Queen Mary
University of London

MAS 335 Cryptography

Notes 6: Public-key cryptography **Spring 2008**

Key distribution

We have seen that it is possible to construct an 'unbreakable' cipher using randomness: this is the one-time pad, whose key is a string of characters as long as the message.

One weakness of all the ciphers we have studied so far is the problem of *key distribution*. If Eve can get hold of the key, then she can decrypt the cipher. On the other hand, Alice and Bob must both know the key, or they cannot communicate. So they must share the key by some secure method which Eve cannot penetrate.

In the classical field of espionage, a spy is given the key (which might be one copy of the one-time pad, the other copy being held by the home agency) before being sent out into the field. Since the key must not be re-used, the spy can only send as much information as the key he possesses. Then he must return to base for a new one-time pad. This system can work well, if the spy keeps the pad on his person and destroys each page when it is used. One of the stories told by Peter Wright in *Spycatcher* relates how MI6 agents found a one-time pad in the possessions of a suspected spy; they copied the pad and returned it, and were subsequently able to read the communications. Of course, having a one-time pad on your person might be extremely dangerous!

Other ciphers use a key which is smaller than the message. For example, a military commander might be issued with a set of keys, and instructed to use a new key every month according to some schedule. But if the enemy captures the keys, then all communications can be read until the whole set of keys is changed; this change may be difficult in wartime.

The commercial use of cryptography since the second world war introduced new problems. Commercial organisations need to exchange secure communications; the only way of exchanging keys seemed to be by using trusted couriers. The amount of courier traffic began to grow out of control. It was the invention of public-key cryptography which gave us a way round the key distribution problem.

That there is a possible way around the problem is suggested by the following fable. Alice and Bob wish to communicate by post, but they know that Eve's agents have control of the postal service, and any letter they send will be opened and read unless it is securely fastened. Alice can put a letter in a chest, padlock the chest, and send it to Bob; but Bob will be unable to open the chest unless he already has a copy of Alice's key!

The solution is as follows. Alice puts her letter in the chest, padlocks it and sends it to Bob. Now Bob cannot open the chest. Instead, he puts his own padlock on the chest and sends it back to Alice. Now Alice removes her padlock and returns the chest to Bob, who then simply has to remove his own padlock and open the chest.

A little more formally, let Alice's encryption and decryption functions be e_A and d_A, and let Bob's be e_B and d_B. This means that Alice encrypts the plaintext p as $e_A(p)$; she can also decrypt this to p, which means that $d_A(e_A(p) = p$.

Now Alice wants to send the plaintext p to Bob by the above scheme. She first encrypts it as $e_A(p)$ and sends it to Bob. He encrypts it as $e_B(e_A(p))$ and returns it to Alice. Now we have to make a crucial assumption:

e_A and e_B commute, that is, $e_A \circ e_B = e_B \circ e_A$.

Now Alice has $(e_B \circ e_A)(p)$, which is equal to $e_A \circ e_B(p) = e_A(e_B(p))$ according to our assumption. Alice can now decrypt this to give $d_A(e_A(e_B(p))) = e_B(p)$ and send this to Bob, who then calculates $d_B(e_B(p)) = p$. At no time during the transaction is any unencrypted message transmitted or any key exchanged.

Note that the operations of putting two padlocks onto a chest do indeed commute! The method would not work if, instead, Bob put the chest inside another chest and locked the outer chest; the operations don't commute in this case.

If the letter that Alice sends to Bob is the key to a cipher (say a one-time pad), then Alice and Bob can now use this cipher in the usual way to communicate safely, without the need for the to-and-fro originally required. The system only depends on the security of the ciphers used by Alice and Bob for the exchange, and the fact that they commute.

Now if Alice and Bob use binary one-time pads for the key exchange, then these conditions are satisfied, since binary addition is a commutative operation.

However, further thought shows that this is not a solution at all! Suppose that Alice wants to send the string l securely to Bob (perhaps for later use as a one-time pad). She encrypts it as $l \oplus k_A$, where k_A is a random key chosen by Alice and known to nobody else. Bob re-encrypts this as $(l \oplus k_A) \oplus k_B$, where k_B is a random key chosen by Bob and known to nobody else. Now $(l \oplus k_A) \oplus k_B = (l \oplus k_B) \oplus k_A$, so when Alice re-encrypts this message with k_A she obtains

$$((l \oplus k_B) \oplus k_A) \oplus k_A = (l \oplus k_B) \oplus (k_A \oplus k_A) = l \oplus k_B,$$

and when Bob finally re-encrypts this with k_B he obtains

$$(l \oplus k_B) \oplus k_B = l.$$

This is the exact analogue of the chest with two keys.

If Eve only intercepts one of these three transmissions, it is impossible for her to read the message, since each is securely encrypted with a one-time pad. However, we must assume that Eve will intercept all three transmissions. Now if she simply adds all three together mod 2, she obtains

$$(l \oplus k_A) \oplus (l \oplus k_A \oplus k_B) \oplus (l \oplus k_B) = l,$$

and she has the message!

Complexity

In trying to wrestle with this problem, Diffie and Hellman came up with an even more radical solution to the problem of key sharing: it is not necessary to share the keys at all! The reason for the insecurity of the above protocol is that decryption is just as simple as encryption for someone who possesses the key; indeed, for binary addition, it is exactly the same operation. (A cipher with this property is called *symmetric*.) The trick is to construct an asymmetric cipher, where decryption is ruinously difficult even if you are in possession of the key.

In order to understand this, we must look at what is meant when we say that a problem is *easy* or *difficult*. This is the subject-matter of *complexity theory*. What follows is a brief introduction to complexity theory. You can find much more detail either in the lecture notes at
`http://www.maths.qmul.ac.uk/%7Epjc/notes/compl.pdf`,
or in books such as M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.*

The subject of computational complexity grew out of computability theory, originally due to Alan Turing (who was also one of the most successful cryptanalysts of the twentieth century). Turing succeeded in showing that there are some mathematical problems which cannot be solved by a machine carrying out an algorithm.

In order to demonstrate this, Turing had to analyse the process of computation. He proposed a model, called a *Turing machine*, and showed that it can carry out any process which can be described algorithmically. Said otherwise, a Turing machine can ‘emulate’ any computer, real or imagined, that has ever been proposed. Seventy years later, despite the efforts of physicists and philosophers, Turing’s claim still stands.

A Turing machine consists of two parts: a *tape* and a *head*.

- The tape is made up of cells stretching infinitely far in both directions. Like the RAM or the hard disc of a computer, it stores information; each cell can either be blank or have a symbol from an alphabet A written on it. The one difference between a Turing machine and a real computer is that the tape is infinite; but we assume that only finitely many tape squares are not blank. So we could regard the memory as finite but unbounded; if more memory is needed for a computation, it is always available.
- The head is a machine which can be in any one of a finite number of states; it resembles the CPU of a computer. The head also has access to one square of the tape.

The configuration of the machine is given by describing

- the string of symbols written on the non-blank squares of the tape;
- the state of the head, and its position (the square which it is scanning).

Now the machine operates as follows. It has a program, a finite set of rules determining what it does at any moment. The action is determined by the state of the head and the symbol on the square which it is scanning. The program can direct the head to change into a specified state, and either to change (or erase) the symbol on the tape square, or to move one place to the left or the right.

One (or more) of the states is distinguished as a 'halting state'. In order to perform a computation, we place a finite amount of information on the tape and put the head in a particular state scanning a particular square. Then the machine starts operating; if it reaches a halting state, its output is the information written on the tape.

Now we can say that a function is computable if there is a Turing machine which computes it. For example, if the tape alphabet is the set of digits $\{0,1,\ldots,9\}$, we could design a machine so that, if the number N is written (in the usual way in base 10) on the tape and the machine is started immediately to the right of the string, it calculates N^2, writes the answer on the tape, and halts. All that such a machine needs is an appropriate program (which might, for example, include the usual multiplication table), and it can square a number of any size.

Clearly this is a very basic kind of machine. But adding facilities such as increasing the number of states, or giving it extra tapes (even changing the tape into a two-dimensional array), or allowing the machine to access any tape square within a fixed distance of the head, we do not change the class of computable functions. Turing showed that there exist mathematical functions which are not computable in this sense.

Now complexity changes the question "Can this function be computed?" to the question "How long will it take to compute it?" Variations are possible, such as "How much memory will I need for the computation?” Clearly the precise answers will depend on the precise details of the Turing machine, so we ask the question in a fairly broad-brush way.

First let us be clear that we are not interested in one-off questions of a general kind such as “Is Goldbach’s Conjecture true?” A *problem* in this context means a whole class of *problem instances*. We specify a problem by saying what data comprises the problem instance, and what answer we require (which might be just ‘Yes’ or ‘No’, or might be some data such as the square of N).

We measure the *size* of a problem instance by the number of tape squares needed to write down the input data. It makes little difference if we decide to use only the binary alphabet, and define the size of a problem instance to be the number of bits of input data. (For example, if we write the number N in base 2 instead of base 10, we need only $\log_2(10) = 2.30\ldots$ times as many tape squares; a constant factor does not matter here.

Now we organise problems into *complexity classes* as in the following examples:

- A problem lies in P, or is *polynomial-time solvable*, if there is a Turing machine which can solve an instance of the problem of size n in at most $p(n)$ for some polynomial p.

- A problem lies in NP, or is *non-deterministic polynomial-time solvable*, if there is a Turing machine which can check the correctness of a proposed solution of a problem instance of size n in at most $p(n)$ steps, for some polynomial p.

- A problem lies in PSpace, or is *polynomial-space solvable*, if there is a Turing machine which can solve an instance of the problem of size n using at most $p(n)$ tape squares, for some polynomial p.

- A problem lies in $\mathsf{ExpTime}$, or is *exponential-time solvable*, if there is a Turing machine which can solve an instance of the problem in at most $2^{p(n)}$ steps, for some polynomial p.

Clearly a problem of higher complexity is harder, and this is a very practical thing to know. If a problem takes n^3 steps to solve, and each step takes a nanosecond, then an instance of size 1000 can be solved in a second, and an instance of size 10000 in three months. However, if it takes 2^n steps, then we can solve an instance of size 30 in a second, while an instance of size 100 will take longer than the age of the universe! The general paradigm is that polynomial-time problems are easy, while exponential-time problems are hard. (Of course much depends on the degree and coefficients of the polynomial; but this works well as a rule of thumb.)

Now we have:

Theorem 12 $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSpace} \subseteq \mathsf{ExpTime}$.

As this is not a course on complexity, we will not prove this in detail; but a few comments on the proof might help explain the concepts. The first inclusion holds because checking a proposed solution is easier than finding a solution.

The second inclusion holds because, if we can check any proposed solution in polynomial time, the check only use a polynomial number of tape squares. So we simply work through all possible solutions until we find one that works.

The last inclusion follows because, if the alphabet has size q, then $p(n)$ tape squares can only hold at most $q^{p(n)}$ possible strings. If the computation took more than this number of steps, we would have to revisit a previous configuration then the machine would be in an infinite loop, and would not finish at all.

One thing remains to be stressed. It is (relatively) easy to show that a problem lies in a particular complexity class. Strictly, we have to show that there is a Turing machine which solves it efficiently. In practice, it is enough to find some algorithm which solves the question efficiently. Then translating that algorithm into a Turing machine may increase the number of steps, but not enough to affect our broad-brush conclusions.

However, it is very difficult to show that a problem is *not* in a particular complexity class, since we would have to show that no possible Turing machine, or no possible algorithm, can solve the problem efficiently enough. There are many instances of problems where the naive algorithm has been superseded by a much more efficient algorithm.

Thus, it is known that P is properly contained in $\mathsf{ExpTime}$, and so at least one of the inclusions in Theorem 12 must be proper. It is conjectured that they are all proper. For example, there are problems in NP (the so-called NP-complete problems) which have been studied for a long time, and nobody has ever managed to find an algorithm to solve any of them in polynomial time.

Our rough equivalences will be:

$$\begin{aligned} \text{`easy'} &= \mathsf{P}, \\ \text{`hard'} &= \mathsf{NP}\text{-complete}. \end{aligned}$$

The NP-complete problems are the ‘hardest’ problems in NP. If a polynomial-time algorithm were ever found for one of them, then we would conclude that $\mathsf{P} = \mathsf{NP}$. It is conjectured that this is not the case. (The Clay Mathematical Institute has offered one million dollars for a proof or disproof of this, as one of its seven millennial problems.)

Public-key cryptography: basics

The idea of public-key cryptography based on the fact that there are easy and hard problems was devised by Diffie and Hellman in the 1970s. This is one of the great ideas of the twentieth century!

In order to explain how a cipher can be secure when the key is publicly available, we now formulate the general setup of cryptography a bit more carefully.

Let $\mathcal{P}$ be the set of plaintext messages that users of the system might wish to send. (Thus, $\mathcal{P}$ might be the set of all strings of letters and punctuation marks, or strings of zeros and ones, or certain strings of dots and dashes.) Let $\mathcal{K}$ be the set of keys, and $\mathcal{Z}$ the set of ciphertexts. Then there is an *encryption function*

$$e : \mathcal{P} \times \mathcal{K} \to \mathcal{Z}$$

and a *decryption function*

$$d : \mathcal{Z} \times \mathcal{K} \to \mathcal{P}$$

which must satisfy the relationship

(PK1) $d(e(p,k),k) = p$.

This simply says that encryption followed by decryption using the same key must recover the original plaintext.

Now the first requirement of public-key cryptography is:

(PK2) Evaluating e should be easy.

(PK3) Evaluating d should be difficult.

(Here, ideally, we should use the equations of the preceding section, that is, 'easy' means 'polynomial-time', while 'hard' means 'NP-complete'. In practice, it almost always means something less precise than this.)

This means that we may assume that Eve not only knows the ciphertext z that Alice sent to Bob, but she also knows the key k and the functions e and d used for encoding and decoding; so all she has to do is to evaluate $d(z,k)$. However, this is a hard problem, and we can assume that, even with the most advanced current technology, it will take her (say) a hundred years to evaluate this function. By that time, the protagonists are all dead and the information has no value.

However, there is a problem here. If decryption is hard, how does Bob (the legitimate recipient) manage to do it? The answer is that there is yet another layer. There is a set $\mathcal{S}$ of *secret keys*, together with an inverse pair of functions

$$g : \mathcal{S} \to \mathcal{K}, \qquad h : \mathcal{K} \to \mathcal{S}.$$

(Think of the mnemonics 'go public' and 'hide'.) Now we make the following requirements:

(PK4) Evaluating the composite function $d^*(z,s) = d(z,g(s))$ is easy.

(PK5) Evaluating g is easy

(PK6) Evaluating h is hard.

Assumption (PK4) means that, given s and z, it is easy to compute p such that $d(z,k) = p$ (or equivalently $e(p,k) = z$) for the unique k which satisfies $h(k) = s$ (or equivalently $g(s) = k$). Note that this does not mean that it is easy to compute $g(s) = k$ and then $d(z,k) = p$, since the latter computation is assumed to be hard; there should be an easy way to compute the composite function d^*.

Now let us see how the system works. Alice wants to send a message to Bob which is secure from the eavesdropper Eve. Bob chooses a 'secret key' from the set $\mathcal{S}$ and tells nobody of his choice. He computes the corresponding 'public key' $k = g(s) \in \mathcal{K}$ and makes this available to Alice. Bob is aware that Eve will also have access to his public key k. We observe that this computation is assumed to be easy.

Alice wants to send Bob the plaintext message p. Knowing his public key k, she computes the ciphertext $e(p,k)$ and sends this to Bob. (This computation is also easy.)

Bob is now faced with the problem of decrypting the message. But Bob already knows the secret key s, and so he only has to do the easy computation of $p = d^*(z,s)$. Since $g(s) = k$, we have $p = d(z,k)$, so that p is indeed the correct plaintext that Alice wanted to send.

What about Eve? Her position is different, since she doesn't know the secret key. Either she has to compute $d(z,k)$ directly (which is hard), or she could decide to compute Bob's secret key s by evaluating the function $s = h(k)$ (which is also hard).

Note that Eve knows in principle how to evaluate either of these functions; the only thing keeping the cipher secure is the complexity of the computations. The important thing is that the secret key, which enables Bob to decrypt the message, is never communicated to anyone else; Bob chooses it, and uses it only to decrypt messages sent to him.

Now in principle we have a method for any set of people to communicate securely. Suppose we have a number of users $A, B, C, \ldots$. Each user chooses his or her own secret key: thus, Alice chooses s_A, Bob chooses s_B, and so on. These choices are never communicated to anyone else. Now Alice computes $k_A = g(s_A)$ and publishes it; and similarly Bob computes $k_B = g(s_B)$ and so on. Then anyone who wishes to send a message p to Alice first obtains her public key k_A (which may be in a directory or on her Web page), and then encrypts it as $z = e(p, k_A)$ and transmits this to Alice. She can calculate $p = d^*(z, s_A) = d(z, k_A)$; but nobody else can read the message without performing a hard calculation.

Some terminology that is often used here is that of 'one-way functions'. A function $f : A \to B$ is said to be *one-way* if it is easy to compute f but hard to compute the inverse function from B to A. It is a *trapdoor one-way function* if there is a piece of information which makes the computation of the inverse function easy. Thus, for public-key cryptography, we want encryption to be a trapdoor one-way function, where the key to the trapdoor is the secret key; the function from secret key to public key should be a one-way function.

Digital signatures

There is a serious potential weakness of public-key cryptography. Eve cannot read Alice's message to Bob. But, since Bob's key is public, Eve can write her own message to Bob purporting to come from Alice, encrypt it with Bob's key, and substitute it for Alice's authentic message on the communication channel. Is there a way around this?

Indeed there is. We make two further assumptions, namely:

(PK7) The set $\mathcal{P}$ of plaintext messages is the same as the set $\mathcal{Z}$ of ciphertexts.

(PK8) $e(d(z,k),k) = z$ for any $z \in \mathcal{Z}$ and $k \in \mathcal{K}$.

The first assumption is not at all restrictive. Almost always, in practice, both sets will consist of all binary strings. The second assumption strictly follows from the others. Condition (PK1) says that decryption is the inverse of encryption; that is, the functions $p \mapsto e(p,k)$ and $z \mapsto d(z,k)$ are inverse bijections (the second undoes the effect of the first). Now inverse functions on finite sets work 'both ways round', so the first undoes the effect of the second; this is exactly what (PK8) claims. The reason that we make this assumption is that in practice the functions may not quite be bijections, or the sets of potential plaintexts may be infinite.

Alice wants to send the plaintext p to Bob, in such a way that it cannot be faked by Eve. First, bizarrely, she pretends that p is a ciphertext and *decrypts it using her own secret key*! In other words, she computes $u = d(p, k_A)$. The result, of course, appears to be gibberish.

Now she writes a preamble in plaintext saying "This is a signed message from Alice", and now encrypts the whole thing using Bob's public key; that is, she calculates $z = e(u, k_B) = e(d(p, k_A), k_B)$. She sends this message to Bob.

Now Bob decrypts this message using his own secret key, obtaining $d(z, k_B) = u$. He sees the statement "This is a signed message from Alice", followed by some gibberish. Now he does another strange thing: he *encrypts* the gibberish, using Alice's public key (as if it were a message he wanted to send to Alice). This gives $e(u, k_A)$, which is equal to p by our assumption (PK8) (since $d(p, k_A) = u$). Then Bob has the intended message.

Assumption (PK8) further tells us that the equation $e(u, k_A) = p$ is equivalent to $d(p, k_A) = u$. Thus, the only person who could compute this is the holder of Alice's secret key, namely Alice herself; so Bob is assured that the message is from Alice. (For Eve to fake such a message, she is faced with the same problem as in decrypting a message from Alice, that is, either compute $d(p, k_A)$, or compute $h(k_A)$; both are hard problems.)