both X and Y . Intuitively, if I tell you about X and then about Y , there may be some information I told you twice, and this overlap should show up as the difference $H(X) + H(Y) - H(X, Y)$. This is illustrated in Figure 5.1, which also illustrates the idea of conditional entropy which we define next.

Conditional entropy is supposed to be a measure of the uncertainty that remains about X given you are told the value Y takes. Now, if you are told that $Y = y_j$, we can find the uncertainty about X in this case by just taking the entropy of the distribution of X that results by conditioning on $Y = y_j$. So we define

$$H(X|Y = y_j) = \sum_{i=1}^n P[X = x_i|Y = y_j] \cdot \log\left(\frac{1}{P[X = x_i|Y = y_j]}\right)$$

Then we define the conditional entropy by taking the weighted average of these values:

Definition 5.9 Give the above definition of $H(X|Y = y_j)$, we define the conditional entropy of X given Y to be

$$H(X|Y) = \sum_j P[Y = y_j] H(X|Y = y_j)$$

In the exercises, you will prove the following results:

Theorem 5.10 *For any random variables X, Y , we have*

$$H(X, Y) = H(Y) + H(X|Y)$$

Furthermore,

$$H(X|Y) \leq H(X)$$

with equality if and only if X and Y are independent.

We emphasize again that these results, like virtually any result in information theory, are exactly what common sense suggests: to tell you about both X and Y , I can first tell you Y , and then I tell you whatever remains to be said about X , given that I just told you Y – which is exactly what $H(X, Y) = H(Y) + H(X|Y)$ expresses. Likewise, $H(X|Y) \leq H(X)$ reflects the intuition that you cannot become more uncertain about X if you are given more information.

5.3.3 Entropy of random variables in cryptography

To analyse cryptosystems using information theory we need to handle 3 random variables, namely the plaintext P , the ciphertext C and the key K . For three variables, one can set up a diagram just like in Figure 5.1, see Figure 5.2. It has been shown that all conclusions you can draw from the diagram on (conditional) entropies are valid, so this gives a very intuitive way of reasoning about these entities.

As an example, we can use the diagram to show a result by Shannon, namely that if a cryptosystem has perfect security, then $H(K) \geq H(P)$. This says that, not only does perfect security require us to have more keys than plaintexts, in fact more is true: if we write key and plaintext down in the most compact way possible, the key will always be at least as long as the plaintext.

First, a couple of things we can say for any cryptosystem: Since we can decrypt correctly if we are given C and K , $H(P|C, K) = f = 0$. Moreover, we always choose K independently of P , so there is no mutual information between K and P , $I(K; P) = b + d = 0$. It follows that

$$H(P) = b + d + c + f = c.$$

Moreover

$$H(K) = a + e + b + d = a + e.$$

Since $e \geq 0$ it will clearly be enough to show that $a \geq c$.

Now, perfect security means that C tells you nothing about P , in other words $H(P) = H(P|C)$ which from the diagram means that $c = d + f = d$. So in fact we may as well show that $a \geq d$. But this follows since $I(C; K) = a + b \geq 0$ and $b + d = 0$.

We end this section with a theorem that will be useful later and speaks about a piece of information that is clearly interesting: how much uncertainty remains about the key given the ciphertext?

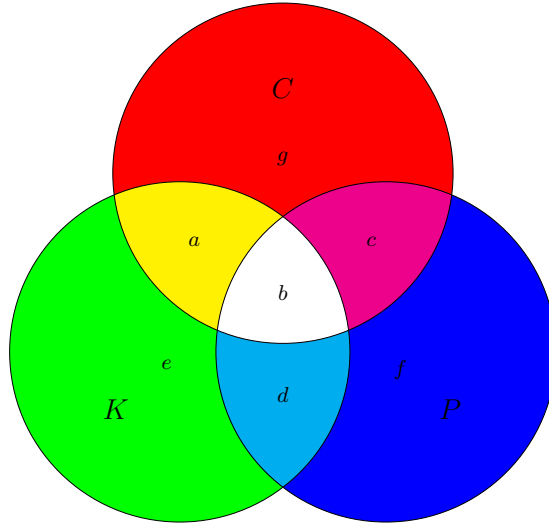


Figure 5.2 Entropy Diagram for three variables. Each circle represents the entropy of a variable, in this case P, C and K . The overlap areas represent the mutual information of two objects, and one gets the measure of a certain region by adding the numbers for the subregion. For instance, $I(K; C) = a + b$ and $H(C|P) = a + g$. All subareas represent amounts of information and are therefore non-negative, except b , which may be negative.

Theorem 5.11 *For any cryptosystem with deterministic encryption function, it holds that*

$$H(K|C) = H(K) + H(P) - H(C)$$

PROOF From the diagram, we get that $H(K|C) = e + d$. Now we simply write down directly what the right-hand side is:

$$H(K) + H(P) - H(C) = (a + b + e + d) + (b + d + c + f) - (a + b + c + g)$$

Since encryption is deterministic, the ciphertext is determined from key and plaintext, so $g = H(C|K, P) = 0$. Also, $f = 0$ as we saw above. Reducing the expression gives $H(K) + H(P) - H(C) = e + d + b + d$, but $b + d = 0$ as we also saw above, and we are done.

For completeness, we also give an argument that uses the formulas and definitions more directly:

By Theorem 5.10, the left side of what we want to show is

$$H(K|C) = H(K, C) - H(C).$$

By Theorem 5.8, the right side is

$$H(K) + H(P) - H(C) = H(K, P) - H(C)$$

because K, P are independent. So it is enough to show that $H(K, C) = H(K, P)$. We will do this by showing that they are both equal to $H(K, C, P)$. We get

that $H(K, C) = H(K, C, P)$ because $H(K, C, P) - H(K, C) = H(P|K, C) = 0$. Similarly $H(K, P) = H(K, C, P)$ because $H(K, C, P) - H(K, P) = H(C|K, P) = 0$ since encryption is deterministic. \square

5.3.4 Unicity distance

We are now finally ready to answer the question that we asked in the beginning of this section: what happens to the unconditional security of a cryptosystem if we are not willing to choose new key material for each message and instead use the same key for many messages? Recall that the adversary knows the plaintext distribution and is given all the ciphertext that is sent. As a concrete example one can think of the plaintext distribution being given because the adversary knows that the plaintext is in English, for instance. One obvious approach the adversary could try in order to determine the key is to simply decrypt the ciphertexts he has, under every possible key (recall that the adversary is unbounded, so the number of keys is not an issue here). For every key, he checks if the decryption result is meaningful English (or more formally, if the resulting plaintext has probability greater than 0). If this is not the case, he discards the key.

Of course, the correct key will pass this test, but other keys might pass it as well, just “by chance”. These incorrect keys are called *spurious keys* in the following. Now, intuitively it seems reasonable that the more ciphertext we send, the more likely it will be that only the correct key passes the test. After all, decrypting with an incorrect key will probably produce a random looking string of letters which - if it is long - is unlikely to be meaningful English. If only the correct key is left, the adversary knows the key, and there is no security left.

In the following, we will make this more precise and come up with a concrete estimate of how much ciphertext will be required to determine the key in the way we just sketched.

The entropy and redundancy of a language

In this subsection we consider a natural language L , such as English. And we let the random variable P_n denote sequences of letters of length n in L . Thus, P_1 will take single letters as values, and for English it will have the distribution given in the single-letter frequency table we saw earlier, this would give an entropy of about 4.19 bits.

Now consider an experiment where we choose two letters independently with distribution as P_1 . Note that the entropy of a pair chosen this way will be $2H(P_1)$. Now, how does this compare to P_2 , which represents pairs as they actually occur in L ? Taking English as an example, we can see that the pair qw , for instance, has non-zero probability if letters are independent, but probability 0 in real English, where q is virtually always followed by u . More generally, once we have seen the first letter in a pair, we can make a prediction about the second letter which is better than what we can do for an independently chosen letter. So there-

fore $2H(P_1) \geq H(P_2)$, or equivalently $H(P_1) \geq H(P_2)/2$. Studies of statistics of English text suggest that $H(P_2)/2$ is about 3.90 (so indeed smaller than 4.19).

For the same reasons, a similar decrease in entropy will happen for longer sequences, so we will have

$$H(P_1) \geq H(P_2)/2 \geq H(P_3)/3 \geq \dots$$

A good way to think of this is that the entropy per letter decreases as we consider longer sequences, because we take more and more dependencies in the language into account. Of course, at some point, we will have included virtually all the statistics of the language, so there will be a limit for this sequence. We define

$$H_L = \lim_{n \rightarrow \infty} H(P_n)/n.$$

H_L is a measure of the number of bits of information each letter contains in the language L , on average. For English, we have that H_L is (very approximately) 1.25 bits per letter.

Recall that \mathcal{P} is the plaintext alphabet. If letters were chosen uniformly, then the entropy per letter would be $\log(|\mathcal{P}|)$, by Theorem 5.7. So $\log(|\mathcal{P}|)$ is the maximal number of bits we have room for in one letter. Now, the language L actually contains H_L bits of information per letter, so this means that we have $\log(|\mathcal{P}|) - H_L$ “unused” bits per letter. This in turn means that the fraction of bits we do not use is $(\log(|\mathcal{P}|) - H_L)/\log(|\mathcal{P}|)$. We call this fraction the *redundancy* of the language, and we define:

$$R_L = \frac{\log(|\mathcal{P}|) - H_L}{\log(|\mathcal{P}|)} = 1 - \frac{H_L}{\log(|\mathcal{P}|)}.$$

The English alphabet has 26 letters, so here $\log(|\mathcal{P}|)$ is approximately 4.70. Using the estimate of 1.25 for the entropy of English, we get that the redundancy of English is approximately 0.75. So this estimate would have us believe that English is about 75% redundant. What this means in practice is that an English text, using the best possible compression, could in principle be compressed to about 25% of its original size, and yet one could reconstruct the original text exactly. Modern compression software actually comes quite close to this ideal goal, compressing to about 30% of the size.

Before we continue with the cryptographic significance of this, it is worth mentioning that the redundancy in natural languages is there for a good reason: redundancy is necessary for us to listen to and understand spoken language. The point is that we can lose some of the data and still be able to know what was said. For instance, “cn y rd th flwng sntnc, vn f t s wrttn wtht vcls?” Possible, right? In the same way, when we listen to spoken language, we miss many of the sounds – maybe we lose focus or the speaker is mumbling a bit. But our brains can still reconstruct what is said because of the redundancy, even without us having to think about it. This is also the reason why data that is to be sent over a noisy channel is coded using a process that adds redundancy to the data so it can be reconstructed after transmission. This is known as error correcting codes and is an entire research field in itself.

So, for error correction, redundancy is a blessing. For cryptography, however, it can be a problem, as we shall see now.

Unicity distance

In this subsection, we will consider a cryptosystem, where we use the same key to encrypt n messages. Each message is of fixed length, and we will think of each such message as a letter in an alphabet \mathcal{P} . We assume that the sequences of letters come from some language L with entropy H_L and redundancy R_L . The random variables P_n, K, C_n will represent the choice of plaintext sequence, key and resulting ciphertext-sequence. We will use boldface: \mathbf{x}, \mathbf{y} , to denote values of P_n , respectively C_n .

Given a ciphertext \mathbf{y} , we use $K(\mathbf{y})$ to denote the set of keys that are possible given this ciphertext. More precisely, a key K is in this set if decryption of \mathbf{y} under K yields a plaintext that could occur with non-zero probability:

$$K(\mathbf{y}) = \{K \in \mathcal{K} \mid P[D_K(\mathbf{y})] > 0\}$$

This is the set of keys that will pass the adversary's test, as we sketched earlier. Of course, the correct key will be in $K(\mathbf{y})$, so the number of spurious keys for this particular ciphertext is $|K(\mathbf{y})| - 1$. The adversary hopes, of course, that the number of spurious keys will be 0, so we are interested in the average number of spurious keys, taken over all choices of ciphertexts of length n . This will be

$$sp_n = \sum_{\mathbf{y} \in \mathcal{C}^n} P[\mathbf{y}] (|K(\mathbf{y})| - 1) = \sum_{\mathbf{y} \in \mathcal{C}^n} P[\mathbf{y}] |K(\mathbf{y})| - 1. \quad (5.1)$$

Note that if a sequence of n letters can occur in a text in L , then if we chop off the last letter, the resulting string can of course also occur somewhere. Hence, if a key K is spurious with respect to a ciphertext \mathbf{y} , then K is also spurious with respect to \mathbf{y}' , where \mathbf{y}' is the ciphertext obtained by removing the last ciphertext symbol. This means that as we increase n , the set of spurious keys can only become smaller, and so sp_n cannot increase as n increases. In fact we expect it to go to 0, although this does not always happen, as we shall see. We therefore define:

Definition 5.12 The unicity distance n_0 of a cryptosystem is the minimal length of plaintexts such that $sp_{n_0} = 0$, if such a value exists, and ∞ otherwise.

We can give an estimate of n_0 , at least for some cryptosystems:

Theorem 5.13 Assume we have a cryptosystem with deterministic encryption function, where the plaintext and ciphertext alphabets have the same size ($|\mathcal{C}| = |\mathcal{P}|$), and where keys are uniformly chosen from \mathcal{K} . Assume we use the system to encrypt sequences of letters from language L . Then

$$n_0 \geq \frac{\log(|\mathcal{K}|)}{R_L \log(|\mathcal{P}|)}$$

Remark 5.14 You will show in the exercises that under a reasonable assumption

on the cryptosystem, the expression in the theorem is not just a lower bound, but is in fact a good estimate of the actual value of n_0 .

PROOF First, note that given some ciphertext \mathbf{y} , the key K will have some conditional distribution, but of course only values in $K(\mathbf{y})$ can occur. Therefore $H(K|C_n = \mathbf{y}) \leq \log(|K(\mathbf{y})|)$, by Theorem 5.7. From this we get

$$H(K|C_n) = \sum_{\mathbf{y} \in \mathcal{C}_n} P[C_n = \mathbf{y}] H(K|C_n = \mathbf{y}) \leq \sum_{\mathbf{y} \in \mathcal{C}_n} P[C_n = \mathbf{y}] \log(|K(\mathbf{y})|)$$

Using Jensen's inequality on the last expression, we obtain

$$H(K|C_n) \leq \log\left(\sum_{\mathbf{y} \in \mathcal{C}_n} P[C_n = \mathbf{y}] |K(\mathbf{y})|\right) = \log(sp_n + 1) \quad (5.2)$$

where the last step follows from (5.1).

We can also compute $H(K|C_n)$ from Theorem 5.11, which says that

$$H(K|C_n) = H(K) + H(P_n) - H(C_n)$$

Observe that $H(C_n) \leq \log(|\mathcal{C}|^n) = n \log(|\mathcal{P}|)$. Moreover, recalling the definition on H_L , let us assume that we take n large enough so that $H(P_n) \approx nH_L$. From this, and the definition of R_L , we obtain the estimate $H(P_n) = n \log(|\mathcal{P}|)(1 - R_L)$. Inserting this into the above, we get

$$H(K|C_n) \geq H(K) - nR_L \log(|\mathcal{P}|) = \log(|\mathcal{K}|) - nR_L \log(|\mathcal{P}|) \quad (5.3)$$

using the assumption that keys are uniformly chosen. Combining (5.3) and (5.2), we obtain

$$\log(|\mathcal{K}|) - nR_L \log(|\mathcal{P}|) = \log\left(\frac{|\mathcal{K}|}{|\mathcal{P}|^{nR_L}}\right) \leq \log(sp_n + 1).$$

Since \log is an increasing function, this implies

$$\frac{|\mathcal{K}|}{|\mathcal{P}|^{nR_L}} \leq sp_n + 1.$$

We now get the desired estimate for n_0 by setting $sp_n = 0$ in the above inequality, taking the log on both sides and solving for n . \square

As an example we can compute an estimate for unicity distance of the general substitution cipher that we saw earlier. On the English alphabet, we have $|\mathcal{P}| = 26$ and $|\mathcal{K}| = 26!$. This means that, using $R_L = 0.75$, we get $n_0 \approx \frac{\log(26!)}{0.75 \log(26)} \approx 25$. So this means that given 25 characters of ciphertext, on average only 1 permutation of the alphabet will take the ciphertext back to a meaningful English text. Experience in practice suggests that one can actually break the cipher “by hand” using only around 35 characters of ciphertext.

Theorem 5.13 shows that if we reuse keys, our unconditional security will always be gone, once we encrypt enough plaintext under the same key. The only exception is the case where $R_L = 0$ which leads to n_0 being ∞ . This makes sense: if every sequence of characters is a plaintext that can occur, the adversary can

never exclude a key. This may seem to suggest that we should compress the plaintext before encryption: if the compression is perfect, then we will be encrypting plaintext for which indeed $R_L = 0$. However, there are two reasons why this is not a solution to all problems: first, compression algorithms are usually not perfect in practice, and second we only considered security against ciphertext-only attacks. If the adversary gets a known plaintext attack, he may be able to compute the key easily, despite the fact that the redundancy is (almost) 0.

A more correct take-home message from Theorem 5.13 is that compression before encryption at least will never hurt you, and can even improve security in some cases. Thus, if you are designing a system where data compression is an easily available option, there is no reason not to use it before encryption. On the other hand, if you need to add redundancy for error correction, this should always be done after encryption and never before.

Finally, one should not forget that all this is about unconditional security: even if only one key is possible given the ciphertext, it may be infeasible to compute that key. This is the basis of computational security which we will explore in the following chapters.

5.4 When the adversary has incomplete information

The above results have shown that in order to have any amount of unconditional security, we need to use new key material all the time. One might think that we could solve this problem by building an unconditionally secure *key exchange protocol*: an interactive protocol where parties A and B share no information initially, then they send some messages back and forth, and in the end they agree on a key K . Still, an unlimited adversary E who sees all the communication has no information on K . If we could pull this off, then we could simply make new key material whenever we need it and have unconditional security indefinitely.

However, it is easy to see that unconditionally secure key exchange is impossible: if A and B agree on the same key K , it must be that K is derived from information they share. If they share nothing initially, the only information they share is what they communicate in the protocol. So it follows that K is determined from the messages sent during the protocol, and hence E (who sees all the messages) can also compute K . To avoid misunderstandings, note that this only means that *unconditionally* secure key exchange is impossible: even if the key is determined from what the adversary sees, that does not mean he can compute it efficiently. This observation is the basis of public-key cryptography that we will get to later.

The above argument that unconditionally secure key exchange is impossible is correct, but makes an assumption that at least deserves closer examination, namely that *E sees all the messages sent*. Is this necessarily true in all scenarios? Taking a closer look at the way we communicate on networks and radio links at the most basic level, one finds that these communication channels introduce errors all the time, simply because of the basic physical characteristics of the channels. We usually don't notice this because our systems have built-in middleware that

puts redundancy into messages before they are sent, so that the errors can be corrected.

This means that if A and B would communicate without such error correction, it is reasonable to assume that E would *not* see everything that is sent, at least only with some number of errors introduced. Of course, A and B would also experience errors, so it is not clear how this helps – but it does mean that the assumption we made when arguing impossibility of key exchange no longer holds, and hence the proof of impossibility is invalid.

To see what is possible in such a setting, let us define a more precise model: we will assume that parties are connected using a so-called *binary symmetric channel* (BSC): such a channel transmits one bit at a time and every time you send a bit b , what is received is $b \oplus n$, where n is a noise bit chosen at random by the channel (independently each time a bit is sent). The probability $p = P[n = 1]$, is called the *error probability* of the channel. This is natural, since if $n = 1$, an incorrect bit is received.

The model we will assume here is that A can send to B over a BSC with error probability p . Furthermore, every time A sends a bit, E will receive that bit over a different and independent BSC with error probability q . One can think of this as a model of the case where A sends to B on a radio channel, and E listens to the signal using her own antenna. Since the signal needs to travel through a different part of the atmosphere to reach B than to reach E , the noise on the two channels is assumed to be independent. This assumption is of course questionable: what if E moves really close to B ? However, for now, we will just accept it and return to the issue later.

It has been shown that if $q > p$, then there exists an unconditionally secure key exchange protocol for this scenario. This is intuitively very reasonable: every time A sends a bit, B receives the correct bit with probability larger than that of E . So if A sends a long bit string $\mathbf{b} = b_1, \dots, b_n$ to B , then B has more information on \mathbf{b} than E has. From what A and B have at this point, it turns out that they can “distill” a shorter string K that they agree on, such that E has (essentially) no information on K . The technical details of this are out of scope for this text, but can be found, e.g., in [15].

So if A and B already have an advantage over E because $q > p$, then key exchange can be done. But this is of course an unreasonable assumption. In the radio signal example, what stops E from buying a bigger antenna? This can be used to reduce q so that $q < p$. Now key exchange is no longer possible, this has even been formally proven.

It is therefore very surprising that a small and seemingly insignificant addition to the model implies that key exchange becomes possible for *all* values of p and q . The addition consists of adding an error-free and public, but authenticated channel from B to A . More precisely, B can send a message to A on this channel, the message will be received by both A and E with no errors, but E cannot modify the message. One can, for instance, think of B calling A on the phone. E can eavesdrop the phoneline, but A will know she talks to B by recognizing his voice.

Note that this new channel does not provide any secrecy per se, so we have not “cheated”. On the other hand, as we discussed, some modification of the model is necessary for key exchange to become possible in the more realistic case where $q < p$.

The following protocol provides a way to send a bit b from B to A so that “magically” A will have more information on b than E .

1. A chooses a bit r uniformly at random and sends it to B over the BSC. Note that B will receive $r \oplus n_B$, while E will get $r \oplus n_E$, where $P[n_B = 1] = p$ and $P[n_E = 1] = q$.
2. B sends $x = m \oplus (r \oplus n_B)$ to A on the error-free channel. Note that E will also see x .
3. A outputs $x \oplus r = m \oplus (r \oplus n_B) \oplus r = m \oplus n_B$.

It is clear that by this trick of sending information back on the public channel, we have turned the noisy channel from A to B into one from B to A with the same error probability. The situation for E , however, is quite different. Note that what E sees is $r \oplus n_E$ and $x = m \oplus r \oplus n_B$. It is intuitively reasonable that the best E can do to get information on m is to xor the two bits to get rid of the random bit r . The result is

$$r \oplus n_E \oplus m \oplus r \oplus n_B = m \oplus n_E \oplus n_B$$

We will argue in a moment that indeed E learns nothing more about m than what is given by $m \oplus n_E \oplus n_B$. But this means that E actually receives the bit in question with B ’s noise added on top of her own!

And hence, with this new channel, E ’s error probability is larger than p . In the exercises, you will argue that since n_E and n_B are independent, and we assume $q > 0$ and $p < 1/2$, then the probability that $n_E \oplus n_B = 1$ is larger than p so indeed E has more noise than A and B .

The two assumptions on p and q assumptions are clearly necessary. if $q = 0$, E gets all information with no errors, and then we are back in the “full-information” scenario where key exchange is impossible, as we argued before. If $p = 1/2$, the BSC outputs essentially a one-time pad encryption of everything you try to send, so it is useless, as nothing gets through.

The conclusion is that the protocol above creates a situation where E ’s error probability is larger than that of A and B and hence we can use the methods we mentioned earlier to get secure key exchange.

We have one sticky point left: is it really true that by seeing the two bits $y = r \oplus n_E$ and $x = m \oplus r \oplus n_B$, E learns nothing more than she learns by seeing $z = m \oplus n_E \oplus n_B$? We will show this by arguing that if E is given z , then she can herself compute two bits with *exactly the same distribution* as (y, x) . This will imply that anything you can compute from (y, x) , you could also compute from the single bit z .

To see how E would pull this off, let us analyse the distribution of (y, x) . Notice that $y = r \oplus n_E$ is uniformly random because r and n_E are independent. Further

we can see that it always holds that $y \oplus z = x$, namely

$$y \oplus z = (r \oplus n_E) \oplus (m \oplus n_E \oplus n_B) = m \oplus r \oplus n_B = x$$

So it is now clear that what E can do is the following: select a random bit s and output the pair $(s, s \oplus z)$. This exactly satisfies the conditions we just derived.

Let us finally return to an issue we left open before: what about the assumption that n_E and n_B are independent? It turns out that key exchange is still possible by essentially the same protocol, as long as they are not completely dependent, that is, give one of the bits, some uncertainty remains about the other. The argument that it works is technically more involved and out of scope for this text. For more information, see [15].

Conclusion on the positive results

We have seen in this section that if the adversary does not have full information on what the honest players send, then key exchange and hence unconditionally secure encryption can be possible, even if parties do not share secret keys initially. One typically needs, however, a public but authenticated channel between the parties. It is easy to see intuitively why such an ingredient is necessary: if from A 's point of view, there is nothing that distinguishes B from any other party, there is no way she can be sure that she agrees on a key with B and not E , for instance.

The scenario with noisy channels may be a bit shaky in the sense that it depends on some limitation we assume on the adversary's technology: she cannot remove all noise on her channel. But there are other, possibly more robust settings, where the same basic principles apply. For instance, using quantum communication, where one sends single elementary particles to communicate, it can be shown that under the laws of quantum mechanics, it *impossible* for an eavesdropper to get full information on what is sent, if the information is coded in the right way into the state of the particles. This can be used to build quantum key exchange protocols that are unconditionally secure under sole assumption that the laws of quantum physics are correct.

The take-home message is that, while it is true that unconditionally secure encryption is usually not practical, this pessimistic view should be taken with grain of salt.

5.5 Exercises

Exercise 5.1 (Perfect security for any plaintext distribution) Assume a cryptosystem is perfectly secure for a given probability distribution of plaintexts. Show that this same cryptosystem is perfectly secure for any plaintext distribution on the same set of plaintexts.

Hint: remember that we have perfect security if and only if $P[y] = P[y|x]$ for any ciphertext y and plaintext x . Write P for probabilities with the given plaintext distribution. Take any other plaintext distribution and write P' for the probabilities we get with this new distribution. Of course, the choice of keys and

the encryption function are not changed. We assume $P[y] = P[y|x]$ and we want to show $P'[y] = P'[y|x]$. Argue first that $P[y|x] = P'[y|x]$. Then use this to show that $P[y] = P'[y]$.

Exercise 5.2 (Wheel of Fortune) Suppose you get to play the following variant of the bonus round in the TV show Wheel of Fortune (Lykkehjulet): you are shown N cards, each of which cover one letter. Each letter has been independently chosen from the same distribution, and you are given the distribution $(p_0, p_1, \dots, p_{25})$. You get to choose one letter from the alphabet, say you choose letter number i . Now every position in the hidden string where letter i occurs (if any) are uncovered. Your goal is to learn (on average) as much information as possible on the hidden string.

Of course this is a very crude model of Wheel of Fortune since we only consider single letter frequencies, but we want something that is feasible to analyze.

In the real-life version of the game, people tend to choose the most frequent letters as their guesses. Let's try to see what information theory has to say about this. Suppose we adopt the convention that Shannon used when defining Entropy: if you know that some event occurs with probability p , and you learn that this event did indeed occur, you have learnt $\log(1/p)$ bits of information.

1. Now, if your guess is letter nr. i , how many bits of information will you learn on average from playing the game (as a function of p_i and N)? Hint: note that you learn something for *every* position in the hidden string, namely either that letter i occurred here, or that it did not occur.
2. What strategy does your result suggest for choosing your guess, given frequencies p_0, \dots, p_{25} as in English?
3. Based on this, does it make sense that players in real life choose the most frequent letter(s)? why or why not?
4. Would this be a good strategy no matter what the frequencies were?

Exercise 5.3 (Conditional Entropy) Show that $H(X, Y) = H(Y) + H(X|Y)$. Show that this implies $H(X|Y) \leq H(X)$ with equality if and only if X and Y are independent.

Exercise 5.4 (Entropy of the key) Show that, in any cryptosystem, it holds that $H(K|C) \geq H(P|C)$. Under which condition do we have equality?

Exercise 5.5 (Entropies in the affine cipher) Compute $H(K|C)$ and $H(K|P, C)$ for the Affine cipher when used to encrypt a single letter from the English alphabet. Assume that keys and plaintexts are uniformly chosen.

Exercise 5.6 (Unicity Distance) In this chapter, we considered an adversary who looks at an n -letter ciphertext y and knows that the plaintext is in language L . He decrypts y under each possible key K . He discards K if the result is not meaningful (i.e., not in L) and keeps it otherwise. Of course, the correct key survives this test, and the adversary hopes that only the correct key survives. The unicity distance n_0 is defined to be the minimal value of n that is still large enough to have only the correct key survive.

We saw the lower bound

$$n_0 \geq \frac{\log_2(|\mathcal{K}|)}{R_L \log_2(|\mathcal{P}|)},$$

The real value of n_0 could in principle be larger than the lower bound. But in this exercise we will see that under a reasonable assumption on the cipher, in fact we expect n_0 to equal the lower bound.

The assumption is the following: *when the adversary decrypts y under an incorrect key, the result is a uniformly distributed plaintext.*

Of course this does not have to be exactly true always, but it is an approximation and a reasonable way to express the intuition that when you decrypt with an incorrect key, you get “random garbage”.

Let p_n be the probability that an incorrect key passes the adversary’s test as described above, when considering n -letter plaintexts. Recall that $|\mathcal{K}|$ is the number of possible keys. Then n_0 can be estimated as the smallest value for which it holds that

$$(*) \quad p_{n_0} \cdot |\mathcal{K}| < 1.$$

This is because $p_{n_0} \cdot |\mathcal{K}|$ is the number of keys we expect will pass the test by chance when we have n_0 letters, so the condition essentially say that no incorrect keys will survive¹.

1. Using the assumption above, find an expression for p_n in terms of n, H_L and $|\mathcal{P}|$.

Hint: It follows from the definition of H_L that $n \cdot H_L$ is a good approximation of the entropy of an n -letter plaintext. The entropy can be thought of as the number of bits you need to write the plaintext down, so the number of meaningful n -letter plaintexts is approximately 2^{nH_L} .

2. Use condition $(*)$ to show that under the assumption above we have $n_0 \approx \frac{\log_2(|\mathcal{K}|)}{R_L \log_2(|\mathcal{P}|)}$.

Exercise 5.7 (E has more noise) This exercise refers to the protocol from Section 5.4. Show that, if n_E and n_B are independent bits with $P[n_B = 1] = p$ and $P[n_E = 1] = q$, and we also have $q > 0, p < 1/2$, then $P[n_E \oplus n_B = 1] > p$. In other words, if we send a bit through two independent BSC’s, then we get more noise than we would have from one channel.

¹ strictly speaking, we should use $p_{n_0}(|\mathcal{K}| - 1)$ to account for the one correct key, but this makes no real difference when there are many keys.

Modern Symmetric Cryptosystems

%minitoc

6.1 The DES Blockcipher

The *Data Encryption Standard*, or DES, was the result of a process initiated by the US National Institute of Standards and Technology (NIST) in 1973. NIST decided that an encryption algorithm for use in the public sector was needed and asked for proposals. Eventually, in 1975, DES was adopted as a standard. It remained a standard much longer than expected, and was renewed last time in 1999. During that time it became an de-facto world wide standard and was the most used cipher in the world.

The algorithm was suggested by IBM, based on an earlier design called Lucifer. Interestingly, the National Security Agency (NSA) was also involved in the design of DES. This design introduced very substantial changes compared to Lucifer, including a much shorter key (56 bits for DES rather than 128 bits for Lucifer). NSA is an intelligence agency. It was and is one of the most secretive organizations in the world, well known to have great expertise in cryptography and very large resources both financially and scientifically. In the public debate, many people suspected that NSA's main interest was to be able to continue to spy on citizens, so when the DES design was published, there was considerable debate and suspicion that the NSA had planted some trapdoor in DES that enabled them to break it easily. This was only made worse by the fact that the design criteria for DES was, and still are, classified.

However, since then a huge amount of public research has been done on DES. It was, after all, the cipher everyone wanted to break for almost 30 years. It is fair to say that today we understand the design of DES quite well. All the evidence found suggests that DES was designed to be as strong as possible and the only problem was - and is - that the key is too short, only 56 bits. This was already pointed out in 1975. Although searching through all 2^{56} possible keys was not quite feasible back then, it certainly is today, and therefore it was clear around 2000 that something new was needed. This led to the Advanced Encryption Standard (AES) that we look at later.

The conclusion of this author is that the NSA most likely did its best to make DES be as strong as possible, but at the same time made sure that the key was a bit on the short side. One should remember that NSA was far ahead of anyone

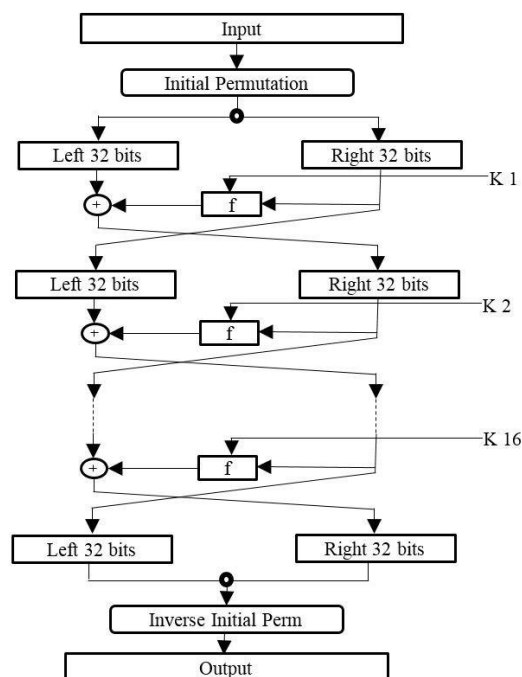


Figure 6.1 Feistel cipher structure, shown here for the case of DES, with the initial and final permutations.

else in terms of computing resources, especially back then, so such an agenda would create a situation where no one else could break DES, but NSA could do it by exhaustive search if they really had to. From the point of view of an intelligence agency, this seems like quite a desirable situation.

Anyhow, the fact remains that DES has an amazing track record: it resisted attacks for 30 years in the sense that the only practical attack is brute-force search for the key (some attacks exist that are only of theoretical value, we return to this later). This makes it well worth a study, even if it is not used anymore.

6.1.1 Feistel ciphers and DES

DES is a so-called Feistel cipher, named after Horst Feistel who was on the IBM team that designed DES. First of all, a Feistel cipher is *block cipher*, which means that the key is a bit string of fixed length and key generation just chooses a uniform key. Furthermore, it takes as input a bitstring of fixed length and outputs a ciphertext of the same length.

Second, a Feistel cipher has the following structure: encryption consists of repeating some computation a number of times, one such computation is called a *round*, and the number of rounds is denoted n . There is an algorithm called the *Key Schedule* that takes the key as input and outputs the rounds keys K_1, \dots, K_n , where each round uses its own round key.

More concretely, each round does the following: The input is a bitstring P that we split in two halves, called L_0, R_0 . Then for $i = 1 \dots n - 1$ we define, for a function f :

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

The last round is slightly different: we set

$$R_n = R_{n-1}, \quad L_n = L_{n-1} \oplus f(R_{n-1}, K_n).$$

The output ciphertext C is now defined to be $C = (L_n, R_n)$.

It is not hard to see that if you know the round key K_i , you can invert the i 'th round: given L_i, R_i , you can compute $L_{i-1} = R_i \oplus f(L_i, K_i)$, $R_{i-1} = L_i$, and so given the key, one can always decrypt a Feistel cipher.

Note that this works no matter what the function f does, so this means that one can design f to get the most secure cipher possible, without having other constraints in mind, for instance that f should be invertible.

Specifically, for DES, the size of the input and output is 64 bits and there are 16 rounds. The key schedule thus outputs 16 round keys, each such key is 48 bits, and contains some of the bits from the original 56 bit key, in some permuted order. We will not cover the details of the key scheduling here. This, and other details of DES that we leave out here, can easily be found on-line in the standard describing DES.

We proceed to explain what the f -function does, on input a 32 bit data block R and a round key K . The output can be written as

$$f(R, K) = P(S(K \oplus E(R)))$$

for functions E, S, P that we now explain.

$E(R)$ is 48 bits long and contains the 32 bits of R in a permuted order specified in the standard. In addition 16 of the bits in R appear twice in the output which is why it is longer.

$S()$ takes as input 48 bits and returns 32 bits. It splits its input, say X in 8 blocks X_1, \dots, X_8 of 6 bits each. Then it computes $S_i(X_i)$ for $i = 1 \dots 8$. Here each S_i is a function that takes 6 bits to 4 bits. These functions are known as substitution boxes, or S-boxes. They are specified by tables that list, for each input, what the output should be. Finally, the output is $S(X) = S_1(X_1) || \dots || S_8(X_8)$ where $||$ means concatenation of strings.

$P()$ is a permutation that takes 32 bits as input and returns 32 bits. It outputs the input bits in a permuted order that is fixed in the standard.

This concludes the description of DES, except that we should mention that in the actual standard there is a so-called initial permutation IP applied first on the bits of the plaintext, and then the inverse permutation is applied at the end. Since these permutations are fixed and public, they have no effect on security, and we will ignore them in the following.

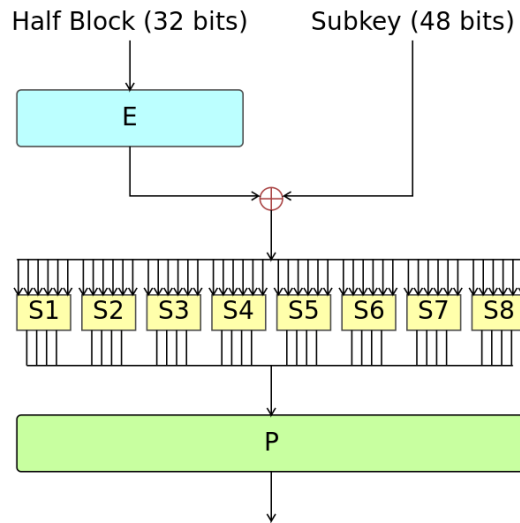


Figure 6.2 The DES f -function

6.1.2 Why is DES the way it is?

Many of the choices in the design of DES may seem quite arbitrary at first sight, but this is far from true, and it is interesting to take closer look at what was done.

The first major thing to notice is that every application of the f -function contains substitution, namely the S-boxes, and permutation, the function P . This is then repeated 16 times. In that sense, DES is a type of substitution-permutation cipher that Shannon was talking about many years earlier.

The second major property is that all the S-boxes are non-linear functions. What that means is that they cannot be described by multiplying the 6-bit input by a 6 by 4 binary matrix. This is crucial for security since everything else in DES *is* linear. So, had the S-boxes been linear as well, that would mean that DES encryption could be described as simply multiplying a vector containing plaintext and key (120 bits) by a 120 by 64 matrix to get the ciphertext. This would have been a disaster, since then, if an adversary would get hold of a plaintext block and the corresponding ciphertext block, he would effectively have 64 equations with the 56 bits of key as unknowns - which is easy to solve. In this sense, the S-boxes are the heart of the security of DES.

Furthermore, the S-boxes have been chosen such that if you flip one bit of the 6 inputs, then at least 2 of the 4 output bits will change. This, together with the design of P implies what is sometimes known as the *avalanche effect*: If we encrypt two almost equal blocks M and M' , say where only one bit differs between the two, we would of course want that the encrypted results look random and unrelated, even if the inputs were very similar. Now consider the encryption process for M and how it changes when we flip 1 bit to get M' instead. At first, 1

input bit to some S-box will change, in the first or in the second round. As a result at least 2 bits will change after the S-boxes. The permutation P is designed such that bits coming out of one S-box will be moved “far apart” so that the bits that changed will go into two different S-boxes in the next round. Once this happens, at least 4 bits will change, and so on. After a small number of rounds, the hope is that these changes will be all over the place and so the encryption of M' will look like it is completely uncorrelated to that of M .

Finally, one may wonder why we have the function E that expands from 32 to 48 bits, when we anyway compress back to 32 bits soon after? The answer can be seen if we consider what happens to the bits of the input R . Recall that 16 bits in R occur twice in the output $E(R)$. Each of these two occurrences are placed such that they go into 2 different S-boxes in the next step. This means that an adversary who does, say, a chosen plaintext attack will have a harder time manipulating input bits in such a way that only 1 S-box is affected. This makes it harder to avoid the avalanche effect. One might say that the effect to some extent is to simulate that we have one large function S , instead of 8 small S-boxes. But then, why did the designers not define one large function from 32 bits to 32 bits? The problem is that it is not clear how to do this in practice. The small S-boxes were designed to have the right properties by choosing carefully the output for each input and writing this down in a table. This is feasible because there are only 2^6 inputs in each case. In contrast, a table with 2^{32} inputs is of course completely out of the question.

6.2 The AES Blockcipher

The AES block cipher was adopted by NIST as a US standard in 2001, and has since then become a de-facto world-wide standard, even to the point where modern INTEL CPUs have an AES implementation on board, which can produce one AES block encryption per clock cycle using a pipelined approach.

AES was explicitly intended to be a replacement for DES, and NIST asked for proposals for the new standard in 1997. The requirements were a block length of 128 bits, and ability to support key lengths of 128, 192 and 256 bits. Contrary to the process leading to DES, the AES process was international and completely open. The authors of the 15 candidates came from 12 countries, and several conferences were held where the candidates were discussed and analyzed. Five candidate ciphers were selected for the final round and NIST stated in its announcement that all five were deemed secure. The winner, proposed by Belgian researchers Johan Daemen and Vincent Rijmen, was selected because it consistently performed well in the efficiency tests that were made in many different execution environments.

6.2.1 Specification of AES

The material in this section assumes you are familiar with finite fields as introduced in Section 3.4. AES uses the field \mathbb{F}_{2^8} which can be constructed using an

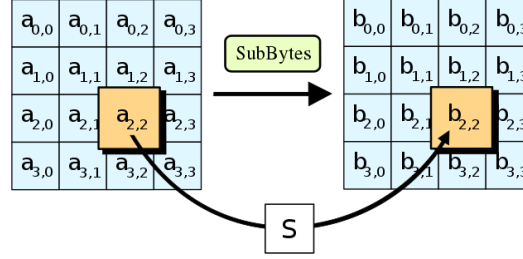


Figure 6.3 The AES SubBytes operation

irreducible polynomial over \mathbb{F}_2 of degree 8. The polynomial used in AES is the same as is used as example in Section 3.4.

As required, AES has a block length of 128 bits. It is an iterated cipher like DES with rounds, each of which consumes a round key, but it is not a Feistel cipher, as we shall see. When the key length is 128 bits, the number of rounds N is 10. For 192 bit keys, $N = 12$, and for 256 bit keys, $N = 14$.

There is a Key Schedule algorithm that we return to later. It takes the key as input and outputs $N + 1$ rounds keys K_0, \dots, K_N .

The plaintext block X is parsed as 16 bytes, which are put into a variable **State** which is a 4 by 4 matrix with bytes as entries. The algorithm puts **State** through a series of invertible operations, and the final value of **State** is the ciphertext. Underway, the operations SubBytes, ShiftRows, MixColumns and AddRoundkey are used, which we will describe below. Here is a summary:

1. Given plaintext X , set $\text{State} = X$, and set $\text{State} = \text{AddRoundkey}(\text{State}, K_0) = \text{State} \oplus K_0$.
2. For $i = 1$ to $N - 1$, do: $\text{State} = \text{SubBytes}(\text{State})$, $\text{State} = \text{ShiftRows}(\text{State})$, $\text{State} = \text{MixColumns}(\text{State})$, $\text{State} = \text{AddRoundkey}(\text{State}, K_i) = \text{State} \oplus K_i$.
3. In the final round, do: $\text{State} = \text{SubBytes}(\text{State})$, $\text{State} = \text{ShiftRows}(\text{State})$, $\text{State} = \text{AddRoundkey}(\text{State}, K_N) = \text{State} \oplus K_N$.
4. Output the content of **State** as the ciphertext.

Here are more details on each of the operations.

SubBytes. Applies the same (invertible) function **Sbox** to each of the bytes in **State**. On input a byte B , we interpret B as an element in the finite field \mathbb{F}_{2^8} , and compute a new byte A . If $B = 00000000$ then we set $A = B$, otherwise, we set $A = B^{-1}$. Now we interpret A as an 8-bit string a_7, \dots, a_0 . The final step is to do an affine transformation, where we multiply A by a fixed binary and invertible matrix, and add a constant string $C = c_7, \dots, c_0 = 01100011$. The computation can be written as follows: the result r_7, \dots, r_0 is computed as

$$r_i = a_i \oplus a_{i+4} \oplus a_{i+5} \oplus a_{i+6} \oplus a_{i+7} \oplus c_i,$$

where all indices are reduced modulo 8.

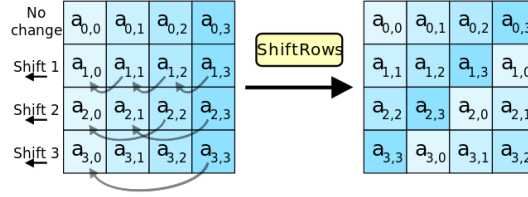


Figure 6.4 The AES ShiftRows operation

ShiftRows. Does a cyclic shift to the left on each of the rows in **State**. The number of positions shifted depends on the row, in that row number 1 is shifted by 0 positions, row 2 by 1 position, row 3 by 2 positions and row 4 by 3 positions.

MixColumns. Applies the same linear mapping to each of the columns in **State**. Let Col be one of the columns, and interpret it as a column vector with 4 entries t_0, t_1, t_2, t_3 in the field \mathbb{F}_{2^8} . Then multiply Col by a fixed and invertible 4 by 4 matrix M . This looks as follows:

$$M \cdot Col = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix} \cdot \begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix}$$

Note here that the elements in M are written as polynomials, as this is how we usually think of elements in \mathbb{F}_{2^8} . For instance, the polynomial $x+1$ corresponds to the byte 00000011. See Section 3.4 for details.

This concludes the specification of AES, except for the Key Schedule that we return to below. It should be clear that AES can be decrypted given the key, as all steps in the algorithm are invertible. We remark without going into details that there is a particularly nice way to decrypt: it turns out that because the last round is AES is a little different from the others, there is an alternative Key Schedule that outputs a modified series of round keys K'_0, \dots, K'_N , and if you do the encryption as specified above with K'_0, \dots, K'_N as round keys, this exactly implements decryption. Thus the same code or chip area can be used for both en- and decryption.

6.2.2 The AES key Schedule

We describe here only the key schedule for 128-bit AES, the algorithm for longer keys is very similar. Recall that we take as input a 128 bit key K and should output round keys K_0, \dots, K_{10} .

The algorithm works with the concept of a word, which is an array with 4 bytes as entries. It works with a sequence words called R_i , these are of the form $R_i = [r_i, 0, 0, 0]$, where r_i is defined by considering it as an element in \mathbb{F}_{2^8} and

defining it as $r_i = x^i$. Note that here, you should think of x as a (degree-1) polynomial. As a byte, it would be 00000010.

We also use the following operations:

$$\text{SubWord}(w_0, w_1, w_2, w_3) = (\text{Sbox}(w_0), \text{Sbox}(w_1), \text{Sbox}(w_2), \text{Sbox}(w_3)).$$

$$\text{RotWord}(w_0, w_1, w_2, w_3) = (w_1, w_2, w_3, w_0)$$

We then define a sequence of words, W_0, W_1, \dots as follows:

Parse K as 4 words K_0, \dots, K_3 and set $W_i = K_i$ for $i = 0, 1, 2, 3$.

The rest of the sequence is defined recursively as follows:

$$W_i = \begin{cases} W_{i-4} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus R_{i/4}, & \text{if } i \equiv 0 \pmod{4} \\ W_{i-4} \oplus W_{i-1} & \text{otherwise.} \end{cases}$$

We now define K_0 to be (W_4, \dots, W_7) , K_1 to be (W_8, \dots, W_{11}) and so on.

6.3 Differential and Linear Cryptanalysis

In this section we give a very brief account of the most powerful techniques known for attacking symmetric ciphers, namely differential and linear cryptanalysis. Both are applicable to any blockcipher and also many primitives for authentication such as hash functions.

6.3.1 Differential Cryptanalysis

Differential analysis is a chosen plaintext attack invented by Biham and Shamir. To understand the principle, consider an iterated cipher consisting of n similar rounds such as DES or AES. In the attack, the adversary selects randomly two plaintexts X, Y such that $\Delta = X \oplus Y$ is very carefully chosen in a sense we explain in a moment. A good way to think of this is that you start from a random X and flip the bits of X in a small number of carefully selected positions to get Y . Δ then has 1's in the positions where the bits were flipped.

Let X_1, \dots, X_n be the outputs of rounds $1, \dots, n$ when the input is X and define Y_1, \dots, Y_n similarly for input Y . Thus X_n, Y_n are the ciphertexts that the adversary will obtain during the attack. The goal of the adversary is to select Δ such that he can predict what $X_i \oplus Y_i$ are going to be, for $i = 1 \dots n$. If the cipher is properly designed, then for most values of Δ the avalanche effect will kick in and make this impossible. But with careful study of the cipher, the S-boxes etc., it may be possible to predict what will happen, at least with some probability. We return to this issue below.

So, the adversary selects X, Y for encryption and gets back the ciphertexts X_n, Y_n . Let Δ_i be his prediction of $X_i \oplus Y_i$, for $i = 1 \dots n$. If $\Delta_n = X_n \oplus Y_n$, we say that (X, Y) is a *good pair*, the idea is that the adversary can take this as evidence that his prediction was true (although he cannot be completely certain). Now, assume the adversary selects some value U for the last round key, and decrypts

both X_n and Y_n for one round, to get values $D_U(X_n)$ and $D_U(Y_n)$. For AES, for instance, this means you XOR with U and then apply the inverses of the MixColumns and SubBytes operations. Now we consider the equation

$$\Delta_{n-1} = D_U(X_n) \oplus D_U(Y_n). \quad (6.1)$$

If this condition holds, we say that the pair (X, Y) *suggests* the round key U . The crucial observation is that if the adversary's prediction holds (which is likely for a good pair) and if U is the correct last round key, then equation (6.1) will definitely be true. But if U is incorrect, it is reasonable to expect that $D_U(X_n)$ and $D_U(Y_n)$ are random garbage values and (6.1) will be true only by coincidence. We therefore conclude that a good pair will suggest the correct round key with high probability and will suggest only a small number of incorrect round keys.

This leads to the following rough sketch of a differential attack, which uses an array *Score* with as many entries as there are round keys.

1. Initialize all entries in *Score* to 0.
2. For $j = 1..T$ do:
 1. Select a random pair X^j, Y^j with $X^j \oplus Y^j = \Delta$, Submit it for encryption and get the ciphertexts X_n^j, Y_n^j back.
 2. If the pair X^j, Y^j is good, then for all values U of the last round key: if the pair X^j, Y^j suggests U then increment $Score[U]$.
3. Output the value U for which $Score[U]$ is maximal.

If the adversary's predictions hold with large enough probability, and if T is large enough, then this algorithm will output the correct last round key: these conditions imply that many of the pairs that are encrypted in the attack are such that the predictions hold. These pairs will all suggest the correct key, and any other key will only be suggested by random coincidence. We will not do a detailed analysis here, but we can say something about how large T must at least be: Say p is the probability that a pair is such that the predictions hold. The adversary certainly needs at least one such pair (and actually a few of them). So this means that T needs to be at least $1/p$ since $1/p$ is the expected number of pairs we need to make until we get one that satisfies the predictions.

The attack may not seem very good at first, as the algorithm will iterate over all possible round keys many times, and for many ciphers, including DES and AES, a round key contains (almost) as many bits as the entire key we are looking for. If we have to do this many iterations, we may as well just search exhaustively for the key. However, this can be dramatically optimized for many ciphers. The idea is to focus only on, say t bits of the last round key and use each possible choice of these bits to decrypt parts of X_{n-1}^j, Y_{n-1}^j . We then need only 2^t entries in *Score*, and though we can only check some of the bit positions involved in (6.1), this can still be enough that the correct choice of the t bits will have the highest score at the end. Once we have t bits of a round key, this usually gives t bits of information on the original key, and it may then be feasible to brute force the rest of the key.

How to make predictions

This leaves the question of how the adversary can hope to make the predictions necessary for the attack? The answer to this lies in the way rounds keys are usually used, namely they are added into the current state of the encryption using bit-wise XOR. To understand how this helps, consider AES as an example, and look at the first round of the encryption of X and Y , with $X \oplus Y = \Delta$. The first two steps is to add the first round key and then apply the **SubBytes** function. Then we apply **ShiftRows** and **MixColumns**. These are linear and hence their composition L is also linear. Now, Δ_1 , the XOR of the values we have after the first round will be

$$\begin{aligned}\Delta_1 &= K_1 \oplus L(\text{SubBytes}(K_0 \oplus X)) \oplus K_1 \oplus L(\text{SubBytes}(K_0 \oplus Y)) \\ &= L(\text{SubBytes}(K_0 \oplus X) \oplus \text{SubBytes}(K_0 \oplus Y)),\end{aligned}$$

where the last step is true because L is linear. So if we can predict

$$\text{SubBytes}(K_0 \oplus X) \oplus \text{SubBytes}(K_0 \oplus Y)$$

we can also predict Δ_1 . Now, note that we also know what the XOR is of the values that go into **SubBytes**, namely

$$(K_0 \oplus X) \oplus (K_0 \oplus Y) = X \oplus Y = \Delta.$$

Therefore, the questions to ask are the following: first, if we choose two random inputs to **SubBytes** with XOR Δ , and compute the XOR of the outputs, which value is most likely to occur? Let p_Δ be the probability that this most likely output-XOR value does occur. And second, for which value of Δ is p_Δ maximal?

This optimal value of Δ is the one for which we have the best possible chance of predicting Δ_1 . Crucially, this analysis depend only on **SubBytes** and not on the key!

The optimal value of Δ may not be easy or even feasible to find, but this still shows how a study of **SubBytes** and hence the AES S-box can help to predict Δ_1 from Δ . Now, you can ask the same questions on the second round, except that now the input XOR is Δ_1 , and look for the value Δ_2 that is most likely to be the XOR of the outputs of the second round. We can go over all rounds this way, and for each round number i , there will be a probability p_i that our prediction holds in that round. The probability that we are lucky in all rounds is then $p = \prod_{i=1}^n p_i$, assuming that what happens in different rounds is independent. This is not really true, but still the product seems to be a good approximation in practice.

This explains why differential analysis gets harder as we increase the number of rounds: for larger n , p gets smaller very fast, and as we saw above, the adversary will need at least $1/p$ pairs to be able to do the attack. So for a large enough number of rounds, the attack will need as many encryption operations as exhaustive key search, and so has become useless.

6.3.2 Linear Cryptanalysis

Linear analysis is a known plaintext attack and was invented by Matsui. It is a known-plaintext type of attack. Some notation: for a bit string X of length m and a subset I of the indices $1, \dots, m$, we define $X[I] = \bigoplus_{i \in I} X[i]$, the XOR of positions in X pointed to by I .

Now, let X be a plaintext, and as before X_1, \dots, X_n be the state after rounds $1, \dots, n$. The adversary's goal now is to find index sets I, J, L such that the bit $X[I] \oplus X_{n-1}[J] \oplus K[L]$ is biased, that is, the probability q that it is 0 for a random X is either close to 1 or close to 0. This will usually not be the case: a properly designed cipher will contain many applications of highly non-linear functions and so most expressions of this form will be random bits: q will be almost exactly $1/2$.

Still, by careful study of the cipher, the adversary may be able to find such a biased expression. If he succeeds, then the following attack can be applied, which uses an array *Score* with as many entries as there are round keys. Note that if we keep the key constant but take a random X , then $K[L]$ is constant. So if $X[I] \oplus X_{n-1}[J] \oplus K[L]$ is biased, then so is $X[I] \oplus X_{n-1}[J]$: it will be 0 with probability q or $1 - q$, depending on the value of $K[L]$.

1. Initialize all entries in *Score* to 0.
2. For $j = 1..T$ do:
 1. Get a plaintext-ciphertext pair X, X_n .
 2. For all values U of the last round key, decrypt X_n for 1 round. If $X[I] \oplus D_U(X_n)[J] = 0$ increment *Score*[U].
3. Output the value U for which $|\text{Score}[U] - T/2|$ is maximal.

The reason why this works is that when U is correct the expression we calculate is very likely or very unlikely to be 0. Whereas an incorrect key is likely to produce a random bit. Therefore we expect that an incorrect key gets a score of $T/2$, while the correct key gets either a much higher or a much lower score. For this to work, T needs to be large enough that the score of the correct key stands out. A detailed analysis of this (which we will not do here) suggests that T needs to be proportional to ϵ^{-2} , where $\epsilon = q - 1/2$ is the *bias* of the expression.

Like differential analysis, this attack does not really need to iterate over all rounds keys: we just need to be able to find the bits that occur in $D_U(X_n)[J]$, and this may only require us to iterate over some of the bits in U .

Finding a biased expression is the hardest part of the attack. It requires a careful study of the non-linear parts of the cipher, usually the S-boxes. A standard approach is find the best possible linear approximation of the S-box, more concretely, we look for a selection of some of the input and output bits for the S-box, such that XOR of these bits is biased. Then the idea is to combine such expressions for one round with similar expressions for S-boxes in the following round. Continuing this way, we can get an expression of the form needed in the attack. The bias we get depends on the bias of the expressions for single rounds.

Unfortunately (for the adversary), the resulting bias tends to vanish very fast (exponentially) as we increase the number of rounds.

6.3.3 Security of DES against Differential and Linear Analysis

When Biham and Shamir invented differential analysis in the late 80-ties, their motivation was to attack DES. They found that they could break reduced versions of DES with a smaller number of rounds, but for 16 rounds the best attack they found was only very slightly faster than exhaustive key search. They also found that many slight modifications to DES, such as a different order of the S-boxes, would make it much weaker against the attack.

Later, Don Coppersmith stated in public that he and the rest of the IBM team that designed DES, were aware of differential analysis at the time, but were mandated by the NSA not to disclose anything.

Thus it seems very likely that DES has 16 round exactly because this is what you need to make differential analysis as slow as exhaustive key search.

Matsui invented linear analysis in the early 90-ties. It was first applied to his own FEAL cipher and soon after to DES. The best known version of the attack requires 2^{47} known plaintexts, and so is faster than exhaustive key search from a theoretical point of view. Thus it seems unlikely that NSA knew about linear analysis in the 70-ties.

However, when considering the practical consequences of these attacks, one should remember that because they are chosen or known plaintext attacks, an adversary who tries to do the attack in real life, will have to get the owner of the key to do all the encryptions required. For linear analysis of DES, for example, the adversary will have to wait until the key has been used to encrypt 2^{47} blocks and then get hold of all the plaintext blocks! This is completely unrealistic: for one thing, any sensible system will replace the key long before it has been used on that much data. Another issue is that if the adversary does his own search for the key, he can optimize as much as he can afford, use parallel hardware, etc., while in a known plaintext attack, he is limited to the speed at which the attacked system runs. Thus a brute force attack remains the only practical threat against DES - quite remarkable for a cipher that dates back to the 70-ties.

6.3.4 Security of AES against Differential and Linear Analysis

AES has been shown to be resistant to both differential and linear cryptanalysis. The reason why this can be done is a combination of the design of the Sbox function, and the carefully designed combination of ShiftRows and MixColumns. For the intuition on this, suppose we flip a single bit in the State-value that is input to a round i . This will change the output of one Sbox-value, and after MixColumns this change can potentially affect an entire column of the state coming out of round i . So, round $i + 1$ will start with changes in an entire column which will propagate through the applications of Sbox, because Sbox is a permutation. Now,

note after **ShiftRows** the entries from a column will appear in all 4 columns, and so after the **MixColumns** of round $i + 1$, the entire state can be affected by a single bit-flip occurring at round i . This makes differential analysis hard and is the basis of the proof that this technique will not be effective despite the relatively small number of rounds.

As for linear analysis, note that not only is **Sbox** not linear, it is very far from being linear! What this means more concretely is that no linear function is a good approximation: any linear function you choose will, on many inputs, produce an output different from that of **Sbox**. The reason why this can be shown is the use of the $B \mapsto B^{-1}$ function, this function is known to be close to optimal with respect to being “far from linear”. In case you were wondering, one of the reasons why there is also the affine transformation at the end is that when we use the inverse function, we have to send 0 to itself, as it has no inverse, so if nothing more was done, the **Sbox** would have a fixed point, and this would be a bad thing. It is because of this high non-linearity of the **Sbox** that AES is resistant to linear cryptanalysis.

As mentioned above, both differential and linear cryptanalysis get harder as the number of rounds increase, in fact the success probability of the attacks decay exponentially with the number of rounds. Since the designers of AES could show bounds on the best possible differential and linear analyses of one round, they could compute exactly how many rounds would be needed in order for both attacks to be as slow as exhaustive key search. This is the reason for AES having 10, 12 and 14 rounds for key lengths 128, 196 and 256.

Of course, all this is not a guarantee that no other attacks exist, but to date nothing has been found.

6.4 Defining Security of Symmetric Encryption

6.4.1 Deterministic Systems as Pseudorandom Functions

As running example cryptosystem for this subsection, think of DES that we just described. DES, and any other block cipher, is deterministic, that is, the encryption algorithm makes no random choices. Therefore, as soon as we have specified the key and the plaintext, the ciphertext is uniquely determined. Put another way, each key specifies a function from \mathcal{P} to \mathcal{C} (in fact a permutation), and hence systems of this kind are also called *Function Families*.

Any such deterministic cryptosystem has severe limits on the kind of security it can provide: if I send the same message today and tomorrow, and I’m using the same key, then I will send the same ciphertext twice, and an adversary listening in can conclude that indeed I sent the same message twice. Even such a limited piece of information can be useful, and we would certainly like to hide even such relations between messages sent at different times.

This means that block ciphers cannot be used directly for encryption of data. Nevertheless, it is useful to talk about their quality because they can be used as building blocks in systems with better security, as we shall soon see.

So what characterizes a good block cipher? One natural property that comes to mind is that even if the adversary knows the plaintext for a given ciphertext, the ciphertext appears to be randomly chosen nonsense, with no obvious relation to the plaintext - as long as the key is unknown. To be more concrete, think of DES as an example, and let's assume we give the adversary a chosen message attack. This means that for some fixed (but random) key k , the adversary gets to choose x_1, x_2, \dots and gets to see $y_1 = \text{DES}_k(x_1), y_2 = \text{DES}_k(x_2), \dots$. Now, according to the intuition we just outlined, we hope that this appears to the adversary as if y_1, y_2, \dots were completely randomly chosen as the images of x_1, x_2, \dots .

Of course, choosing a random DES key, and using it to map x 's to y 's is not the same as choosing a random function. You can get at most 2^{56} different mappings this way, whereas the total number of functions from $\{0, 1\}^{64}$ to itself is astronomically larger: $2^{64 \cdot 2^{64}}$. But it might still be the case that to an attacker with limited resources, DES encryption with a random key *appears to* behave like a random function, we say that the set of DES encryption functions is a *pseudorandom* function family.

To define this concept precisely, and more generally, we consider a family of functions $\{f_K \mid K \in \{0, 1\}^k\}$ where each f_K is a function $f_K : \{0, 1\}^n \mapsto \{0, 1\}^m$. We also consider a probabilistic algorithm A (the adversary) which is placed in one of the following two scenarios, and is asked to guess which one it is in (so it outputs its guess as one bit):

The ideal world: A gets access to an oracle O_{Ideal} which initially chooses a random mapping R from $\{0, 1\}^n$ to $\{0, 1\}^m$ (uniformly among all such mappings), and then, when A sends an input x , it answers with $R(x)$.

The real world: The adversary gets access to an oracle O_{Real} which initially chooses K at random from $\{0, 1\}^k$, and fixes K for the duration of the game. After this, on input x , it answers with $f_K(x)$.

Defining the Distinguishing Advantage

We want to define security of the function family by saying that good pseudorandom function security means that an adversary would have a hard time distinguishing between the real and the ideal world. So we need a way to measure how good a job an adversary is doing when he tries to guess which of the two oracles he is talking to. We do this by defining the *advantage* with which an algorithm can distinguish two oracles:

So consider a probabilistic algorithm A that runs with access to either oracle O_0 or O_1 . At the end A outputs a bit which we think of as its guess at which oracle it was talking to. Let $p(A, 0)$ be the probability that A outputs 1 when it was talking to O_0 . Likewise let $p(A, 1)$ be the probability that A outputs 1 when it was talking to O_1 . These probabilities are taken over A 's random choices, and also over the choices made by the oracle in the two cases. The *advantage of A in distinguishing O_0 from O_1* is defined to be

$$\text{Adv}_A(O_0, O_1) = |p(A, 0) - p(A, 1)|.$$

If the advantage is very small (close to 0), this implies that A is guessing in essentially the same way, regardless of which oracle he talks to, so small advantage indeed means that A has almost no idea which case he is in. Conversely, if the advantage is very large (close to 1), it means that from A 's answer we can infer almost with certainty which case he was in.

Using the notion of advantage, we can then define PRF security

Definition 6.1 (PRF security) We say that $\{f_K \mid K \in \{0,1\}^k\}$ is a (t, q, ϵ) -secure *pseudorandom function family* (PRF), if any adversary A that runs in time at most t and makes at most q calls to the oracle, satisfies $\text{Adv}_A(O_{\text{Real}}, O_{\text{Ideal}}) \leq \epsilon$.

The convention is that we include in the running time also the time spent by the oracle, this makes some of the results in the following simpler to state. Intuitively, you can think of the first two parameters in the definition (t, q) as a measure of how powerful the adversary is, and ϵ as a measure of how successful his attack is. So the if a cryptosystem satisfies the definition for some choice of parameters, this basically says that as long as the adversary is not too powerful, there is a limit to how successful he can be.

Note that a deterministic cryptosystem (G, E, D) where keys are chosen uniformly (as is virtually always the case for symmetric encryption) is a special case, by letting $\{f_K \mid K \in \{0,1\}^k\} = \{E_K \mid K \in \{0,1\}^k\}$.

If we take DES as an example, for which values of t, q and ϵ could we hope that DES satisfies this definition? First thing to notice is that DES with a fixed key is not just a function on \mathcal{P} , it is a *permutation*. So no matter how many inputs and corresponding outputs the adversary knows, he will never see a case where two inputs form a *collision*, i.e. they are mapped to the same output. By contrast, a random function will tend to have such collisions, indeed, if the adversary asks for outputs resulting from about $\sqrt{2^{64}} = 2^{32}$ distinct inputs, then there is significant probability that collisions will occur. This follows by the well-known "birthday paradox". So we cannot hope for security as we defined it unless the adversary asks well under 2^{32} queries. In fact, if you ask q queries, then for small values of q , the probability of a collision in the ideal world will be approximately q^2 divided by the number of possible blocks (2^n). So it will certainly not be more than 2^{-20} in case of DES and 2^{20} queries. Another type of brute-force attack is that the adversary can always first ask 1 or 2 queries and then search through all possible keys until one is found that matches the data he has. Such a match will indeed happen in the real world if the adversary happens to choose the right key, but will only happen by coincidence (and so with very small probability) in the ideal world. So we also cannot hope for security if the adversary has time comparable to the number of possible keys, 2^{56} . In fact, if he is able to test x keys, he will find the right one with probability $x/2^{56}$, so for an attack that has time enough to test x keys the advantage will be approximately $x/2^{56}$. See also Exercise 1. Even if we are optimistic on behalf of the adversary, he will certainly need at least one CPU instruction per round of DES, so in time 2^{40} , he can test at most $2^{40}/16 = 2^{36}$ keys, so the probability of hitting the right one is $2^{36}/2^{56} = 2^{-20}$.

Taking this into account, it seems that $(2^{40}, 2^{20}, 2^{-20})$ -security is reasonable to

conjecture. Given the current lack of lower bounds in complexity theory, this can be nothing but a conjecture. Nevertheless, some assumption of this type is needed to have security at all, and the numbers reflect the security we can assume for DES if there are no other attacks on it than the two generic ones we considered. If we had looked instead at AES, which has block- and keylength 128 bits, then much better parameters can be expected, say $(2^{80}, 2^{45}, 2^{-30})$, by the same arguments as above, and allowing some security margin for attacks that may do better than simple search for the key.

The general real/ideal world approach to a definition we have used may not seem like the most natural one at first sight. Why don't we just say that it should be difficult to find the secret key? or that it should be difficult to find any partial information about the plaintext once it has been encrypted? Indeed many such sensible definitions could be proposed. In fact the list of variants is almost infinite. The nice thing about the real-or-ideal-world definition is that it is strong enough such that if a cryptosystem satisfies it, then it also automatically satisfies *any* of the variants we just sketched. The reason is that the ideal world is constructed such that any type of non-trivial "break" is clearly impossible. And then if the adversary cannot even tell if he is attacking the real or the ideal system, he certainly can't succeed in the real world either, for any reasonable definition of what "succeeding" means.

Another question one might ask about the definition is why we try to compare a block cipher to a random *function*. After all, a block-cipher encryption function must be a permutation, so would it not be more natural to ask in the definition that the block cipher looks like a random *permutation*? – that is, we could define the ideal world so it will never output the same answer to different queries. This would in fact be a meaningful definition, namely of *pseudorandom permutations* (PRP). The reason why we prefer to define PRF's is that in the following section we will look at various "modes of operation" for block ciphers such as CBC and Counter mode, and we will sketch a proof that these modes form secure encryption algorithms if the underlying block cipher is a secure PRF. If we were to assume instead that the block cipher is a PRP, the proof of the same result would be more cumbersome and give worse parameter values.

6.4.2 Probabilistic Systems and CPA security

Probabilistic encryption schemes are cryptosystems where the encryption algorithm is probabilistic (but usually the decryption is deterministic). Using such schemes is necessary in practice to solve the problem we pointed out before, namely that for a deterministic scheme, encryption of the same message twice under the same key will give identical ciphertexts and this can be observed by the adversary.

The approach to defining secure symmetric encryption in the probabilistic case is similar to what we did before: we will compare it to an "ideal situation", where it is obvious that an attacker will get nowhere. If an attacker A cannot tell

whether he is attacking the real system (G, E, D) or the ideal one, then we say the system is secure against A .

However, a probabilistic cryptosystem can achieve more security than a deterministic one, in fact, we will require that the adversary cannot tell the difference between a real encryption of a message x he chooses, and a completely random ciphertext chosen with no relation to x .

Clearly, in a world where random encryptions are being sent in place of encryptions of the actual messages, the adversary has no chance of figuring out any interesting information at all. For instance, he cannot tell if we send the same message twice. So if he cannot tell the real world from the ideal one, then we have the best possible security.

To model this, we consider a cryptosystem (G, E, D) and an adversary A which is placed in one of the following two scenarios, and is asked to guess which one he is in (so he outputs his guess as one bit):

The ideal world: A gets access to an oracle O_{Ideal} which on input a plaintext x answers with $E_K(r)$, where r is a randomly chosen message with the same length as x , and K is produced by G , but fixed in the entire attack.

The real world: A gets a access to normal chosen message attack: an oracle O_{Real} which on input a plaintext x answers with $E_K(x)$, where K is produced by G , but fixed in the entire attack.

Definition 6.2 (Chosen-Plaintext Attack(CPA)-security) We say the cryptosystem (G, E, D) is (t, q, μ, ϵ) -secure, if for any adversary A that runs in time at most t , and makes at most q calls to the oracle, with plaintexts consisting of a total of μ bits, it holds that $Adv_A(O_{Real}, O_{Ideal}) \leq \epsilon$.

Probabilities and running times can be formalized in the same as we explained in the previous section for pseudorandom functions.

The reason for the parameter μ , which we did not have in the PRF definition, is that some of the systems we will look at in the following can handle plaintexts of many different lengths. Thus, if we want to measure how much information the adversary has obtained from the oracle, it is not enough to count the number of calls, we must also look at how many bits the adversary asked to have encrypted.

6.5 Good Symmetric Encryption from Pseudorandom Functions.

Several standards give schemes by which cryptosystems like AES and DES can be expanded to cryptosystems which can first handle plaintexts of any length, and second can be probabilistic, with the security advantages this can imply. With the machinery developed above, it is possible to prove that several of these mechanisms are actually CPA-secure, provided the block cipher used forms a good PRF.

6.5.1 CBC Encryption

Let a block cipher be given, i.e., a deterministic cryptosystem (G', E', D') , where \mathcal{P}, \mathcal{C} are fixed, and $\mathcal{P} = \mathcal{C} = \{0, 1\}^n$ for some n .

CBC encryption can be thought of as a way to make a cryptosystem (G, E, D) from the PRF defined by the block cipher (G', E', D') . The first thing is to set $G' = G$. The plaintext set for the new system will be all strings of length divisible by n . We make this restriction only for simplicity, a small modification of the construction will allow handling strings of any length. To do encryption we choose a random n -bit string y_0 , and we split the input x into n -bit strings x_1, \dots, x_t . Then we define for $i > 0$ that $y_i = E_K(y_{i-1} \oplus x_i)$. The output ciphertext is y_0, y_1, \dots, y_t . Decryption is straightforward.

It is now possible to show the following [3]:

Theorem 6.3 *Suppose (G', E', D') is a (t', q', ϵ') -secure PRF. Then CBC encryption based on this system is CPA (t, q, μ, ϵ) -secure for any q , and for*

$$\epsilon = \epsilon' + \left(\frac{\mu}{n}\right)^2 \cdot \frac{1}{2^n}$$

provided that

$$t \leq t', \quad \frac{\mu}{n} \leq q'$$

Intuitively, what this results says is the following: as long as CBC encryption using (G', E', D') is attacked by an adversary who is no more powerful than what (G', E', D') can handle, the success will be no better than $\epsilon = \epsilon' + \epsilon' + \left(\frac{\mu}{n}\right)^2 \cdot \frac{1}{2^n}$.

So what the result basically talks about is the difference between ϵ and ϵ' . Let's call a string of n bits a *block*. This is the amount of text the original system can handle in one go. Then we see that the difference between ϵ and ϵ' is essentially the square of the number of blocks we encrypt, divided by 2^n . In other words, if we always CBC encrypt much less than $2^{n/2}$ blocks before we change the key, then the advantage with which CBC mode can be broken is essentially the same as the advantage with which the original system can be distinguished from a random function.

Note that although the results says that ϵ may be greater than ϵ' , this does NOT mean that CBC mode is less secure than the block cipher itself. We have to remember that the two types of security involved are completely different. What the result really says is that if the original system is secure in a weak (PRF) sense, then CBC mode based on this system is secure in the much stronger CPA-sense, but essentially with the same advantage, as long as we do not encrypt too much data.

Proof of Theorem 6.3

We introduce a new game we call a *hybrid*, which we think of as being somewhere “in between” the real and the ideal world as defined above. In this game, the Oracle chooses a random function R from n bits to n bits, and on input some message m from the adversary, the oracle “encrypts” m using CBC mode, but where the

block cipher encryption function is replaced by R . We define $p(A, hybrid)$ as the probability that A outputs 1 when playing this hybrid game, and likewise $p(A, ideal), p(A, real)$ are the probabilities of output 1 when A talks to O_{ideal} or O_{real} , respectively.

Now since the *only* difference between the hybrid and the real game is that $R()$ is replaced by $E_K()$, we must have that

$$|p(A, real) - p(A, hybrid)| \leq \epsilon'.$$

If this was not the case, A could be used to distinguish between $E_k()$ and a random function with advantage greater than ϵ' , contradicting our assumption about the block cipher.

Then observe that the ideal game is completely equivalent to a game where the Oracle on input an N -block message always returns a randomly chosen sequence of $N + 1$ blocks. We can then argue that the hybrid with high probability does exactly this (and so it is also hard to tell the hybrid and ideal cases apart. Define the event BAD as follows: BAD occurs, if at any point during the hybrid game, the function R receives an input that it has received before in this game, in this case we say a *collision* occurs. Now, if BAD does not occur, since R is a random function, all outputs generated by R will be random blocks, chosen independently of anything else¹ - which means that the output we generate is completely random, exactly as in the ideal case. Hence, A 's only chance of telling the ideal and the hybrid game apart is if BAD occurs, i.e.,

$$|p(A, hybrid) - p(A, ideal)| \leq Pr(BAD).$$

This, together with $|p(A, real) - p(A, hybrid)| \leq \epsilon'$ from above implies

$$Adv_A(O_{real}, O_{ideal}) = |p(A, real) - p(A, ideal)| \leq \epsilon' + Pr(BAD).$$

So now we just have to estimate $Pr(BAD)$. Let M_j be the event that a collision occurs after j calls to R . Clearly $P(M_1) = 0$. Using the standard formula for conditional probabilities, we can see that

$$P(M_j) \leq P(M_{j-1}) + P(M_j | \neg M_{j-1})$$

The last probability on the right hand side is equal to $(j-1)/2^n$: First, since M_{j-1} did not occur we have seen $j-1$ different inputs before. Second, the new input nr. j is the XOR of some message block and an independently chosen random block (either a y_0 -value chosen by the oracle or an output from R), it is therefore uniformly chosen.

We conclude that in fact

$$P(M_j) \leq (1 + 2 + \dots + (j-1))/2^n \leq j^2/2^n$$

¹ Note that this part of the argument depends crucially on the fact that we start from a block cipher that we assume is a PRF, since this means we can replace it by the random function R . Had we assumed a PRP, R would be a random permutation, and it would no longer be true that all output values are independently chosen!

Since the total number of calls is at most μ/n , it follows that $P(BAD) \leq \frac{\mu^2}{n^2 2^n}$ and we are done.

Tightness of the bound for CBC.

What happens if we do encrypt $2^{n/2}$ blocks? The result above does not say that anything really bad happens: the bound on ϵ becomes useless, but this does not mean that any attack exists. However, it turns out that CBC encryption does leak information in this case. The reason is the following: because we have a total of 2^n possible blocks, in a set of $2^{n/2}$ random blocks there will be two that are equal with significant probability. So it happens with quite large probability that two ciphertext blocks y_i, y_j , where $i, j > 0$ are equal (they need not come from the same plaintext). Let x_i, x_j be the corresponding plaintext blocks. Then it follows that

$$E_{k_e}(y_{i-1} \oplus x_i) = y_i = y_j = E_{k_e}(y_{j-1} \oplus x_j)$$

and consequently that $y_{i-1} \oplus y_{j-1} = x_i \oplus x_j$. So from the ciphertext we can compute at least some non-trivial information about the plaintext. In a sense, the above theorem says that if we assume the underlying block cipher is secure, then the attack we just outlined is the best possible against CBC encryption.

For DES, this means that one should not encrypt more than 2^{32} blocks of data under CBC with the same key. This is not too bad a restriction: it corresponds to about 1000 Gbyte (!). Nevertheless, it is an important observation that the blocksize of a cryptosystem influences the security of the CBC construction in this way.

Let us look at a numeric example. Suppose we are willing to believe that DES is a $(2^{40}, 2^{20}, 2^{-20})$ -secure system. Then the parameters for DES based CBC will be $t = 2^{40}, \mu = 2^{26}, \epsilon = 2^{-20} + 2^{-24}$. Of course we do not know how to rigorously prove that DES has the parameters we assumed - but they are not unreasonable, given the known attacks on DES. If we were to use two-key triple DES or AES instead as the basic system, much better parameter values could be reasonably assumed, i.e. larger t, μ and smaller ϵ . In particular, assuming that AES is a $(2^{80}, 2^{45}, 2^{-30})$ -secure PRF, we would get that CBC based on AES is secure for $t = 2^{80}, \mu = 2^{53}$ and $\epsilon = 2^{-30} + 2^{-38}$.

6.5.2 Counter (CTR) Mode

This is a construction of the same form as CBC encryption, so we use the same notation as before. The only difference is that the encryption function is defined as follows:

We choose a random n -bit string y_0 , and we split the input x into n -bit strings x_1, \dots, x_t (we assume for simplicity that the length of x is always a multiple of n). Then we define for $i > 0$ that $y_i = E_K(y_0 + i) \oplus x_i$. The output ciphertext is y_0, y_1, \dots, y_t . Here $y_0 + i$ means that you write i in binary and do the addition modulo 2^n . Decryption is straightforward and is left to the reader.

It is interesting to note some differences between CBC and CTR mode, that

make CTR mode preferable to CBC in most practical cases: first, CTR mode only requires the encryption function of the underlying block cipher to be implemented, and second CTR can be very easily parallelized, in contrast to CBC that is inherently sequential.

Just as before there is a result linking the security of CTR mode to that of the original scheme:

Theorem 6.4 *Suppose (G', E', D') is (t', q', ϵ') -secure PRF. Then Counter mode based on this system is CPA (t, q, μ, ϵ) -secure for any q , and where:*

$$t \leq t' \quad \frac{\mu}{n} \leq q' \quad \epsilon = \epsilon' + \left(\frac{\mu}{n}\right)^2 \cdot \frac{1}{2^n}$$

PROOF Just like in the proof for Theorem 6.3, we define a game called Hybrid, where the oracle encrypts the correct message, but uses a random function R in place of the block cipher. Again, we can see that as long as there are no collisions on the inputs to R , the hybrid game will produce output with exactly the same distribution as the ideal game. We define the event BAD to mean that such collision occur, and then we can show in exactly the same way as in Theorem 6.3 that the adversary's advantage is bounded by $P(BAD)$.

We let B_j denote the probability that any of the blocks used as input to R during the j 'th call to the oracle collide with blocks used in other calls. And we let b_j denote the number of blocks encrypted during the j 'th call. Consider a single block input to R during the j 'th call. This block is of form $y_0 + i$ for some i and is uniformly chosen because y_0 is uniform, and it is also chosen independently of all blocks used in other calls. There are at most μ/n such other blocks, so the probability that $y_0 + i$ collides with any of these is at most $\frac{\mu}{n2^n}$. And so by the union bound, $P(B_j) \leq \frac{b_j \mu}{n2^n}$. Clearly if BAD occurs, at least one of B_j 's must occur, so again by union bound and because the total number of blocks encrypted is μ/n , we have

$$P(BAD) \leq \sum_{j=1}^q P(B_j) \leq \frac{\mu}{n2^n} \sum_{j=1}^n b_j = \left(\frac{\mu}{n}\right)^2 \cdot \frac{1}{2^n}.$$

The theorem follows. □

6.6 Exercises

Exercise 6.1 (Feistel Decryption) Show that one can decrypt any Feistel cipher by running the encryption algorithm while using the round keys in reverse order.

Exercise 6.2 (The DES complementation property) For a bit string X , let \bar{X} denote the complement of X , that is, the string obtained by flipping all bits in X . Show that for any plaintext block X and DES key K , it holds that if $Y = DES_K(X)$, then $\bar{Y} = DES_{\bar{K}}(\bar{X})$. Also show that, given a chosen plaintext

attack where you may ask for the encryption of 2 plaintexts, you can use this property to do exhaustive key search in half the time it would normally take.

Exercise 6.3 (Meet in the middle attack) . When it became clear that a single DES key is too short - but before AES was available, many people tried to devise ways to use DES to build a better block cipher with larger key. One simply suggestion is Double DES encryption: use two keys K_1, K_2 and define encryption of a block X to be $DES_{K_2}(DES_{K_1}(X))$. This would seem to be a block cipher with 112 bit keys, and one would hope that a brute force attack would require time corresponding to 2^{112} encryptions. However, this not at all true, as we shall see in this exercise.

Note that if $Y = DES_{K_2}(DES_{K_1}(X))$, then $DES_{K_2}^{-1}(Y) = DES_{K_1}(X)$, where DES^{-1} denotes decryption. If $(K'_1, K'_2) \neq (K_1, K_2)$, then what is the probability that $DES_{K'_2}^{-1}(Y) = DES_{K'_1}(X)$? You may assume that if you decrypt or encrypt with an incorrect key, then you get a uniformly random block (strictly speaking, not quite true, but an OK approximation in practice).

Assume you get a known message attack on Double DES encryption, where you get two pairs of matching plaintext/ciphertext blocks, $(X_1, Y_1), (X_2, Y_2)$. Design an algorithm for finding the key. You need to build a table with 2^{56} entries and you should do no more than $4 \cdot 2^{56}$ encryptions. Use the answer to the first question to find a bound on the probability that your algorithm outputs an incorrect key.

Exercise 6.4 (PRF security) In this exercise, we verify that the definition of security for pseudorandom functions, (or block ciphers) correctly captures the intuition that there cannot be much security if the adversary has enough computing power to search exhaustively for the key. So consider a block cipher with k -bit keys and n -bit blocks, and where key generation just consists of choosing a random k -bit string. Let $time(m)$ be the time needed to do m encryption operations. Consider the following adversary A who plays the game defined in section 4.1:

A chooses T distinct plaintext blocks M_1, \dots, M_T , and submits them to the encryption oracle. The oracle sends back blocks C_1, \dots, C_T . A now chooses m different keys and computes for each key the encryption under the key of M_1, \dots, M_T . If a key is found where the encryption results match all C_1, \dots, C_T , then A outputs 1, else A outputs 0.

Clearly, A runs in time $time(mT)$ (we ignore the time needed for the comparisons). Now:

- Compute a lower bound on the probability that A outputs 1 in the real world, where he is talking to the real block cipher.
- Compute an upper bound on the probability that A outputs 1 in the ideal world where he is talking to a random function.
- Use the bounds to show that the advantage of A is at least $\frac{m}{2^k} - \frac{m}{2^{nT}}$. You may assume that $nT > k$, this will always be the case with realistic values of parameters.
- Suppose someone claims that the blockcipher is a $(time(mT), T, \epsilon)$ -secure PRF.

The existence of A implies that for some values of ϵ , this claim must be false. Concretely, specify an interval I with the property that if $\epsilon \in I$, then the existence of A implies the claim is false.

Exercise 6.5 (Partial Plaintext Knowledge) Suppose we are given a cryptosystem (G, E, D) . Assume an adversary develops an algorithm Alg running in time T that can take a ciphertext $E_K(x)$ for an m -bit plaintext x and compute the first bit of x . Describe an adversary that plays the game in the CPA security definition and uses Alg to try to distinguish the real and ideal case. Which advantage can you obtain by asking a single query to the oracle? In terms of the parameters (t, q, μ, ϵ) , which parameter values does your adversary obtain? How would your result change if Alg cannot compute the first bit with certainty but can only guess it with probability $p > 1/2$?

Exercise 6.6 (AES implementation) This concerns a trick that is very useful in implementing AES. Let S be the AES S-box function, i.e., it inputs and outputs a single byte. Let MC be the mix-column function, i.e., it takes as input a column consisting of 4 bytes and outputs such a column.

Note that MC is linear, i.e., we have for any two columns C_1, C_2 that $MC(C_1 \oplus C_2) = MC(C_1) \oplus MC(C_2)$.

Define a function T_0 as follows: it takes as input a byte b , and the output $T_0(b)$ is a 4-byte column computed as follows: you first form a column $C(b)$ by placing $S(b)$ in the top byte and all-0 bytes in the lower three positions. Then set $T_0(b) = MC(C(b))$. We also define functions T_1, T_2, T_3 . They are similar to T_0 , except that when we form $C(b)$, we place $S(b)$ in the second, respectively third, respectively fourth entry from the top, and put in 0's elsewhere.

Now consider the state of an AES encryption at the start of some round. Name the bytes in this state a_{ij} as in figure 6.3 and let R be the state after we have done SubBytes, ShiftRow and MixColumn. So R is a 4 by 4 matrix of bytes.

Show that the first column of R is

$$T_0(a_{00}) \oplus T_1(a_{11}) \oplus T_2(a_{22}) \oplus T_3(a_{33})$$

Give similar expressions for the other 3 columns of R .

Sketch how this result can be used to implement AES based only on table look-up and XOR, instead of explicitly computing the operations. How much memory would you need for this?

Exercise 6.7 (DES Cycles) A *cycle* for a cryptosystem with encryption function E is defined by a plaintext block M and a key K , such that

$$M = E_K(\dots E_K(E_K(M))\dots)$$

where the *length* of the cycle is the number of encryptions involved. Some years ago, Don Coppersmith and his research team from IBM made a research announcement entitled “DES is not a group”. They said that they had found a number of cycles for DES, with lengths n_1, \dots, n_t such that the least common multiple of n_1, \dots, n_t was greater than 2^{56} . They claimed that this implies that

there must exist DES keys K_1, K_2 such that the function $M \rightarrow E_{K_1}(E_{K_2}(M))$ is NOT the same function as $M \rightarrow E_K(M)$, for any DES key K . In other words, multiple encryption under different keys does buy you something, compared to single key DES. Prove this claim.

The RSA 1-way trapdoor function

Contents

7.1	Introduction	78
7.2	RSA	80
7.3	RSA decryption works	80
7.4	Implementation of RSA	81
	7.4.1 Modular Exponentiation	81
	7.4.2 Multiplicative Inverses	81
	7.4.3 Generating random prime numbers	83
7.5	Optimizations of RSA	86
7.6	Security of RSA	87
	7.6.1 Factoring Algorithms	90
7.7	Exercises	91

7.1 Introduction

In what is probably the most cited paper in cryptography, “New Directions in Cryptography” from 1976, Diffie and Hellman proposed the idea of public-key encryption for the first time in open research. Actually, Clifford Cocks from the British intelligence service had similar ideas already in the 73, but this remained classified until many years later.

What Diffie and Hellman proposed was the basic idea that one could create a matching pair of a public and a secret key, such that the public key could be used only for encryption, while secret key would be required in order to decrypt. In this way, anyone could encrypt something meant to be received by the owner of the secret key without having to agree on a shared key in advance. They proposed several ideas for realizing this, but one turned out be particularly fruitful, namely the idea of a trapdoor one-way function.

Simplistically speaking, this would be an injective function $f : X \mapsto Y$, where X, Y are finite sets, such that computing $f(x)$ from input x is easy, but computing x from $f(x)$, say for a random $x \in X$ is infeasible. In addition, they suggested that f might come with a so-called trapdoor t_f , and given this extra information, it would be easy to compute x from $f(x)$.

Diffie and Hellman suggested that if such pairs (f, t_f) of function and trapdoor could be generated efficiently, then f could serve as the public key. Then anyone

could encrypt $x \in X$ to ciphertext $y = f(x)$, but only the party who knew t_f would be able to recover x from y . Of course, this is not really a secure encryption scheme, because the encryption is deterministic: a plaintext always goes to the same ciphertext, and we have already seen that this is a serious problem. Nevertheless, this was a very good starting point.

Diffie and Hellman did not know how to construct such functions, instead they suggested something related that we now know as Diffie-Hellman Key Exchange, which we will return to later. However, soon after, in 1977, Rivest, Shamir and Adleman proposed the first construction of trapdoor 1-way functions, that we now know as RSA.

In this chapter, we describe RSA and its implementation, and we survey what is known about its security. In the following chapter, we consider how we can use RSA for actual encryption of data (as mentioned above, it is insecure to encrypt by simply applying the public function to the plaintext).

Let us first define more precisely what we are trying to build:

Definition 7.1 A public-key cryptosystem consists of 3 algorithms (G, E, D) , satisfying the following:

G , algorithm for generating keys: This algorithm is probabilistic, takes a security parameter k as input and always outputs a pair of keys (pk, sk) , the public and secret keys. We assume that the public key contains (perhaps implicitly) a description of \mathcal{P} , the set of plaintexts and \mathcal{C} , the set of ciphertexts. These do not have to be the same for every key.

E , algorithm for encryption: this algorithm takes as input pk and $x \in \mathcal{P}$ and produces as output $E_{pk}(x) \in \mathcal{C}$. Note that E may be probabilistic, that is, even though we fix x and K , many different ciphertexts may be produced as output from E , as a result of random choices made during the encryption process. In other words, the ciphertext will have a probability distribution that is determined from x and K , typically uniform in some subset of the ciphertexts.

D , algorithm for decryption: this algorithm takes as input $sk, y \in \mathcal{C}$ and produces as output $D_{sk}(y) \in \mathcal{P}$. It is allowed to be probabilistic, but is in most cases deterministic.

For a public-key cryptosystem, we always require that for any key pair (pk, sk) output by G , correct decryption is possible, i.e. it holds that for any $x \in \mathcal{P}$, $x = D_{sk}(E_{pk}(x))$.

The exact meaning of the security parameter k will become more clear later, one should think of it as a number that specifies how large keys we want and hence how much security we want.

As we mentioned above, the basic RSA construction we will see in this chapter can only give a cryptosystem where E is deterministic.

7.2 RSA

On a high level the key generation for RSA works as follows, where we leave some steps unspecified for now, and return to them below.

1. On input (even) security parameter value k , choose random $k/2$ -bit primes p, q , and set $n = pq$.
2. Select a number $e \in \mathbb{Z}_{(p-1)(q-1)}^*$ and set $d = e^{-1} \bmod (p-1)(q-1)$.
3. Output public key $pk = (n, e)$ and secret key $sk = (n, d)$. For RSA, we always have $\mathcal{P} = \mathcal{C} = \mathbb{Z}_n$.

Encryption and decryption works as follows:

$$E_{(n,e)}(x) = x^e \bmod n, \quad D_{(n,d)}(y) = y^d \bmod n.$$

Of course, this specification leaves several questions unanswered:

- Why does this even work, that is, why does decryption return the correct plaintext?
- How do we implement the required operations efficiently? That is, selection of random primes of a given bitlength, computation of inverses and exponentiations with large numbers.
- Why should we believe this is secure? That is, if you are given (n, e) and $x^e \bmod n$, for random $x \in \mathbb{Z}_n^*$, is it actually hard to recover x ?

We will answer these questions in the following subsections.

7.3 RSA decryption works

The material in this section assumes you are familiar with Section 3.5. We want to show that $D_{(n,d)}(E_{(n,e)}(x)) = x$ for all $x \in \mathbb{Z}_n$. We will do the case of $x \in \mathbb{Z}_n^*$ here and leave the rest to an exercise. Note that the order of the group \mathbb{Z}_n^* , also known as $\phi(n)$, is $(p-1)(q-1)$. This is because there are pq numbers in \mathbb{Z}_n , and to get $\phi(n)$ from this we should subtract the number of elements that are not relatively prime to pq . Because p, q are primes, these are the multiples of p (of which there are q) and all multiples of q (of which there are p). But then we have subtracted pq (or 0 if you will) twice so we should add one. As a result, $\phi(n) = pq - q - p + 1 = (p-1)(q-1)$.

Also note that because d is the inverse of e modulo $(p-1)(q-1)$, we have $ed \bmod (p-1)(q-1) = 1$.

Armed with these observations and Corollary 3.11, we compute as follows:

$$\begin{aligned} D_{(n,d)}(E_{(n,e)}(x)) &= (x^e \bmod n)^d \bmod n = x^{ed} \bmod n \\ &= x^{ed \bmod (p-1)(q-1)} \bmod n \\ &= x^1 \bmod n \\ &= x \end{aligned}$$

The intuition here is as follows: after we encrypt and decrypt, we have effectively

raised the plaintext x to power ed . But because we work in \mathbb{Z}_n^* , arithmetic in the exponent actually happens modulo the order, $(p-1)(q-1)$. So we might as well have raised to power $ed \bmod (p-1)(q-1) = 1$.

7.4 Implementation of RSA

As we will see later, RSA cannot be secure unless the modulus n is large enough that factoring it, i.e., computing p and q from n , is infeasible. It is therefore clear that to implement RSA we will need to add and multiply large numbers. On a computer, this can be done in a way similar to what we learned in school: to multiply two k -digit numbers, we multiply each digit from one number by each digit from the other one, and then we add up all these results, after shifting them up as required. There will be k^2 such pairs of digits to multiply, so when a computer multiplies k -bit numbers using this approach, we will get an algorithm that runs in time $O(k^2)$. There are other multiplication algorithms that are asymptotically better, but they only beat the standard algorithm for numbers that are much larger than those we need in practice for RSA. It is well known that one can also divide in time $O(k^2)$ if the machine used can divide two digits by a single digit in constant time, which all modern CPUs can do.

7.4.1 Modular Exponentiation

Next, we show that if we can multiply and divide, we can also do exponentiation, that is, compute $x^d \bmod n$ from n, d, x . The trivial approach is to just multiply x by itself modulo n d times. But this will be horribly inefficient: to have security we will see that n needs to have several thousand bits, and d can be expected to be as large as n . Doing 2^{1000} multiplications will take longer than the estimated life time of the universe. Instead we do the following: first observe that we can compute $x, x^2 \bmod n, x^{2^2} \bmod n, x^{2^3} \bmod n, \dots, x^{2^k} \bmod n$ using k squarings. Next, let us write d in binary notation, that is, $d = \sum_{i=0}^k d_i 2^i$, where d_0, d_1, \dots are the bits in the binary expansion of d . Now we can compute as follows:

$$x^d \bmod n = x^{\sum_{i=0}^k d_i 2^i} \bmod n = \prod_{i=0}^k (x^{2^i})^{d_i} \bmod n$$

This expression says that we get the result we need by multiplying together mod n those $x^{2^i} \bmod n$ for which $d_i = 1$: when $d_i = 0$ the corresponding factor vanishes. Thus the worst case for this step is when all d_i are 1, in which case we need k multiplications for this part. In total, we need $O(k)$ modular multiplications, which means total time $O(k^3)$ using the standard algorithm for multiplication.

7.4.2 Multiplicative Inverses

With efficient exponentiation, we can do the encryption and decryption in practice. However, in key generation, we also need to compute $d = e^{-1} \bmod (p -$

1)($q - 1$). In general, multiplicative inverses can be computed using the so-called *Extended Euclidian algorithm*. It is based on the fact that the greatest common divisor of two numbers can always be expressed as a linear combination of the two numbers. More precisely, for any a, b , let $r = \gcd(a, b)$. Then there exist integers s, t such that $r = sa + tb$. Recall that b has an inverse modulo a exactly if $r = 1$. In this case, we get $tb = 1 - sa$ which is equivalent to $tb \bmod a = 1$. In other words $t = b^{-1} \bmod a$.

The Extended Euclidian algorithm indeed computes r, s, t from positive integers a, b , and hence also lets you compute inverses when they exist.

To see what the algorithm does, we will first define inductively two sequences of integers, r_0, r_1, \dots and q_1, q_2, \dots . This is done as follows: First we set $r_0 = a$ and $r_1 = b$. Then we define, for $i = 1, 2, \dots$

$$q_i = \lfloor \frac{r_{i-1}}{r_i} \rfloor, \quad r_{i+1} = r_{i-1} - q_i r_i$$

Put differently, we set $r_{i+1} = r_{i-1} \bmod r_i$, and let q_i be the quotient involved in the modular reduction. Note that r_{i+1} is the remainder resulting from a division by r_i , and so $r_{i+1} < r_i$. So, since the r_i 's are positive, and must decrease in every step, there must exist some index m for which $r_{m+1} = 0$, and when we reach this step, it will always hold that $\gcd(a, b) = r_m$. This is because we have

$$\gcd(r_0, r_1) = \gcd(r_1, r_2) = \dots = \gcd(r_{m-1}, r_m) = r_m$$

which follows by an easy inductive argument.

We now get the Extended Euclidian algorithm by defining 2 additional sequences, t_0, t_1, \dots and s_0, s_1, \dots as follows:

$$t_0 = 0, t_1 = 1, s_0 = 1, s_1 = 0,$$

$$t_i = t_{i-2} - q_{i-1} t_{i-1}$$

$$s_i = s_{i-2} - q_{i-1} s_{i-1},$$

where q_i is as defined above. By induction on i , one can now show that for $0 \leq i \leq m$ it holds that

$$r_i = s_i r_0 + t_i r_1 = s_i a + t_i b.$$

Recall that m was defined such that $\gcd(a, b) = r_m$. Therefore we have $\gcd(a, b) = r_m = s_m a + t_m b$.

So the Extended Euclidian algorithm computes the sequences r_i, q_i, s_i, t_i as described, stops when we reach the index m where $r_{m+1} = 0$ and returns r_m, s_m, t_m .

It can be shown that on input two k -bit numbers, the algorithm runs in time $O(k^2)$.

7.4.3 Generating random prime numbers

The final missing piece in order to implement RSA is the problem of generating random primes of a specified bit-length. On a high level the solution to this is very simple. To generate a k -bit prime, we do the following:

1. On input k , select a uniformly random k -bit number x , or more precisely, choose x at random such that $2^k \leq x < 2^{k+1}$.
2. Test if x is prime. If yes, then output x and stop, else go to Step 1.

Of course, this leaves two main questions we will have to answer: first, how many iterations can we expect to have to do before finding a prime? and second, how do we test efficiently if a given number is prime? (certainly not by trying to factor it - this is exactly what we hope is infeasible!)

The number of primes.

The answer to the first question is given by the following famous theorem:

Theorem 7.2 (The Prime Number Theorem) *Let π_N be the number of primes less than N , and \ln be the natural logarithm. Then*

$$\lim_{n \rightarrow \infty} \left(\frac{\pi_N}{N / \ln(n)} \right) = 1$$

In other words, for large N , $N / \ln(N)$ is a good approximation to π_N . The proof of this is beyond scope of this text, but we note that in fact the approximation is also known to be very good: explicit constants c_1, c_2 are known such that $c_1 N / \ln(N) < \pi_N < c_2 N / \ln(N)$ for large enough N .

In the algorithm above, we can see that the number of primes in the range we consider is

$$\pi_{2^{k+1}} - \pi_{2^k} \approx \frac{2^{k+1}}{\ln(2^{k+1})} - \frac{2^k}{\ln(2^k)} = 2^k \left(\frac{2}{(k+1) \ln(2)} - \frac{1}{k \ln(2)} \right) = 2^k \frac{k-1}{(k+1)k \ln(2)}$$

When k grows, $\frac{k-1}{k+1}$ converges to 1, so we see that the number of primes in our range for large k is $\Omega(2^k/k)$, that is, at least a constant times $2^k/k$.

The total number of candidates we choose from is $2^{k+1} - 2^k = 2^k$, so for the probability p_k that the candidate x is prime we have

$$p_k \in \Omega\left(\frac{2^k/k}{2^k}\right) = \Omega(1/k)$$

If each iteration succeeds with probability p_k , then the expected number of iterations we need until success is $1/p_k$, so we conclude that the expected number of iterations is $O(k)$.

Testing for primality.

We now turn to the question of how we test for primality. There exists a polynomial time algorithm that will always answer whether a given number is prime, the so-called AKS algorithm. However, it is not efficient enough to be used in

practice. Instead, we will look at the Miller-Rabin test, which is efficient, but may sometimes make mistakes: it will always accept a prime, but will in rare cases accept a composite, even though it should not. The test works as follows:

Miller-Rabin Primality Test

1. On input an odd x , compute a, b such that $x - 1 = 2^a b$ where b is odd (divide $x - 1$ by 2 as many times as possible).
2. Choose a non-zero $w \in \mathbb{Z}_x$ at random and compute $d = \gcd(w, x)$. If d is not 1, reject x and stop.
3. At this point we know that $w \in \mathbb{Z}_x^*$. Compute the following list of numbers

$$w^b \bmod x, w^{2b} \bmod x, w^{2^2 b} \bmod x, \dots, w^{2^{a-1} b} \bmod x$$

If any number on the list is -1 modulo x , or if $w^b \bmod x = 1$, then accept x , else reject.

To analyze the test we need this first:

Lemma 7.3 *Let x be prime and let $y \in \mathbb{Z}_x^*$ be such that $y^2 \bmod x = 1$. Then $y \equiv 1 \bmod x$ or $y \equiv -1 \bmod x$.*

PROOF The assumption implies that $y^2 - 1 \bmod x = 0$, or equivalently x divides $y^2 - 1 = (y + 1)(y - 1)$. But since x is prime this means x divides $y - 1$ or $y + 1$, so the lemma follows. \square

Here is the first basic fact about the test:

Lemma 7.4 *The Miller-Rabin test always accepts a prime*

PROOF Note that each number on the list generated by the test is formed by squaring the previous one modulo x . Note also that if we square the last number, we get $(w^{2^{a-1} b})^2 \bmod x = w^{2^a b} \bmod x = w^{x-1} \bmod x = 1$ because x is assumed to be prime: then \mathbb{Z}_x^* has order $x - 1$ and raising to the group order always gives 1 (Corollary 3.10).

But then by Lemma 7.3, this must mean that $w^{2^{a-1} b} \bmod x$ is 1 or -1 modulo x . If it is -1 we are done, the test will accept. If it is 1, we can repeat the same argument, considering that we now know that $(w^{2^{a-2} b})^2 \bmod x = 1$. The only way this can fail to lead to accept at some point, is if all numbers on the list turn out to be 1. But in that case also the first one, namely $w^b \bmod x$, is 1 and this also makes the test accept. \square

It is also possible to show that the test accepts a composite with bounded probability. This follows by a long and rather tedious counting argument that we will not give here. The result is:

Lemma 7.5 *The Miller-Rabin test (MR) accepts a composite with probability at most $1/4$, put differently, for all odd composite x , it holds that*

$$P[\text{MR}(x) = \text{accept} \mid x \text{ is composite}] \leq 1/4$$

Of course, $1/4$ is not an acceptable error probability for our key generation. A natural idea for improving this is to repeat the test a number of times, with independent choices of the value w in the test: if x is composite, we may not spot this first time, if we are unlucky with the choice of w , but we can try again, and will have a new independent chance (at $3/4$ probability) of spotting that x is composite. More precisely, we have that if we let MR^t denote t independent iterations of the test, then

$$P[MR^t(x) = \text{accept} \mid x \text{ is composite}] \leq 4^{-t}. \quad (7.1)$$

This suggests the following more concrete version of the random prime generation algorithm:

1. On input k , select a uniformly random k -bit number x , or more precisely, choose x at random such that $2^k \leq x < 2^{k+1}$.
2. If $MR^t(x) = \text{accept}$ then output x and stop, else go to Step 1.

Based on (7.1), it is very tempting to conclude that the probability that this algorithm makes an error and outputs a composite is at most 4^{-t} . Indeed in several older texts, and even in some standards, you will find this type of argument. However, we emphasize that *this line of reasoning is incorrect!*

To see the problem, let C be the event that x is composite, and let A be the event that $MR^t(x) = \text{accept}$. Note that what we observe when we run the algorithm is that A occurs, and given that this happened we want to know what is the probability that x is composite, that is, we want to know $P[C|A]$. However, (7.1) talks about $P[A|C]$ which is not the same probability!

Actually, if we take a closer look, it is clear that the probability we care about cannot be estimated by looking only at the MR test. We also need to consider the distribution of primes. To illustrate let's do a thought experiment: assume we lived in an unfriendly universe where primes are extremely rare. This would mean that our prime generation algorithm would have to test gazillions of composites before finally arriving at a prime. This would be bad news: sure, each individual composite has a very small chance of being accepted. But still, the probability that we accidentally accept one of them might still be large because there are so many.

Fortunately, we live in a friendly universe where there are quite a few primes: for bit-length k we just need to make sure we reject the first $O(k)$ composite candidates with high probability, since by then we expect to have found a prime.

Furthermore, it turns out that even though composites do exist, that will pass the MR test with probability $1/4$, they are extremely rare. A typical composite is accepted with much smaller probability, that one can show converges to 0 as we increase the bit length k . As a result one gets the following theorem, from [12], which shows that for large values of k , even a single iteration of the MR test is enough:

Theorem 7.6 *Let $p_{k,t}$ be the probability that the prime generation algorithm outputs a composite on input k and when using MR^t as primality test. Then*

for $k \geq 2$ we have $p_{k,1} \leq k^2 4^{2-\sqrt{k}}$. Furthermore, for $3 \leq t \leq k/9$ and $k \geq 88$, we have $p_{k,t} \leq k^{3/2} t^{-1/2} 4^{2+t/2-\sqrt{tk}}$.

In most practical applications, one would certainly consider an error probability of, say, 2^{-100} to be small enough to not matter. We should remember that in any application using crypto, we always need to live with the fact that things go wrong with non-zero probability, for instance, the adversary might guess a password or a key just by chance. So we basically just need to bring the error probability down so that it is comparable to the other error probabilities we have to live with anyway.

Therefore, Theorem 7.6 shows that in practice, we can think of the number of iterations t as a (small) constant. This means that the expected cost of generating a random k -bit prime is no more than running the MR test a constant number of times on the k candidates we expect to have to handle. Since the cost of running the MR test is dominated by an exponentiation, the total running time is $O(k \cdot k^3) = O(k^4)$.

It should be mentioned that in practice one would optimize the algorithm by test dividing each candidate by some of the small primes before we run the MR test, say all primes less than some upper bound T . The rationale is that dividing by all primes less than T is faster than the MR test, if we do not choose T too large. So, if we use this approach to reject all candidates that are 0 modulo one of the small primes without running MR, we improve the running time. One can show that by choosing T optimally (which turns out to be logarithmic in k), we get an expected run time of $O(k^4 / \log(k))$.

7.5 Optimizations of RSA

One easy way to speed up RSA encryption is to choose a small public exponent e . For this we have to fix e before we generate n , so one should change the key generation such that when considering candidates for the prime p (or q) we reject those that do not satisfy $\gcd(e, p-1) = 1$. The smallest value that will work is $e = 3$, but also $e = 2^{16} + 1$ is a popular choice. Since the time for encryption is linear in the bit length of the exponent, such a small e will make encryption several orders of magnitude faster than for a random e .

One should of course not do something similar with the secret d as this would make it easier to guess.

However, it is still possible to speed up RSA decryption using the Chinese Remainder Theorem 3.12.

For secret key (n, d) , let $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$. Also, we set $a_q = p \cdot (p^{-1} \bmod q)$ and $a_p = q \cdot (q^{-1} \bmod p)$. These numbers can be precomputed and stored with the secret key.

Now, on input a ciphertext y , we compute

$$x_p = y^{d_p} \bmod p \text{ and } x_q = y^{d_q} \bmod q.$$

and then

$$x = (x_p a_p + x_q a_q) \bmod n$$

To see why this is the correct plaintext, note first that the x output by the algorithm is constructed such that $x = f_n^{-1}(x_p, x_q)$, see Theorem 3.12. What we would like is that this output equals $y^d \bmod n$, so what we want to show is that $y^d \bmod n = f_n^{-1}(x_p, x_q)$. Since f_n is injective and surjective, we can apply f_n on both sides, and instead show that

$$f_n(y^d \bmod n) = (x_p, x_q)$$

Since f_n is a homomorphism, we have

$$f_n(y^d \bmod n) = f_n(y)^d = (y \bmod p, y \bmod q)^d = (y^d \bmod p, y^d \bmod q),$$

so the equation we want to show becomes

$$(y^d \bmod p, y^d \bmod q) = (x_p, x_q)$$

By definition of x_p, x_q , this is equivalent to $y^d \bmod p = y^{d \bmod (p-1)} \bmod p$, and $y^d \bmod q = y^{d \bmod (q-1)} \bmod q$. This follows immediately from Corollary 3.11, since the orders of \mathbb{Z}_p^* and \mathbb{Z}_q^* are $p-1$ and $q-1$.

The cost of this decryption algorithm is dominated by the two exponentiations modulo p and q . If n is a k -bit number, we are looking at two exponentiations on $k/2$ bit numbers. We saw earlier that exponentiation takes time roughly $O(k^3)$. So then an exponentiation on $k/2$ bit numbers takes time proportional to $(k/2)^3 = k^3/8$, that is, 8 times faster than on k bit numbers. It follows that we expect the entire decryption to be 4 times faster than computing $x^d \bmod n$ directly.

In practice we cannot expect quite as large an improvement since an actual implementation will not be exactly cubic, there will also parts that are quadratic and linear. But in practice one typically sees at least a factor 3 improvement.

7.6 Security of RSA

As hinted at earlier, the security property we can hope to get from basic RSA as defined above, is that given $E_{(n,e)}(x)$ for a random $x \in \mathbb{Z}_n$, it is hard to compute x . We will define more precisely what this means in the next chapter.

But even without the precise definition, we can make a few basic observations: first, it is clear that if an adversary could factor the modulus n and recover p, q , then the system is completely broken: the adversary can then compute $d = e^{-1} \bmod (p-1)(q-1)$, exactly as in the key generation, and can now decrypt everything. So we need to assume that for large enough moduli, it is infeasible to factor them.

But we should still ask ourselves whether there might a different and easier way of breaking RSA than factoring the modulus. For instance, perhaps one could compute the secret exponent d from (n, e) without having to factor n ? This, however, turns out to be impossible, as we shall see in a moment. First, we need the following basic result:

Lemma 7.7 *Let n, x, y be such that $x^2 \equiv y^2 \pmod{n}$ and furthermore $x \not\equiv \pm y \pmod{n}$. Then from n, x, y one can compute a non-trivial factor in n in polynomial time: a number that divides n but is not 1 or n .*

PROOF The first assumption implies that $x^2 - y^2 = (x + y)(x - y) \equiv 0 \pmod{n}$, so n divides $(x - y)(x + y)$. On the other hand the second assumption implies that n does not divide $(x - y)$ or $(x + y)$. Now consider $\gcd(n, x + y)$. It cannot be n , this would contradict the second assumption. But it also cannot be 1. If this was the case then since n divides $(x + y)(x - y)$, it would have to divide $(x - y)$ – intuitively, none of the prime factors of n are present at all in $x + y$ so they would all have to be in $x - y$. So since $\gcd(n, x + y)$ is neither n nor 1, it is a non-trivial factor. \square

Now we proceed to prove that is you can compute the secret exponent, then you can factor. As we shall see, we can use the knowledge of d to compute $y \in \mathbb{Z}_n^*$ such that $y^2 \pmod{n} = 1$. This fits into the assumption of the above lemma with $x = 1$. So we can factor n if $y \not\equiv \pm 1 \pmod{n}$. To see that there are such values of y in \mathbb{Z}_n^* , consider the two elements $(1, -1)$ and $(-1, 1)$ in $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$. Recall from the Chinese Remainder theorem that we can map such pairs into \mathbb{Z}_n using the function f_n^{-1} . We claim that $f_n^{-1}(-1, 1)$ and $f_n^{-1}(1, -1)$ are the values of y we are looking for. Indeed, because f_n^{-1} is a homomorphism, if we set $y = f_n^{-1}(-1, 1)$, then we have

$$y^2 \pmod{n} = f_n^{-1}(-1, 1)^2 \pmod{n} = f_n^{-1}((-1)^2 \pmod{p}, 1^2 \pmod{q}) = f_n^{-1}(1, 1) = 1.$$

The other value, $f_n^{-1}(1, -1)$, can be verified similarly. So there are 2 values of y that we like and 2 useless values, namely 1 and -1 . This suggests that there should be an algorithm that succeeds with probability at least $1/2$, and this is indeed what happens:

Theorem 7.8 *Assume we have an algorithm A that on input an RSA public key (n, e) will output the corresponding secret exponent d . From any such algorithm we can construct a new algorithm B that on input (n, e) factors n with probability at least $1/2$ using one invocation of A and polynomial time computation.*

PROOF The claimed algorithm B works as follows:

1. Run A on input (n, e) to get d .
2. Let $f = ed - 1$ and note that since $ed \pmod{(p - 1)(q - 1)} = 1$, $(p - 1)(q - 1)$ divides f . Therefore, we can use the algorithm from Lemma 7.9 below to factor n .

\square

Lemma 7.9 *There exists a PPT algorithm which, on input an RSA modulus $n = pq$ and a number f that is divisible by $(p - 1)(q - 1)$, will factor n with probability at least $1/2$.*

PROOF The claimed algorithm works as follows:

1. Write $f = 2^a b$ where b is odd. Since f is divisible by $(p-1)(q-1)$, for some integer u we have $f = u(p-1)(q-1)$.
2. Choose at random $w \in \mathbb{Z}_n$. If $d = \gcd(w, n) \neq 1$, then d must be p or q so return $d, n/d$ and stop.
3. If we get to this point we can assume that $w \in \mathbb{Z}_n^*$. Compute the following list of numbers:

$$w^b \bmod n, w^{2b} \bmod n, w^{2^2 b} \bmod n, \dots, w^{2^{a-1} b} \bmod n = w^f \bmod n$$

4. Note that since \mathbb{Z}_n^* has order $(p-1)(q-1)$ we can use Corollary 3.11 to argue that the list must end with the number 1:

$$w^f \bmod n = (w^{(p-1)(q-1)})^u \bmod n = 1^u \bmod n = 1,$$

If all numbers on the list are 1, the algorithm fails. Otherwise, locate the first time 1 occurs on the list, and let y be the number right before this location. Clearly, $y^2 \bmod n = 1$.

5. Now, if $y \equiv \pm 1 \bmod n$, the algorithm fails. Otherwise use Lemma 7.7 with $x = 1$ to factor n .

It is clear that this algorithm actually factors n when it does not fail. So we just need to estimate the probability of success. To this end, recall that, by the Chinese remainder theorem, we can think of the algorithm as being executed in $\mathbb{Z}_p \times \mathbb{Z}_q$, and so we write w as a pair $(w_p, w_q) = (w \bmod p, w \bmod q)$. When algorithm B raises w to power b , for instance, we can think of this as $(w_p^b \bmod p, w_q^b \bmod q)$, and when the squaring process gets to the value 1 or -1 in \mathbb{Z}_n , this corresponds in our notation to $f_n(1) = (1, 1)$ or $f_n(-1) = (-1, -1)$.

Note that because w is random in \mathbb{Z}_n^* , w_p is random in \mathbb{Z}_p^* and w_q is random in \mathbb{Z}_q^* .

Now, we know that $(p-1)(q-1)$ divides f , so $(p-1)$ does too. So we have $f = (p-1)v$ for some v . We can write both $(p-1)$ and v as a 2-power times an odd number, to get $(p-1) = 2^{t_p} p'$ and $v = 2^{t_v} v'$.

Now we can describe what happens to w_p as Algorithm B executes. First we raise to power $b = p'v'$. And then we start squaring. Consider the result we have after $t_p - 1$ squarings, namely $w_p^{p'2^{t_p-1}v'} \bmod p = (w_p^{(p-1)/2})^{v'} \bmod p$. By Lemma 3.16, this result is 1 with probability 1/2 and -1 with probability 1/2. We can write $(q-1) = 2^{t_q} q'$ and do the same argument for w_q .

Then we finish the proof by a case analysis: if $t_p = t_q$, then after $t_p - 1 = t_q - 1$ squarings, we will have 1 out of 4 results each with probability 1/4, namely $(\pm 1, \pm 1)$. This means we will have $(1, -1)$ or $(-1, 1)$ with probability 1/2, and either of these values will enable us to factor as discussed earlier. So the success probability is at least 1/2 in this case. On the other hand, if $t_p \neq t_q$, assume without loss of generality that $t_p > t_q$. Then after $t_p - 1$ squarings, we will have raised to a power divisible by $q-1$. So the result at this point will be $(1, 1)$ or $(-1, 1)$ with probability 1/2 each, and we have success in the latter case. So also here, the success probability is at least 1/2.

Note that the success probability may in both cases be larger than 1/2: if we

get $(1, 1)$ at the points in time we consider, the algorithm may still have success with a value that occurs earlier. \square

A final question we can ask about “shortcuts” to breaking RSA is: maybe there is way to compute x from n, e and $x^e \bmod n$ without computing d and without factoring n ? This question is open, but it is generally strongly believed that one really needs to factor n to find x . Moreover, there are variants of RSA that are known to be equivalent to factoring, so the issue is not seen as a very serious one.

7.6.1 Factoring Algorithms

Therefore, the most important question about security of RSA is how hard it really is to factor large numbers. We do not know how to prove anything conclusive about this. The factoring problem has been studied for centuries, and no efficient algorithm has been found. It would therefore be extremely surprising if one turned out to exist. On the other hand, a result saying that no efficient algorithm exists would resolve the P versus NP question, so this is not likely to be shown any time soon.

Therefore, the best we can do in practice is to look at the best known algorithms and state of the art of factoring in practice, and use this to estimate how large numbers we should use.

Currently all the best factoring algorithms use Lemma 7.7: they try to find x, y satisfying the condition in that lemma and differ only in how they approach this problem.

Currently, the best algorithm is the Number Field Sieve. It has a running time that is essentially $\exp(O(\sqrt[3]{k}))$ where k is the bit length of the input¹. As you can see, the algorithm is subexponential, it has $\sqrt[3]{k}$ in the exponent and not k – and so it is much faster than simple exhaustive search. This and experience with implementations of the algorithm suggest that if you want your system to be secure now and 10-15 years into the future, you should use at least a 2000 bit modulus. At the time of writing, this size is believed to provide 128 bits of security, meaning that breaking such a modulus is believed to take time roughly corresponding to exhaustive search for a 128 bit key.

We give a rough overview of how the best factoring algorithms work: first we choose a *factor base* which is the $F_t = \{p_1, p_2, \dots, p_t\}$ of the t smallest primes. There is a tradeoff involved in choosing t that we return to below.

Then the basic step in the first part of the algorithm is to choose $x \in \mathbb{Z}_n^*$ and compute $x^2 \bmod n$. Now the hope is that $x^2 \bmod n$ as an integer factors over F_t , i.e., has only small prime factors. If this is not the case, we delete x , else we remember it, we say we have *found a relation*. Say that at some point we have found the relation $x_1^2 \bmod n = p_1 p_2^2$. Suppose also that when we are lucky the next time, we get $x_2^2 \bmod n = p_1 p_3^2$. The observation now is that if we multiply the

¹ we have simplified the expression, but the other factors that occur in the real expression are of much less importance.

left sides and the right sides of the two relations, all the exponents become even:

$$(x_1x_2)^2 \bmod n = p_1^2p_2^2p_3^2 = (p_1p_2p_3)^2$$

Therefore we can set $x = x_1x_2 \bmod n$ and $y = p_1p_2p_3$ because this gives a pair x, y on which we may apply Lemma 7.7, we just need to hope that $x \not\equiv \pm y \bmod n$ which happens half the time on average.

We can see that it does not really matter what the actual exponents are on the right side of our relations, only whether they are even or odd. So we can represent the first equation as x_1 and then a bit vector $(1, 0, 0, \dots, 0)$ where 1 means odd and 0 means even. The second equation also corresponds to $(1, 0, 0, \dots, 0)$ and the fact that we get something interesting by multiplying the relations is represented by the fact that if we XOR the two bit vectors together, we get $(0, \dots, 0)$ corresponding to all-even.

So the algorithm goes as follows: we generate relations and write down a bit vector for each, a t -bit vector if we use F_t as factor base. Once we have enough of them, we try to compute a subset that XOR to the all-0 vector. This is actually the same as solving a linear system of equations over \mathbb{F}_2 . Namely, if we form a matrix M where our bit vectors are the columns, we are looking for a column vector \mathbf{v} such that $M\mathbf{v} = 0$. Once we have more than t relations, the bit vectors must be linearly dependent and we are able to find a solution.

The trade-off involved here is that if we choose t large, there is a better probability that each attempt will give a relation, but we will need many relations before we can solve the equations. And the other the way around for small t . So there is an optimal value that one usually needs to experiment to find.

To make it more likely to find relations, we can choose our x 's such that they are close to \sqrt{n} , because this means that when we square $x \bmod n$ we will get a relatively small number: say $x = s + \lceil \sqrt{n} \rceil$ where $s \ll n$. Then

$$x^2 \bmod n = x^2 - n = s^2 + (\lceil \sqrt{n} \rceil)^2 - n + 2s\lceil \sqrt{n} \rceil$$

Clearly all of s^2 , $(\lceil \sqrt{n} \rceil)^2 - n$ and $2s\lceil \sqrt{n} \rceil$ are much smaller than n , so the sum is much more likely to have only small prime factors than a random number modulo n would. Perfecting this idea results in the so called Quadratic Sieve algorithm which is the best for numbers of size up to around 500 bits. After this point the Number Field Sieve is better. It uses a different and smarter technique to choose the x 's, which is outside of the scope of this text.

7.7 Exercises

Exercise 7.1 (RSA decryption works on the entire domain) Show that for any $x \in \mathbb{Z}_n$, we have $D_{n,d}(E_{n,e}(x)) = x^{ed} \bmod n = x$. In the text we showed this for $x \in \mathbb{Z}_n^*$. Be careful not to repeat that argument, you have to include the case where $x \notin \mathbb{Z}_n^*$. Hint: by the Chinese Remainder theorem, $x^{ed} \equiv x \bmod n$ if and only if $x^{ed} \equiv x \bmod p$ and $x^{ed} \equiv x \bmod q$.

Exercise 7.2 (Slightly smaller secret exponent) Let $n = pq$ be an RSA modulus. Define

$$\lambda(n) = \frac{(p-1)(q-1)}{\gcd(p-1, q-1)}$$

Suppose we redefine the key generation by constructing e, d such that $ed \bmod \lambda(n) = 1$. Show decryption still works with this modification, i.e., we still have $x^{ed} \bmod n = x$ for any x . This can lead to a smaller secret exponent than the standard key generation, because $\lambda(n) < (p-1)(q-1)$, since $\gcd(p-1, q-1) \geq 2$. However, one should not construct p, q such that $\gcd(p-1, q-1)$ is very large. This would mean there is strong connection between p and q and this potentially dangerous.

Hint: You can use the Chinese remainder theorem and an argument similar to the one from the previous exercise.

Exercise 7.3 (Multiple exponent problem) Suppose someone generate one RSA modulus n and two different public exponents e_1, e_2 corresponding to secret exponents d_1, d_2 , so that $d_i = e_i^{-1} \bmod (p-1)(q-1)$ for $i = 1, 2$. The hope might be that we can support two users without having to generate two moduli. This is a bad idea for many reasons.

Here is one: suppose the same plaintext is sent under both keys, that is the adversary will get to see $x^{e_1} \bmod n$ and $x^{e_2} \bmod n$. And suppose that $\gcd(e_1, e_2) = 1$. Show that the adversary can compute x efficiently without factoring n .

Hint: recall that on input a, b , the Extended Euclidian Algorithm computes integers r, t such that $\gcd(a, b) = ra + tb$.

Exercise 7.4 (Small exponent problem) Suppose we are given three RSA public keys $(n_1, 3), (n_2, 3), (n_3, 3)$. So three users have chosen public exponent 3. Assume someone sends the same plaintext x to all three keys using basic RSA (he did not take this course, so he doesn't know that basic RSA should never be used for actual encryption of data). This means that the adversary sees $x^3 \bmod n_1, x^3 \bmod n_2, x^3 \bmod n_3$. Show that he can easily compute x , without breaking RSA.

Exercise 7.5 (A Fault Attack on the Chinese Remainder optimization) User A has a hardware device containing his secret RSA key (n, d) . The device performs decryption using the optimized decryption procedure described in Section 7.5.

Now, an adversary steals the device. His plan is to try to get the secret key out of the device and give it back to A quickly, hoping that A will not suspect anything and will continue to use the device and the key.

The device is well shielded and the adversary cannot directly read the key, so he does the following attack instead: He chooses a plaintext x and computes the ciphertext $y = x^e \bmod n$ (he knows, of course, the public key). He sends y to the device for decryption. However, while the device is doing the exponentiation modulo p he shoots some radiation at the device making its computation fail, so that it will obtain $x'_p \neq y^{d_p} \bmod p$. He leaves the computation modulo q alone.

As a result, the device will return $x' = (a_p x'_p + a_q x'_q) \bmod n$. Show that given x and x' , the adversary can easily factor n and hence break the cryptosystem.

Exercise 7.6 (RSA is hard to break almost everywhere) Are there particular RSA ciphertexts that are "weak", that is, ciphertexts that the adversary can very easily decrypt? There are certainly some, just think of the ciphertext 0. But in this exercise, we will show that if RSA is any good at all, there cannot be *large* sets of weak ciphertexts.

Let A be an algorithm that gets as input an RSA public key (n, e) and a ciphertext y . A will either return the correct plaintext x , or will return "no answer". Suppose A is able to decrypt if and only if y is in some subset S of \mathbb{Z}_n^* . Assume also that the size of S is $\epsilon \cdot (p-1)(q-1)$, for $0 < \epsilon < 1$.

Your task: construct a probabilistic algorithm B that uses A as a subroutine. B gets input public key (n, e) and ciphertext z , where z can be any number in \mathbb{Z}_n . We will assume that z is not 0, as 0 is easy to decrypt anyway.

You must construct B such that for *any fixed* z , B returns the correct plaintext for z with probability at least ϵ .

Hint: first show that if z is non-zero, but not in \mathbb{Z}_n^* , you can decrypt it easily without using A , by first using n , z and e to compute the secret key. If $z \in \mathbb{Z}_n^*$, you do need to use A . You will also need to use the so-called multiplicative property of RSA, namely $a^e \cdot b^e \bmod n = (ab \bmod n)^e \bmod n$ for any $a, b \in \mathbb{Z}_n^*$. That is, if you multiply an RSA encryption of a and one of b , you get an encryption of the product $ab \bmod n$.

Note: you cannot simply run A on input (n, e) and z . If z is not in S , A would always return "no answer" so for such z the success probability would be 0 and not ϵ as required.

[this is one of those irritating exercise where you are stuck if you don't get the right idea, so don't hesitate to ask your teacher if you are stuck.]

The Theory of Secure Public-Key Encryption

Contents

8.1	Public-Key Cryptosystems	94
8.2	Security of Public Key Encryption	95
8.2.1	Families of Trapdoor One-Way Functions (aka Deterministic Public-key Systems)	95
8.2.2	Security of Probabilistic Systems: CPA (Semantic) Security	96
8.2.3	Chosen Ciphertext Security	100
8.3	Exercises	103

8.1 Public-Key Cryptosystems

For convenience, we repeat here the syntax for public-key cryptosystems that we also saw in the previous chapter. Such a system consists of 3 algorithms (G, E, D) , satisfying the following:

G , algorithm for generating keys: This algorithm is probabilistic, takes a security parameter k as input and always outputs a pair of keys (pk, sk) , the public and secret keys. We assume that from the public key one easily derive a description of \mathcal{P} , the set of plaintexts and \mathcal{C} , the set of ciphertexts. These do not have to be the same for every key.

E , algorithm for encryption: this algorithm takes as input pk and $x \in \mathcal{P}$ and produces as output $E_{pk}(x) \in \mathcal{C}$. Note that E may be probabilistic, that is, even though we fix x and K , many different ciphertexts may be produced as output from E , as a result of random choices made during the encryption process. In other words, the ciphertext will have a probability distribution that is determined from x and K , typically uniform in some subset of the ciphertexts.

D , algorithm for decryption: this algorithm takes as input $sk, y \in \mathcal{C}$ and produces as output $D_{sk}(y) \in \mathcal{P}$. It is allowed to be probabilistic, but is in most cases deterministic.

For a public-key cryptosystem, we always require that for any key pair (pk, sk) output by G , correct decryption is possible, i.e. it holds that for any $x \in \mathcal{P}$, $x = D_{sk}(E_{pk}(x))$.

8.2 Security of Public Key Encryption

8.2.1 Families of Trapdoor One-Way Functions (aka Deterministic Public-key Systems)

The basic RSA system from the previous chapter is the primary example of a public-key cryptosystem where the encryption is deterministic, that is, the ciphertext is determined from the public key and the plaintext.

This means that the RSA construction defines an infinite family of functions, namely the family consisting of all RSA encryption functions, one for each possible public RSA key. The family is infinite because there is - in principle - no limit on the size of public key we can choose. In, fact any deterministic public-key system defines a family of functions in this way. So secure deterministic public-key cryptosystems are also called *families of trapdoor one-way functions*.

What properties can we hope to get from such a family of encryption functions? Well, at least the encryption functions should be efficiently computable, given the public key, whereas the decryption functions should be hard to compute if you know only the public key, but easy if you know the secret trapdoor: the decryption key. Since we have an infinite family of functions, we should be careful about what we mean by “easy” and “hard” in this context: if the adversary is given $E_{pk}(x)$ we cannot demand that he *never* computes x : he can always make a guess, and will be right with some non-zero probability. But we also cannot demand that the probability of computing x is less than some fixed constant, like 2^{-100} . This will be false if the public key is small: if we use RSA modulus $n = 15$, then the adversary will succeed with probability 1. Of course, no one would use such small numbers, the point is just that we cannot demand that the adversary’s success probability is always small, we can only ask that it becomes small if we choose large enough keys. However, even this cannot be true, unless we limit the adversary’s computing time: if there is no limitation, the adversary can always try all possible values for x and will succeed with probability 1.

So at the end of the day what we want is that, for any polynomial time bounded adversary, it holds that as the length of the key goes to infinity, the adversary’s success probability goes to 0. Moreover, we would like a *fast* convergence, so we do not have to choose huge keys in order to get small error probability.

Therefore, we need a precise way to state what it means that a probability converges to 0 “fast”:

Definition 8.1 A probability $\epsilon(k)$ that depends on natural number k is said to be *negligible* if it holds that for any polynomial p , we have $\epsilon(k) \leq 1/p(k)$ for all large enough k . In other words, asymptotically in k , it vanishes to zero very quickly.

This is complexity theoretic inspired way to say that “for all practical purposes”, the probability might as well have been zero. Of course, there are many conditions we could have used that would ensure convergence to 0. But the motivation for this one is as follows: suppose some event occurs with negligible probability $\epsilon(k)$. This might be the probability that an adversary succeeds when

he tries to break a certain system. How many attempts do we expect that the adversary will need until he has success? By Theorem 2.2, this is $1/\epsilon(k)$, and by Definition 8.1 we have $1/\epsilon(k) \geq p(k)$, for all polynomials $p()$ and all large enough k . So if the adversary has only polynomial time available, he cannot expect to be successful.

Intuitively, we can say that events that occur with negligible probability occur so seldom that polynomial time algorithms do not have time to wait for them to happen.

Now we can define trapdoor one-way functions:

Definition 8.2 We will say that the system (G, E, D) forms a *family of trapdoor one-way functions* if the following is satisfied:

- The algorithms (G, E, D) define a public-key crypto system according to Definition 8.1, and they all run in time polynomial in the security parameter k . In addition, the algorithm E is deterministic, that is, it defines an injective function from its input set to the output set.
- Let any probabilistic polynomial time algorithm (PPT) A be given. Consider the following experiment: we run G on input k , let (pk, sk) be the output. Then we select x at random in the set of plaintexts \mathcal{P} and finally we run A on input $pk, E_{pk}(x)$. Let $p(A, k)$ be the probability that A outputs x . Then we require that for any PPT algorithm A , $p(A, k)$ is *negligible* in k .

For the case of RSA, in order to build any security on this system, we have to at least assume that it satisfies the above definition. This is the standard

RSA assumption: the basic RSA algorithm (including the standard method for generating keys) defines a family of trapdoor one-way functions.

Applying a one-way trapdoor function directly to encrypt a message gives only very limited security, and is not something one should use for encryption. One concrete reason is the following issue: suppose our worst enemy knows that tomorrow we will send one of two messages x_0, x_1 . Then he can just use the public key and apply the encryption function to both messages and store the two ciphertexts he obtains. Now he just waits and watches which of the two show up on the communication line tomorrow. Then he knows what we sent.

Furthermore, even if it is hard to compute x from $E_{pk}(x)$, this does not guarantee that an adversary cannot compute part of x . Even if, say, half the bits of x are easy to compute from $E_{pk}(x)$, the system might still satisfy the definition above. Of course this is not a satisfactory security guarantee.

The solution to all this turns out to add randomness to the encryption process. Then we can get better security.

8.2.2 Security of Probabilistic Systems: CPA (Semantic) Security

We will first look at security against a passive attack, i.e. one where the adversary just observes ciphertext, and then tries to get information on some of the hidden

plaintext. The following definition is known as *semantic security*, or *CPA security*, it has become standard in the area, and turns out to imply any other reasonable definition.

The idea is as follows: I give you the public key, you choose any message you want, and I give you an encryption of either your message, or a complete random message. If you cannot efficiently guess which kind of ciphertext I gave you, we say the system is secure. So this definition is designed to capture the idea that an encryption should tell you nothing at all about the message: for all you care, encryptions might contain only random garbage. This is basically the same idea as in the corresponding definition for secret key (symmetric) encryption.

A bit more formally, Let (G, E, D) be the system in question. Then we consider the following games that the adversary plays with an oracle. They are very similar to the corresponding ones we considered for conventional encryption. The only real difference is that since this is public-key crypto, the adversary should know the public key like anyone else, this also means he can by himself make as many encryptions as he likes.

The ideal world: Input to both adversary A and oracle O_{Ideal} is the security parameter k . The oracle runs $G(k)$ to get (pk, sk) and gives pk to A . A computes a plaintext $x \in \mathcal{P}$ and gives it to O . The oracle responds with $E_{pk}(r)$, where r is randomly chosen in \mathcal{P} of the same length as x . Finally A outputs a bit b .

The real world: Input to both adversary A and oracle O_{Real} is the security parameter k . The oracle runs $G(k)$ to get (pk, sk) and gives pk to A . A computes a plaintext $x \in \mathcal{P}$ and gives it to O . The oracle responds with $E_{pk}(x)$. Finally A outputs a bit b .

Definition 8.3 We say that (G, E, D) is *CPA (semantically) secure*, if for all probabilistic polynomial time adversaries A , it holds that $Adv_A(O_{real}, O_{Ideal})$ is negligible in k .

In more human language: the adversary may be able to see from a ciphertext how long the plaintext is, but other than that, no efficient adversary can tell meaningful encryptions from random encryptions. So intuitively, this means that ciphertexts reveal no useful information to the adversary.

One may wonder if the definition really captures what we need in real life? In Definition 8.3 the adversary is only allowed to submit a single plaintext and get back the ciphertext. But in a real attack he may be see several encryptions on different messages. However, it turns out that the definition implies a seemingly stronger one where the adversary can submit any polynomial number of plaintexts and still cannot distinguish real encryptions from random ones. See Exercise 8.3.

Clearly, no deterministic system can be CPA secure: as we saw in the previous section, if the system is deterministic, then the adversary can just encrypt his message by himself and compare the results to what he gets from the oracle.

However, it is possible to use deterministic systems to build new ones that do have this type of security. Taking RSA as an example, we build the following:

Probabilistic cryptosystem based on RSA (PCRSA)

Key Generation Generates a pair of RSA keys (n, e) , (n, d) in the usual way.

The set of messages is just $\{0, 1\}$ whereas the set of ciphertexts is Z_n^* .

Encryption The encryption algorithm will encrypt a bit b by choosing a random number $x_b \in Z_n^*$ such that the least significant bit of x_b is b . The ciphertext is now $c = x_b^e \bmod n$.

Decryption Compute $x_b = c^d \bmod n$ and extract the least significant bit.

This cryptosystem is of course not very efficient: a one message bit is expanded to a ciphertext that takes up an entire number mod n . However, the theorem below that says that it is CPA secure can be extended to say we can also securely encrypt $O(\log k)$ bits at once by placing them in the least significant end of an RSA block and padding with random bits. This could, for instance, be used to encrypt a key for a symmetric cipher under RSA. Such keys, say AES keys, are indeed usually much shorter than an RSA modulus. Even though the ciphertext is still much longer than the plaintext, this is not a serious problem in an application where one only sends an AES key once and then encrypts the actual data under that key.

Theorem 8.4 *Under the RSA assumption, PCRSA is CPA secure.*

This theorem follows from results in [1]. The proof is quite complicated and out of scope for this text. However, we will show a weaker result that nevertheless gives a feeling for why the theorem should be true, namely if one can compute *with certainty* the least significant bit of x from $x^e \bmod n$, then one can invert the RSA encryption function. Even this is quite a remarkable result: given the ability to compute only a single bit of the plaintext, we are able to somehow compute the entire plaintext!

Before we get there we need to define two functions: $H : Z_n^* \mapsto \{0, 1\}$ and $P : Z_n^* \mapsto \{0, 1\}$. The idea in both cases is that the functions take an RSA ciphertext as input and output a bit of information about the corresponding plaintext. Assuming throughout that $y = x^e \bmod n$, the definitions are as follows:

$$P(y) = P(x^e \bmod n) = \text{lsb}(x),$$

where lsb denotes the least significant bit.

$$H(y) = H(x^e \bmod n) = \begin{cases} 0 & \text{if } 0 \leq x < n/2 \\ 1 & \text{otherwise.} \end{cases}$$

Note that an algorithm for computing P would be an algorithm for computing the plaintext from a ciphertext in PCRSA. It turns out that there is a close connection between the two functions. This is due to the multiplicative property of RSA, that if you multiply two ciphertexts, you get an encryption of the product of the two plaintexts. Concretely for our case, we note that if $y = x^e \bmod n$ then

$$2^e \cdot y \bmod n = 2^e x^e \bmod n = (2x \bmod n)^e \bmod n.$$

In other words, given a ciphertext, even if you don't know the secret key or the

plaintext, you can still efficiently compute an encryption of 2 times the hidden plaintext. Using this fact we can show that given any algorithm for computing $H(y)$, you can compute $P(y)$ efficiently, and vice versa. Namely, we have

$$H(y) = P(2^e y \bmod n) \quad P(y) = H(2^{-e} y \bmod n) \quad (8.1)$$

To see the first equation, note that the plaintext corresponding to $2^e y \bmod n$ is $2x \bmod n$. Now, if $0 \leq x < n/2$, then $2x \bmod n = 2x$ which is an even number so P will return 0 as required. Otherwise, if x is in the top half of the interval from 0 to n , then $2x \bmod n = 2x - n$ which is odd, so P returns 1 as we would like. The second equation is left as an exercise.

It is not hard to see that repeated application of the H function can be used to reveal information about an unknown plaintext. Namely, for ciphertext $y = x^e \bmod n$, $H(y)$ tells us whether $0 \leq x < n/2$ or $n/2 < x \leq n - 1$. Let's assume that we find that $0 \leq x < n/2$. Then $H(2^e y \bmod n) = H((2x)^e \bmod n)$ will tell us whether $0 \leq x < n/4$ or $n/4 < x < n/2$. Repeating this step of multiplying by 2^e modulo n will allow us to find x , essentially by binary search. Concretely, we can show:

Theorem 8.5 *Given an algorithm that computes the function P correctly on every input, there exists an algorithm that on input n, e, y computes x such that $x^e \bmod n = y$ in polynomial time, using $O(k)$ calls to P , where k is the bit length of n .*

PROOF Note that if we can compute P , we can also compute H . We then use the binary search technique sketched above. This results in the following algorithm that takes y, n, e as input, where we set $k = \lfloor \log_2(n) \rfloor$:

1. For $i = 0$ to k do $h_i = H(y)$, $y = 2^e y \bmod n$.
2. $lo = 0$, $hi = n$
3. for $i = 0$ to k do
 1. $mid = (hi + lo)/2$
 2. if $h_i = 1$ then $lo = mid$ else $hi = mid$.
4. Return $\lfloor hi \rfloor$.

□

The main result from [1] is essentially a stronger version of this theorem, saying that one can compute plaintext x from ciphertext y , even given a very weak “oracle”, namely one that on input y returns a guess at $P(y)$ that is correct with probability $1/2 + \epsilon(k)$ where $\epsilon(k)$ is not negligible. Now, an adversary who breaks the CPA security of of PCRSA would in fact be such an oracle. Therefore, such a successful CPA-adversary would lead to an algorithm that breaks the RSA assumption.

It turns out that RSA is not special w.r.t. the possibilities of building CPA-secure systems. In general it can be shown that:

Theorem 8.6 *If a family of one-way trapdoor permutations exist, then there exists a CPA-secure public-key cryptosystem.*

While the proof of this is technical, the construction itself is quite simple: let f be the given one-way trapdoor function, for simplicity assume it maps k -bit strings to k -bit strings. Then to encrypt a bit b , we choose two k -bit strings x, r at random and send

$$f(x), r, (r \cdot x) \oplus b,$$

where $r \cdot x$ means the inner product of $r = r_1, \dots, r_k$ and $x = x_1, \dots, x_k$, i.e., $r \cdot x = \bigoplus_{i=1}^k r_i \wedge x_i$.

8.2.3 Chosen Ciphertext Security

Everything we said so far has been about CPA attacks which model security against a passive adversary, i.e., an adversary that simply looks at the public key and some ciphertext and does his best to figure out what the plaintext was. What happens if the adversary is also allowed to modify the communication and observe the effect of what he did? We model this using a chosen ciphertext attack.

To define this type of attack, we reuse the same idea as before, but in addition, the oracle will now kindly decrypt ciphertext for the adversary:

The ideal world: Input to both adversary A and oracle O_{Ideal} is the security parameter k .

1. The oracle runs $G(k)$ to get (pk, sk) and gives pk to A .
2. A may submit an input string y to O_{Ideal} , and O_{Ideal} will return $D_{sk}(y)$ to A . This is repeated as many time as A wants.
3. A computes a plaintext $x \in \mathcal{P}$ and gives it to O . The oracle responds with $y_0 = E_{pk}(r)$, where r is randomly chosen in \mathcal{P} of the same length as x .
4. A may now again submit an input string y to O_{Ideal} , the only restriction is that y must be different from y_0 . O_{Ideal} will return $D_{sk}(y)$ to A . This is repeated as many time as A wants.
5. A outputs a bit b .

The real world): Input to both adversary A and oracle O_{Real} is the security parameter k .

1. The oracle runs $G(k)$ to get (pk, sk) and gives pk to A .
2. A may submit an input string y to O_{Real} , and O_{Real} will return $D_{sk}(y)$ to A . This is repeated as many time as A wants.
3. A computes a plaintext $x \in \mathcal{P}$ and gives it to O_{Real} . The oracle responds with $y_0 = E_{pk}(x)$.
4. A may now again submit an input string y to O_{Real} , the only restriction is that y must be different from y_0 . O_{Real} will return $D_{sk}(y)$ to A . This is repeated as many time as A wants.
5. Finally A outputs a bit b .

Definition 8.7 We say that (G, E, D) is *chosen ciphertext (CCA)-secure*, if for all probabilistic polynomial time adversaries A , it holds that $\text{Adv}_A(O_{\text{Real}}, O_{\text{Ideal}})$ is negligible in k .

The general theorem from before can be expanded to this case as well:

Theorem 8.8 *If there exists a family of trapdoor one-way functions, then there exists a chosen ciphertext secure probabilistic public-key system.*

The construction behind this general result leads to very inefficient systems. We would like to have more practical constructions, because chosen ciphertext security is not just a theoretical notion, but something we need to have in practice. Granted, it may seem that the attack we give the adversary in the CCA definition is very strong. One could ask whether this is too pessimistic - perhaps in real life, the adversary would not be in such a favorable situation? However, there are in fact cases where the adversary has something that is close to a full CCA attack. For instance, many Internet servers can set up secure connections using, for instance, the SSL protocol. Part of this is to decrypt a ciphertext received from a client. Until some years ago the cryptosystem used was an RSA variant described in a (now outdated) version of the PKCS #1 standard. In this system, the decryption process could lead to different types of errors, if the input was not a legal ciphertext. If this occurred, servers would typically send an error message back to the client, specifying the type of error. But this means that an outsider now has access to (part of) the result of applying the decryption algorithm to an input he chooses. This is a chosen ciphertext attack! Indeed, Bleichenbacher has shown that this can be used in practice to break the old version of PKCS #1.

Fortunately, we have today various methods one can use to get chosen ciphertext security efficiently. For example, in [2] the so called OAEP-system is proposed (Optimal Asymmetric Encryption Padding). OAEP is basically a general method for turning a public-key system with only deterministic security into a chosen ciphertext secure scheme. The basic idea is to encode a message m to be sent in special way before it is sent into the encryption algorithm. This means that only strings with a very special structure are ever encrypted. Put another way, if the original encryption algorithm can handle messages in some set \mathcal{P} , we will only use messages in some subset \mathcal{P}' . The decryption algorithm is defined such if after decryption, it finds that the result is not in \mathcal{P}' , it will simply output an error message. Ciphertexts that lead to plaintexts in \mathcal{P}' are called *legal*.

Why would this help against a chosen ciphertext attack? well, if \mathcal{P}' is very small compared to \mathcal{P} , then it may be a reasonable assumption that the only way in which the adversary could produce efficiently a legal ciphertext, is by choosing some plaintext m and encrypt it in the normal way. But if he submits this to the decryption oracle, he already knows the answer will be m . If he produces a ciphertext in any other way, by assumption it will be illegal with overwhelming probability. So again the adversary already knows what the oracle will say. Of course an oracle is not of any use if you can always predict in advance what it

will answer! It follows that we can remove the option for the adversary to have ciphertexts decrypted, and then what is left is the definition of CPA security.

Concretely, OAEP works as follows: suppose the original encryption $E_{pk}(\cdot)$ works on k -bit strings. Then we choose two parameters k_0, k_1 such that $k_0 + k_1 < k$. The scheme can encrypt messages with n bits, where $n = k - k_0 - k_1$. We need two functions G, H , where $G : \{0, 1\}^{k_0} \rightarrow \{0, 1\}^{n+k_1}$ and $H : \{0, 1\}^{n+k_1} \rightarrow \{0, 1\}^{k_0}$. Usually, as G, H we use so called cryptographic hash functions, i.e., one-way functions that produce random looking output - they can be built, for instance, from symmetric encryption schemes and can therefore be very efficiently computable.

We encrypt a message m as follows:

OAEP-based Encryption

1. Choose $r \in \{0, 1\}^{k_0}$ at random.
2. Compute $s = G(r) \oplus (m || 0^{k_1})$, $t = H(s) \oplus r$, $w = s || t$, where $||$ means concatenation of strings.
3. Let the ciphertext be $E_{pk}(w)$.

For decryption, one uses the secret key to reconstruct w and hence what should be s, t if the ciphertext was legal. Then we can find $r = t \oplus H(s)$, and finally $s \oplus G(r)$ should be some n -bit string m followed by k_1 0's. If this is not the case, the ciphertext was illegal and we return an error, otherwise we output m .

No method of the OAEP type have been *proved* secure in the sense that their security follows only from the RSA assumption, for instance. The problem is that it seems very difficult to design the encoding method such that one can prove that it is hard to generate legal ciphertexts without knowing the plaintext. But they can nevertheless be quite adequate in practice, and OAEP is a part of many international standards for encryption, particularly in connection with RSA.

This is because there is something we *can* prove about OAEP, namely that it is CCA secure in the so-called Random Oracle (RO) model, which we will return to in more detail in the chapter on Authentication and Hash functions. In the RO model, we model the functions G and H as completely random functions. More concretely, we assume that when users of the scheme or the adversary would call one of the functions, instead of evaluating the relevant function, they submit the input to an oracle which will return a randomly chosen value. In this model, the adversary is forced to treat the functions as “black boxes” with no structure that can be exploited. So a security proof in the RO model only guarantees security against such attacks. In real life, the adversary is of course free to try to exploit detailed knowledge of G and H , as they must be specified and known by everyone. But we may hope that if they are sufficiently complicated and without any “nice” structure, the adversary will not get any advantage from this.

So, proofs in the RO model do not guarantee security in real life, but they are certainly better than nothing.

On the other hand, it is in fact possible to build practical CCA secure cryp-

tosystems where security can be reduced to well established intractability assumptions, without using the RO model. Cramer and Shoup [4] have built a quite practical system based on discrete logarithms which they can prove is chosen ciphertext secure under a reasonable assumption, the so called Decisional Diffie-Hellman assumption which we will see in more detail later. This is the first practical system with such provable security. Some years later, Hofheintz and Kiltz [5] constructed the first CCA secure and practical system based only on the assumption that factoring is hard.

8.3 Exercises

Exercise 8.1 (Computing H and P) Show the second part of Equation (8.1).

Exercise 8.2 (Repeated Squaring) The "Repeated squaring"-algorithm for exponentiation is essential for all known public-key cryptosystems. It comes in several variants, scanning the exponent from the least or the most significant end. This exercise shows that scanning from the most significant end has some advantages in terms of the available optimizations:

Assume we want to compute $a^z \bmod n$, where z is a k -bit number, and where z is $z_{k-1}z_{k-2}\dots z_0$ in binary notation (all z_i are 0 or 1).

The algorithm does the following:

1. $x := 1$
2. For $i = k - 1, \dots, 0$, do:
 1. $x := x^2 \bmod n$.
 2. $x := x \cdot a^{z_i} \bmod n$ (note that this step is empty if $z_i = 0$).
3. Return x .

Let Z_i be the number which is binary notation is $z_{k-1}z_{k-2}\dots z_i$.

1. Show that after each execution of step 2, we have $x = a^{Z_i} \bmod n$ and conclude that the algorithm returns $x = a^z \bmod n$.
2. If the bits in z are randomly chosen, what is the expected number of multiplications mod n done in the algorithm?
3. The algorithm can be optimized: assume that we first compute the values $a^2 \bmod n$ and $a^3 \bmod n$, this can be done using 2 multiplications. So we now know $a^v \bmod n$ for any value of v that can be written with 2 bits. Describe a variant of the algorithm above which reads off 2 bits of the exponent in one step. Argue that it is correct and give the expected number of multiplications you need for a random exponent.
4. (Optional) generalize the idea from above. What is the optimal number of bits to scan in one step?

Exercise 8.3 (CPA Security with several messages) Consider a security definition that is exactly the same as Definition 8.3 of CPA security, except that the adversary can send two messages m_1, m_2 to the oracle. It either gets back ciphertexts c_1, c_2 that are either encryptions of m_1, m_2 , in case of the real oracle

(that we will call O_{Real}^2 to emphasize that it takes two messages as input). Or it gets back encryptions of random messages, in case of the ideal oracle O_{Ideal}^2 . We will say the system has CPA_2 security if no PPT algorithm can distinguish the two cases with non-negligible advantage.

Show that CPA security implies CPA_2 security.

Hint: consider an oracle O_{Hybrid} that on input m_1, m_2 will return $c_1 = E_{pk}(m_1)$, $c_2 = E_{pk}(r)$ for a random message r . Show that for any PPT algorithm A , we have $Adv_A(O_{Real}^2, O_{Hybrid})$ is negligible and that also $Adv_A(O_{Hybrid}^2, O_{Ideal})$ is negligible. For this you need to use the assumption that the cryptosystem is CPA secure. Use this to conclude that the system is CPA_2 secure.

Note that we can define CPA_l -security for any l in the natural way and prove, similarly to above, that CPA security implies CPA_l security as long as l is polynomial.

Exercise 8.4 (Alternative CPA definition) An alternative definition to the one we give here for CPA security of public-key schemes goes as follows: from security parameter k the oracle generates a key pair (pk, sk) and gives the public key to the adversary A . The adversary chooses two plaintexts m_0, m_1 of the same length and sends them to the oracle. The oracle chooses a random bit b and sends $E_{pk}(m_b)$ to the adversary. Finally, the adversary outputs a bit b' . Let $p_A(k)$ be the probability that $b = b'$. We say the cryptosystem is secure if for all polynomial time adversaries it holds that $|1/2 - p_A(k)|$ is negligible.

The goal is to show that a cryptosystem is secure according to this definition if and only if it is secure according to the one from Definition 8.3. This is done by showing that if there is an adversary that breaks Definition 8.3, you can construct an adversary that breaks the alternative definition, and vice versa.

For the first part, you are given A , who breaks Definition 8.3, and you must construct A' who uses A as subroutine, and breaks the alternative definition. Consider the following algorithm for A' :

1. Get pk from the oracle and give it to A .
2. A outputs a message m . A' now chooses a random message r of the same length as m , sets $(m_0, m_1) = (m, r)$ and sends the pair to the oracle.
3. The oracle returns a ciphertext $y = E_{pk}(m_b)$ for random bit b . Then A' gives y to A , who outputs a guess “real” or “ideal”.
4. If A said “real”, output $b' = 0$, else output $b' = 1$ (this makes sense because the m that A chose is the first message in the pair, and output “real” means that A thinks y contains m .)

Now, show that if A has advantage ϵ , then $|1/2 - p_{A'}(k)| = \epsilon/2$.

You can start by using the fact that

$$\begin{aligned} P[b = b'] &= P[b = b' | b = 0] \cdot P[b = 0] + P[b = b' | b = 1] \cdot P[b = 1] \\ &= \frac{1}{2} (P[b = b' | b = 0] + P[b = b' | b = 1]) \end{aligned}$$

Of course, if ϵ is not negligible, then $\epsilon/2$ is also not negligible, so we are done with the first part.

For the second part, you are given A' who breaks the alternative definition and you must construct an adversary A who uses A' as subroutine, and breaks Definition 8.3.

Specify an algorithm for A , showing how it uses A' to play the oracle game from Definition 8.3, and show that your construction works. If you do the algorithm the right way, you will be able to show that if $|1/2 - p_{A'}(k)| = \epsilon$, then the advantage of your A is also ϵ .

Public-Key Encryption Based on Discrete Logarithms

Contents

9.1	Introduction	106
9.2	Preliminaries	106
9.3	Three computational problems	107
9.3.1	Hardness of the problems	108
9.4	The El Gamal Cryptosystem	109
9.5	Some Example Groups	111
9.5.1	\mathbb{Z}_p^*	111
9.5.2	Prime Order Subgroups of \mathbb{Z}_p^*	112
9.5.3	Groups based on Elliptic Curves.	114
9.6	Algorithms for solving discrete log	118
9.6.1	The generic algorithm	118
9.6.2	Index Calculus	119

9.1 Introduction

This chapter describes how public-key encryption schemes can be constructed from finite groups with certain properties, and studies some related computational problems that must be hard in order for the cryptosystems to be secure.

9.2 Preliminaries

Consider the group \mathbb{Z}_p^* , where p is a prime. We may think of the prime p as information that *specifies* the group \mathbb{Z}_p^* , so that we can work with it on a computer. What we mean by this in general is that knowing the specification of a group, you can

- compute the order
- write down elements in the group, i.e., represent them on our computer as bit strings
- compute the group operation and compute inverses

For \mathbb{Z}_p^* , this is clear: if we know p , we know the order of the group ($p-1$), elements in the group are numbers from 1 to $p-1$, and we can multiply and invert elements modulo p . When in the following, we talk about *generating a group* G , this means

running some algorithm to obtain the specification of G . Also, when we say that some algorithm is given G as input, this means it is given the specification.

Any group \mathbb{Z}_p^* is cyclic, i.e., there exists some α - a primitive element, or generator - such that $\langle \alpha \rangle = \mathbb{Z}_p^*$, see Theorem 3.14. In the following, we will consider cyclic groups in general, i.e., we are given the specification of some group G and a generator α of G . Throughout, t will be the order of G .

9.3 Three computational problems

We may now consider the following

The discrete log (DL) problem

Given a group G , generator α , and $\beta \in G$, find integer a , such that $\alpha^a = \beta$.

The DL problem is in many groups notoriously hard, for instance in \mathbb{Z}_p^* . A related problem is

The Diffie-Hellman (DH) problem

Given a group G , generator α , and α^a, α^b , where a, b are randomly and independently chosen from \mathbb{Z}_t , compute α^{ab} .

Clearly, if we could find a from α^a , we could solve DH by a single exponentiation, so

Lemma 9.1 *The DH problem is no harder than the DL problem.*

It is not known if the opposite direction is true in general, but in some groups, the problems are equivalent. In particular, suppose the factorisation of t (the order of G is known), suppose that for each *large* prime factor q in t , q^2 does not divide t . Then under plausible number theoretic assumptions, it can be shown that the DH problem is at least as hard as the DL problem [6].

Note that the DH problem has a peculiar property, namely if I give you a group element and I claim it solves a DH instance, it is not clear that you can verify that the solution is correct, at least not unless you can solve DL. You would need to decide if, for given $\alpha^a, \alpha^b, \alpha^c$, it holds that $c = ab \bmod t$. This seems to require that you solve DL (although it is NOT clear that this would be necessary). This is the motivation for defining a final related problem. The idea is that you get an instance of the DH problem, plus an extra group element which is either a correct DH solution, or is a random element. You are then supposed to guess which case you are in.

The Decisional Diffie-Hellman (DDH) problem

Given a group G , generator α , and $\alpha^a, \alpha^b, \alpha^c$, where a, b are randomly and independently chosen from \mathbb{Z}_t ; and where c is chosen either as $c = ab$, or uniformly random from \mathbb{Z}_t . Now guess which of the two cases we are in.

Clearly, if you could solve DH, then you could solve DDH, by computing α^{ab} and comparing this to α^c . So we have:

Lemma 9.2 *The DDH problem is no harder than the DH problem.*

There are no types of groups known (other than trivial cases) for which we can show that DDH is equivalent to DH. In fact, there are cases where DDH is known to be easy, but DH is conjectured to be hard. In certain subgroups of \mathbb{Z}_p^* , however, DL, DH and DDH all seem to be hard. In the next section, we define more precisely what it means for these problems to be hard.

9.3.1 Hardness of the problems

There exist several methods by which you can efficiently choose a prime p of a particular size (bit length) that you want together with a primitive element. We generalize this notion by assuming that we have a *group generator* $GGen$, i.e., an efficient probabilistic algorithm which takes as input an integer k and outputs a group G and an element $\alpha \in G$ that generates G . The idea is that k controls the size of the group that is generated, and the specification produced allows us to work in G , just as knowing p allows us to work in \mathbb{Z}_p^* .

We need this kind of tool to talk about hardness of the DL, DH and DDH problems. The point is that if, for instance, we want to work in groups of the type \mathbb{Z}_p^* , it is possible to choose p in such a way that none of our problems are hard. As an example, consider an algorithm that always outputs primes p , where $p-1$ has only small prime factors. There is an algorithm known as Pollard-Hellmann that in this case solves DL in \mathbb{Z}_p^* efficiently (and hence also DH and DDH). On the other hand, DL does seem to be hard, if we choose *random* primes of length k . So therefore, these problems can only have the kind of hardness we need relative to some algorithm that chooses the group we are in ¹.

So hardness of DL w.r.t. $GGen$ means that if we use $GGen$ to make an instance of the DL problem, then any polynomial time algorithm can solve such an instance with probability essentially 0. More formally:

Definition 9.3 Consider the following experiment with an algorithm A : run $GGen$ on input k to get G and α . Choose a at random in \mathbb{Z}_t , and give A input G, α, α^a . The DL problem is said to be hard (with respect to $GGen$) if for any polynomial (in k) time algorithm A , the probability that A outputs a is negligible in k .

We can define hardness of DH and DDH in a similar way:

Definition 9.4 Consider the following experiment with an algorithm A : run $GGen$ on input k to get G and α . Choose a, b at random in \mathbb{Z}_t , and give A input specification of $G, \alpha, \alpha^a, \alpha^b$. The DH problem is said to be hard (with respect to $GGen$) if for any polynomial (in k) time algorithm A , the probability that A outputs α^{ab} is negligible in k .

¹ This is more or less the same reason that is also behind the fact that it is meaningless to talk about security of a cryptosystem without taking the algorithm for generating keys into account

Definition 9.5 Consider the following experiment with an algorithm A : run $GGen$ on input k to get G and α . Choose a, b at random in \mathbb{Z}_t . Now, we can be in one of two cases. In the “real” case, set $c = ab$, in the “ideal” case, choose c at random from \mathbb{Z}_t . Give A input specification of $G, \alpha, \alpha^a, \alpha^b, \alpha^c$. A then outputs one bit, namely its guess at whether we are in the real or the ideal case. Now, let $p_{A,0}(k)$ be the probability that A outputs 1 in the real case, and $p_{A,1}(k)$ be the probability that A outputs 1 in the ideal case. The *advantage* of A is as usual defined to be

$$Adv_A(k) = |p_{A,0}(k) - p_{A,1}(k)|$$

The DDH problem is said to be hard (with respect to $GGen$) if for any polynomial (in k) time algorithm A , $Adv_A(k)$ is negligible in k .

Note that hardness of DDH is defined in the same way we have defined security of cryptosystems, namely it is hard for the adversary to distinguish the “real” from the “ideal” world with non-negligible advantage.

9.4 The El Gamal Cryptosystem

The motivation for the DH problem is that Diffie and Hellman in 1977 suggested to use it as a basis for exchanging a secret between two parties A and B that share no secret key in advance. The method for this is very simple, assuming that we already agreed (in public) on a group G and a generator α :

1. A chooses s_A at random in \mathbb{Z}_t , B chooses s_B at random in \mathbb{Z}_t .
2. A sends $y_A = \alpha^{s_A}$ to B , B sends $y_B = \alpha^{s_B}$ to A .
3. A computes $y_B^{s_A}$ and B computes $y_A^{s_B}$

The point is of course that A and B compute the same value in the last step, since $y_B^{s_A} = y_A^{s_B} = \alpha^{s_A s_B}$. Furthermore, an adversary observing the communication would need to solve an instance of the DH problem to find the shared value. El Gamal suggested a way to turn this idea into a regular public key cryptosystem. What we do is essentially to consider A ’s first message in the above as a part of his public key, then B ’s part of the protocol can be modified to be an encryption:

El Gamal cryptosystem (general version)

Key generation On input security parameter k , run $GGen$ on input k to obtain specification of a group G and generator α . Choose a at random from \mathbb{Z}_t . Then the public key is the specification of G and $\beta = \alpha^a$, while the secret key is a . The plaintext space is G while the ciphertext space is $G \times G$.

Encryption To encrypt $m \in G$, we choose r at random from \mathbb{Z}_t , and the ciphertext is $(\alpha^r, \beta^r m)$.

Decryption To decrypt ciphertext (c, d) , compute $c^{-a}d$.

To see that decryption works, simply plug in $(\alpha^r, \beta^r m)$ for (c, d) in the decryption algorithm.

Note that there may be some difficulties in applying this scheme in practice: we have said that the plaintext space is the group G . But in real life, we probably want to encrypt some arbitrary piece of data, given as a bit string. It may not be clear how to consider such a string in a unique way as group element. For \mathbb{Z}_p^* this is easy, as long as the string is not too long: just think of it as a binary number. Other cases are less obvious - we return to this later.

Given a ciphertext $(\alpha^r, \beta^r m)$, it is clear that you can compute m if and only if you can compute $\beta^r = \alpha^{ar}$, so we have:

Lemma 9.6 *The problem of decrypting an El Gamal ciphertext (without the secret key) is equivalent to solving the DH problem.*

However, this does not say much about the CPA security of El Gamal: even if it is hard to compute the entire plaintext, we may be able to compute partial information about it. However, if DDH is hard, then even you are given $\alpha, \alpha^a, \alpha^r$, it is infeasible to distinguish α^{ra} from a random element. This means that for all you care, the last part of the ciphertext that determines m (namely $\alpha^{ra}m$), might as well have been a random element – which would contain no information on m . So in this case, we can expect to have semantic security. More formally:

Theorem 9.7 *If the DDH problem is hard (w.r.t. $GGen$), then the El Gamal cryptosystem is CPA secure.*

Exercise 9.1 (CPA security of El Gamal) Prove the above theorem. You will do this by contradiction: Assume that there exists some adversary Adv that plays the CPA security game and has advantage at least ϵ . Now:

1. Construct an algorithm B that uses Adv as subroutine and attempts to solve DDH. Concretely, this means that B gets as input $G, \alpha, \alpha^a, \alpha^b, \alpha^c$, and must eventually output a guess, either “ c is random” or “ $c = ab$ ”.
2. Show that your algorithm achieves advantage at least ϵ .

The conclusion is that if Adv is polynomial time and ϵ is not negligible, the existence of B demonstrates that DDH cannot be hard, we have a contradiction, and so such an Adv cannot exist.

There are also results about El Gamal similar to what is known for RSA:

Exercise 9.2 (Multiplicative property of El Gamal) RSA is multiplicative, i.e. the product of two encryptions is an encryption of the product of the messages. Formulate and prove a similar property for EL Gamal encryption. Use this to prove the following: given an algorithm A which on input a public El Gamal key $pk = (G, \alpha, \beta)$ and a random ciphertext, outputs the correct plaintext with probability ϵ . Construct an algorithm which on input pk and *any fixed* ciphertext (c, d) , decrypts it correctly with probability ϵ .

9.5 Some Example Groups

9.5.1 \mathbb{Z}_p^*

These are probably the most well-known examples. For this kind of group, a natural candidate for the *GGen* algorithm would be: On input k , choose a random k -bit prime p and an arbitrary generator $\alpha \in \mathbb{Z}_p^*$. Note that, by Theorem 3.14, a generator always exists.

Choosing a random k -bit prime is something we have already seen in RSA key generation. As for finding a generator, it turns out that there are so many generators, that it will work fine to simply choose random candidates for α until we find a generator. Moreover, if we know the factorization of $p - 1$, we can recognize a generator when we see one:

Lemma 9.8 $\alpha \in \mathbb{Z}_p^*$ is a generator if and only if $\alpha^{(p-1)/q} \neq 1$ for every prime q that divides $p - 1$.

PROOF α is a generator if and only if its order is $p - 1$. The order of α can be characterized as follows: consider $\alpha^i \bmod p$, for $i = 1, 2, \dots$. The order is the first value of i where we get 1. This means that if α is a generator it clearly satisfies the condition $\alpha^{(p-1)/q} \neq 1$ since the exponent $(p - 1)/q$ is less than $p - 1$.

Conversely assume α satisfies the condition and call its order t . We want to argue that $t = p - 1$. By Lagrange's theorem at least we know that t divides $p - 1$. Note first $\alpha^{p-1} \bmod p = 1$ because we are in the group \mathbb{Z}_p^* which has order $p - 1$ and raising to group order always gives 1. So we have $p - 1 = at$ for some a . We claim that $a = 1$. Assume for contradiction that $a > 1$ and take a prime q that divides a and hence also divides $p - 1$. Since α satisfies the condition, we have

$$1 \neq \alpha^{(p-1)/q} \bmod p = \alpha^{at/q} \bmod p = (\alpha^t)^{a/q} \bmod p = 1^{a/q} \bmod p = 1$$

which is a contradiction, so we conclude $a = 1$ and hence $t = p - 1$ as desired. \square

Fortunately, there are methods for building random primes p in such a way that we also know the factorization of $p - 1$. Details of this in general are out of scope for this note, but a special case of the method is simple to understand: we can choose a random prime q , set $p = 2q + 1$ and test whether p is a prime. If not, we choose a new q . When this succeeds, we certainly know the factors of $p - 1$, namely 2 and q . What about the efficiency? Heuristically, we would expect that p is prime with about the same probability with which a random number of that size is prime, so by the prime number theorem if we want a k -bit p , we would expect to generate $O(k)$ values of q before p happens to be prime. This is indeed exactly what happens in practice.

A prime p generated this way is sometimes called a safe prime, and such primes turn out to be especially useful for encryption, as we explain later.

Unfortunately, however, if we use $G = \mathbb{Z}_p^*$ in El Gamal, we will *never* get a CPA-secure cryptosystem. To see this, we need the following basic fact:

Lemma 9.9 Let α be a generator of \mathbb{Z}_p^* . Then $(\alpha^i)^{(p-1)/2} \bmod p = 1$ if and only if i is even, and is -1 otherwise.

PROOF Clearly we have for any $\gamma \in \mathbb{Z}_p^*$ that $\gamma^{(p-1)/2} \bmod p$ is congruent 1 or -1 – this follows since $(\gamma^{(p-1)/2})^2 \bmod p = 1$ by Langrange, and because \mathbb{Z}_p is a field, the quadratic equation $X^2 = 1 \bmod p$ can have at most 2 solutions, namely 1 and -1.

Therefore since α is a generator, it must be that $\alpha^{(p-1)/2} \bmod p$ is congruent to -1: it cannot be 1 by Lemma 9.8.

We can now rewrite the result we want as follows:

$$(\alpha^i)^{(p-1)/2} \bmod p = (\alpha^{(p-1)/2})^i \bmod p = (-1)^i \bmod p$$

which is clearly 1 if i is even and -1 otherwise. \square

This implies:

Lemma 9.10 *In the group \mathbb{Z}_p^* , the DDH problem is not hard*

PROOF The adversary is given $\alpha, \alpha^a, \alpha^b, \alpha^c$. By the above, he can easily compute parity of a, b , i.e., whether a, b are even or odd, and hence the parity of ab . He can also find the parity of c . Of course, if $c = ab$, these two parities always match, whereas if c is randomly and independently chosen, they will match with probability $1/2$. So the adversary compares the parities of ab and c and if they are equal he guesses that he is in case $ab = c$, otherwise he guesses that c was random. Clearly, his advantage is $1 - 1/2 = 1/2$ which certainly is not negligible. \square

In fact, the problem we just saw with using \mathbb{Z}_p^* is a special case of something more general:

Exercise 9.3 Assume we are given group G with generator α , where a prime q divides the order t of G .

1. Show that the order of $\alpha^{t/q}$ is q .
2. Assume you are given α^i . Show that, by computing $(\alpha^i)^{t/q}$, you can determine $i \bmod q$ in time proportional to q .
3. Show that, using the answer to the previous question, one can solve DDH in G with advantage $1 - 1/q$ in time proportional to q .

This exercise shows that if the group order is divisible by any prime of size polynomial in the security parameter, then DDH is not hard. So we really need large and prime order groups.

9.5.2 Prime Order Subgroups of \mathbb{Z}_p^*

To obtain CPA-security, we need to use, instead of \mathbb{Z}_p^* , a subgroup of large prime order. The reason for this is that the problem we spotted before with the DDH problem really comes from the fact that the order of the group we had ($p-1$) was divisible by a very small prime, namely 2. This means that by raising elements to the power $(p-1)/2$, we can “squeeze” things into a very small subgroup (consisting of 1 and -1) where DDH is not hard to handle.

The simplest way to solve the problem is to use the so called safe primes we mentioned earlier, that is, a prime p where $p = 2q + 1$ and q is also prime.

Now, for every prime q that divides $p - 1$, there is exactly one subgroup G of order q . In case of a safe prime, if α_0 generates all of \mathbb{Z}_p^* , then $\alpha = \alpha_0^2$ generates this subgroup. This follows, simply from the fact that there are q different multiples $2i$ between 0 and $p - 1$, and so G simply consists of the elements $\{\alpha^1, \alpha^2, \dots, \alpha^q\} = \{\alpha_0^{2 \cdot 1}, \alpha_0^{2 \cdot 2}, \dots, \alpha_0^{2q}\}$.

This leads to the following algorithm for generating G and α :

1. On input k , use known methods to generate a k -bit prime p with $p = 2q + 1$, and generator α_0 of \mathbb{Z}_p^* .
2. Let G be the subgroup of order q , and set $\alpha = \alpha_0^2$. Output p, q, α .

Then we can use general El Gamal, with these choices of G, α . There is one problem, however: in practice, we want to encrypt some bit string str . The plaintext space is really G , and it consists of only some of the numbers in \mathbb{Z}_p^* . So we cannot just interpret str as a binary number modulo p : there is no reason to expect that this number will be in G . The simplest way to solve this problem is to use safe primes, that we defined above. Recall that these are primes of form $p = 2q + 1$, where q is also prime. With such a prime, the following encoding enables us to encrypt any number in Z_q . It provides a way to map efficiently and 1-1 from Z_q to G :

- On input $x \in Z_q$, set $y = x + 1$ and compute $y^{(p-1)/2} \bmod p$. If this is 1, output y , else output $-y \bmod p$.

When we decrypt, we get an element in G , and we can then bring this back to Z_q as follows:

- On input $g \in G \subset \mathbb{Z}_p^*$, test if $g \leq q$. If so, set $y = g$, else set $y = -g \bmod p$. Output $y - 1$.

Exercise 9.4 (Encoding for Safe Primes) Note that when $p = 2q + 1$, the subgroup G of order q in \mathbb{Z}_p^* consists of all even powers of α , where α is a generator of all of \mathbb{Z}_p^* . Use this and Lemma 9.9 to prove that the above encoding method works, i.e., the mapping introduced is 1-1, and maps to the subgroup G .

Exercise 9.5 (Malleability of El Gamal) Use an idea similar to that of Exercise 2 to show the following: using only the public key, one can transform an El Gamal ciphertext that encrypts message m efficiently into a different ciphertext that also encrypts m .

Use this to show that the El Gamal cryptosystem is not CCA secure, see Definition 8.7.

One might hope that we could make El Gamal be CCA secure by using a procedure similar to what we saw for OAEP: select a small subset of the messages and define only those to be legal. If the decryption sees an illegal message it will output an error. We shall now see that this does not work for El Gamal.

Concretely, suppose we change the cryptosystem as follows: say we are given

an injective and easy to invert function $f : \{0, 1\}^t \mapsto G$ (supposedly for a small value of t). to encrypt a bit string m , we encrypt $w = f(m)$ using El Gamal. The decryption first does El Gamal decryption to get w . if $w \in \text{Im}(f)$, outputs $f^{-1}(w)$, and outputs an error if $w \notin \text{Im}(f)$.

Show that this cryptosystem is not CCA secure either, regardless of which function f we use.

Optional question: can you think of a way to modify El Gamal that might make it CCA secure? (you do not need to prove that your suggestion works).

Note: you may find that the attack you came up with for the second question is more or less the same as for the first one. This is fine, you did not overlook something.

9.5.3 Groups based on Elliptic Curves.

A widely used family of candidate groups is Elliptic Curve groups.

The interesting point about them is that only the (very slow) generic discrete log algorithm (which we will see later) is known to solve DL in these groups, when they are properly constructed. We will discuss this further when we have seen what elliptic curves are about.

We will not go into many details here, but we will explain some basic notions. One way to think of an elliptic curve is to see it as a set of points (x, y) in the real plane defined by the equation

$$y^2 = x^3 + ax + b,$$

where a, b satisfy $4a^3 + 27b^2 \neq 0$. To the set of points defined this way we add a “point at infinity”, called \mathcal{O} . If we draw the curve in a coordinate system, we can think of \mathcal{O} as being infinitely far away from the origin, in the y -direction. The reason for this strange-looking addition to our collection of points will become clear in a moment.

Quite surprisingly, this set of points has a group structure. The group operation is usually written as addition, and in this case, where we can draw the curve, the group operation has a nice geometric representation. To add 2 points P and Q , we first draw the line through both points. Because of the constraint on a, b , this line will intersect the curve in a third point (with some exceptions we handle in a moment). We take this point and reflect it over the x -axis, or equivalently flip the sign of its y -coordinate. Because the equation of the curve involves the y -coordinate only as y^2 , this new point is also on the curve, and this point is defined to be $R = P + Q$, see Figure 9.1. We can also add a point P to itself, in which case we take the tangent to the curve that goes through P instead of the line through both points. Moreover, if P and Q are directly above each other, the line through both points is vertical and there is not a third intersection point. In this case we say that the third intersection point is infinitely far away, and so is the point \mathcal{O} that we added. Minus this point is also infinitely far away, so it makes sense to define $P + Q = \mathcal{O}$ for such P and Q .

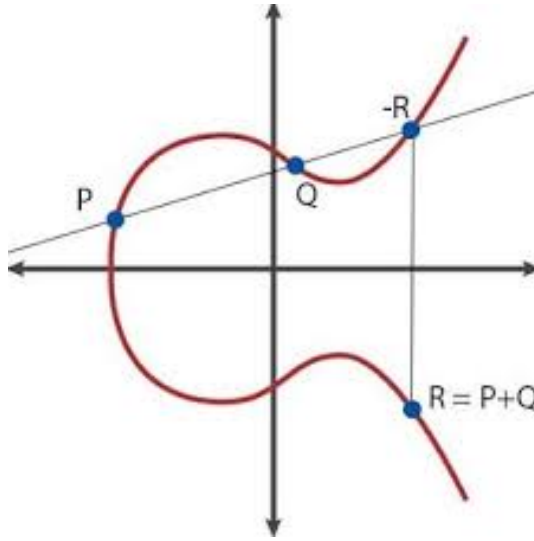


Figure 9.1 Elliptic curve point addition

Finally, consider what happens if we add \mathcal{O} to some point P , say with coordinates (x, y) . Naturally, the line through these points is a vertical line through P , and hence the third intersection point will have coordinates $(x, -y)$. The final step is to flip the sign of the y -coordinate which takes us back to $(x, y) = P$. So what we see is that \mathcal{O} is the neutral element in our group, and the inverse of P , that is, $-P$ is $(x, -y)$. We will not show formally that this gives a group structure, in particular the proof of associativity is a bit cumbersome.

What is clear, however, is that we can derive formulas for the group operation in terms of the coordinates of the points we add. So let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and define $P + Q = (x_3, y_3)$. Then we have the following cases:

- If $x_1 \neq x_2$, we have the general case where there is a third intersection point on the actual curve. Here we set $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$, the slope of the line through P and Q . Then it turns out that $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = \lambda(x_1 - x_3) - y_1$.
- If $x_1 = x_2$ and $y_1 = -y_2$, we have the case where P and Q are above each other, and then $P + Q = \mathcal{O}$.
- If $x_1 = x_2$ and $y_1 = y_2$, we need the tangent to the curve in point P , the slope of which can be calculated to be $\lambda = \frac{3x_1^2 + a}{2y_1}$. Then the rest is the same as in the first case, so $x_3 = \lambda^2 - x_1 - x_2$ and $y_3 = \lambda(x_1 - x_3) - y_1$.

The details of how these formulas are derived are not very important. What is important is that they exist. The point is that we can use them to define elliptic curves over the field \mathbb{Z}_p :

Let $p > 3$ and let $a, b \in \mathbb{Z}_p$ be such that $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. Then the elliptic curve $E_{a,b,p}$ is defined as follows:

$$E_{a,b,p} = \{(x, y) \mid y^2 \equiv x^3 + ax + b \pmod{p}\} \cup \mathcal{O},$$

in other words, it is defined exactly as for the case of real coordinates, we just do everything in \mathbb{Z}_p instead. Furthermore $E_{a,b,p}$ is a group, where the group operation is defined using exactly the same formulas we just saw for the case of real coordinates.

So now that we have a finite group, the question is if we can use it as the basis for El Gamal encryption? For this to be the case, the group certainly needs to be large, so that there is a chance that the DL problem is hard in this group. Fortunately, we know roughly how large the order is going to be, by a theorem of Hasse:

Theorem 9.11 (Hasse) *The order N of $E_{a,b,p}$ satisfies $p + 1 - 2\sqrt{p} \leq N \leq p + 1 + 2\sqrt{p}$,*

However, it is not enough that the order is large, as we know from our study of \mathbb{Z}_p^* , we actually need a large prime order group. It turns out that there is a method known as *complex multiplication*, which allows us to construct p, a, b from a prime q such that the order of $E_{a,b,p}$ is q . The details are quite complicated and out of scope for this text. It should also be mentioned that there is an algorithm by Rene Schoof which computes in polynomial time the order of an elliptic curve group.

Using Elliptic curve based El Gamal for encryption

A problem in using elliptic curve groups for encryption is that if we use the El Gamal crypto system directly as we described it before, the plaintext space is going to be the underlying group, so in this case it will be $E_{a,b,p}$. This means that the ciphertexts we can create will have form $E_{pk}(x, y)$ where the plaintext (x, y) is a point on the curve. Usually, however, we would want to encrypt a bit-string that is determined by some application that knows nothing about elliptic curves. And it is not immediately clear how to encode a bit string as a point on a curve.

One way out of this problem is to do a two-step process. First encrypt a random point (x, y) (on the curve) using El Gamal, and then somehow use (x, y) as a secret key for encrypting the message itself. Intuitively, the adversary sees an El Gamal encryption of (x, y) , but if that encryption is CPA secure, then the adversary has no idea what (x, y) is. So then it should be fine to use (x, y) for encrypting the message. The question, however, is exactly how we do this? Perhaps we can just write down (x, y) as a binary string and use it as a one-time pad? Well, the good news is that, from the adversary's point of view, (x, y) is indistinguishable from a random group element. if DDH is hard in group. The bad news is that is not a totally random pair of elements in \mathbb{Z}_p because it must satisfy the equation defining the curve. This means that not every pair (x, y) can occur. Hence (x, y) does not form a uniformly random bit string and so we cannot use (x, y) directly as the key in a one-time pad encryption.

Instead, we can try to produce a random string from (x, y) . One approach is to use a cryptographic hash function H (see the chapter on authentication for more info) with k -bit output, where $k < c \cdot \log_2(p)$ where $0 < c < 1$ is a constant. And

then encrypt a k -bit message m as

$$E_{pk}((x, y)), H(x, y) \oplus m.$$

Here $E_{pk}((x, y))$ denotes the El Gamal encryption of the group element (x, y) . The receiver uses his secret key to decrypt (x, y) and computes $H(x, y) \oplus (H(x, y) \oplus m) = m$.

We can argue intuitively why this should be secure: if we assume DDH in the elliptic curve group is hard, then El Gamal encryption is CPA-secure, and this means that from the adversary's point of view, (x, y) might as well be a completely random group element. So, we can argue as if this really was the case. This means there are roughly p possibilities for (x, y) by Hasse's theorem. Since we assumed $k < c \cdot \log_2(p)$, we have $2^k < 2^{c \log_2(p)} = p^c$. This number is much less than p : since $c < 1$ and p is exponentially large in the security parameter, we see that the quotient $p^c/p = p^{-1+c}$ is negligible.

So when we apply the hash function, we are hashing from a set containing roughly p elements into the much smaller set of k -bit strings. It is therefore a perfectly reasonable assumption that when (x, y) is a random group element, $H(x, y)$ is (very close to being) a random k -bit string. This can be proved in an idealized model of the hash function, where each output is randomly chosen, the so-called random oracle model. To see why this is the case note that we can model the idealized case by imagining a ball for each input and a bin for each possible output values. Choosing random outputs for each input corresponds to throwing each ball into a random bin. Since we have many more balls than we have bins, we can expect that each bin has approximately the same number of balls. This in turn implies that choosing a random input (ball) leads to choosing a random output (bin).

In conclusion, this means that $H(x, y)$ can indeed be used as a secure one-time pad to encrypt m .

In real life and in various standard texts, one finds a large number of variations on this theme, but the basic ideas remain the same.

Attacks on and advantages of elliptic curve crypto

As we have seen the group $E_{a,b,p}$ has order approximately p . In the general case the only algorithm that is known to solve DDH, DL or CH on a curve is the generic algorithm (which we will see below). This algorithm takes time proportional to the square root of the order of the group. That is, we need to do approximately \sqrt{p} group operations, which implies that p should be a 200-300 bit prime with current state of the art. This compares favorably with the 2000 or more bits one would need if we used \mathbb{Z}_p^* directly, so that keys and ciphertexts can be shorter, and (in some cases) the encryption/decryption faster. This is the reason why elliptic curve crypto is taking over from RSA in many market segments. On the other hand, the key generation is considerably more complicated, needing advanced methods such as complex multiplication that we mentioned earlier.

Super-singular elliptic curves: curse or blessing?

When a curve $E_{p,a,b}$ is such that $|E_{p,a,b}| \bmod p = 1$, the curve is said to be *super-singular*. Such curves are problematic from one point of view. Namely, it turns out that one can solve discrete log on a curve by instead solving it in a finite field \mathbb{F}_{p^d} where for super singular curves $d \leq 6$. Simplifying a bit, it turns out that there is a homomorphism $\phi : E_{p,a,b} \mapsto \mathbb{F}_{p^d}$, which can be computed efficiently, and so if we are given $\alpha, \beta = \alpha^a$ and we want to find a we can compute $\phi(a)$ and $\phi(\beta) = \phi(\alpha)^a$ which is a discrete log problem in \mathbb{F}_{p^d} . In the finite field \mathbb{F}_{p^d} one can apply a variant of the Index calculus algorithm (see below) and since d is not very large we can solve the problem much faster than if we had to attack it directly on the curve using the generic algorithm.

It should be mentioned that any curve has such a homomorphism into \mathbb{F}_{p^d} for some d called the embedding degree. However, usually d is very large and hence mapping the problem into the field is useless for an attacker. It is only for curves of a very special form such as super-singular curves that the embedding degree is small.

The take-home message is that if the goal is simply public-key encryption, one should avoid super-singular curves, since they need to be very large to be secure. In practice, it is a good idea to use standardized curves which have been verified to have large embedding degree.

On the other hand, super-singular and similar curves also have advantages: they come with a so-called pairing which is a function with very nice algebraic properties. With pairings, various interesting cryptographic constructions become possible that we do not know how to implement otherwise. If we choose the curves large enough, discrete log will still be hard despite the attacks mentioned here. Pairing based crypto is a large and interesting research field, but is not in scope of this text and will not be described in detail here.

9.6 Algorithms for solving discrete log

9.6.1 The generic algorithm

The most basic algorithm we will consider is the *generic discrete log algorithm* which has its name from the fact that it works in any group and uses no special properties of the group in question.

So assume we are given G, α and β , and we want to find the discrete log of β base α . Let x be the solution, and set $t = \lceil \sqrt{|G|} \rceil$. We can imagine dividing x by t to get a quotient q and a remainder r , that is, we can define q, r by

$$x = qt + r, \text{ where } 0 \leq r < t.$$

Note that since $0 \leq x < |G|$, we also have that $0 \leq q < t$. Clearly, if we can find q and r , we can also easily compute x .

Now, consider that $\beta = \alpha^x = \alpha^{qt+r}$ which implies that the correct values of q and r satisfy $\beta\alpha^{-r} = (\alpha^t)^q$. Therefore, the generic algorithm does the following:

1. Compile a table T with entries of the form $(r, \beta\alpha^{-r})$ for $r = 0, 1, \dots, t$ and sort according to the last component.
2. For $q = 1$ to t , compute $(\alpha^t)^q$, test if this value is in T . If yes, exit the loop and record the corresponding values of r, q .
3. return $x = qt + r$.

Clearly, this algorithm runs in time $O(\sqrt{|G|})$. Of course, the memory consumption is very large, but there are tricks that allow to make this smaller. Importantly, it can be shown that the generic algorithm is optimal in the sense that any algorithm that uses no special assumptions on the group must use time $\Omega(\sqrt{|G|})$.

9.6.2 Index Calculus

The Index Calculus algorithm is specifically targeted against the group \mathbb{Z}_p^* . It exploits the fact that elements in this group can also be seen as integers and as such can be factored into primes. Similarly to the best factoring algorithms it uses a factor base which is a set F consisting of the first small primes, $F = \{p_1, \dots, p_B\}$.

The algorithm first does a preprocessing stage, where the input for which we want the discrete log is not used at all, the only input used is p and the base element α , which we assume here for simplicity is a generator, so it has order $p - 1$. The goal of this step is to find the discrete log of all the primes in F . It looks as follows:

1. Choose $r \in \mathbb{Z}_{p-1}$ at random and compute $y = \alpha^r \bmod p$.
2. If y (interpreted as an integer) does not factor over F , go to Step 1. Else write y as $y = \prod_{i=1}^B p_i^{t_i}$, and save (r, t_1, \dots, t_B) in a table T .
3. If T has more than B entries, go to Step 4, else go to Step 1.
4. Let x_i be the discrete log of p_i base α , so $p_i = \alpha^{x_i} \bmod p$. Note that each entry in T defines a linear equation in the x_i 's, as follows:

$$\alpha^r \equiv y = \prod_{i=1}^B p_i^{t_i} \equiv \alpha^{\sum_{i=1}^B t_i x_i} \bmod p$$

This equation is true modulo p if and only if the exponents are equal modulo the order of α , so we have

$$r \equiv \sum_{i=1}^B t_i x_i \bmod (p - 1)$$

If we arrive at this stage, we have more than B equations in B unknowns, so we can solve a linear equation system to get the x_i 's².

² In practice, some difficulties arise from the fact that $p - 1$ is not a prime so one may need to divide by numbers that are not actually invertible modulo $p - 1$. Most of this can be handled using the Chinese Remainder Theorem to solve the system modulo each of the prime factors in $p - 1$, and then assemble a complete solution at the end.

In the final stage, we are given a β for which we want to find the discrete log base α . Say $\beta = \alpha^x \bmod p$, so x is the solution we are looking for. We proceed as follows:

1. Choose $s \in \mathbb{Z}_{p-1}$ at random, and compute $z = \alpha^s \beta \bmod p$.
2. If z does not factor over F , go to Step 1. Otherwise, let $z = \prod_{i=1}^B p_i^{s_i}$. Substituting α^{x_i} for p_i , we see that this equation is equivalent to

$$x + s \equiv \sum_{i=1}^B s_i x_i \bmod (p-1),$$

so we output $\sum_{i=1}^B s_i x_i - s \bmod (p-1)$.

This algorithm spends nearly all its time in the preprocessing stage, so this means that once that stage is done for a particular prime and generator, a lot of discrete logs can be computed very efficiently. This means that if one uses a subgroup of \mathbb{Z}_p^* as the basis of El Gamal encryption, one should not let many users share the same prime and generator. If we do, it means that a powerful adversary could attack a large set of users at smaller amortized cost because the preprocessing can be used for all users in that system.

Because Index Calculus shares its high-level structure with the best factoring algorithms, the optimizations that have been found for factoring also apply here. More concretely, clever ways have been found to choose the y -values in the preprocessing in such a way that they tend to be smaller than if the choice had been uniform. Hence, the probability that they factor of the factor base is larger. This also means that the complexity of finding a discrete log modulo a prime p is similar to the cost of factoring a number of the same bit length.

Public-Key Encryption Based on Learning With Error (LWE)

Due to the fact that quantum computers (if they could be built in large enough size) can break both RSA and all discrete log based cryptosystems, it is of clear interest to build public-key crypto from problems that are hard even for quantum computers (as far as we know).

One example of this is cryptosystems based on the Learning with errors (LWE) problem. This problem is defined over a finite field F_q where q is a prime, and all computations in the following will be modulo q , even if we do not say so explicitly.

Also a secret vector $\mathbf{s} \in F_q^n$ will be involved. The LWE problem is now: You are given random $\{\mathbf{a}_i \in F_q^n\}_{i=1}^m$ and $\{\mathbf{a}_i \cdot \mathbf{s} + e_i\}_{i=1}^m$ where the e_i 's are random but numerically "small". The goal is now to find \mathbf{s} .

The e_i 's being random and numerically "small" can mean different things, but always means that they are much smaller than q . For instance, e_i can be chosen uniformly in the interval $[-\sqrt{q}, \sqrt{q}]$. Or it can be chosen using a discrete Gaussian distribution with mean 0 and standard deviation much smaller than q . It turns out that the exact details of this are not very important as to how hard the problem is. In the following we will simply assume that the e_i 's are chosen with a distribution D_e with the following important property: *even if you sample m values distributed according to D_e , take numeric value and sum them, the result will be smaller than $q/4$ with overwhelming probability.* The reason for this exact condition will become clear later.

The motivation for the name of the problem is that the adversary gets some partial information on \mathbf{s} and tries to learn it from this information, but the problem is non-trivial because the "errors" e_i are added in. Of course, without the errors the problem would be easy once $m > n$ since then one just solves the linear equations to get \mathbf{s} .

There is also a decision version of LWE: you are given random $\{\mathbf{a}_i \in F_q^n\}_{i=1}^m$ and then either $\{\mathbf{a}_i \cdot \mathbf{s} + e_i\}_{i=1}^m$ or $\{u_i\}_{i=1}^m$ where the u_i are uniformly random. The goal is now to decide which case you are in.

We say that decision LWE is hard if any PPT algorithm can distinguish the two cases with only negligible advantage in n (where m is polynomial in n).

Not here the similarity between this and discrete log based problems: decision LWE is to LWE what DDH is to the DH problem.

As a warm-up, it is easy to see that we can make a secret-key cryptosystem from LWE: the secret key is \mathbf{s} and to encrypt a message bit w , we will send the

ciphertext

$$(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e + \lceil q/2 \rceil w)$$

To decrypt a ciphertext (\mathbf{u}, v) , the receiver can compute $v - \mathbf{u} \cdot \mathbf{s}$. For a valid ciphertext we see that

$$v - \mathbf{u} \cdot \mathbf{s} = \mathbf{a} \cdot \mathbf{s} + e + \lceil q/2 \rceil w - \mathbf{a} \cdot \mathbf{s} = e + \lceil q/2 \rceil w.$$

Clearly, if $e \ll q$, then this value will be close to 0 if $w = 0$ and close to $\lceil q/2 \rceil$ if $w = 1$.

Note that if we represent F_q by the numbers $0, 1, \dots, q-1$, then we will need to interpret “close to 0” as “close to 0 or close to q ”. This makes sense as we are computing mod q whence q is treated as being equivalent to 0. For instance, if e is small but negative, such as $e = -2$, then indeed $e \bmod q = q - 2$.

It is not hard to see that, if decision LWE is hard, then the adversary cannot distinguish a ciphertext from something of form $(\mathbf{a}, u + \lceil q/2 \rceil w)$ where u is uniformly random, and of course from the latter data, there is no information on w available at all.

The problem with turning this into a public-key system is that it seems one needs access to the secret key to form a ciphertext. But fortunately, one can bypass this problem if the owner of the secret key gives away some extra data. As a first step, note that a ciphertext containing w can be written as

$$(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e) + (0, \lceil q/2 \rceil w)$$

where the first part can be interpreted as a ciphertext that contains 0. So the problem of making a valid encryption of m can be rephrased to: first make a random encryption of 0 and then add something that only depends on w .

The next observation is that if one adds together two encryptions of 0, one gets an encryption of 0, albeit with an e -value that can be twice as large. So the idea is that the owner of the secret key publishes m random encryptions of 0, c_1, \dots, c_m . The sender can then form a random encryption of 0 by adding together a random subset of the c_i 's. This leads to the following cryptosystem:

Key Generation Secret key: random vector $\mathbf{s} \in F_q^n$. Public key: $\{c_i\}_{i=1}^m$, where $c_i = (\mathbf{a}_i, \mathbf{a}_i \cdot \mathbf{s} + e_i)$, where \mathbf{a}_i are uniformly random and the e_i are chosen according to D_e .

Encryption Message is a bit w . Choose random bits b_1, \dots, b_m and then the ciphertext is $\sum_{i=1}^m b_i c_i + (0, \lceil q/2 \rceil w)$

Decryption To decrypt (\mathbf{u}, v) , compute $v - \mathbf{s} \cdot \mathbf{u}$, and output 0 if this value is closer to 0 than it is to $\lceil q/2 \rceil$. Output 1 otherwise.

One can now show that because of the restriction on D_e that we assumed, decryption will work correctly, except with negligible probability (see exercise below).

Next, we want to argue that the cryptosystem is CPA secure. For this, we note that if decision LWE is hard, we can replace the c_i 's in the public key by pairs of form (\mathbf{a}_i, u_i) where the u_i are random, and the adversary would not be able

to tell the difference. One can now finish the proof by showing that if m is large enough, then encrypting w with this altered public key will result in a ciphertext that contains essentially no information on w .

The reason why this can be shown is the following result, which is stated here only informally:

Lemma 10.1 *If $m \geq n + (n + 1) \log_2(q)$, then the following two distributions can be distinguished with only negligible in n advantage (even by an unbounded adversary):*

$$\{(\mathbf{a}_i, u_i)\}_{i=1}^m, \sum_{i=1}^m b_i(\mathbf{a}_i, u_i)$$

$$\{(\mathbf{a}_i, u_i)\}_{i=1}^m, \mathbf{r}$$

where $\mathbf{r} \in F_q^{n+1}$ is chosen uniformly and independently of anything else.

What the lemma says is that $\sum_{i=1}^m b_i(\mathbf{a}_i, u_i)$ is essentially uniformly random even if you are given $\{(\mathbf{a}_i, u_i)\}_{i=1}^m$ (but not the b_i 's).

The intuition behind the condition on m is that to specify a random vector of form (\mathbf{a}, u) , one needs $(n + 1) \log q$ bits, so what the lemma says is that if we invest that many random bits b_i plus a few more, we will be able to produce a random vector \mathbf{r} as required.

This lemma implies that if we encrypt with the altered public key as mentioned above, we will be masking the message (essentially) by a totally random value, and so there is no information on w to be found in the ciphertext.

This basic form of cryptosystem is quite inefficient: to get reasonable security one needs q to be around 1000, and n to be a few hundred. The size of a ciphertext is $(n + 1) \log q$ bits, so this around 5000 if $n = 500$. It is of course not practical to expand a single bit to 5000 bits of ciphertext. But much more compact versions based on variants of LWE do exist. For details on this and much more, see [7].

Exercise 10.1 (Correctness of LWE-based encryption) Prove the claim made in the text: assume that the distribution D_e satisfies that $\sum_{i=1}^m |e_i| < q/4 - 1$ when each e_i is chosen according to D_e . Then the decryption defined for the public key LWE scheme always works correctly.

Symmetric Authentication Systems

Contents

11.1 Introduction	124
11.2 Hash Functions	126
11.2.1 Hash functions based on factoring and discrete log	127
11.2.2 Collisions-intractable hash functions are one-way	127
11.2.3 Domain extension for hash functions	128
11.2.4 Hash Functions in Practice	130
11.2.5 The random oracle model	133
11.3 Definition of Message Authentication Codes (MACs)	134
11.3.1 Security of Secret-Key Systems (MAC's)	134
11.4 Existence of good MAC Schemes	135
11.4.1 MACs from block ciphers	135
11.4.2 MACs from hash functions	137

11.1 Introduction

In this chapter, we look at systems for authentication of data, that is, how do we protect data from unwanted modification? Clearly, this is of utmost importance in E-commerce, home banking, database security, etc. If we could physically prevent unauthorized access to the data, this would of course solve the problem, but in most cases this is infeasible or even impossible. So if our data must be in an environment where we do not have physical control, of course we cannot *prevent* that the data is modified, but we can make sure that it cannot happen without us noticing that something is wrong. In other words, an adversary cannot make an honest user believe that a piece of data is authentic when it is not.

Usually, we model the scenario by having a sender that communicates data to a receiver over some channel, where an adversary can modify the communication any way he likes. Loosely speaking, the adversary wins if he can make the receiver accept a message that was not sent by the sender. Note that this also covers the case where a user stores data in a database and later, when retrieving the data, wants to make sure that it was not tampered with. In this case, sender and receiver is the same person, and we can think of the database as playing the role of communication channel.

All systems we know of for this purpose work by computing, from the data (say

a message m) and possibly a secret key, some *authenticator value* s . The idea is that later, the message can be checked against the authenticator, and we hope that we will see a mismatch if the data or the authenticator was changed. The way the systems work on a more technical level then varies, depending on whether the adversary is able to tamper with both the message and the authenticator, or only the message; and also depending on whether sender and receiver share in advance a secret key or not. In slightly more detail:

Adversary can only change the message: a typical application would be a user that computes an authenticator s for the data m . He stores m in some insecure location, but brings the authenticator with him in secure storage (say, on a chipcard). Later, he wants to check that the data has not been changed. Such systems can be built from *cryptographic hash functions*.

Adversary can change both message and authenticator: This will be the case when a sender communicates with a receiver over an insecure channel. There are two subcases:

Sender and receiver share a secret key: In this case, authenticator s is computed from both data m and the secret key k . m, s is sent on the channel, and m', s' is received. Then the receiver can, using the *same* key k , test if m', s' look OK. Of course, this should always be the case if $m', s' = m, s$, and we hope that anyone who does not know k cannot come up with a fake pair m', s' that the receiver would accept. Such systems can be built using *Message Authentication Codes* (MAC's) also called *Keyed Hash Functions*.

Sender and receiver share no secret keys in advance In this case, we have a scenario similar to that of public-key encryption: the sender has a secret key sk , and has published a public key pk . Now, the sender computes authenticator s from data m and secret key sk . m, s is sent on the channel, and m', s' is received. Now the receiver takes the public key pk of the user he believes sent the message and tests, using pk , if m', s' look OK. Such systems can be built using *Digital Signature Schemes*. The name comes from the fact that the sender is the only one who can produce valid authenticators using sk , since he is the only who knows sk – but using the public key pk , anyone can verify them. Ordinary handwritten signatures also (supposedly) has this type of property: I'm the only one who can produce my signature, but it can be verified by anyone. We will look at digital signatures in the following chapter.

In the following, we give some definitions, constructions and general results for the different types of schemes.

11.2 Hash Functions

A hash function is defined by a *generator* \mathcal{H} , which on input a security parameter k outputs the description of a function $h : \{0, 1\}^* \rightarrow \{0, 1\}^k$, i.e., a function mapping any bit string to a string of length k . By “the description” we simply mean information allowing us to compute the function efficiently (formally speaking, in polynomial time in k and length of the input string). Note that there is no secret key involved here - hence in applications of this concept, we assume that everyone (including the adversary) knows the function we are using.

One application of this is for the case where a data string m is to be stored in an insecure location. We then compute the authenticator $h(m)$ and store it in a secure location. Later, we retrieve a (possibly different) string m' , we then compute $h(m')$ and compare to $h(m)$. This application motivates why we want the output to have fixed length: it may be feasible to store a short fixed size value securely, while several megabytes are in most cases infeasible to handle. The value $h(m)$ is also sometimes called a *message digest* or a *message fingerprint*.

What properties do we need for such a function in order for the system to be secure? Clearly, an adversary sees the function h and the message m we stored. He would be in business, if he could find a different message m' such that $h(m) = h(m')$. This is called a *second preimage attack*. But it might also be relevant for the adversary if he could find *any* pair of messages $m \neq m'$, such that $h(m) = h(m')$ - a so-called collision. Now, if he can trick the honest user into storing m , he can later change this to m' without being detected. This is called a *collision attack*.

Clearly, if you can do a second preimage attack, you can also do a collision attack, using the same amount of time and with the same success probability: just choose m yourself and then run whatever algorithm you have for a second preimage attack. Therefore, the best security is obtained, if even a collision attack is infeasible, so this is what one usually uses to define security of a hash function:

Definition 11.1 Consider the game where we run \mathcal{H} on input k to get function h . We give h as input to adversary algorithm A , who outputs 2 strings m, m' . We say that A has success if $m \neq m'$ and $h(m) = h(m')$. We say that \mathcal{H} is collision intractable, if any PPT algorithm A has success with negligible probability (as a function of k).

The idea of defining a family of hash functions based on a generator may seem strange at first. Why can't we consider a single function and ask that it be hard to find collisions for that fixed function? The answer is that a definition requiring that no polynomial time adversary can find collisions for a *single* function would make no sense: for any fixed hash function, of course such an adversary exists, namely one that has a collision hardwired in its code and just outputs that collision. On the other hand as soon as we consider a large family of functions, we get around this problem: no single polynomial-time adversary can be “prepared” to break all of the many functions he might be subjected to.

11.2.1 Hash functions based on factoring and discrete log

Collision-intractable (aka collision-resistant) hash functions exist under well-known intractability assumptions:

We define \mathcal{H} as follows: choose a prime $p = 2q + 1$, where q is a $k - 1$ -bit prime. Choose α, β of order q in \mathbb{Z}_p^* . Now define a function $h : \mathbb{Z}_q \times \mathbb{Z}_q \rightarrow \mathbb{Z}_p^*$ by

$$h(m_1, m_2) = \alpha^{m_1} \beta^{m_2} \bmod p.$$

We claim that discrete log is a hard problem in \mathbb{Z}_p^* then this (family of) function(s) is collision intractable. We prove this by contradiction, namely assume one could find collisions and then show that this allows us to find discrete logs. So, if one could find a collision for h , this would give $(m_1, m_2) \neq (m'_1, m'_2)$ with $\alpha^{m_1} \beta^{m_2} = \alpha^{m'_1} \beta^{m'_2} \bmod p$. We must have either that $m_1 \neq m'_1$ or that $m_2 \neq m'_2$. Suppose without loss of generality that we have the first case, which means that $(m'_1 - m_1)^{-1} \bmod q$ exists. Then by rearranging the equation, we get

$$\alpha = \beta^{(m_2 - m'_2)(m'_1 - m_1)^{-1} \bmod q} \bmod p,$$

so we have now found the discrete logarithm of α base β .

Of course, this construction does not quite fit the definition because we cannot handle any input length. Fortunately, there is a general result we shall see later that solves this problem – it turns out to be enough if we can compress the input by only a single bit. The example we consider here does much better, it compresses (essentially) a $2(k - 1)$ bit input to a k -bit output.

Also the RSA assumption is sufficient to make collision-intractable functions:

Exercise 11.1 (Hash functions from Factoring) Given an RSA modulus $n = pq$, where $p = 2p' + 1, q = 2q' + 1$ and p', q' are also primes. Show that there exists elements in \mathbb{Z}_n^* of order $2p'q'$. Hint: remember that by the Chinese remainder theorem \mathbb{Z}_n^* is isomorphic to the direct product $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$, and both of those are cyclic. Let g be an element of order $2p'q'$, and define a function $h : \{0, 1\}^* \rightarrow \mathbb{Z}_n^*$ by $h(m) = g^m \bmod n$. Show that, given a collision for h , one easily factors n . Hint: recall that we already know from Lemma 7.9 that given any multiple of $(p - 1)(q - 1)$, we can easily factor.

11.2.2 Collisions-intractable hash functions are one-way

How does the question of existence of collision intractable hash functions relate to the other big theoretical questions in cryptography? it is very easy to see that if it is easy to invert a hash function h , then it is also easy to find collisions:

Lemma 11.2 Given function $h : \{0, 1\}^{2k} \rightarrow \{0, 1\}^k$, and assume we are given an algorithm A running in time t that, when given $h(m)$ for uniform m , returns a preimage of $h(m)$ with probability ϵ . Then a collision for h can be found in time t plus one evaluation of h and with probability at least $\epsilon/2 - 2^{-k-1}$.

PROOF The algorithm for finding a collision will simply choose a random m

and give $h(m)$ as input to A , which will return m' . If A has success, meaning that $h(m') = h(m)$ and $m' \neq m$, then we have a collision.

To see what the probability of getting a collision is, we call an input value m an *only child* if no other input is mapped to $h(m)$ (otherwise it has siblings that are also mapped to $h(m)$). Since there are 2^k output values, at most 2^k m 's can be an only child. Therefore, since the algorithm chooses m randomly, $P[m \text{ is an only child}] \leq 2^{-k}$. So if we let G denote the event that m is not an only child *and* that A has success, then we have $P[G] \geq \epsilon - 2^{-k}$. Note that if m is not an only child, there is at least one input other than m that maps to $h(m)$, and A has no information on which preimage we have chosen as m . So if we let C denote the event that we get a collision we have that $P[C|G] \geq 1/2$. Therefore the theorem follows from:

$$P[C] \geq P[C \cap G] = P[C|G]P[G] \geq 1/2 \cdot P[G] \geq (\epsilon - 2^{-k})/2$$

□

If ϵ is not negligible, then $\epsilon/2 - 2^{-k-1}$ is also not negligible. This means that if we have a hash function generator \mathcal{H} for which no probabilistic polynomial time algorithm can win the game from definition 11.1, there cannot exist an algorithm A that will invert a function generated by \mathcal{H} with non-negligible probability: if so, we could use A and lemma 11.2 to get an algorithm contradicting Definition 11.1. In other words, if collision-intractable hash functions exist, then one-way functions exist. But the converse implication is not known to be true, and also no connection in either direction is known between existence of collision-intractable hash functions and one-way (trapdoor) functions.

11.2.3 Domain extension for hash functions

When constructing collision intractable hash functions, it is always enough to build collision-intractable functions that map a fixed number of bits, say m bits to k bits, as long as $m > k$:

Theorem 11.3 *If there exists a collision-intractable hash function generator \mathcal{H}' producing functions with finite input length $m > k$, then there exists a collision-intractable generator \mathcal{H} that produces functions taking arbitrary length inputs.*

The construction behind the proof of the theorem is known as the Merkle-Damgård construction.

We will first consider the case where $m - k > 1$, and return later to the case of $m - k = 1$. Set $v = m - k - 1$ and note that $v > 0$. We now define the generator \mathcal{H} promised in the theorem. First use \mathcal{H}' to get a function $f : \{0, 1\}^m \mapsto \{0, 1\}^k$. Now define our new function $h : \{0, 1\}^* \mapsto \{0, 1\}^k$ as follows:

Split the input string x in v -bit blocks x_1, \dots, x_n , padding the last block with 0s if needed. Finally, add a block x_{n+1} containing in binary notation the number of 0s we used in padding x_n .

Now we define a series of m -bit blocks z_1, z_2, \dots, z_{n+1} as follows: We set

$$z_1 = 0^k || 1 || x_1,$$

where $||$ denotes concatenation of bit strings. And then, for $i = 2, \dots, n+1$ we set

$$z_i = f(z_{i-1}) || 0 || x_i.$$

Finally we define $h(x) = f(z_{n+1})$.

In other words, we define a “starting state” as being some 0s concatenated by a 1 followed by the first message block. Then we apply f , concatenate the result with a 0 followed by the next message block and apply f again. We continue this way until we have processed all message blocks. The single bit-field in the middle is used for distinguishing the starting state from all intermediate states. We will see shortly why this is important.

We show by contradiction that h is collision intractable, namely if one can find a collision for h , we can efficiently find one for f . So assume we are given $x \neq x'$ where $h(x) = h(x')$. Let z_1, \dots, z_{n+1} and $z'_1, \dots, z'_{n'+1}$ be the sequence of inputs to f defined as above when we process x and x' , respectively. Assume without loss of generality that $n \geq n'$. By assumption, we know that $f(z_{n+1}) = h(x) = h(x') = f(z'_{n'+1})$.

Now, we have two cases: either $z_{n+1} \neq z'_{n'+1}$, which immediately gives a collision for f and we are done. Or $z_{n+1} = z'_{n'+1}$. This implies that the number of 0's used for padding x_n , respectively $x'_{n'}$, are equal. But it also implies that $f(z_n) = f(z'_{n'})$. Now we have again two cases: either $z_n \neq z'_{n'}$, which implies a collision for f , or $z_n = z'_{n'}$. The latter case implies that $x_n = x'_{n'}$ (because we know the number of padding bits are the same), and that $f(x_{n-1}) = f(x'_{n'-1})$. It is now clear that we can repeat this argument, and we claim that this must lead to a collision for f . Namely, if it did not we would always have that the inputs to f are equal and hence we would eventually reach a stage where we could conclude $z_{n-n'+1} = z'_1$: At this point we would also have

$$x_{n+1} = x'_{n'+1}, x_n = x'_{n'}, \dots, x_{n-n'+1} = x'_1$$

But this cannot happen: if $n = n'$, then it would imply $x = x'$ which is false by assumption, and if $n > n'$, it cannot be the case that $z_{n-n'+1} = z'_1$ because then $z_{n-n'+1}$ has a 0 in position $k+1$ while z'_1 has a 1.

Finally, we sketch the case where $m - k = 1$. We will do a similar construction of new function \bar{h} from f . We call it \bar{h} because we will later modify it to get the actual hash function h . Now there is no room for an extra bit that marks the start of the hashing process. So we simply define $z_1 = 0^k || x_1$ where in this case x_1 is simply the first bit of x . And then, for $i = 2, \dots, n+1$ we set

$$z_i = f(z_{i-1}) || x_i.$$

Finally we define $\bar{h}(x) = f(z_{n+1})$. Now, suppose we try to argue in exactly the same way as before that if we have a collision for \bar{h} then we get one for f . It is easy to see that exactly the same argument works, with only one caveat: this time

the process where we work our way backwards through x and x' can actually fail to produce a collision, but the only case where it can fail is if $n > n'$ and

$$x_n = x'_{n'}, x_{n-1} = x'_{n'-1}, \dots, x_{n-n'+1} = x'_1,$$

in other words, if x' is a suffix of x . We can make sure that this case does not happen, if we first pass the input through a so-called suffix-free encoding $E()$, this is just an efficiently computable function from bit strings to bit strings with the property that there exist no pair x, x' such that $E(x')$ is a suffix of $E(x)$. And we then define $h(x) = \bar{h}(E(x))$. It is then clear that we are guaranteed to get a collision for f from a collision for h .

A suffix-free encoding is simple to construct. For instance, let $D(x)$ be the function that repeats every bit twice, that is $D(x_1, \dots, x_n) = x_1, x_1, x_2, x_2, \dots, x_n, x_n$. Then we set $E(x) = 0, 1 || D(x)$. Since now the pair 01 can only occur at the start of an encoding, the suffix-free property follows easily.

Exercise 11.2 (Simplified Merkle-Damgård construction) Assume we are given a function $f : \{0, 1\}^m \mapsto \{0, 1\}^k$, where $m > k$, exactly as in the proof of Theorem 11.3. However, in this exercise, we assume f has an extra property we call zero-preimage resistant, namely it is infeasible to find an input w such that $f(w) = 0^k$. Now define a new function $h : \{0, 1\}^* \mapsto \{0, 1\}^k$ as follows:

Set $v = m - k$. Split the input string x in v -bit blocks x_1, \dots, x_n , padding the last block with 0s if needed. Finally, add a block x_{n+1} containing in binary notation the number of 0s we used in padding x_n .

Now, define a series of m -bit blocks z_1, z_2, \dots, z_{n+1} as follows: We set

$$z_1 = 0^k || x_1,$$

where $||$ denotes concatenation of bit strings. And then, for $i = 2, \dots, n + 1$ we set

$$z_i = f(z_{i-1}) || x_i.$$

Finally we define $h(x) = f(z_{n+1})$. Show that given a collision for h , you can find either a collision for f or a preimage under f of 0^k . In other words, if f is collision resistant and zero-preimage resistant, then h is collision resistant.

11.2.4 Hash Functions in Practice

How large should we choose k in practice? There is one general lower bound that comes from an attack one could try against *any* hash function: simply choose a set of random inputs, evaluate the function on these and hope that we happen to run into a collision. If we choose the inputs at random of sufficiently large length, it is a reasonable assumption that the outputs would then be uniformly chosen in $\{0, 1\}^k$. So we want to know how many random values we must choose in $\{0, 1\}^k$ before there is a significant probability that at least two of the are the same. This is the well known “birthday paradox” question: if the number of values is the square root of the total number of values, then there is a constant probability of collision – as we go below the square root, the probability drops very quickly towards 0. In other words:

Theorem 11.4 *Given function h , and assume that it is possible to sample input values to h causing the outputs to be uniformly random in $\{0,1\}^k$. Then a collision for h can be found with constant probability in time corresponding to $2^{k/2}$ evaluations of h .*

With current state of the art, this implies that k should be at least 160, and some would prefer even larger values.

The constructions we have seen based on RSA and Discrete log are nice from a theoretical point of view, but not efficient enough in practice, in particular because we also want to use hash functions together with digital signature schemes, as we shall see later. There we will need hash functions that are much faster to evaluate than standard public-key like operations. Therefore people have put much effort into designing hash functions that would be efficient both in hard- and software implementations. Well known historic examples are MD5 (Message Digest nr. 5, 128 bits output) and SHA-1 (Secure Hash Algorithm, 160 bits output). MD5 was broken, however, in the fall of 2004, and was considered to be of shaky security status even before that, so the US standard SHA-1 was the most used for a while after that. Also for SHA-1, however, attacks have been found in the fall of 2005 that run somewhat faster than brute force. Fortunately, SHA-1 exists these days in 256 and 512 bit variants, and SHA-256 is a common choice at the time of writing.

All these functions were designed specially with a 32-bit architecture in mind and can process several Mbits pr. second on a decent machine. This is due to the fact that the designs use almost exclusively operations that are available as CPU instructions on most standard CPUs, such as bit-wise XOR, left/right shift and addition. They are also all based on the Merkle-Damgård paradigm, namely the word-wise instructions are used in order to design a hash function with fixed size input, which is then iterated in a way similar to what we saw in the proof of Theorem 11.3 to get a function that accepts any size of input.

In 2012 a new US standard called SHA-3 has been announced. Part of the background for developing a new standard has been that it was felt that previous constructions had been too much targeted against software architectures and that the designs were perhaps too aggressive w.r.t. speed. Another issue has been that there has been a growing need to use hash functions for many different purposes other than basic authentication. Some of these applications, for instance for pseudorandom bit generation, for construction of message authentication codes and as “random oracle” replacement in OAEP, require other properties than just collision resistance. The problem now is that the older hash functions were only designed with collision resistance in mind and hence this also creates a need for a new approach.

The new hash function is indeed based on a completely different design principle than the SHA and MD families. Practical applications are now gradually moving towards adapting SHA-3, also known as KECCAC.

The idea behind SHA-3 is known as the “sponge” design, where you imagine the function as a sponge that first absorbs whatever input it gets, and in the end

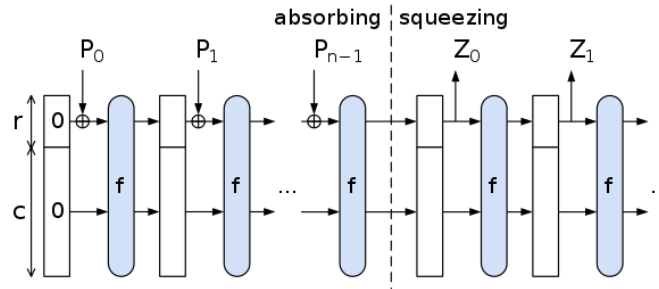


Figure 11.1 Design of SHA-3

you squeeze the sponge to get the output. The design can be seen in Figure 11.1, where the P_i 's are the input blocks, of size r bits, and the Z_i 's are the output. Thus the construction has variable length output. The value r is called the rate. Note that construction maintains a state consisting of c bits which are carried forward from the previous iteration. The standard uses $c + r = 1600$, and sets $r = 1088$ when the hash output is supposed to be 256 bits, so that then $c = 512$. In this case the output is just the leading 256 bits of the final state, and there is no need for further iterations.

The heart of SHA-3 is the function f which is a permutation and is a typical cryptographic function: highly non-linear and hard to invert. It is constructed to be fast in both soft and hardware and hence uses many word-wise operations that readily implementable on a 32 or 64 bit architecture. The basic bit-operations involved are XOR, NOT and AND, as well as rotations of the words involved so that the information in a state is well mixed during the computation.

It is an important design choice in SHA-3 that the state carried forward to the next round is much larger than the final output (here 512 versus 256 bits). This is done to prevent what is known as *length extension attacks*. To understand this, note that in older designs such as SHA-256, the output is simply the final state once all of the message has been processed. Writing H for the hash function in question, this allows an adversary who knows $H(m)$, but not m , to compute $H(m||m')$ for an m' of his choice by simply continuing the hashing process. However, if H is instead SHA-3, then such an attack is not possible.

In a basic application for authentication where only collision intractability is required, length extension is not an issue. However, as mentioned above, new applications of hash functions have emerged, requiring other properties. In particular, for applications that construct message authentication codes from hash functions, length extension attacks can be relevant.

With all this talk about concrete and fixed hash functions, the reader might complain that this does not really fit our definition: SHA-3, for instance, is just one given function, and not a generator we can run to make a function with any desired output size. One should remember, however, that a function like SHA-3 does not just come falling from the sky, it is the result of a long design process.

In fact, it makes sense to think of this design process as being the generator and SHA-3 as the function produced as output from the generator.

11.2.5 The random oracle model

Suppose I give you hash function h and input/output values m , $h(m)$. Suppose m' is equal to m , but with 1 bit flipped. Can we say anything about what $h(m')$ will look like? for instance, will it resemble $h(m)$ in any obvious way? if h is secure, and so is a complex and hard to invert function, the answer seems to be that we cannot say very much in general. We can evaluate h on m' , and the value coming out will probably “seem like” a random value with no obvious relation to $h(m)$.

In the Random Oracle Model, we try to formalize this intuitive idea that the output values “might as well” have been random. The idea is that we take whatever application of the hash function we have in mind and replace the hash function by a *random oracle*. A random oracle is an oracle that will receive any bit string m as input and will return $R(m)$, where $R(m)$ is a randomly chosen k -bit string. If it later receives m as input, the same string $R(m)$ will be returned. But every time a new string m' is given as input, a fresh random string $R(m')$ is returned, chosen independently from anything else. In the random oracle model, we assume that *everyone*, including the adversary, has access to such an oracle, and it is the *same* oracle for everyone, i.e., whoever sends input m will receive back the same string $R(m)$.

Of course, no such oracle could be available in a real-life scenario. In real life, we have a concrete, given function h which is of course not random. What we can hope for (but not prove) is that an adversary would not be able to do better with the concrete hash function than he could with a random oracle. Now, in the random oracle model, any adversary would have to only use attacks that exploit no particular properties of the hash function. Consider for instance the application where we store m in an insecure location and later check it against $h(m)$. In the random oracle model this becomes a random value $R(m)$, and the adversary still has to find a collision to attack the system. Thus all he can do is to send a set of inputs to the oracle and hope that some of the answers happen to be the same. In other words, he has to use the generic attack from Theorem 11.4.

Summarizing, if an application of a hash function has been proved secure in the random oracle model, what this means for the real life application is that any attack that considers the hash function h as a “random black-box” and does not use any special properties of h , is doomed to failure. But there is no guarantee for attacks directed specifically against h . Of course, there would be no reason to even consider this model, if we were able to go and prove our constructions secure in the standard way. But unfortunately, there are many constructions that are important in practice, where no security proof is known, but where we are able to say something in the random oracle model. This is certainly not what we would like best, but it is much better than having no proof at all. For instance,

the theorem claiming that OAEP combined with RSA gives CCA security, we only know how to prove in the random oracle model.

11.3 Definition of Message Authentication Codes (MACs)

A *secret-key authentication system* consists of three probabilistic algorithms (G, A, V) . G which outputs a key K , usually it works by simply choosing K as a random bit string of a certain length. Algorithm A gets input a message m and the key K and produces an authenticator value $s = A_K(m)$. Finally algorithm V gets as input an authenticator s , a message m and key K , and outputs $V_K(s, m)$ which is equal to *accept* or *reject*. It is required that we always have $V_K(A_K(m), m) = \text{accept}$.

A secret-key authentication system is also called a MAC scheme, where MAC stands for message authentication codes (MACs). MAC is also the standard name for the authenticator value s .

11.3.1 Security of Secret-Key Systems (MAC's)

To define the security of a secret-key authentication system, we use a similar approach as for cryptosystems: Given some system (G, A, V) , the adversary E gets access to an oracle, which initially runs G to get a key K . We now play the following game: E may, as many times as it wants, send some message m to the oracle, who will return to E the MAC $A_K(m)$. We say that E gets to do a chosen message attack (CMA). At the end of the game, E outputs a message m_0 and an authenticator s_0 . E wins the game m_0 is not one of the messages the oracle was asked to authenticate, and $V_K(s_0, m_0) = \text{accept}$.

Note that the adversary is not able to check himself whether a given string is a valid MAC on a given message. But if the MAC scheme is deterministic, i.e., every message has only one valid MAC, then adversary can do such a check by asking for the MAC on the given message and comparing to the candidate value. Thus, the CMA game we described is without loss of generality for deterministic schemes, and we will only look at such MAC schemes in the following.

Definition 11.5 (CMA Security of MACs) We say that a MAC scheme is (t, q, ϵ, μ) CMA-secure if any adversary that runs in time at most t and asks at most q queries, on messages of total length μ bits, wins the above game with probability at most ϵ .

Some remarks: Some schemes have to put restrictions on the maximal length of messages in order to guarantee security, or one needs restrictions on the content. We will mention such restrictions explicitly later when we talk about schemes where they are needed. As usual, the definition is constructed to give the adversary as much power as it is meaningful to give him. The idea is that in a real application, the adversary will be worse off than he is in the game, so if we can guarantee security in the game, then we should have security in any application as well.

11.4 Existence of good MAC Schemes

11.4.1 MACs from block ciphers

One of the the most well-known scheme for MAC's is called CBC-MAC and is based on blockciphers: given such a system, say DES or AES, one simply encrypts the input message in CBC mode (using $IV = 0$ always) and define the MAC to be the final block of the ciphertext. This has the added advantage that the MAC has fixed length, no matter how long the message is.

Suppose the cipher we start from is a good deterministic cryptosystem, i.e., it forms a good family of pseudorandom functions (PRFs). Then the resulting MAC we get this way is also good to a certain extent, as was shown in [8]. However, the problem is that we can only have security if we assume that no message we ever authenticate will be a prefix of another message.

This is of course a rather unpleasant restriction to deal with. It is possible to ensure this, if one writes down the length of the message in a block put this in the beginning of the string we authenticate, we say we prepend the length. But this is often not a practical approach: if the message we want to handle arrives as a stream, we may not initially know how long it is going to be.

By contrast, appending the length, i.e., putting the length at the end is practical, but not secure. For instance suppose we take the number of blocks contained in the message, construct a block containing this number in binary¹, *append* this block to the end of the message, and then finally use CBC-MAC on the resulting extended message. Indeed, this is not secure, as we now explain:

We assume that all messages have length divisible by the block length (so we show that appending the message length is insecure even under this extra assumption) Take any three blocks b, b', c with $b \neq b'$. And ask the oracle for MAC's on messages b, b' and $b||1||c$, where $||$ means concatenation and 1 is the block containing the number 1 in binary. Note that the three messages are specified as they look before the extra block with the length is appended. So to compute the MAC on the first message with just one block b , one would compute CBC-MAC on the two-block message $b||1$. Say the three queries to the oracle returns MACs $t_b, t_{b'}, t_{b1c}$, respectively. One can then verify that t_{b1c} is also a valid MAC for the message $b'||1||t_b \oplus t_{b'} \oplus c$.

Exercise 11.3 (MAC Attack) Show that indeed, t_{b1c} is a valid MAC for the message $b'||1||t_b \oplus t_{b'} \oplus c$.

Fortunately, there is a simple solution to the problem we just saw, namely we define a new MAC scheme using 2 keys K_1, K_2 . We compute the CBC-MAC on the message using K_1 and then we encrypt this MAC under K_2 and output the result of this. This method is known as EMAC or ECBC and is found in the US NIST standards, among others. A bit more formally, we define

$$EMAC_{K_1, K_2}(m) = E_{K_2}(CBC - MAC_{K_1}(m))$$

¹ we assume some fixed way of converting a binary number to a block of bits, whether this is done in a big- or a small-endian way is irrelevant here

Even this definition is not quite complete: we have only defined CBC-MAC on messages whose lengths are divisible by the block length k of the underlying block cipher. In practice, we would like a MAC that works on messages of any length. To handle this, one may first use a padding scheme to make sure that all messages have length a multiple of k bits after padding. For instance, always append a 1, and then append enough 0's to make the string length a multiple of k . Note that this padding has the property that we can always reconstruct the original message from the padded version. Any padding rule with this property will do. The reason this is important is that the receiver must be able to distinguish the bits added for padding from the actual message bits. Otherwise, the same MAC would be valid for several different messages and the scheme would not be secure.

For EMAC, we can indeed show that if the underlying block cipher is a good PRF, then EMAC is CMA secure:

Theorem 11.6 *Suppose the block cipher (G, E, D) is a (t', q', ϵ') -secure PRF and has block length k . Then EMAC based on this block-cipher is a (t, q, ϵ, μ) CMA-secure MAC scheme, where*

$$t \leq t', \quad \mu/k \leq q', \quad \epsilon = 2\epsilon' + \frac{2(\mu/k)^2 + 1}{2^k}.$$

The main conclusion from this result is similar to the one for modes of operation in encryption, namely that as long as we do not use the system for too large amounts of data before changing the key, then EMAC inherits most of the strength of the original cipher.

The proof of this result can be found in [9] and is rather complicated. Thus we give only a sketch here: the overall plan is very similar to what we saw for the encryption modes: first we define a hybrid game where we replace the applications of $E_{K_1}()$ and $E_{K_2}()$ by calls to two different random functions $R()$ and $S()$. This can be distinguished by the adversary with advantage at most $2\epsilon'$, where the factor 2 is because we switch 2 functions. Then an information theoretic result is shown, namely that EMAC based on 2 random functions can be distinguished from a totally random function (from bit strings to $\{0, 1\}^k$) with advantage at most $\frac{2(\mu/k)^2}{2^k}$. This is where the proof gets complicated.

Finally, observe that a totally random function is a secure MAC where the adversary clearly can win with probability at most $1/2^k$: the adversary must output a new message, but since it is new, the random function has not been queried on input that message. So no matter what the adversary submits as his guess at the right function value, it will be correct with probability 1 divided by the number of possible values, namely 2^{-k} .

We are now now done, since the total advantage of the adversary can be at most $2\epsilon' + \frac{2(\mu/k)^2}{2^k} + \frac{1}{2^k}$. This follows by an application of the triangle inequality, just like what we did in the proof for security of CBC mode.

11.4.2 MACs from hash functions

Another construction of MAC-schemes is known as HMAC and can be based on any collision intractable hash function. We describe it here in the concrete standardized version that is based on SHA-1: the key is a random 512-bit string K . The scheme uses two 512 bit constants $ipad = 3636...36$, $opad = 5C5C...5C$, in HEX notation. Then we define

$$HMAC_K(m) = SHA1((K \oplus opad) || SHA1((K \oplus ipad)||m))$$

where $||$ means concatenation of bit strings. This scheme generalizes naturally to be based on any hash function instead of SHA-1. It can be proved secure when the hash function is modelled as a random oracle, and no better result is known when we want to base HMAC on an *arbitrary* hash function. If security on the random oracle model was all we wanted, then a much simpler scheme where one just appends the key to the message before hashing, would be sufficient. However, in many cases the hash function is built by iterating a compression function with fixed size input, like SHA-1. Then the extra complexity of applying the key twice, xor'ed with different strings, allows to prove a result "in the real world". Namely, HMAC is secure if the hash function is collision resistant *and* if the basic compression function is secure in a weak sense when used as a MAC : part of the input to the function is used as key, the rest as message, and the adversary must forge MAC's even when the messages involved are partly unknown to him. No proof assuming only collision-resistance is known. Nevertheless, this is a popular scheme, due to its efficiency and because it does not rely on cryptosystems, which are sometimes under export restrictions.

Exercise 11.4 (MAC Constructions) Let E_K be the encryption function of a block cipher with block length ℓ , and define a MAC scheme that takes as input messages that can be written as sequence of ℓ -bit blocks. On input a message x_1, \dots, x_n , the MAC is $E_K(x_1) \oplus \dots \oplus E_K(x_n)$. Consider the definition of CMA security and show how to break this MAC scheme with probability 1 using no queries to the oracle.

This shows that one can forge the MAC on *some* message (that was not sent to the oracle). Suppose now that you are given a fixed message y_1, \dots, y_n and the goal is to forge a MAC on exactly this message. For $n > 1$ and any y_1, \dots, y_n , show how to do this with probability 1 and 1 query. Note that you have to handle the case where $y_1 = \dots = y_n$.

Finally a not quite so trivial construction: suppose we change the MAC to be defined as follows: $E_{K_1}(x_1) \oplus \dots \oplus E_{K_n}(x_n)$, where the keys K_1, \dots, K_n are random and independent. Show that you can still forge a MAC on some message with probability 1 and 3 queries.

Digital Signatures

Contents

12.1	Definition of Digital Signature Schemes	138
12.1.1	Security of Signature Schemes	139
12.2	Signature schemes based on RSA	139
12.3	Signature schemes based on discrete logarithms	141
12.3.1	The Schnorr signature scheme	141
12.3.2	The ECDSA signature scheme	144
12.4	Combining Signatures and Hashing in general	145
12.5	Existence of good Signature Schemes	147
12.6	Dealing with replay attacks	149

12.1 Definition of Digital Signature Schemes

An *Digital Signature System* (G, S, V) is defined, first by a probabilistic key generation algorithm G which gets a security parameter k as input and produces as output a pair of keys (pk, sk) , the public and the secret key. Algorithm A gets input a message m and the secret key sk and produces a signature $S_{sk}(m)$. Finally algorithm V gets as input a signature s , a message m and public key pk , and outputs $V_{pk}(s, m)$ which is equal to *accept* or *reject*. It is required that we always have $V_{pk}(A_{sk}(m), m) = \text{accept}$.

As usual, the security parameter can be thought of as a measures of the amount of security we are after: the larger k is, the more secure the keys are (hopefully). In addition, all algorithms should be efficient (polynomial time in k).

An example would be plain RSA signatures, where the key generation is the same as in the RSA cryptosystem, so $sk = (n, d)$, $pk = (n, e)$ and $\mathcal{P} = \mathcal{A} = \mathbb{Z}_n$. Here, k is usually the bit length of the modulus n . The signature on message m is $s = m^d \bmod n$, and to verify s , one checks that $s^e \bmod n = m$.

Note a very important difference between MAC's and digital signatures: if use MAC's, then since both parties know the secret key, there is no way a third party can later tell if a given authenticated piece of data was created by the sender or the receiver. Thus, such a system cannot be used to resolve, for instance, a conflict between a bank and a bank customer over the authenticity of a payment order.

A digital signature system can do this, provided the implementation ensures that only the user could have accessed his own secret key.

12.1.1 Security of Signature Schemes

Just as for MAC schemes, security of a signature scheme is defined by an oracle game:

The oracle runs G on input k to get sk, pk . We give pk to the adversary E , and keep sk inside the oracle. The adversary can submit as many messages as he wants and for each message m he will get $S_{sk}(m)$ back.

The adversary wins the game if he produces m_0, s_0 , such that m_0 is not one of the messages the oracle was asked to sign, and that $V_{pk}(s_0, m_0) = \text{accept}$. The probability that E wins is a function of the security parameter k and is called $Adv_E(k)$.

Definition 12.1 (Security for Signature Schemes) We say that a signature scheme is CMA-secure if for any probabilistic polynomial time adversary E , $Adv_E(k)$ is negligible as a function of k . This is the strongest sense in which a signature system can be secure.

12.2 Signature schemes based on RSA

We already defined the basic RSA signature scheme above. Namely, $sk = (n, d), pk = (n, e)$ and $\mathcal{P} = \mathcal{A} = \mathbb{Z}_n$. The signature on message m is $s = m^d \bmod n$, and to verify s , one checks that $s^e \bmod n = m$.

It is easy to see that this scheme is not CMA secure: given only the public key, the adversary simply chooses any value $s \in \mathbb{Z}_n^*$, and computes $m = s^e \bmod n$. He now outputs the pair (m, s) , which is clearly a successful forgery. Of course, this may not be a useful attack in an application since the adversary most likely will have a hard time controlling the value of m . It is, after all, the output of a complicated one-way function. Nevertheless, this breaks CMA security and besides one has to be careful in dismissing an attack as being useless in practice: granted, it seems reasonable to claim that if you obtain m as $m = s^e \bmod n$ for an s you chose, then m most likely will not be a message that makes sense. But what “makes sense” depends on the context! For instance, if we have a system that makes measurements on, say, seismic activity, messages may simply be numbers, and then it is much less clear that a random number in \mathbb{Z}_n^* cannot be a valid message.

Another problem with the basic RSA scheme is that it is very inefficient. If we had to process the entire message using RSA this would hardly work well in practice.

The solution to both problems is to combine RSA with a hash function. The idea is simply to hash a message before you sign it. By collision intractability, signing the hash value is as convincing as signing the message.

More formally, we assume we have a hash function generator \mathcal{H} , and then the combined scheme works as follows:

Key generation Run standard RSA key generation on input k to get $((n, e), (n, d))$.
 Run $\mathcal{H}(k)$ to get a hash function h . We will assume that we can generate h such that $\text{Im}(h) = \mathbb{Z}_n$ - thus this scheme is known as “Full Domain Hash”.
 Output $(pk, sk) = ((n, e, h), (n, d, h))$. The message space is $\{0, 1\}^*$ (any bit string can be signed).

Signing The signature on message m is $S_{sk}(m) = (h(m))^d \bmod n$.

Verification Given message and signature (m, s) , check that $s^e \bmod n = h(m)$.

It is trivial to see that the scheme always accepts a correct signature. As for security, we can first observe that the attack from before no longer works: the adversary can of course still choose some s and compute $s^e \bmod n$. But now he has to interpret s as a possible output from the hash function and must find m such that $h(m) = s$. This is hard: if h is collision intractable it is also one-way, as we saw above.

The best result we know how to show for this scheme is the following:

Theorem 12.2 (Security of Full Domain Hash) *If we model the hash function used in Full Domain Hash as a random oracle, then under the RSA assumption, Full Domain Hash is CMA secure.*

The intuition why the theorem holds is as follows: the RSA assumption says that no efficient adversary can find x from $x^e \bmod n$ when x is random in \mathbb{Z}_n . But then $x^e \bmod n$ is random as well, so we can rephrase the assumption as: given random $y \in \mathbb{Z}_n$ it is hard to find x such that $x^e \bmod n = y$. But when the hash function is modelled as a random oracle it follows that no matter what message m the adversary tries to forge a signature on, $h(m)$ is random, so the adversary needs to solve exactly the problem that the RSA assumption says is hard.

Actually proving this the theorem - which we will not do here - requires that you build a concrete reduction that uses a CMA adversary to build an algorithm that breaks the RSA assumption. This is not trivial, in particular not if we want an efficient reduction. For the proof, see [13].

Full Domain Hash is not often used completely as is in practice, since it is not immediately clear how we get a hash function whose image is exactly \mathbb{Z}_n (although SHA-3 comes close because it has flexible output length). Instead, a padding scheme is used, that will map a hash value into \mathbb{Z}_n , thus correcting for the issue that the hash value often is much shorter than n . Concrete methods for this are specified in several international standards, for instance the ISO 9796, or the PKCS series. The result $\text{pad}(h(m))$ is then signed using RSA.

We mention that it is in fact possible to construct a signature scheme that is CMA secure based on the so called strong RSA assumption, without using the random oracle model, see [14]. The strong RSA assumption says that given a random y it is hard to compute e, x such that $x^e \bmod n = y$. In other words RSA is hard to invert even if you can choose the public exponent yourself. The

best known attack against this assumption is the same as for the standard RSA assumption, namely to factor the modulus.

12.3 Signature schemes based on discrete logarithms

Historically, the first signature scheme based on discrete logarithms was proposed by El Gamal. However, nowadays other schemes are much more widely used and supported, so we will not cover the details of his scheme here. However, it is fair to say that the later schemes build on his ideas, but they have various advantages in terms of efficiency and what we can prove about them.

12.3.1 The Schnorr signature scheme

The Schnorr signature scheme is based on a subgroup of \mathbb{Z}_p^* . More concretely, we assume we have primes p, q such that $q|p-1$, and we also have $\alpha \in \mathbb{Z}_p^*$ with $|\alpha| = q$. We can get such an element by setting $\alpha = \alpha_0^{(p-1)/q}$ where α_0 is a generator of \mathbb{Z}_p^* .

The idea behind this set-up is that because q is much smaller than p , we can get smaller signatures than one would expect from a scheme based on arithmetic mod p – since a signature turns out to be a pair of numbers in \mathbb{Z}_q .

The key generation of the scheme is very simple: we choose primes p, q such that $q|p-1$. Finally, we choose $\alpha \in \mathbb{Z}_p^*$ of order q and $a \in \mathbb{Z}_q$ at random. Then the public key is $p, q, \alpha, \beta = \alpha^a \bmod p$ and the secret key is a .

An Interactive game

To understand how Schnorr signatures work, we will start by looking at an interactive game that we can imagine the *signer* (the owner of the secret key) plays in order to convince someone else – the *verifier* – that he knows the secret key a – but without telling the verifier what the secret key is. This game is an example of a *zero-knowledge proof of knowledge*, a very interesting notion that we will not cover here, however. In this context, one can think of the game as a mental experiment, from which we will derive the actual signature scheme.

The game takes the public key p, q, α, β as input and goes as follows:

1. The signer sends $c = \alpha^r$ to the verifier, where r is chosen at random in \mathbb{Z}_q .
2. The verifier sends a random value $e \in \mathbb{Z}_q$ to the signer.
3. The signer sends $z = (r + ea) \bmod q$ to the verifier.
4. The verifier checks that $\alpha^z = c\beta^e \bmod p$. He concludes that he believes the signer knows a , if this check is satisfied, we say the verifier accepts.

If the signer executes this protocol correctly, the verifier will always be happy. We see this by just plugging into the expression the verifier checks:

$$c\beta^e \bmod p = \alpha^r (\alpha^a)^e \bmod p = \alpha^{r+ea} \bmod p = \alpha^z \bmod p$$

Loosely speaking, the reason why a is not revealed to the verifier is that what he

sees is only $(r + ea) \bmod q$, and here the part that depends on a is masked by adding the random value r .

Finally, we claim that if the signer did not actually know a , he could only make the verifier accept with negligible probability $1/q$. We do this by showing that if the signer can send c and then answer correctly to more than 1 of the q possible values of e , then he could in fact easily compute a , and so he knows it after all. So, assume the signer sends c and can answer correctly both e and $e' \neq e$. In other words, he can compute z and z' such that $\alpha^z = c\beta^e \bmod p$ and $\alpha^{z'} = c\beta^{e'} \bmod p$. By dividing one equation by the other, we get

$$\alpha^z - \alpha^{z'} = \beta^{e-e'} \bmod p$$

Since $e - e' \neq 0$ and q , the order of α , is prime the inverse $(e - e')^{-1} \bmod q$ exists. Raising both sides to this power, we get

$$\beta = \alpha^{(z-z')(e-e')^{-1} \bmod q} \bmod p.$$

We conclude from this that we get the secret key as $a = (z - z')(e - e')^{-1} \bmod q$.

From interactive game to signature scheme

We will now turn the interactive game into a signature scheme. The first step is to use a hash function h which we assume maps to \mathbb{Z}_q , and note that the signer can use h to execute the interactive game by himself without having to involve the verifier. This is done by computing e as $h(c)$ instead of having the verifier choose e . Note that if we model the hash function as a random oracle, then this is essentially the same as playing the interactive game, since e is random in both cases.

We do a twist in order to link this to a message m by hashing m as well as c , so we set $e = h(c, m)$. The prover can do all this by himself and then send the “conversation” (c, e, z) and m to the verifier. The verifier will do the check from the interactive game and if it is OK, he can take this as evidence that m must come from the signer: Values (c, e, z) leading to accept could only feasibly be created by the signer who is only party who knows a . Moreover, e is linked to m by the hash function: it is infeasible to change m and still make the hash function output e .

As a final touch, we can optimize the data sent to the verifier to make it smaller: it is enough to send (e, z) , since the verifier knows that the equation $\alpha^z = c\beta^e \bmod p$ should hold, so he can compute c himself as $c = \alpha^z \beta^{-e} \bmod p$ and then check that $e = h(c, m)$.

We summarize the scheme as follows:

Key generation We assume an algorithm \mathcal{K} that on input k will output a prime p , a k -prime q so that $q|(p-1)$ and $\alpha \in \mathbb{Z}_p^*$ of order q . Run hash function generator $\mathcal{H}(k)$ to get hash function h that maps to \mathbb{Z}_q . Choose $a \in \mathbb{Z}_q$ at random.

Output $pk = (h, p, q, \alpha, \beta = \alpha^a \bmod p)$ and $sk = a$.

Signing On input message m and pk, sk , choose $r \in \mathbb{Z}_q$ at random, set $c = \alpha^r \bmod p$ and output the signature $(e, z) = (h(c, m), (r + ea) \bmod q)$.

Verification On input message m , signature (e, z) and pk , set $c = \alpha^z \beta^{-e} \bmod p$ and check that $e = h(c, m)$, accept if this is the case, otherwise reject.

The algorithm \mathcal{K} will choose the prime q first, and then repeat choosing s at random until $p = 2sq + 1$ is prime. Here s needs to be large enough so that p has the right size, which we discuss in a moment. Finally choose α_0 to be a generator of \mathbb{Z}_p^* and set $\alpha = \alpha_0^{(p-1)/q} \bmod p$.

As shown in the lemma we shall see in a moment, security of Schnorr signatures follows from the difficulty of finding discrete logarithms base α . We therefore have to choose p, q large enough that this problem is indeed hard. We must consider the Index calculus algorithm since this is the best known for the DL problem in \mathbb{Z}_p^* . An important point is that even if α has order much smaller than p , Index calculus will not be able to exploit this: its running time depends only on the size of p , not on the order of α . On the other hand, one cannot make the order of α too small since then one could apply the generic algorithm.

This means that, with current state of the art, one would choose p to have at least around 2000 bits, while q can be about 300 bits. This will provide at least 128 bits of security, that is, (loosely speaking) the best attack requires at least 2^{128} iterations of a non-trivial computation.

It can be shown that

Lemma 12.3 *If we model h as a random oracle, then given a probabilistic polynomial time algorithm that breaks CMA security of the Schnorr signature scheme, we can build a probabilistic polynomial time algorithm that computes a from $\alpha^a \bmod p$.*

The proof of this is long and technical and will not be covered here. The intuition is that seeing genuine signatures from the signer does not help adversary to get a : each signature corresponds to an execution of the interactive game which does not reveal anything about a . On the other hand, to forge a signature on his own, the adversary would need to do the interactive game successfully in the role of the signer, and this is not feasible unless you can compute a .

Exercise 12.1 ((Mis)use of randomness in Schnorr signatures) It is absolutely essential that the random value r in the signature scheme is chosen independently for each signature:

Show that given Schnorr signatures $(e_1, z_1), (e_2, z_2)$ on messages m_1, m_2 , one can efficiently decide whether the same r has been used for both signatures (given only the public key).

Show that if the same r has been used for two signatures as above, one can efficiently compute the secret key a .

Assume someone tries to avoid the problem by choosing the r -values as follows: for the first signature choose a random r_0 , then for each new signature add a constant value u , so that the r -value for the i 'th signature is $r_0 + (i - 1)u \bmod q$.

Show that given signatures on the i 'th and j 'th message, as well as i, j and u , one can still compute a efficiently.

12.3.2 The ECDSA signature scheme

This signature scheme is a US standard that is based on elliptic curves and is used, for instance, in the well known BitCoin system.

It has many similarities with Schnorr signatures but with some important variations.

The scheme is based on an elliptic curve $E_{p,a,b}$ as described previously. The curve is constructed such that the order is a prime q , and furthermore, by Hasse's Theorem 9.11, p and q are extremely close to each other (we will see later why this is important). Let α be a generator of the group.

To describe the scheme, we will use standard multiplicative notation for the group operation (rather than the additive one traditionally used for curves). We will also adapt the notation ones finds in standard texts to make it correspond better to the one we used for Schnorr signatures.

Key generation Choose an elliptic curve group $E_{p,a,b}$ as specified above, of order a k -bit prime q . Run hash function generator $\mathcal{H}(k)$ to get hash function h that maps to \mathbb{Z}_q . Choose $a \in \mathbb{Z}_q$ at random.

Output $pk = (h, E_{p,a,b}, \beta = \alpha^a)$ and $sk = a$.

Signing On input message m and pk, sk , choose $r \in \mathbb{Z}_q$ at random.

Set $c = \alpha^r \bmod p$.

Set z to be the leftmost k bits of $h(m)$.

Note that in this case, group elements are points on the curve, so actually $c = (c_x, c_y) \in \mathbb{Z}_p \times \mathbb{Z}_p$.

Set $s = r^{-1}(z + c_x \cdot a) \bmod q$.

Output the signature (c_x, s) .

Verification On input message m , signature (c_x, s) and pk , compute the point

$$d = \alpha^{zs^{-1} \bmod q} \beta^{c_x s^{-1} \bmod q},$$

where x is as in the signing procedure, let d_x be its x -coordinate and check that $d_x = c_x$, accept if this is the case, otherwise reject.

Note that c_x is really a number modulo p , but after extracting it from c , we use it in computations modulo q , so strictly speaking, we should explicitly reduce it modulo q before proceeding. However, since p and q are so close, such a reduction changes nothing, except with negligible probability.

We can see that ECDSA, like Schnorr signatures, is also based on the verifier reconstructing from signature and message a group element that the signer had in mind. To see that correct signatures are always accepted, note that since $\beta = \alpha^a$, the element d computed during verification is:

$$d = \alpha^{zs^{-1} \bmod q} \beta^{c_x s^{-1} \bmod q} = \alpha^{(z + ac_x)s^{-1} \bmod q}$$

We can now plug in what s is and get that the exponent of α is

$$(z + ac_x)s^{-1} \bmod q = (z + ac_x)(r^{-1}(z + c_x a))^{-1} = r,$$

hence $d = \alpha^r = c$, and so of course $d_x = c_x$.

No security proof is known for ECDSA. However, the intuition is that the factor in s that depends on a is masked by the random value r . So we may hope that seeing correct signatures do not help to get a . And then the adversary needs to essentially start from only the public key to forge a message and for this we do not know any method other than using the secret key.

Exercise 12.2 ((Mis)use of randomness in ECDSA) Also here, it is essential that the random value r in the signature scheme is chosen independently for each signature:

Show that given ECDSA signatures $(c_x, z_1), (c_x, z_2)$ on messages m_1, m_2 , where the same r has been used for both signatures, one can efficiently compute the secret key a .

Derandomizing Schnorr and ECDSA

The following remark applies to both Schnorr and ECDSA signatures: Even if the implementer of the scheme understands the need to generate r randomly, it may not always be easy to get hold of random bits in reliable way - although modern Intel processors solve a lot of the problem by having hardware based random number generators onboard. One workaround that is often proposed is to generate r deterministically from a and m . This can be done, for instance, by using a as the key in a pseudorandom function and apply this function to m . Since a is unknown to the adversary (if not we are dead anyway), this can reasonably be assumed to be indistinguishable from random to the adversary, and hence the scheme should be as secure as before.

12.4 Combining Signatures and Hashing in general

We have seen above that combining RSA or discrete log based schemes with a hash function can upgrade the security of an otherwise insecure scheme - or so we hope. We only know how to actually prove this in the random oracle model.

The best general result we can show without random oracles is from [10], and gives in some sense the next-best thing: if both hash function and signature scheme *were already secure*, then the combination of them is also secure. Given signature scheme Σ and hash function generator \mathcal{H} , we can define a new scheme Σ' : To generate keys, we first make keys for Σ by running G , then we run \mathcal{H} to get h where we assume we can make h such that it maps into the plaintext space defined by Σ . h is included in the public key of Σ' . You can sign any bit string m in the combined scheme Σ' , by first computing $h(m)$, and then the signature $A_{sk}(h(m))$. It should be obvious how to verify a signature in the new scheme.

We now have

Theorem 12.4 *If \mathcal{H} is collision intractable and Σ is secure, then Σ' is secure.*

To prove the theorem, we assume that there is an enemy E' that breaks Σ' under a chosen message attack.

Consider the following algorithm E : it gets as input a hash function h , a public key pk from Σ , and an oracle O that on input a message x returns the Σ -signature $A_{sk}(x)$, where sk is the secret key corresponding to pk , i.e. E gets to do a chosen message attack on Σ .

Now E does the following:

- It starts E' on input pk, h (this is a public key in the combined scheme Σ').
- When E' makes an oracle call, i.e. it wants the Σ' signature on a message m , this is handled by computing $h(m)$, calling O on input $h(m)$ and returning to E' the result $A_{sk}(h(m))$.
- When E' outputs a message m_0 and a signature s_0 , E also outputs these values, plus the set of messages M for which E' made oracle calls.

The fact that E' by assumption is successful with large (non-negligible) probability means that there is a good chance that $m_0 \notin M$, and that s_0 is a valid Σ -signature on $h(m_0)$. Given that this happens, either $h(m_0) \in h(M_0)$ or $h(m_0) \notin h(M_0)$ so at least one of these cases must occur with non-negligible probability.

1. If $h(m_0) \notin h(M_0)$, we have forged the Σ -signature on a new message $h(m_0)$: we never asked O for a signature on $h(m_0)$.
2. If $h(m_0) \in h(M_0)$, we have found a collision for h .

It follows that the assumption that E' break Σ' leads to a contradiction with at least one of the assumptions in the theorem: if case 1 occurs with large probability, we can take a public key P as input, choose h ourselves and run E on h, pk to break Σ under a chosen message attack, i.e. if we are given an oracle O . If case 2 occurs with large probability, we can take a hash function h as input, run G ourselves to get sk, pk , and run E on input h, pk to find a collision for h (note that in this case, we chose sk ourselves, so the oracle O required by E is trivial to implement).

Note that it is essential for the proof of the result that h and (sk, pk) are chosen independently when we generate keys for the combined scheme. One can make artificial examples where h depends on sk, pk such that even though h and sk, pk are secure by themselves, the combination is not.

The results says that we can combine a signature scheme and a hash function that are already maximally secure, without losing security. But it does not apply to signature schemes with less security. For instance, plain RSA is not secure against a chosen message attack because of the multiplicative property. Thus, combining it with a strong hash function may certainly result in a secure scheme, but we cannot use the above theorem to establish this fact.

12.5 Existence of good Signature Schemes

It is easy to see that signature systems cannot exist unless one-way functions exist: the generator algorithm G can be seen as a mapping from the random choices it makes to the public key pk . If this mapping is not one-way, an adversary could reconstruct a good set of random choices, and run G himself using these choices (and the right value of pk , which is public). This will lead the right public key pk being generated, but will also produce the matching secret key, and so the signature scheme could not be secure.

On the other hand, it can be shown [11] that secure signatures can be constructed from any one-way function, and so we have

Theorem 12.5 *One-way functions exist iff secure signature schemes exist.*

The proof of this is very long and complicated and will not be given here. It is interesting to note, however, that apparently this puts public key signature schemes on a different status than public key *encryption* schemes: those schemes seem to require something stronger, either trapdoor functions such as RSA or one-way functions with other extra properties, such discrete log based functions.

To give a flavor of how one can prove the general result, we look briefly at a weaker result which says that collision resistant hash functions are sufficient. We begin by a scheme that may look rather silly at first sight, but nevertheless is the main building block. This is known as the Lamport-Diffie (LD) one-time signature scheme:

To generate keys, we run \mathcal{H} on input k to get hash function f (we use the notation f in order to reserve h for another function we need later). Then we choose t pairs of inputs

$$(x_0^1, x_1^1), (x_0^2, x_1^2), \dots, (x_0^t, x_1^t)$$

at random, say, from $\{0, 1\}^{k+1}$. The public key is now

$$f, (y_0^1 = f(x_0^1), y_1^1 = f(x_1^1)), \dots, (y_0^t = f(x_0^t), y_1^t = f(x_1^t)),$$

while the secret key is the pairs of x -values. The message space is $\{0, 1\}^t$. The signature on a bit string b_1, \dots, b_t is $x_{b_1}^1, \dots, x_{b_t}^t$, so signature verification is obvious: one just evaluates f on the x -values in the signature and verifies if the result matches the corresponding y -values in the public key.

This scheme will be secure, if it is used to sign at most ONE message. Already if we sign 2 messages with the same public key, we may be in trouble:

Exercise 12.3 (LD Signatures are 1-time only) Assume the LD scheme has been used to sign 2 different messages $b_1, \dots, b_t, b'_1, \dots, b'_t$. Let ℓ be the number of indices i for which $b_i \neq b'_i$. Show that given the two signatures, an adversary can now easily sign $2^\ell - 2$ new messages.

To prove security when only one message is signed, the point is that a chosen message attack on this scheme allows the adversary to get the signature on one message b_1, \dots, b_t , and so he learns the values $x_{b_1}^1, \dots, x_{b_t}^t$. But if he can sign a new

message b'_1, \dots, b'_t , note that for at least one i , we have $b'_i \neq b_i$, and so he needs to come up with $x_{b'_i}^i = x_{1-b_i}^i$, which was not given in the signature. This means inverting f in a random point $h(x_{1-b_i})$ which we know is hard by Lemma 11.2. The fact that you know other x -values does not help, since the x 's were chosen independently. More formally, here's a reduction showing how to use a successful forger F to build a machine that inverts f in a given point:

1. We are given f and y (where $y = f(x)$ for random x). Now run a normal key generation for the LD scheme using f as the one-way function, with one exception: we choose indices $i \in \{1, \dots, t\}, b \in \{0, 1\}$ at random, and set $y_b^i = y$. Give the resulting public key as input to F . So we now know a preimage of all the y 's in the public key, except for one.
2. Get a message b_1, \dots, b_t to sign from F . If $b_j = b$, return ? and stop. Otherwise sign the message as usual and give the signature to F .
3. Get a signature on a new message b'_1, \dots, b'_t from F , this consists of t values x_1, \dots, x_t . If the signature is valid, and $b'_j = b$, then return x_j , otherwise return ?.

Exercise 12.4 (Proof of LD Security) Assume that F succeeds in making a valid signature on a new message with probability ϵ . Show that the probability of returning ? in Step 2 above is $1/2$. Use this to show that we succeed in finding a preimage of y with probability at least $\epsilon/2t$.

This scheme could in fact be based on just a one-way function, but is of course quite useless in the form we gave it here. But since we assume we have collision-intractable functions, we can do better: first note that by the Theorem 12.4 above, we can combine the LD scheme with a hash function, and we will still have a secure scheme. This will of course still be a one-time scheme, but since we now hash messages and then sign the hash value in the LD scheme, we can sign messages of *any* length. This allows us to build a scheme where we can sign any number of messages securely. The idea is that whenever we sign a message, we will also authenticate a new LD public key. We can then use the corresponding secret key to sign the next message, this again involves generating a new key pair, etc..

Key generation Construct a key pair pk_0, sk_0 for the LD scheme allowing to sign t -bit messages, for even t and choose a hash function $h : \{0, 1\}^* \rightarrow \{0, 1\}^{t/2}$. The public key is pk_0, h , while the secret key is sk_0 . Initialize a list L containing for each previously signed message: an LD key pair, an LD signature, and a hash value (on a previously signed message). Initially, L has only one element, namely $((pk_0, sk_0), -, -)$, where the last two entries are empty.

Signing a message Assume we are about to sign message m_i , where this is the i 'th message we sign. Thus the last entry in L is a key pair pk_{i-1}, sk_{i-1} a signature s_{i-1} and a hash value $h(m_{i-1})$. Generate an LD key pair pk_i, sk_i with parameters as above. Compute $h(pk_i), h(m_i)$ and compute

the signature s_i on the concatenation of these two strings under the key pair pk_{i-1}, sk_{i-1} . Return as signature on m_i the sequence

$$(s_i, pk_i, h(m_i)), (s_{i-1}, pk_{i-1}, h(m_{i-1})), \dots, (s_1, pk_1, h(m_1)).$$

Append $((pk_i, sk_i), s_i, h(m_i))$ to L .

Verifying a signature We are given a message m , public key h, pk_0 and a signature $(s_i, pk_i, h(m_i)), (s_{i-1}, pk_{i-1}, h(m_{i-1})), \dots, (s_1, pk_1, h(m_1))$. First verify that $h(m)$ matches the hash value in the first triple in the signature. Then, for $j = i$ down to 1, verify that s_j is a valid LD signature on $h(pk_j)||h(m_j)$ under public key pk_{j-1} . Note that for $j = 1$, this involves verifying a signature under pk_0 , which is part of the public key.

It is quite straight forward to show that this scheme is secure, based only on collision resistance of h and security of the LD scheme (which was again based on one-wayness of the function f). A problem with the scheme is that the length of a signature grows linearly with the number of messages signed. But this is only because we organize data on previous signatures as a linear list (for simplicity). If instead we build a tree structure, where each public key authenticates a message and 2 new public keys, we immediately get a scheme where the signature length only grows logarithmically.

The more general result that any 1-way function implies secure signature schemes, was obtained by first by observing that requiring collision-resistance for h in the scheme above is actually overkill: a slightly weaker type of hash functions known as One-way Universal Hash Functions (OWUHF's) will suffice. The security property of a OWUHF says that no efficient adversary can win a game where he must first produce a string x , then receive the function h and must now find x' such that $x' \neq x$ and $h(x) = h(x')$. One can construct signature schemes from OWUHF's in the same way as we have just seen, and the proof of security is essentially the same. The proof is then completed by showing that one can construct OWUHF's from one-way functions. This is a non-trivial result that requires some quite advanced techniques from probability theory.

12.6 Dealing with replay attacks

Let us finally return to the scenario where A sends messages to B over an insecure channel. Depending on which precise security goals we have, the authentication schemes we have seen may or may not be enough to solve the problems by themselves.

The reason why there may be problems is that if I receive for instance a message m with a digital signature from A , this only proves that *at some point*, A produced this message. It leaves open the possibility for an adversary to take a copy of the signed message and send it to me as many times as he wants. This is known as a *replay attack*. If A is a bank customer, the receiver is a bank, and the message is a request to transfer money from A 's account to someone else, the security problems with replay should be evident.

So what we really want in practice is often not just to protect the integrity of messages, but to have a real authentic channel, that is, we want B to receive the exact same sequence of messages that A sends, and if this not possible, we want to come as close to this as we can. If we do not have physical control over the communication line, we cannot prevent an adversary from physically blocking some messages or from reordering them before they are received, but we should at least be able to make sure that replayed messages are filtered out.

One trivial way to ensure this is have the sender make sure that he never sends exactly the same message twice, for instance by appending a sequence number, and also add an authenticator computed over both message and sequence number. Then we can have the receiver store every message he ever receives (or at least the sequence numbers). This will allow the receiver to filter out every replayed message, and also to correctly place all messages he gets in the order they were sent.

Of course, this is hardly a practical solution. Even if we do use sequence numbers, we cannot expect the receiver to store more than the sequence number of a small number of recently accepted message. Suppose, for instance, that he stores only the last accepted message. Then if the receiver has stored number n and the next message does not have number $n + 1$, we need a rule for the receiver to decide what to do. The problem is that in some cases, messages may get lost or be reordered even if no attacker is present. One possibility is to require that the next message must have number $\geq n$ to be accepted. This will prevent replay, but may also mean that good messages get rejected if they arrive too late.

A different approach appends a timestamp to messages. To be of any use, this of course requires that the receiver checks the time stamp, to ensure that it is not too far from his own time. Some compromise has to be made here, so that on one hand good messages are accepted, but on the other hand replayed messages are rejected. This requires that there is some synchronization between sender and receiver, and also that messages are delivered quickly enough. On the other hand, there is no need here to remember any information about earlier messages.

Finally, one may use interaction: we can have the receiver first send a number R to the sender. This number can be chosen at random, or be a sequence number, the only real requirement is that it has never been used before. Then the sender sends the message plus a MAC computed over the message and R . This will prevent replay, and there is no need for sychronization, but one does need to at least remember some state in order to ensure that R -values are not reused. In addition this is not trivial to implement securely if the receiver has to handle several connections in parallel.

List of Exercises

theorem.2.1 theorem.2.2 theorem.2.3 theorem.3.1 theorem.3.2 theorem.3.3 theorem.3.4 theorem.3.5	
3.1 Distributive law in \mathbb{F}_2 s	16
exer.3.1 theorem.3.6 theorem.3.7 theorem.3.8 theorem.3.9 theorem.3.10 theorem.3.11 theorem.3.12 theorem.3.13 theorem.3.14 theorem.3.15 theorem.3.16	
4.1 Attacking Historic Ciphers	31
exer.4.1	
4.2 Involutory Keys	32
exer.4.2	
4.3 An “Improvement” of Vigenere	32
exer.4.3 theorem.5.1 theorem.5.2 theorem.5.3 theorem.5.4 theorem.5.5 theorem.5.6 theorem.5.7 theorem.5.8 theorem.5.9 theorem.5.10 theorem.5.11 theorem.5.12 theorem.5.13 theorem.5.14	
5.1 Perfect security for any plaintext distribution	51
exer.5.1	
5.2 Wheel of Fortune	52
exer.5.2	
5.3 Conditional Entropy	52
exer.5.3	
5.4 Entropy of the key	52
exer.5.4	
5.5 Entropies in the affine cipher	52
exer.5.5	
5.6 Unicity Distance	52
exer.5.6	
5.7 E has more noise	53
exer.5.7 theorem.6.1 theorem.6.2 theorem.6.3 theorem.6.4	
6.1 Feistel Decryption	74
exer.6.1	
6.2 The DES complementation property	74
exer.6.2	
6.3 Meet in the middle attack	75
exer.6.3	
6.4 PRF security	75
exer.6.4	
6.5 Partial Plaintext Knowledge	76
exer.6.5	
6.6 AES implementation	76

exer.6.6	
6.7 DES Cycles	76
exer.6.7 theorem.7.1 theorem.7.2 theorem.7.3 theorem.7.4 theorem.7.5 theorem.7.6	
theorem.7.7 theorem.7.8 theorem.7.9	
7.1 RSA decryption works on the entire domain	91
exer.7.1	
7.2 Slightly smaller secret exponent	91
exer.7.2	
7.3 Multiple exponent problem	92
exer.7.3	
7.4 Small exponent problem	92
exer.7.4	
7.5 A Fault Attack on the Chinese Remainder optimization	92
exer.7.5	
7.6 RSA is hard to break almost everywhere	93
exer.7.6 theorem.8.1 theorem.8.2 theorem.8.3 theorem.8.4 theorem.8.5 theorem.8.6	
theorem.8.7 theorem.8.8	
8.1 Computing H and P	103
exer.8.1	
8.2 Repeated Squaring	103
exer.8.2	
8.3 CPA Security with several messages	103
exer.8.3	
8.4 Alternative CPA definition	104
exer.8.4 theorem.9.1 theorem.9.2 theorem.9.3 theorem.9.4 theorem.9.5 theorem.9.6	
theorem.9.7	
9.1 CPA security of El Gamal	110
exer.9.1	
9.2 Multiplicative property of El Gamal	110
exer.9.2 theorem.9.8 theorem.9.9 theorem.9.10	
9.3	112
exer.9.3	
9.4 Encoding for Safe Primes	113
exer.9.4	
9.5 Malleability of El Gamal	113
exer.9.5 theorem.9.11 theorem.10.1	
10.1 Correctness of LWE-based encryption	123
exer.10.1 theorem.11.1	
11.1 Hash functions from Factoring	127
exer.11.1 theorem.11.2 theorem.11.3	
11.2 Simplified Merkle-Damgård construction	130
exer.11.2 theorem.11.4 theorem.11.5	
11.3 MAC Attack	135
exer.11.3 theorem.11.6	
11.4 MAC Constructions	137
exer.11.4 theorem.12.1 theorem.12.2 theorem.12.3	
12.1 (Mis)use of randomness in Schnorr signatures	143
exer.12.1	

12.2 (Mis)use of randomness in ECDSA	145
exer.12.2 theorem.12.4 theorem.12.5	
12.3 LD Signatures are 1-time only	147
exer.12.3	
12.4 Proof of LD Security	148
exer.12.4	

References

- [1] W.Alexi, B.Chor, O.Goldreich and C.P.Schnorr: *RSA and Rabin Functions: Certain parts are as hard as the Whole*, SIAM J.Computing, 17(1988), 194-209.
- [2] Bellare and Rogaway: *Optimal Asymmetric Encryption*, Proc. of EuroCrypt 94, Springer Verlag LNCS series, 950.
- [3] Bellare, Desai, Jokipii and Rogaway: *A concrete security treatment of symmetric encryption*, FOCS 97, full paper available from <http://www-cse.ucsd.edu/users/mihir>.
- [4] Cramer and Shoup: *A Practical Public Key Cryptosystem Secure Against Adaptive Chosen Ciphertext Attacks* Proceedings of Crypto 98, Springer Verlag LNCS series 1462.
- [5] Hofheintz and Kiltz: *Practical Chosen Ciphertext Secure Encryption from Factoring*, proc. of EuroCrypt 2009.
- [6] Ueli M. Maurer, Stefan Wolf: *The Diffie-Hellman Protocol*. Des. Codes Cryptography 19(2/3): pp.147-171 (2000).
- [7] Chris Peikert; A Decade of Lattice Cryptography. <https://web.eecs.umich.edu/~cpeikert/pubs/lattice-survey.pdf>
- [8] Bellare, Pietrzak and Rogaway: *Improved Security Analysis for CBC-MACs*, Proc. of Crypto 05.
- [9] Petrank and Rackoff: CBC MAC for real-time data sources, Journal of Cryptology, 13, 315-338, 2000.
- [10] Damgård: *Collision-Free hash functions and public-key signature schemes*, Proc. of EuroCrypt 87.
- [11] Rompel: *One-way functions are necessary and sufficient for digital signatures*, proc. of STOC 90.
- [12] Damgrd, Ivan, Peter Landrock, and Carl Pomerance. *Average case error estimates for the strong probable prime test*. Mathematics of Computation 61.203 (1993): 177-194.
- [13] Jean-Sébastien Coron: *On the Exact Security of Full Domain Hash*, Proceedings of Crypto 2000, Springer Verlag.
- [14] Ronald Cramer, Victor Shoup: *Signature schemes based on the strong RSA assumption*. ACM Trans. Inf. Syst. Secur. 3(3): 161-185 (2000)
- [15] Stefan Wolf: *Unconditional security in cryptography*, Proceedings from School organized by the European Educational Forum 1998, pp. 217-250, Springer Verlag.