85 Fundamentals of Cryptography and Encryption

Ronald A. Gove

This chapter presents an overview of some basic ideas underlying encryption technology. The chapter begins by defining some basic terms and follows with a few historical notes so the reader can appreciate the long tradition that encryption, or secret writing, has had. The chapter then moves into modern cryptography and presents some of the underlying mathematical and technological concepts behind private and public key encryption systems such as DES and RSA. We will provide an extensive discussion of conventional private key encryption prior to introducing the concept of public key cryptography. We do this for both historical reasons (private key did come first) and technical reasons (public key can be considered a partial solution to the key management problem).

85.1 Some Basic Definitions

We begin our discussion by defining some terms that will be used throughout the chapter. The first term is *encryption*. In simplest terms, encryption is the process of making information unreadable by unauthorized persons. The process may be manual, mechanical, or electronic, and the core of this chapter is to describe the many ways that the encryption process takes place. Encryption is to be distinguished from message-hiding. Invisible inks, microdots, and the like are the stuff of spy novels and are used in the trade; however, we will not spend any time discussing these techniques for hiding information. Exhibit 85.1 shows a conceptual version of an encryption system. It consists of a sender and a receiver, a message (called the "plain text"), the encrypted message (called the "cipher text"), and an item called a "key." The encryption process, which transforms the plain text into the cipher text, may be thought of as a "black box." It takes inputs (the plain text and key) and produces output (the cipher text).

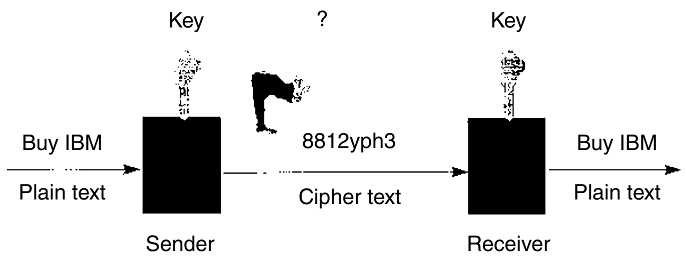


EXHIBIT 85.1 Conceptual version of an encryption system.

The messages may be handwritten characters, electromechanical representations as in a Teletype, strings of 1s and 0s as in a computer or computer network, or even analog speech. The black box will be provided with whatever input/output devices it needs to operate; the insides, or cryptographic algorithm will, generally, operate independently of the external representation of the information.

The *key* is used to select a specific instance of the encryption process embodied in the machine. It is more properly called the "*cryptovariable*." The use of the term "key" is a holdover from earlier times. We will discuss cryptovariables (keys) in more detail in later sections. It is enough at this point to recognize that the cipher text depends on both the plain text and the cryptovariable. Changing either of the inputs will produce a different cipher text. In typical operation, a cryptovariable is inserted prior to encrypting a message and the same key is used for some period of time. This period of time is known as a "cryptoperiod." For reasons having to do with cryptanalysis, the key should be changed on a regular basis. The most important fact about the key is that it embodies the security of the encryption system. By this we mean the system is designed so that complete knowledge of all system details, including specific plain and cipher text messages, is not sufficient to derive the cryptovariable.

It is important that the system be designed in this fashion because the encryption process itself is seldom secret. The details of the data encryption standard (DES), for example, are widely published so that anyone may implement a DES-compliant system. In order to provide the intended secrecy in the cipher text, there has to be some piece of information that is not available to those who are not authorized to receive the message; this piece of information is the cryptovariable, or key.

Inside the black box is an implementation of an algorithm that performs the encryption. Exactly how the algorithm works is the main topic of this chapter, and the details depend on the technology used for the message.

Cryptography is the study of the means to do encryption. Thus cryptographers design encryption systems. Cryptanalysis is the process of figuring out the message without knowledge of the cryptovariable (key), or more generally, figuring out which key was used to encrypt a whole series of messages.

85.2 Some Historical Notes

The reader is referred to Kahn[1] for a well-written history of this subject. We note that the first evidence of cryptography occurred over 4000 years ago in Egypt. Almost as soon as writing was invented, we had secret writing. In India, the ancients' version of Dr. Ruth's Guide to Good Sex, the *Kama-Sutra*, places secret writing as 45th in a list of arts women should know. The Arabs in the 7th century AD were the first to write down methods of cryptanalysis. Historians have discovered a text dated about 855 AD that describes cipher alphabets for use in magic.

One of the better known of the ancient methods of encryption is the Caesar Cipher, so called because Julius Caesar used it. The Caesar Cipher is a simple alphabetic substitution. In a Caesar Cipher, each plain

[1]Kahn, D. 1996. *The Codebreakers; The Comprehensive History of Secred Communication from Ancient Times to the Internet*. Scribner.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Plain text: Omnia gallia est divisa in partes tres ....

Cipher text: RPQLD JDOOLD HVW GLYLVD LQ SDUWHV WUHV ...

EXHIBIT 85.2. The caesar cipher.

text letter is replaced by the letter 3 letters away to the right. For example, the letter A is replaced by D, B by E, and so forth. (See Exhibit 85.2, where the plain-text alphabet is in lower case and the cipher text is in upper case.)

Caesar's Cipher is a form of a more general algorithm known as monoalphabetic substitution. While Julius Caesar always used an offset of 3, in principal one can use any offset, from one to 25. (An offset of 26 is the original alphabet.) The value of the offset is in fact the cryptovariable for this simplest of all monoalphabetic substitutions. All such ciphers with any offset are now called Caesar Ciphers.

There are many ways to produce alphabetic substitution ciphers. In fact, there are 26! (26 factorial or 26X25X24 … X2X1) ways to arrange the 26 letters of the alphabet. All but one of these yields a nonstandard alphabet. Using a different alphabet for each letter according to some well-defined rule can make a more complicated substitution. Such ciphers are called polyalphabetic substitutions.

Cryptography underwent many changes through the centuries often following closely with advances in technology. When we wrote by hand, encryption was purely manual. After the invention of the printing press various mechanical devices appeared such as Leon Batista Alberti's cipher disk in Italy. In the 18th century, Thomas Jefferson invented a ciphering device consisting of a stack of 26 disks each containing the alphabet around the face of the edge. Each disk had the letters arranged in a different order. A positioning bar was attached that allowed the user to align the letters along a row. To use the device, one spelled out the message by moving each disk so that the proper letter lay along the alignment bar. The bar was then rotated a fixed amount (the cryptovariable for that message) and the letters appearing along the new position of the bar were copied off as the cipher text. The receiver could then position the cipher text letters on his "wheel" and rotate the cylinder until the plain text message appeared.

By World War II very complex electromechanical devices were in use by the Allied and Axis forces. The stories of these devices can be found in many books such as Hodges.[2] The need for a full-time, professional cryptographic force was recognized during and after WWII and led to the formation of the National Security Agency by Presidential memorandum signed by Truman. See Bamford[3] for a history of the NSA.

Except for a few hobbyists, cryptography was virtually unknown outside of diplomatic and military circles until the mid-seventies. During this period, as the use of computers, particularly by financial institutions, became more widespread, the need arose for a "public," (non-military or diplomatic) cryptographic system. In 1973 the National Bureau of Standards (now the National Institute of Standards and Technology) issued a request for proposals for a standard cryptographic algorithm. They received no suitable response at that time and reissued the request in 1974. IBM responded to the second request with their Lucifer system, which they had been developing for their own use. This algorithm was evaluated with the help of the NSA and eventually was adopted as the Data Encryption Standard (DES) in 1976. See Federal Information Processing Standard NBS FIPS PUB 46.

[2]Hodges, A. 1983. *Alan Turing: The Enigma of Intelligence*, Simon and Schuster.

[3]Bamford, J. 1982. *The Puzzle palace*. Houghton Mifflin.

The controversy surrounding the selection of DES[4] stimulated academic interest in cryptography and cryptanalysis. This interest led to the discovery of many cryptanalytic techniques and eventually to the concept of public key cryptography. Public key cryptography is a technique that uses distinct keys for encryption and decryption, only one of which need be secret. We will discuss this technique later in this chapter, as public key cryptography is more understandable once one has a firm understanding of conventional cryptography.

The 20 years since the announcement of DES and the discovery of public key cryptography have seen advances in computer technology and networking that were not even dreamed of in 1975. The Internet has created a demand for instantaneous information exchange in the military, government, and most importantly, private sectors that is without precedent. Our economic base, the functioning of our government, and our military effectiveness are more dependent on automated information systems than any country in the world. However, the very technology that created this dependence is its greatest weakness: the infrastructure is fundamentally vulnerable to attacks from individuals, groups, or nation-states that can easily deny service or compromise the integrity of information. The users of the Internet, especially those with economic interests, have come to realize that effective cryptography is a necessity.

85.3 The Basics of Modern Cryptography

Since virtually all of modern cryptography is based on the use of digital computers and digital algorithms, we begin with a brief introduction to digital technology and binary arithmetic. All information in a computer is reduced to a representation as 1s and 0s. (Or the "on" and "off" state of an electronic switch.) All of the operations within the computer can be reduced to logical OR, EXCLUSIVE OR, and AND. Arithmetic in the computer (called binary arithmetic) obeys the rules shown in Exhibit 85.3 (represented by "addition" and "multiplication" tables):

The symbol $\oplus$ is called modulo 2 addition and $\otimes$ is called modulo 2 multiplication. If we consider the symbol '1' as representing a logical value of TRUE and '0' as the logical value FALSE then $\oplus$ is equivalent to exclusive OR in logic (XOR) while $\otimes$ is equivalent to AND. For example, A XOR B is true only if A or B is TRUE but not both. Likewise, A AND B is true only when both A and B are TRUE.

All messages, both plain text and cipher text, may be represented by strings of 1s and 0s. The actual method used to digitize the message is not relevant to an understanding of cryptography so we will not discuss the details here.

We will consider two main classes of cryptographic algorithms:

- Stream Ciphers—which operate on essentially continuous streams of plain text, represented as 1s and 0s
- Block Ciphers—which operate on blocks of plain text of fixed size.

These two divisions overlap in that a block cipher may be operated as a stream cipher. Generally speaking, stream ciphers tend be implemented more in hardware devices, while block ciphers are more

⊕	0	1
0	0	1
1	1	0

⊗	0	1
0	0	0
1	0	1

EXHIBIT 85.3 Binary Arithmetic rules.

[4]Many thought that NSA had implanted a "trap door" that would allow the government to recover encrypted messages at will. Others argued that the cryptovariable length (56 bits) was too short.

suited to implementation in software to execute on a general-purpose computer. Again, these guidelines are not absolute, and there are a variety of operational reasons for choosing one method over another.

85.4 Stream Ciphers

We illustrate a simple stream cipher in the table below and in Exhibit 85.4. Here the plain text is represented by a sequence of 1s and 0s. (The binary streams are to be read from right to left. That is, the right-most bit is the first bit in the sequence.) A keystream[5] generator produces a "random" stream of 1s and 0s that are added modulo 2, bit by bit, to the plaintext stream to produce the cipher-text stream.

The cryptovariable (key) is shown as entering the keystream generator. We will explain the nature of these cryptovariables later. There are many different mechanisms to implement the keystream generator, and the reader is referred to Schneier[6] for many more examples. In general, we may represent the internal operation as consisting of a finite state machine and a complex function. The finite state machine consists of a system state and a function (called the "next state" function) that cause the system to change state based on certain input.

The complex function operates on the system state to produce the keystream. Exhibit 85.5 shows the encryption operation. The decryption operation is equivalent; just exchange the roles of plain text and cipher text. This works because of the following relationships in modulo two addition: Letting p represent a plain-text bit, k a keystream bit, and c the cipher text bit

$$c = p \oplus k,$$

Plain text:	1	0	1	1	0	1	1	0	0
	$\ominus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\ominus$	$\oplus$	$\oplus$	$\oplus$
Keystream	1	1	0	1	0	0	0	1	1
Cipher text	0	1	1	0	0	1	1	1	1

EXHIBIT 85.4 Stream cipher.

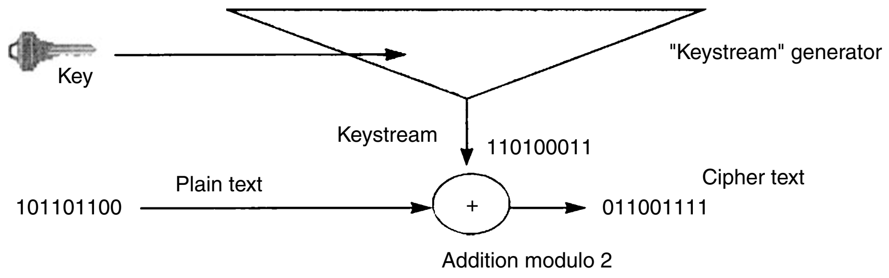


EXHIBIT 85.5 Stream ciphers.

[5]The reader is cautioned not to confuse "keystream" with key. The term is used for historical reasons and is not the "key" for the algorithm. It is for this reason that we prefer the term "cryptovariable."

[6]Schneier, B. 1996. *Applied Cryptography.* John Wiley.

```
1111 add 3rd and 4th bits, shift right -->
0111
0011
0001
1000
0100
0010
1001
1100
0110
1011
0101
1010
1101
1110
1111 begins to repeat
```

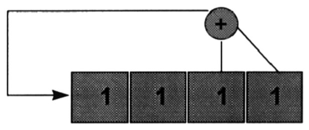


EXHIBIT 85.6 Simple LFSR.

$$\text{so, } c \oplus k = (p \oplus k) \oplus k = p \oplus (k \oplus k) = p \oplus 0 = p,$$

since in binary arithmetic $x \oplus x$ is always 0. ($1 \oplus 1 = 0 \oplus 0 = 0$).

These concepts are best understood with examples. Exhibit 85.6 shows a simple linear feedback shift register (LFSR). A LFSR is one of the simplest finite state machines and is used as a building block for many stream ciphers (see Schneier's text). In Exhibit 85.6, the four-stage register (shown here filled with 1s) represents the state. During operation, at each tick of the internal clock, the 4 bits shift to the right (the right-most bit is dropped), and the last 2 bits (before the shift) are added (mod 2) and placed in the left-most stage. In general, an LFSR may be of any length, n, and any of the individual stages may be selected for summing and insertion into the left-most stage. The only constraint is that the right-most bit should always be one of the bits selected for the feedback sum. Otherwise, the length is really $n-1$, not n. Exhibit 85.6 shows the sequence of system states obtained from the initial value of 1111. In some systems, the initial value of the register is part of the cryptovariable.

Note that if we started the sequence with 0000, then all subsequent states would be 0000. This would not be good for cryptographic applications since the output would be constant. Thus the all-0 state is avoided. Note also that this four-stage register steps through $15 = 2^4 - 1$ distinct states before repeating. Not all configurations of feedback will produce such a maximal sequence. If we number the stages in Exhibit 85.6 from left to right as 1, 2, 3, 4, and instead of feeding back the sum of stages 3 and 4 we selected 2 and 4, then we would see a very different sequence. This example would produce 2 sequences (we call them cycles) of length 6, one cycle of length 3, and 1 of length 0. For example, starting with 1111 as before will yield:

$$1111 \rightarrow 0111 \rightarrow 0011 \rightarrow 1001 \rightarrow 1100 \rightarrow 1110 \rightarrow 1111$$

It is important to have as many states as possible produced by the internal state machine of the keystream generator. The reason is to avoid repeating the keystream. Once the keystream begins to repeat, the same plain text will produce the same cipher text. This is a cryptographic weakness and should be avoided. While one could select any single stage of the LFSR and use it as the keystream, this is not a good idea. The reason is that the linearity of the sequence of stages allows a simple cryptanalysis. We can avoid the linearity by introducing some more complexity into the system. The objective is to produce a keystream that looks completely random.[7] That is, the keystream will

[7]The output cannot be truly random since the receiving system has to be able to produce the identical sequence.

pass as many tests of statistical randomness as one cares to apply. The most important test is that knowledge of the algorithm and knowledge of a sequence of successive keystream bits does not allow a cryptanalyst to predict the next bit in the sequence. The complexity can often be introduced by using some nonlinear polynomial $f(a_1, a_2, \ldots, a_m)$ of a selection of the individual stages of the LFSR. Nonlinear means that some of the terms are multiplied together such as $a_1a_2 + a_3a_4 + \ldots a_{m-1}a_m$. The selection of which register stages are associated with which inputs to the polynomial can be part of the cryptovariable (key). The reader is encouraged to refer to texts such as Schneier[6] for examples of specific stream-cipher implementations. Another technique for introducing complexity is to use multiple LFSRs and to select output alternately from each based on some pseudorandom process. For example, one might have three LFSRs and create the keystream by selecting bits from one of the two, based on the output of a third.

Some of the features that a cryptographer will design into the algorithm for a stream cipher include:

1. Long periods without a repetition.
2. Functional complexity—each keystream bit should depend on most or all of the cryptovariable bits.
3. Statistically unpredictable—given n successive bits from the keystream it is not possible to predict the $n+1$st bit with a probability different from ½.
4. The keystream should be statistically unbiased—there should be as many 0s as 1s, as many 00s as 10s, 01s, and 11s, etc.
5. The keystream should not be linearly related to the cryptovariable.

We also note that in order to send and receive messages encrypted with a stream cipher the sending and receiving systems must satisfy several conditions. First, the sending and receiving equipment must be using identical algorithms for producing the keystream. Second, they must have the same cryptovariable. Third, they must start in the same state; and fourth, they must know where the message begins.

The first condition is trivial to satisfy. The second condition, ensuring that the two machines have the same cryptovariable, is an administrative problem (called key management) that we will discuss in a later section. We can ensure that the two devices start in the same state by several means. One way is to include the initial state as part of the cryptovariable. Another way is to send the initial state to the receiver at the beginning of each message. (This is sometimes called a message indicator, or initial vector.) A third possibility is to design the machines to always default to a specific state. Knowing where the beginning of the message is can be a more difficult problem, and various messaging protocols use different techniques.

85.5 Block Ciphers

A block cipher operates on blocks of text of fixed size. The specific size is often selected to correspond to the word size in the implementing computer, or to some other convenient reference (e.g., 8-bit ASCII text is conveniently processed by block ciphers with lengths that are multiples of 8 bits). Because the block cipher forms a one-to-one correspondence between input and output blocks it is nothing more or less than a permutation. If the blocks are *n* bits long, then there are 2^n possible input blocks and 2^n possible output blocks. The relationship between the input and output defines a permutation. There are $(2^n)!$ possible permutations, so theoretically there are $(2^n)!$ possible block cipher systems on n bit blocks.[8]

A simple block cipher on 4-bit blocks is shown in Exhibit 85.7.

With such a prodigious number of possible block ciphers, one would think it a trivial matter to create one. It is not so easy. First of all, the algorithm has to be easy to describe and implement. Most of the $(2^n)!$ permutations can only be described by listing the entries in a table such as the one in Exhibit 85.8. For a 32-bit block cipher this table would have on the order of $10^{9.6}$ entries, which is quite impractical. Another consideration is that there needs to be a relation between the cryptovariable and the permutation.

[8]For $n=7$, $2^n!$ is about 10^{215}. The case $n=8$ is more than I can calculate. Clearly, there is no lack of possible block ciphers.

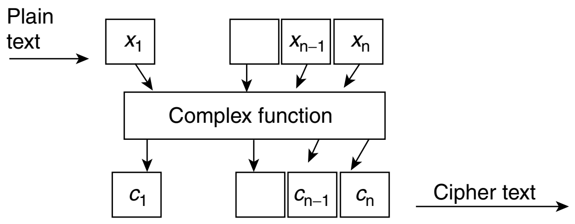
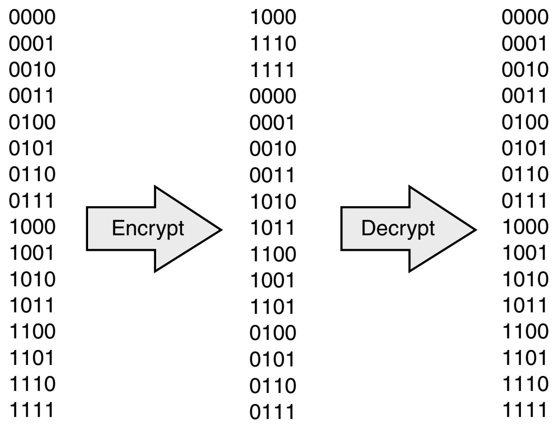


EXHIBIT 85.7 Block ciphers.


Encrypt
Decrypt

0000	1000	0000
0001	1110	0001
0010	1111	0010
0011	0000	0011
0100	0001	0100
0101	0010	0101
0110	0011	0110
0111	1010	0111
1000	1011	1000
1001	1100	1001
1010	1001	1010
1011	1101	1011
1100	0100	1100
1101	0101	1101
1110	0110	1110
1111	0111	1111

EXHIBIT 85.8 Simple Block cipher.

In most implementations, the cryptovariable selects a specific permutation from a wide class of permutations. Thus one would need as many tables as cryptovariables. We conclude from this that it is not easy to design good block ciphers.

The most well-known block cipher is the Data Encryption Standard, DES. The cryptovariable for DES is 64 bits, 8 of which are parity check bits. Consequently the cryptovariable is effectively 56 bits long. DES operates as follows: a 64-bit plain text block, after going through an initial permutation (which has no cryptographic significance) is split onto left and right halves, L_0 and R_0. These two halves are then processed as follows for $i=0, 1, \ldots, 15$

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

The blocks K_i are derived from the cryptovariable. The function *f* is a very complex function involving several expansions, compressions, and permutations by means of several fixed tables called the S-boxes and P-boxes. The reader is referred to FIPS PUB 46 for a detailed description of the S-boxes and P-boxes.

As was the case with the DES cryptovariable, there has been much discussion about the significance of the S-boxes. Some people have argued that the NSA designed the S-Boxes so as to include a “trap door” that would allow them to decrypt DES-encrypted messages at will. No one has been able to discover such a trap door. More recently it has been stated that the S-boxes were selected to minimize the danger from an attack called differential cryptanalysis.

Because of the widespread belief that the DES cryptovariable is too small, many have suggested that one encrypt a message twice with DES using two different cryptovariables. This “Double DES” is carried out in the following way. Represent the operation of DES encryption on message *P* and cryptovariable *K* as $C=E(P; K)$; and the corresponding decryption as $P=D(C; K) =D(E(P; K); K)$. The “Double DES” with cryptovariables K and K' is

$$C = E(E(P;K);K')$$

Since each cryptovariable is 56 bits long, we have created an effective cryptovariable length of $56+56=112$ bits. However, we shall see in the section on cryptanalysis that there is an attack on double-DES that requires about the same amount of computation as that required to attack a single DES. Thus double DES is really no more secure than single DES.

A third variant is triple DES, which applies the DES algorithm three times with two distinct cryptovariables. Let K and K' be DES cryptovariables. Then triple DES is

$$C = E(D(E(P;K);K');K).$$

That is, apply the encrypt function to P using the first cryptovariable, K. Then apply the decrypt function to the result using the second cryptovariable, K' Since the decrypt function is using a different cryptovariable, the message is not decrypted; it is transformed by a permutation as in any block cipher. The final step is to encrypt once again with the encrypt function using the first key, K. By using the D in the middle, a triple DES implementation can be used to encrypt a single DES message when $K=K'$:

$$C = E(D(E(P;K);K);K) = E(P;K).$$

Thus, someone using triple DES is still able to communicate securely with persons using single DES. No successful attacks have been reported on triple DES that are any easier than trying all possible pairs of cryptovariables. In the next section we deal with cryptanalysis in more detail.

85.6 Cryptanalysis

As we stated in the introduction, cryptography is the science of designing algorithms for encrypting messages. Cryptanalysis is the science (some would say art) of "breaking" the cryptographic systems. In the following we will try to explain just what "breaking" a cryptosystem means, as there are many misconceptions in the press.

There is an obvious analogy between cryptanalysis and cryptography and burglars and locks. As the locksmiths design better locks the burglars develop better ways to pick them. Likewise, as the cryptographer designs better algorithms the cryptanalyst develops new attacks. A typical design methodology would be to have independent design teams and attack teams. The design team proposes algorithms, and the attack teams tries to find weaknesses. In practice, this methodology is used in the academic world. Researchers publish their new algorithms, and the rest of the academic world searches for attacks to be published in subsequent papers. Each cycle provides new papers toward tenure.

Breaking or attacking a cryptosystem means recovering the plain-text message without possession of the particular cryptovariable (or key) used to encrypt that message. More generally, breaking the system means determining the particular cryptovariable (key) that was used. Although it is the message (or the information in the message) that the analyst really wants, possession of the cryptovariable allows the analyst to recover all of the messages that were encrypted in that cryptovariable. Since the cryptoperiod may be days or weeks, the analyst who recovers a cryptovariable will be able to recover many more messages than if he attacks a single message at a time.

Determining the specific details of the algorithm that was used to encrypt the message is generally not considered part of breaking an encryption system. In most cases, e.g., DES, the algorithm is widely known. Even many of the proprietary systems such as RC4 and RC5 have been published. Because it is very difficult to maintain the secrecy of an algorithm it is better to design the algorithm so that knowledge of the algorithm's details is still not sufficient to determine the cryptovariable used for a specific message without trying all possible cryptovariables.

Trying all cryptovariables is called a “brute force” or “exhaustion” attack. It is an attack that will always work as long as one is able to recognize the plain-text message after decryption. That is, in any attack you need to be able to decide when you have succeeded. One also has to be able to find the cryptovariable (and hence the message) in time for it to be of use. For example, in a tactical military environment, to spend one week to recover a message about an attack that will occur before the week is over will not be useful. Last, one has to be able to afford to execute the attack. One may often trade off time and computer power; an attack that may take one year on a PC might take only one day on 365 PCs. If one must have the message within a day for it to be valuable, but one does not have the funds to acquire or run 365 PCs, then one really doesn’t have a viable attack.

Often a cryptanalyst might assume that she possesses matched plain and cipher text. This is sometimes possible in real systems because military and diplomatic messages often have stereotyped beginnings. In any case it is not a very restrictive condition and can help the cryptanalyst evaluate the cryptographic strength of an algorithm.

Let us look at a brute force attack on some system. We suppose that the cryptovariable has n binary bits (e.g., DES has $n=56$). We suppose that we have a stream cipher and that we have matched plain and cipher text pairs P_i and C_i for $I=1, 2, \ldots$. For each possible cryptovariable there is some fixed amount of computation (“work”) needed to encrypt a P_i and see if it results in the corresponding C_i. We can convert this work into the total number, W, of basic bit operations in the algorithm such as shifts, mod 2 additions, compares, etc. Suppose for definiteness that $W=1000$ or 10^3.

There is a total of 2^n n-bit cryptovariables. For $n=56$, 2^{56} is about $10^{16.8}$ or 72,000,000,000,000,000. If we select one of the possible cryptovariables and encrypt P_1 we have a 50:50 chance of getting C_1 since the only choices are 1 and 0. If we do not obtain C_1 we reject the selected cryptovariable as incorrect and test the next cryptovariable. If we do get C_1 then we must test the selected cryptovariable on P_2 and C_2. How many tests do we need to make in order to be sure that we have the correct cryptovariable? The answer is: at least 56. The rationale is that the probability of the wrong cryptovariable successfully matching 56 or more bits is 2^{-56}. Since we potentially have to try 2^{56} cryptovariables the expected number of cryptovariables passing all the tests is $(2^{56})(2^{-56})=1$. With one “survivor” we may correctly assume it is the cryptovariable we want. If we tested only 2^{55} cryptovariables, then we would expect two survivors. (Cryptanalysts call a cryptovariable that passes all of the tests by chance a “non-causal survivor.”) If we test a few more than 56, the expected number of non-causal survivors is much less than 1. Thus we can be sure that the cryptovariable that does successfully match the 56 P_i and C_i is the one actually used. In a block cipher, such as DES, testing one block is usually sufficient since a correct block has 64 correct bits.

A natural question is how long does it take to execute a brute force attack (or any other kind of attack for that matter). The answer depends on how much computational power is available to the analyst. And since we want cryptographic systems to be useful for many years we also need to know how much computational power will be available in years hence. Gordan Moore, one of the founders of Intel, once noted that processing speeds seem to double (or costs halved) every 18 months. This is equivalent to a factor of 10 increase in speed per dollar spent about every 5 years. This trend has continued quite accurately for many years and has come to be known as “Moore’s law.”

Using Moore’s law we can make some predictions. We first introduce the idea of a MIPS year (*M.Y.*). This is the number of instructions a million-instruction-per-second computer can execute in one year. One *M.Y.* is approximately $10^{13.5}$ instructions. At today’s prices, one can get a 50 MIPS PC for about \$750. We can then estimate the cost of a MIPS year at about \$750/50 or \$15, assuming we can run the computer for one year.

Let’s look at what this means in two examples. We consider two cryptographic systems. One with a 56-bit cryptovariable (e.g., DES) and the other a 40-bit cryptovariable. Note that 40 bits is the maximum cryptovariable length allowed for export by the U.S. government. We assume that each algorithm requires about 1000 basic instructions to test each cryptovariable. Statistics tells us that, on average, we may expect to locate the correct cryptovariable after testing about ½ of the cryptovariable space.

There are two perspectives: how much does it cost? And how long does it take? The cost may be estimated from:

$$(\tfrac{1}{2})(1000N(15))/M.Y.,$$

where N equals the number of cryptovariables (in the examples, either 2^{56} or 2^{40}), and $M.Y. = 10^{13.5}$. The elapsed time requires that we make some assumptions as to the speed of processing. If we set K equal to the number of seconds in one year, and R the number of cryptovariables tested per second, we obtain the formula:

$$\text{Time(in years)} = (\tfrac{1}{2})(N/KR).$$

The results are displayed in Exhibit 85.9.

One of the first public demonstrations of the accuracy of these estimates occurred during the summer of 1995. At that time a student at Ecole Polytechnique reported that he had "broken" an encrypted challenge message posted on the Web by Netscape. The message, an electronic transaction, was encrypted using an algorithm with a 40-bit cryptovariable. What the student did was to partition the cryptovariable space across a number of computers to which he had access and set them searching for the correct one. In other words he executed a brute force attack and he successfully recovered the cryptovariable used in the message. His attack ran for about 6 days and processed about 800,000 keys per second. While most analysts did not believe that a 40-bit cryptovariable was immune to a brute force attack, the student's success did cause quite a stir in the press. Additionally the student posted his program on a Web site so that anyone could copy the program and run the attack. At the RSA Data Security Conference, January 1997, it was announced that a Berkeley student using the idle time on a network of 250 computers was able to break the RSA challenge message, encrypted using a 40-bit key, in three and one-half hours.

More recently a brute force attack was completed against a DES message on the RSA Web page. We quote from the press release of the DES Challenge team (found on www.frii.com/~rtv/despr4.htm):

LOVELAND, COLORADO (June 18, 1997). Tens of thousands of computers, all across the U.S. and Canada, linked together via the Internet in an unprecedented cooperative supercomputing effort to decrypt a message encoded with the government-endorsed Data Encryption Standard (DES).

Responding to a challenge, including a prize of $10,000, offered by RSA Data Security, Inc., the DESCHALL effort successfully decoded RSA's secret message.

According to Rocke Verser, a contract programmer and consultant who developed the specialized software in his spare time, "Tens of thousands of computers worked cooperatively on the challenge

Year	M.Y. cost	On 56 bit cryptovariable	On 40 bit cryptovariable
1998	$15	$17 million	$260
2003	$1.50	$1.7 million	$26
2008	$0.15	$170 thousand	$2.60

Number of cryptovariables tested per second	On 56 bit cryptovariable	On 40 bit cryptovariable
1,000	300 million years	17.5 years
1,000,000	300,000 years	6.2 days
1,000,000,000	300 years	9 minutes
1,000,000,000,000	109 days	0.5 seconds

EXHIBIT 85.9 Cost and time for brute force attack.

> in what is believed to be one of the largest supercomputing efforts ever undertaken outside of government."
>
> Using a technique called "brute-force," computers participating in the challenge simply began trying every possible decryption key. There are over 72 quadrillion keys (72,057,594,037,927,936). At the time the winning key was reported to RSADSI, the DESCHALL effort had searched almost 25% of the total. At its peak over the recent weekend, the DESCHALL effort was testing 7 billion keys per second.
>
> … And this was done with "spare" CPU time, mostly from ordinary PCs, by thousands of users who have never even met each other.

In other words, the DESCHALL worked as follows. Mr. Verser developed a client-server program that would try all possible keys. The clients were available to any and all who wished to participate. Each participant downloaded the client software and set it executing on their PC (or other machine). The client would execute at the lowest priority in the client PC and so did not interfere with the participant's normal activities. Periodically the client would connect to the server over the Internet and would receive another block of cryptovariables to test. With tens of thousands of clients it only took 4 months to hit the correct cryptovariable.

Another RSA Data Security Inc.'s crypto-cracking contest, launched in March 1997, was completed in October 1997. A team of some 4000 programmers from across the globe, calling themselves the "Bovine RC5 Effort," has claimed the $10,000 prize for decoding a message encrypted in 56-bit -RC5 code. The RC5 effort searched through 47 percent of the possible keys before finding the one used to encrypt the message.

RSA Data Security Inc. sponsored the contest to prove its point that 128-bit encryption must become the standard. Under current U.S. policy, software makers can sell only 40-bit key encryption overseas, with some exceptions available for 56-bit algorithms.

A second DES challenge was solved in February 1998 and took 39 days (see Exhibit 85.10). In this challenge, the participants had to test about 90 percent of the keyspace.

This chapter has focused mostly on brute force attacks. There may be, however, other ways to attack an encryption system. These other methods may be loosely grouped as analytic attacks, statistical attacks, and implementation attacks.

Analytic attacks make use of some weakness in the algorithm that enables the attacker to effectively reduce the complexity of the algorithm through some algebraic manipulation. We will see in the section on public key systems, that the RSA public key algorithm can be attacked by factoring with much less work than brute force. Another example of an analytic attack is the attack on double DES.

Start of contest:
January 13, 1998 at 09:00 PST
Start of distributed.net effort: January 13, 1998 at 09:08 PST
End of contest: February 23, 1998 at 02:26 PST

Size of keyspace: 72,057,594,037,927,936
Approximate keys tested: 63,686,000,000,000,000

Peak keys Per second: 34,430,460,000

EXHIBIT 85.10 RSA project statistics.

Double DES, you recall, may be represented by:

$$C = E(E(P;\ K);\ L),$$

where K and L are 56-bit DES keys. We assume that we have matched plain and cipher text pairs C_i, P_i. Begin by noting that if $X = E(P;\ K)$. Then $D(C;\ L) = X$. Fix a pair C_1, P_1, and make a table of all 2^{56} values of $D(C_1;\ L)$ as L ranges through all 2^{56} possible DES keys. Then try each K in succession, computing $E(P_1;\ K)$ and looking for matches with the values of $D(C_1;\ L)$ in the table. Each pair K, L for which $E(P_1;\ K)$ matches $D(C_1;\ L)$ in the table is a possible choice of the sought-for cryptovariable. Each pair passing the test is then tested against the next plain-cipher pair P_2, C_2.

The chance of a non-causal match (a match given that the pair K, L is not the correct cryptovariable) is about 2^{-64}. Thus of the 2^{112} pairs K, L, about $2^{(112-64)} = 2^{48}$ will match on the first pair P_1, C_1. Trying these on the second block P_2, C_2 and only $2^{(48-64)} = 2^{-16}$ of the non-causal pairs will match. Thus, the probability of the incorrect cryptovariable passing both tests is about $2^{-16} \sim 0$. And the probability of the correct cryptovariable passing both tests is 1.

The total work to complete this attack (called the "meet in the middle" attack) is proportional to $2^{56} + 2^{48} = 2^{56}(1 + 2^{-8}) \sim 2^{56}$. In other words an attack on double DES has about the same work as trying all possible single DES keys. So there is no real gain in security with double DES.

Statistical attacks make use of some statistical weakness in the design. For example, if there is a slight bias toward 1 or 0 in the keystream, one can sometimes develop an attack with less work than brute force. These attacks are too complex to describe in this short chapter.

The third class of attacks is implementation attacks. Here one attacks the specific implementation of the encryption protocol, not simply the cryptographic engine. A good example of this kind of attack was in the news in late summer 1995. The target was Netscape; and this time the attack was against the 128-bit cryptovariable. Several Berkeley students were able to obtain source code for the Netscape encryption package and were able to determine how the system generated cryptovariables. The random generator was given a seed value that was a function of certain system clock values.

The students discovered that the uncertainty in the time variable that was used to seed the random-number generator was far less than the uncertainty possible in the whole cryptovariable space. By trying all possible seed values they were able to guess the cryptovariable with a few minutes of processing time. In other words, the implementation did not use a randomization process that could, in principle, produce any one of the 2^{128} possible keys. Rather it was selecting from a space more on the order of 2^{20}. The lesson here is that even though one has a very strong encryption algorithm and a large key space, a weak implementation could still lead to a compromise of the system.

85.7 Key (Cryptovariable) Management

We have noted in the previous sections that each encryption system requires a key (or cryptovariable) to function and that all of the secrecy in the encryption process is maintained in the key. Moreover, we noted that the sending and receiving party must have the same cryptovariable if they are to be able to communicate. This need translates to a significant logistical problem.

The longer a cryptovariable is used the more likely it is to be compromised. The compromise may occur through a successful attack or, more likely, the cryptovariable may be stolen by or sold to an adversary. Consequently, it is advisable to change the variable frequently. The frequency of change is a management decision based on the perceived strength of the algorithm and the sensitivity of the information being protected.

All communicating parties must have the same cryptovariable. Thus you need to know in advance with whom you plan to exchange messages. If a person needs to maintain privacy among a large number of different persons, then one would need distinct cryptovariables for each possible communicating pair. In a 1000-person organization, this would amount to almost one million keys.

Next, the keys must be maintained in secrecy. They must be produced in secret, and distributed in secret, and held by the users in a protected area (e.g., a safe) until they are to be used. Finally they must be destroyed after being used.

For centuries, the traditional means of distributing keys was through a trusted courier. A government organization would produce the cryptovariables. And couriers, who have been properly vetted and approved, would distribute the cryptovariables. A rigorous audit trail would be maintained of manufacture, distribution, receipt, and destruction. Careful plans and schedules for using the keys would be developed and distributed.

This is clearly a cumbersome, expensive, and time-consuming process. Moreover the process was and is subject to compromise. Many of history's spies were also guilty of passing cryptovariables (as well as other state secrets) to the enemy.

As our communications systems became more and more dependent on computers and communication networks, the concept of a key distribution center was developed. The key distribution center concept is illustrated in Exhibit 85.11. The operation is as follows: Initially each user, A, B, …, is given (via traditional distribution) a user-unique key that we denote by K_A, K_B, etc. These cryptovariables will change only infrequently, which reduces the key distribution problem to a minimum. The KDC maintains a copy of each user-unique key. When A calls B, the calling protocol first contacts the KDC and tells it that user A is sending a message to user B. The KDC then generates a random "session key," K, i.e., a cryptovariable that will be used only for this communicating session between A and B. The KDC encrypts K in user A's unique cryptovariable, $E(K; K_A)$ and sends this to A. User A decrypts this message obtaining K. The KDC likewise encrypts K in user B's unique cryptovariable, $E(K; K_B)$ and sends this result to B. Now A and B (and no other party) have K, which they use as the cryptovariable for this session.

A session here may be a telephone call or passing a message through a packet switch network; the principles are the same. In practice the complete exchange is done in seconds and is completely transparent to the user.

The KDC certainly simplifies the distribution of cryptovariables. Only the user-unique keys need to be distributed in advance, and only infrequently. The session key only exists for the duration of the message so there is no danger that the key might be stolen and sold to an unauthorized person at some later date. But the KDC must be protected, and one still has to know with whom they will be communicating. The KDC will not help if one needs to send an electronic mail message to some new party (i.e., a party unknown to the KDC) for example.

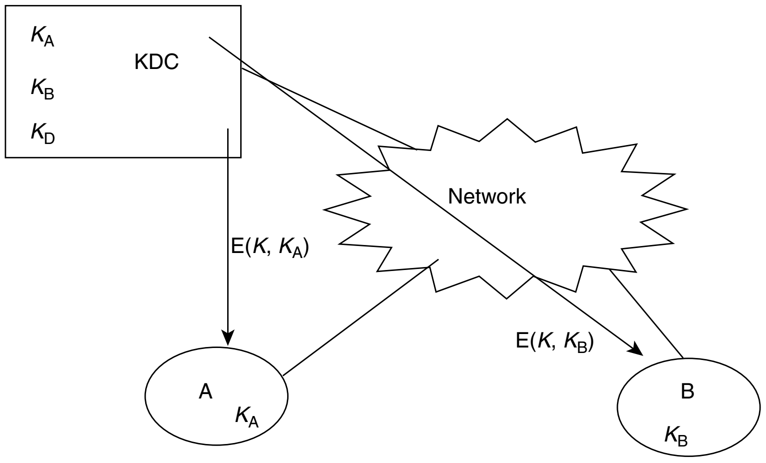


EXHIBIT 85.11 Key distribution center.

It is clear that cryptovariable (or key) management is difficult and does not provide much in the way of flexibility. Many people have wondered if it would be possible to develop an encryption system that did not require secret keys; a system where one could have a directory of public keys. When you wanted to send an encrypted message to someone, you would look up that person's cryptovariable in a "telephone book," encrypt the message, and send it. And no one intercepting the message would be able to decrypt it except the intended recipient. Can such a system be designed? The answer is yes. It is called public key cryptography.

85.8 Public Key Cryptography

The concept of public key cryptography was first discovered and publicly announced by Whitfield Diffie and Martin Hellman (and independently by Ralph Merkle) in 1976. Adm. Bobby Inmann, a former director of the National Security Agency once stated publicly that NSA knew of the idea for many years prior to the publication by Diffie and Hellman.

The public key concept is rather simple (as are most great ideas, once they are explained). We assume that we have two special functions, E and D, that can operate on messages M. (In actual applications large integers will represent the messages, and E and D will be integer functions.) We assume that E and D satisfy the following conditions:

1. $D(E(M))=M$
2. $E(D(M))=M$
3. Given E it is not possible to determine D
4. Given D it is not possible to determine E.

The use of the function E in encryption is straightforward. We assume that each person, A, B, C, has pairs of functions E_A, D_A, E_B, D_B, … that satisfy the conditions 1., 2., and 3. given above. Each user X makes their E_X publicly available but keeps their D_X secret and known only to themselves. When A wants to send a message, M, to B, A looks up E_B in the published list and computes $E_B(M)$. By property 2, $D_B(E_B(M)) = M$ so B can decrypt the message. From property 3, no person can determine D_B from knowledge of E_B so no one but B can decipher the message.

The functions can also be used to sign messages. Perhaps A wants to send a message M to B and she does not care if anyone else sees the message, but she does want B to know that it really came from her. In this case A computes $D_A(M)$, called a signature, and sends it along with M. When B gets these two messages, he looks up A's function E_A and computes $E_A(D_A(M))$ and obtains M from property 2. If this computed M agrees with the message sent as M, then B is sure that it came from A. Why? Because no one else has or can compute D_A except A and the likelihood of someone producing a fictitious X such that $E_A(X)=M$ is infinitesimally small.

Now suppose A wants to send B a secret message and sign it. Let M be the message. A first computes a "signature" $S=D_A(M)$ and concatenates this to the message M, forming M, S. A then encrypts both the message and the signature, $E_B(M, S)$ and sends it to B. B applies D_B to $E_B(M, S)$ obtaining $D_B(E_B(M, S))=M, S$. B then computes $E_A(S)=E_A(D_A(M))=M$ and compares it to the message he decrypted. If both versions of M are the same, he can be assured that A sent the message.

The question the reader should be asking is "Do such functions exist?" The answer is yes, if we relax what we mean by conditions 3 and 4 above. If we only require that it be computationally infeasible to recover D from E (and vice versa) then the functions can be shown to exist. The most well-known example is the RSA algorithm, named for its discoverers, Rivest, Shamir, and Adleman.

A description of RSA requires a small amount of mathematics that we will explain as we proceed. We start with two large prime numbers, p and q. By large we mean they contain hundreds of digits. This is needed in order to meet conditions 3 and 4. A prime number, you recall, is a number that has no divisors except the number itself and 1. (In dealing with integers when we say a divides b we mean that there is no remainder; i.e., $b=ac$ for some integer c.) The numbers 2, 3, 7, 11, 13, 17 are all prime. The number 2 is the only even prime. All other primes must be odd numbers.

We then define a number n as the product of p and q:

$$n = pq$$

We also define a number t as:

$$t = (p-1)(q-1)$$

As an example, take $p=3$ and $q=7$. (These are not large primes, but the mathematics is the same.) Then $n=21$ and $t=12$. The next step in the construction of RSA is to select a number e that has no common divisors with t. (In this case e and t are said to be relatively prime.) In our numerical example we may take $e=5$ since 5 and 12 have no common divisors. Next we must find an integer d such that $ed-1$ is divisible by t. (This is denoted by ed $=1$ mod t.) Since 5*5−1=25−1=24=2*12=2*t, we may take $d=5$. (In most examples e and d will not be the same.)

The numbers d, p, and q are kept secret. They are used to create the D function. The numbers e and n are used to create the E function. The number e is usually called the public key and d the secret key. The number n is called the modulus. Once p and q are used to produce n and t, they are no longer needed and may be destroyed, but should never be made public.

To encrypt a message, one first converts the message into a string of integers, m_1, m_2, … all smaller than n. We then compute:

$$c_i = E(m_i) = m_i^e \text{mod } n$$

This means that we raise m_i to the e^{th} power and then divide by n. The remainder is $c_i=E(m_i)$. In our example, we suppose that the message is $m_1=9$. We compute:

$$\begin{aligned} c_1 &= 9^5 \text{mod } 21 \\ &= 59049 \text{ mod } 21 \end{aligned}$$

Because 59049=89979*21+18, we conclude that $c_1=18$ mod 21.

The decryption, or D function, is defined by:

$$D(c_i) = c_i^d \text{mod } n$$

In our example,

$$\begin{gathered} 18^d \text{mod } n \\ = 18^5 \text{mod } 21 \\ = 1889668 \text{mod } 21 \end{gathered}$$

As 1889568=889979*21+9, we conclude that $D(18)=9$, the message we started with.

To demonstrate mathematically that the decryption function always works to decrypt the message (i.e., that properties 1 and 2 above hold) requires a result from number theory called Euler's generalization of Fermat's little theorem. The reader is referred to any book on number theory for a discussion of this result.

The security of RSA depends on the resistance of n to being factored. Since e is made public, anyone who knows the corresponding d can decrypt any message. If one can factor n into its two prime factors, p and q, then one can compute t and then easily find d. Thus it is important to select integers p and q such that it is not likely that someone can factor the product *n*. In 1983, the best factoring algorithm and the best computers could factor a number of about 71 decimal (235 binary) digits. By 1994, 129 digit (428 bits) numbers were being factored. Current implementations of RSA generate *p* and *q* on the order 256 to 1024 bits so that *n* is about 512 to 2048 bits.

The reader should note that attacking RSA by factoring the modulus *n* is a form of algebraic attack. The algebraic weakness is that the factors of *n* lead to a discovery of the "secret key." A brute force attack, by definition, would try all possible values for *d*. Since *d* is hundreds of digits long, the work is on the order of 10^{100}, which is a prodigiously large number. Factoring a number, *n*, takes at most on the order of square root of n operations or about 10^{50} for a 100-digit number. While still a very large number it is a vast improvement over brute force. There are, as we mentioned, factoring algorithms that are much smaller, but still are not feasible to apply to numbers of greater than 500 bits with today's technology, or with the technology of the near future.

As you can see from our examples, using RSA requires a lot of computation. As a result, even with special purpose hardware, RSA is slow; too slow for many applications. The best application for RSA and other public key systems is as key distribution systems.

Suppose A wants to send a message to B using a conventional private key system such as DES. Assuming that B has a DES device, A has to find some way to get a DES cryptovariable to B. She generates such a key, *K*, through some random process. She then encrypts *K* using B's public algorithm, $E_B(K)$ and sends it to B along with the encrypted message $E_{DES}(M; K)$. B applies his secret function D_B to $E_B(K)$ and recovers *K*, which he then uses to decrypt $E_{DES}(M; K)$.

This technique greatly simplifies the whole key management problem. We no longer have to distribute secret keys to everyone. Instead, each person has a public key system that generates the appropriate *E* and *D* functions. Each person makes the *E* public, keeps *D* secret and we're done. Or are we?

85.8.1 The Man-in-the-Middle

Unfortunately there are no free lunches. If a third party can control the public listing of keys, or *E* functions, that party can masquerade as both ends of the communication.

We suppose that A and B have posted their E_A and E_B, respectively, on a public bulletin board. Unknown to them, *C* has replaced E_A and E_B with E_C, his own encryption function. Now when A sends a message to B, A will encrypt it as $E_C(M)$ although he believes he has computed $E_B(M)$. *C* intercepts the message and computes $D_C(E_C(M)) = M$. He then encrypts it with the real E_B and forwards the result to B. B will be able to decrypt the message and is none the wiser. Thus this man in the middle will appear as B to A and as A to B.

The way around this is to provide each public key with an electronically signed signature (a certificate) attesting to the validity of the public key and the claimed owner. The certificates are prepared by an independent third party known as a certificate authority (e.g., VeriSign). The user will provide a public key (*E* function) and identification to the certificate authority (CA). The CA will then issue a digitally signed token binding the customer's identity to the public key. That is, the CA will produce $D_{CA}(ID_A, E_A)$. A person, B, wishing to send a message to A will obtain A's public key, E_A and the token $D_{CA}(ID_A, E_A)$. Since the CA's public key will be publicized, B computes $E_{CA}(D_{CA}(ID_A, E_A)) = ID_A, E_A$. Thus B, to the extent that he can trust the certification authority, can be assured that he really has the public key belonging to A and not an impostor.

There are several other public key algorithms, but all depend in one way or another on difficult problems in number theory. The exact formulations are not of general interest since an implementation will be quite transparent to the user. The important user issue is the size of the cryptovariable, the speed of the computation, and the robustness of the implementation. However, there is a new implementation that is becoming popular and deserves some explanation.

85.9 Elliptic Curve Cryptography

A new public key technique based on elliptic curves has recently become popular. To explain this new process requires a brief digression. Recall from the previous section, that the effectiveness of public key algorithms depend on the existence of very difficult problems in mathematics. The security of RSA depends, for example, on the difficulty of factoring large numbers. While factoring small numbers is a simple operation, there are only a few (good) known algorithms or procedures for factoring large integers, and these still take prodigiously long times when factoring numbers that are hundreds of digits long. Another difficult mathematical problem is called the discrete logarithm problem. Given a number b, the base, and x, the logarithm, one can easily compute b^x or $b^x \bmod N$ for any N. It turns out to be very difficult to solve the reverse problem for large integers. That is, given a large integer y and a base b, find x so that $b^x = y \text{ Mod } N$. The known procedures (algorithms) require about the same level of computation as finding the factors of a large integer. Diffie and Hellman[9] exploited this difficulty to define their public key distribution algorithm.

85.9.1 Diffie and Hellman Key Distribution

Suppose that Sarah and Tanya want to exchange a secret cryptovariable for use in a conventional symmetric encryption system, say a DES encryption device. Sarah and Tanya together select a large prime p and a base b. The numbers p and b are assumed to be public knowledge. Next Sarah chooses a number s and keeps it secret. Tanya chooses a number t and keeps it secret. The numbers s and t must be between 1 and $p-1$. Sarah and Tanya then compute (respectively):

$$x = b^s \text{Mod } p \text{(Sarah)}$$
$$y = b^t \text{Mod } p \text{ (Tanya)}$$

In the next step of the process Sarah and Tanya exchange the numbers x and y; Tanya sends y to Sarah, and Sarah sends x to Tanya. Now Sarah can compute

$$y^s = b^{ts} \text{Mod } p$$

And Tanya can compute

$$x^t = b^{st} \text{Mod } p$$

But,

$$b^{ts} \text{Mod } p = b^{st} \text{Mod } p = \text{K}$$

which becomes their common key. In order for a third party to recover K, that party must solve the discrete logarithm problem to recover s and t. (To be more precise, solving the discrete logarithm problem is sufficient to recover the key, but it might not be necessary. It is not known if there is another way to find b^{st} given b^s and b^t. It is conjectured that the latter problem is at least as difficult as the discrete logarithm problem.) The important fact regarding the Diffie-Hellman key exchange is that it applies to any mathematical object known as an Abelian group. (See Exhibit 85.12.)

Now we can get into the idea of elliptic curve cryptography, at least at a high level. An elliptic curve is a collection of points in the $x-y$ plane that satisfy an equation of the form

[9]Hellman, M.E. and Diffie, W. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* IT-22, 644–654.

Groups:

A group is a collection of elements, G, together with an operation * (called a "product" or a "sum") that assigns to each pair of elements *x*, *y* in G a third element *z* = *x***y*. The operation must have an identify element e with *e***x* = *x***e* = *x* for all x in G. Each element must have an inverse with respect to this identify. That is, for each *x* there is an *x*' with *x***x*' = *e* = *x*'**x*. Last, the operation must be associative. If it is also true that *x***y* = *y***x* for all x and y in G, the group is said to be commutative, or Abelian. (In this case the operation is often written as –.

EXHIBIT 85.12 Definition of Abelian groups.

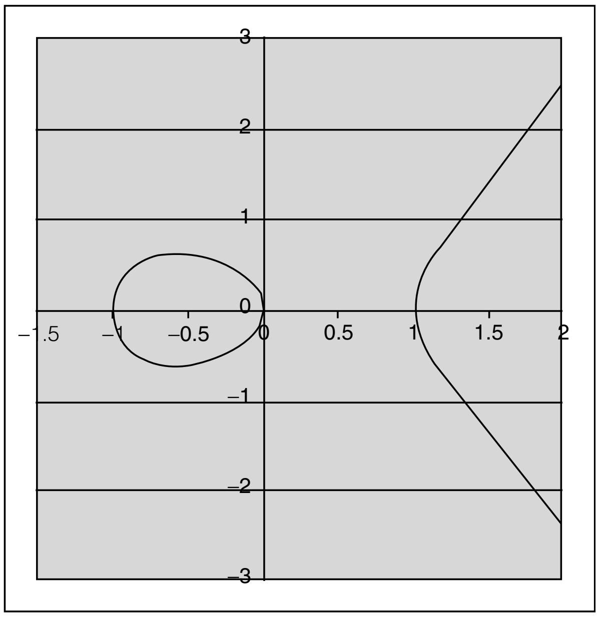


EXHIBIT 85.13 Graph of elliptic curve.

$$y^2 = x^3 + ax + b. \tag{85.1}$$

The elements a and b can be real numbers, imaginary numbers, or elements from a more general mathematical object known as a field. As an example, if we take $a = -1$ and $b = 0$. The equation is:

$$y^2 = x^3 - x. \tag{85.2}$$

A graph of this curve is shown in Exhibit 85.13. It turns out that the points of this curve (those pairs (x, y) that satisfy the equation 2) can form a group under a certain operation. Given two points $P = (x, y)$ and $Q = (x', y')$ on the curve we can define a third point $R = (x'', y'')$ on the curve called the "sum" of P and Q.

Furthermore this operation satisfies all of the requirements for a group. Now that we have a group we may define a Diffie-Hellman key exchange on this group. Indeed, any cryptographic algorithm that may be defined in a general group can be instantiated in the group defined on an elliptic curve. For a given size key, implementing an elliptic curve system seems to be computationally faster than the equivalent RSA. Other than the speed of the implementation there does not appear to be any advantage for using elliptic curves over RSA. RSA Data Security Inc. includes an elliptic curve implementation in their developer's kit (BSAFE) but they strongly recommend that the technique not be used except in special circumstances. Elliptic curve cryptographic algorithms have been subjected to significantly less analysis than the RSA algorithm so it is difficult to state with any confidence that elliptic curves are as secure or more secure than RSA. See Koblitz[10] for a complete discussion.

85.10 Conclusions

This short chapter presented a quick survey of some basic concepts in cryptography. No attempt was made to be comprehensive; the object was to help the reader better understand some of the reports about encryption and "breaking encryption systems" that often appear in the trade press and newspapers. The reader is referred to any of the many fine books that are available for more detail on any of the topics presented.

[10]Koblitz, N. 1994. *A Course in Number Theory and Cryptography. 2nd Ed.*, Springer-Verlag.