Chapter XI
Introduction to Cryptography

Rajeeva Laxman Karandikar
Indian Statistical Institute, India

ABSTRACT

The chapter introduces the reader to various key ideas in cryptography without going into technicalities. It brings out the need for use of cryptography in electronic communications, and describes the symmetric key techniques as well as public key cryptosystems. Digital signatures are also discussed. Data integrity and data authentication are also discussed.

INTRODUCTION

With a many-fold increase in digital communication in the recent past, cryptography has become important not only for the armed forces, who have been using it for a long time, but for all the aspects of life where Internet and digital communications have entered. *Secure and authenticated communications* are needed not only by the defense forces but, for example, in banking, in communicating with customers over the phone, automated teller machines (ATM), or the Internet.

Cryptography has a very long history. Kahn (1967) describes early use of cryptography by the Egyptians some 4,000 years ago. Military historians generally agree that the outcomes of the two world wars critically depended on breaking the codes of secret messages. In World War II, the breaking of the Enigma code turned the tide of the war against Germany. The term cryptography comes from the Greek words kryptós, meaning "hidden," and gráphein, meaning "to write." The first recorded usage of the word "cryptography" appears in Sir Thomas Browne's Discourse of 1658 entitled "The Garden of Cyrus," where he describes "the strange Cryptography of Gaffarel in his Starrie Booke of Heaven."

This chapter provides an introduction to the basic elements of cryptography. In the next section, we discuss the need for cryptography. The following four sections describe the four pillars of cryptology: confidentiality, digital signature, data integrity, and authentication. The final section concludes the chapter.

WHY WE NEED CRYPTOLOGY

First, if a company that has offices in different locations (perhaps around the globe) would like

to set up a link between its offices that guarantees secure communications, they could also need it. It would be very expensive to set up a separate secure communication link. It would be preferable if secure communication can be achieved even when using public (phone/Internet) links.

Second, e-commerce depends crucially on secure and authenticated transactions–after all the customers and the vendors only communicate electronically, so here too secure and secret communication is a must (customers may send their credit card numbers or bank account numbers). The vendor (for example, a bank or a merchant), while dealing with a customer, also needs to be convinced of the identity of the customer before it can carry out instructions received (say the purchase of goods to be shipped or transfer of funds). Thus, authenticated transactions are required. Moreover, if necessary, it should be able to prove to a third party (say a court of law) that the instructions were indeed given by said customer. This would require what has come to be called a *digital signature*. Several countries have enacted laws that recognize digital signatures. An excellent source for definitions, description of algorithms, and other issues on cryptography is the book by Menezes, van Oorschot, & Vanstone (1996). Different accounts can be found in Schneier (1996), and Davies and Price (1989).

Thus, the objectives of cryptography are:

1. **Confidentiality-secrecy-privacy:** To devise a scheme that will keep the content of a transaction secret from all but those authorized to have it (even if others intercept the transcript of the communication, which is often sent over an insecure medium).
2. **Digital signature:** Requires a mechanism whereby a person can sign a communication. It should be such that at a later date, the person cannot deny that it (a communication signed by him) was indeed sent by him.
3. **Data integrity:** Requires a method that will be able to detect insertion, substitution, or deletion of data (other than by the owner). (Say on a Web server or in a bank's database containing the information such as the balance in various accounts.)
4. **Authentication:** Two parties entering into a communication identify each other. This requires a mechanism whereby both parties can be assured of the identity of the other.

CONFIDENTIALITY-SECRECY-PRIVACY: ENCRYPTION

Encryption is necessary to secure confidentiality or secrecy or privacy. This requires an understanding of the encryption process. Most of such encryption in the past involved linguistic processes.

Consider the following example. Suppose two persons, A and B, would like to exchange information that may be intercepted by someone else. Yet A and B desire that even if a transmitted message is intercepted, anyone (other than A and B) should not be able to read it or make out what it says. Two friends may be gossiping or two senior executives in a large company may be exchanging commercially sensitive information about their company. This may be executed via e-mail (which can be intercepted rather easily). The most widespread use of secure communication is in the armed forces, where strategic commands are exchanged between various officers in such a fashion that the adversary should not be able to understand the meaning, even if they intercept the entire transcript of communication.

Let us first see how this objective could be achieved. Consider a *permutation* of the 26 letters of the Roman alphabet:

```
abcdefghijklmnopqrstuvwxyz
sqwtynbhgzkopcrvxdfjazeilm
```

Suppose that A and B both have this permutation (generated randomly). Now when A would

like to send a message to B, he/she replaces every occurrence of the letter a by the letter s, letter b by q, and so on (a letter is replaced by the letter occurring just below it in the list given). Since B knows this scheme and also has this permutation, he can replace every letter by the letter occurring just above it in the list, and he can recover the message. This scheme has the disadvantage that word lengths remain the same and thus could be a starting point for *breaking the code.* This could be done using the same techniques that linguists have used to decode ancient scripts based on a few sentences written on stone. The word length remaining the same could be rectified by adding a *space* to the character set used (for ease of understanding we will denote a space by &). We will also add the punctuations , . and ?. Let us also add the 10 digits 0,1,2,3,4,5,6,7,8,9. Thus, our character set now has 40 elements. Let us write the character set and its permutation (randomly generated) as follows:

abcdefghijklmnopqrstuvwxyz0123456789&.,?
s&q69w5ty.n,b4hg0zk7opc8r?vxd1fjaze2ilm3

Now the coding scheme is a goes to s, b goes to & (space) and so on. Now even the word lengths would not be preserved, and so the attack based on word lengths would not be a threat. However, if a rather long message is sent using this scheme, say 15 pages of English text, then the scheme is not safe. A statistical analysis of the coded message would be a giveaway. It is based on the observation that frequencies of characters are different: vowels, in particular the letter e, occur most frequently in any large chunk of English text. Thus, word frequencies in the encoded text are a starting point for an attack in an attempt to recover an original message.

These are naive examples of encoding schemes. We can construct more complicated schemes, thereby making it more difficult for an attacker to recover an original message. For example, instead of one character at a time, we can form words of two characters, and have a permutation of two character words that could act as an encoding scheme. Perhaps too difficult to encode or decode manually, machines or computers could be used for these operations. For schemes that have these characters as the basic alphabet, linguists, along with mathematicians, could attempt to break the code, as was done for the Wehrmacht Enigma cipher used by Nazis during World War II. Different Enigma machines have been in commercial use since the 1920s. However, the German armed forces refined it during the 1930s. The breaking of the Werhmacht version was considered so sensitive that it was not even officially acknowledged until the 1970s.

Today, the information to be communicated is typically stored on a computer, and is thus represented using a binary code (as a string of 0s and 1s). So coding and decoding is of strings of 0s and 1s and in such a scheme, linguists have a minimal role, if any.

BASIC NUTS AND BOLTS OF CRYPTOLOGY

Let us now introduce terminology that we will use in the rest of the article. The message to be sent secretly is called *plaintext* (though it could be say a music file, or a photograph). We will assume that the text or the music file or photograph has been stored as a file on a storage device (using a commonly used format such as ASCII, mp3, or jpg). We will regard the plaintext as a string of 0s and 1s. The scheme, which transforms the plaintext to a secret message that can be safely transmitted, is called an *encryption algorithm,* while the encrypted message (secret message) is called the *ciphertext*. The shared secret, which is required for recovery of original plaintext from the ciphertext, is called the *key.* The scheme that recovers the ciphertext from plaintext using the key is called *decryption algorithm*.

The encryption/decryption algorithms that require a common key to be shared are known as the *symmetric key ciphers*. In this framework, there is an algorithm, Encrypt, that takes a plaintext M_0 and a key K_0 as input and outputs ciphertext C_0 :

Encrypt: $(M_0, K_0) \longrightarrow C_0$

and an algorithm, Decrypt, that takes a ciphertext C_1 and a key K_1 as input and outputs plaintext M_1:

Decrypt: $(C_1, K_1) \longrightarrow M_1$.

The two algorithms, Encrypt and Decrypt, are related as follows: If the input to Decrypt is $C_{0,}$ the output of Encrypt, and the key is $K_{0,}$ the same as the key used in Encrypt, then the output of Decrypt is the plaintext M_0 that had been used as input to Encrypt. Thus, if A and B share a key K, A can take a plaintext M, use Encrypt with input (M,K) to obtain ciphertext C, and transmit it to B. Since B knows K, using (C,K) as input to Decrypt, B gets back the original message M that A had encrypted. The important point is that even if an interceptor obtains C, unless he has the original key K that was used as input to Encrypt, the message M cannot be recovered.

An adversary will try to systematically recover plaintext from ciphertext, or even better, to deduce the key (so that future communications encrypted using this key can also be recovered). It is usually assumed that the adversary knows the algorithm being used (i.e., the functions Encrypt and Decrypt), and he has intercepted the communication channel and has access to the ciphertext, but he does not know the true key that was used. This is the worst-case scenario. The algorithm has to be such that the adversary cannot recover the plaintext, or even a part of it in this worst-case scenario. The task of recovering the message without knowing the key or recovering the key itself is called cryptanalysis.

Here are different situations against which we need to guard the algorithm depending upon the usage:

- A *ciphertext-only attack* is one where the adversary (or cryptanalyst) tries to deduce the decryption key or plaintext by only observing ciphertext.
- A *chosen-plaintext attack* is one where the adversary chooses plaintext and is then given corresponding ciphertext in addition to the ciphertext of interest that he has intercepted. One way to mount such an attack is for the adversary to gain access to the equipment used for encryption (but not the encryption key, which may be securely embedded in the equipment).
- An *adaptive chosen-plaintext attack* is a chosen plaintext attack wherein the choice of plaintext may depend on the ciphertext received from previous requests.
- A *chosen-ciphertext attack* is one where the adversary selects the ciphertext and is then given the corresponding plaintext. One scenario where such an attack is relevant is if the adversary had past access to the equipment used for decryption (but not the decryption key, which may be securely embedded in the equipment), and has built a library of ciphertext-plaintext pairs. At a later time without access to such equipment, he will try to deduce the plaintext from (different) ciphertext that he may intercept.

One kind of attack that an adversary can always mount (once he knows the algorithm being used) is to sequentially try all possible keys one by one, and then the message will be one of the outputs of the decryption function. It is assumed that based on the context, the adversary has the capability to decide which of the decrypted outputs is the message. Thus, the total number of possible keys has to be large enough in order to rule out exhaustive search. Note that if all keys of p-bits

are allowed, then p = 128 would suffice for this purpose, for now, as there will be 2^{128} possible keys. Let us examine why.

Suppose we have a machine that runs at 4GHz clock speed, and we have an excellent algorithm that decides in one cycle if a given p-bit string is the key or not. Then in 1 second, we will be able to scan through $4 \times 1024 \times 1024 \times 1024 = 2^{32}$ keys. In 1 year, there are approximately 2^{25} seconds, and thus in 1 year, we can scan 2^{57} keys. Even if we use 1,000 computers in parallel, we would still have covered only 2^{67} keys. Thus, the fact that there are 2^{128} possible keys assures us that exhaustive search will not be feasible or *exhaustive search is computationally infeasible*. While designing crypto algorithms, the designer tries to ensure that the total number of possible keys is so large that exhaustive search will take a very long time (given the present computing power), and at the same time ensuring that no other cryptanalytic attacks can be mounted.

We will now describe a commonly used family of ciphers, known as *stream ciphers*.

Many of the readers may be aware of *pseudo random number generators*. These are algorithms that start from a seed (or an initial value, typically an integer) that generates a sequence of 0s and 1s that appear to be random or generated by tossing of fair coin, where 0 corresponds to tails and 1 corresponds to heads. Another way to put it is the output is indistinguishable from the output of a sequence of fair coin tosses. For any integer N, this algorithm, with the seed as input, produces x_1, x_2,… , x_N, where each x_i is 0 or 1. Such algorithms are part of every unix/Linux distribution, and also of most C/C++ implementations.

Suppose A wants to send plaintext (message) m_1, m_2, …, m_N (each m_i is either 0 or 1) to B. Let the shared secret key be an integer K. Using K as the seed, it generates random bits $x_1, x_2, \ldots, x_N$ and defines:

$$c_i = x_i \oplus m_i$$

(Here $\oplus$ is the addition modulo 2, $0 \oplus 0=0$, $0 \oplus 1=1$, $1 \oplus 0=1$, $1 \oplus 1=0$).

Now $c_1, c_2, \ldots, c_N$ is the ciphertext that can be transmitted by A to B. On receiving it, B uses the shared key K as the seed, generates the bits $x_1, x_2, \ldots, x_N$ and computes:

$$d_i = c_i \oplus x_i.$$

It can be verified easily that for all a, b in {0,1}, $(a \oplus b) \oplus b=a$, and thus $d_i = (x_i \oplus m_i) \oplus m_i$ for all i.

Thus, having access to the same random number generator algorithm and the same seed enables B to generate the same random bit-sequence x_1, x_2,… , x_n and thereby recover the message m_1, m_2,…, m_N from the ciphertext c_1, c_2,…, c_N.

It should be noted that even if an adversary knew the algorithm or the scheme being used including the random number generator, as long as she does not know the secret shared K, the generation of $\{m_i\}$ would not be possible, and hence recovery of $\{x_i\}$ will not be possible.

So the strength of this algorithm is in the pseudorandom number generator. There are algorithms that are very good for simulation (for example, the Mersenne Twister algorithm), but are not good for using in a stream cipher in the manner described, since the entire sequence can be computed if we know a few of the previous bits. This can be utilized to mount an attack on a stream cipher based on the Mersenne Twister random number generation algorithm. There are other methods to generate pseudorandom numbers for cryptographic purposes that yield good stream ciphers. Most commonly used stream ciphers are based on linear feedback shift registers (LFSR). Several LFSRs are combined via a nonlinear combining function to yield a good random bit generator. The stream cipher encrypts the binary digits of the plaintext one by one using an encryption transformation that varies with time or the position of the bit to be encrypted in the sequence (Golomb, 1967).

Another type of symmetric key (or shared key) cipher that is commonly used is a *block cipher*. This divides the plaintext into blocks of fixed size (m-bits, say), and transforms each block into another block (preferably of the same size) in such a way that the operation can be inverted (necessary for decryption). This transformation is dependent on the key, and thus decryption is possible only when we know the key that was used for encryption. The encryption transformation that operates on a block does not vary with the position of the block (in contrast with the stream cipher). Also, if the ciphertext encrypted using a block cipher is transmitted over a noisy channel, a single transmission error would lead to erroneous decryption of the entire block. Thus, errors propagate over an entire block. When ciphertext encrypted using a stream cipher is transmitted over a noisy channel, errors do not propagate. Apart from this advantage that errors do not propagate, stream ciphers are faster than block ciphers when implemented in hardware. However, as of now, no single algorithm is accepted as a standard. Lots of algorithms that are in use are proprietary.

In the early 1970s, IBM established a set of standards for encryption. It culminated in 1977 with the adoption as a U.S. Federal Information Processing Standard for encrypting unclassified information. The data encryption standard (DES) thus became the most well-known cryptographic mechanism in history. It remains the standard means for securing electronic commerce for many financial institutions around the world. (see, Menezes et al., 1996, chapter I). Block ciphers have been extensively studied from the point of view of cryptanalysis. Two techniques, called differential cryptanalysis and linear cryptanalysis, are used in cryptanalysis. In differential cryptanalysis, two plaintexts are chosen with specific differences, and each is encrypted with the same key. The resulting ciphertexts are then studied for possible mathematical relationships. If a relationship can be found that, in turn, can be used to mount an attack, it is thus a chosen plaintext attack. It is widely believed that designers of DES were aware of this technique and thus ensured that DES is secure against differential cryptanalysis. Linear cryptanalysis consists of studying the relationship between specific bits of plaintext, key, and ciphertext. If it can be established that such a linear relationship exists with high probability, it can be used to recover (a part of) the plaintext.

With increasing computing power, experts realized in the nineties that they need a new standard. Several proposals were considered and finally, in 2000, an algorithm named, *Rijndael* had been chosen by experts as a standard, now known as *dvanced Encryption Standard* or AES (see Daemen & Rijmen, 2002, also see AES1, AES2, AES3). It is used extensively in many communication devises. This uses a 128-bit key, and is considered secure for commercial transactions.

While symmetric key ciphers (stream-ciphers and block-ciphers) require that the encryption and decryption is done using the *same* key, thus requiring the sender and receiver to share the key, another framework, called *public key encryption*, does away with this requirement. Originally proposed by Diffie and Hellman (1976), this scheme consists of two algorithms, encryption algorithm and decryption algorithm, and uses a pair of (distinct) keys, one for encryption and one for decryption. The scheme works as follows: each person (or entity) in a group generates a pair of keys; one key, called the *public key,* is available in a directory of all members (or stored with a trusted third party), and the other key, called the *private key,* is kept secret by each member. A public key E_0 and the corresponding private key D_0 are related as follows: a message encrypted with the key E_0 can be decrypted using the key D_0.

Let us denote the public keys of A, B by E_A and E_B, and private keys by D_A and D_B respectively. When A wants to send a message M to B, A obtains the public key E_B of B from the directory of members, and then encrypts the message M using this key- E_B and sends the ciphertext (encrypted message) to B. Now since B knows his/her private

key D_B, he/she can decrypt the ciphertext using D_B and recover the message. Also, since D_B is known only to B, only B can recover the message.

Thus, public key encryption also has two algorithms, Encrypt and Decrypt. In this framework Encrypt that takes a plaintext M_0 and a key K_0 as input and outputs ciphertext C_0:

pubEncrypt: $(M_0, K_0) \longrightarrow C_0$

and an algorithm, Decrypt that takes a ciphertext C_1 and a key K_1 as input and outputs plaintext M_1:

pubDecrypt: $(C_1, K_1) \longrightarrow M_1$.

The two algorithms, pubEncrypt and pubDecrypt, are related as follows: Let K_0 be the public key of an individual and K_1 be the corresponding private key (of the same entity). If the ciphertext input to pubDecrypt is C_0 (the output of pubEncrypt) and the key is K_1, then the output of pubDecrypt is the plaintext M_0 that had been used as input to Encrypt. Note that in symmetric key encryption, the requirement was that K_0 is the same as K_1, where as in public key encryption, the requirement is that the pair (K_0,K_1) are respectively the public and private keys of the same entity.

Thus, A and B no longer need to share a key K, A only needs to know the public key K_0 of B and then he/she can take a plaintext M, use Encrypt with input (M,K_0) to obtain ciphertext C, and transmit it to B. Since B has the corresponding private key K_1, using (C, K_1) as input to Decrypt, B gets back the original message M that A had encrypted. The important point is that even if an interceptor obtains C, (and knows K_0, which is usually the case since the public key of all entities can be obtained) unless he/she has the corresponding private K_1, the message M cannot be recovered.

It is of course required that it should not be possible to compute the private key from the public key. Here, the phrase "not possible" is to be interpreted as computationally difficult in the sense that even several computers working in parallel would take years to compute the same. A commonly used algorithm used for public key cryptography is known as the RSA (The name RSA comes from the initials of the last names of authors Ron Rivest, Adi Shamir, and Len Adleman, who first described it). Yet another algorithm is ElGamal. The ElGamal algorithm is an asymmetric key encryption algorithm for public key cryptography, which is based on the Diffie-key agreement. Taher Elgamal discovered it in 1984.

The RSA was proposed in 1977. Initially, the computational requirements were thought to be so large, that it remained a mathematical curiosity. In fact, Clifford Cocks, in the British Intelligence Agency, proposed the same algorithm in 1973. MIT took out a patent in 1983 (which expired in 2000). Had Cocks' work been made public, it would not have been possible for MIT to patent it.

RSA is based on the widely accepted belief among experts that if p and q are two *large* prime numbers and n is their product, then given n is computationally difficult to factorize n (i.e., given n to determine p,q). What is large depends on the computational power available. Even in the late nineties, 24-bit primes, (prime numbers which in binary would require 24 bits to represent, roughly 8 digits in the decimal system) were considered very safe (p, q have to satisfy some additional restrictions). In this case n is about 48 bits. Currently, 64-bit primes are considered large enough for commercial applications, though for military and defense application, 512- or 1024-bit primes are used.

Why not use only public key cryptography and forget about symmetric key cryptography, since the former does not require sharing of a common secret key? Moreover, why not use 1024-bit primes even for commercial applications? The answer to both of these questions lies in the fact that the computational power required to encrypt or

decrypt a message using public key cryptography is high, and it grows with the size of n. So public key cryptography has its limitations. Also, for a commercial communication involving a few thousand dollars, the adversary is not going to spend huge sums on breaking the code, whereas when it comes to a nation's security, the adversary can (and probably will) have a lot more resources (in terms of computing power) to try and break the code. Thus, we see that we can increase the *cost* of breaking the code for the adversary, but in turn we have to put in more resources for encryption and decryption. So like many other things in life, we have to strike a balance between costs and benefits.

So if a small piece of plaintext is to be sent, it can be encrypted using RSA. But if a long plaintext (say several pages of text along with some data files) is to be sent, it would be too time (and resource) consuming to encrypt it using RSA and we have to use a symmetric key cipher. An interesting solution is to use a combination of both. First a (random) key K of required size (for the chosen symmetric key cipher) is generated by the sender, and then this key K is encrypted using RSA and sent to the receiver, who retrieves K. Subsequently, the plaintext is encrypted using the agreed symmetric key algorithm and the key K and the ciphertext so generated is sent. The receiver already has K, and so she can decrypt it to get the plaintext.

In the evolution of the Internet, secrecy and authentication were not built into the design. Thus, unless otherwise instructed, the information is exchanged as it is and can be retrieved easily as other users also have access to the "packets" of information that are being exchanged. To be specific, say a student or a professor at a university accesses the server at her university via Telnet. After typing the login id, she types her password, which on her terminal shows as ******** so that someone looking at her screen cannot see it. However, it is very easy for someone at a university with access to another terminal on the same local area network (LAN) to capture the entire transcript of her session (all communications between her terminal and the server) including her password. The same is the story when someone accesses his or her Web-mail account through the "http" protocol from a browser. Thus, if someone gives her credit card number to a vendor while she has connected over a connection that is not "secure," the same information can be trapped by anyone who is monitoring the traffic.

To work around this is to build a secure layer over the Internet. Let us explain this with ssh, namely "secure shell" (which replaces Telnet), and sftp, "secure FTP" (which replaces FTP). Both ssh and sftp are built in most Linux distributions and are also available for Windows/Mac operating systems. At installation, the ssh client as well as the ssh server both generate a public key/private key pair. When an ssh client seeks a connection to an ssh server, they agree on one symmetric key algorithm and one public key algorithm to be used that are supported by both the server and client, then exchange their public keys. The ssh client then generates a session key for the agreed symmetric key cipher, encrypts it with the server's public key using the agreed public key algorithm, and transmits the resulting ciphertext. The ssh server now decrypts this and has the session key for a symmetric key cipher. Once this is done (all this is over in fraction of a second), all further communications are encrypted (and on receiving decrypted) using the session key. The same protocol applies when we use the sftp.

When a user accesses a Web site with an https protocol, the same steps as are carried out (provided the Web site is running the https server and the browser at the client end has the capability). So over a secure https connection, secret information can be transmitted safely. The user still has to ensure that others do not have access to his/her computer for the private key; (in principle) the session key as well as the information transmitted can be recovered from the user computer unless special precautions are taken. This now brings us

to an important question. When a user accesses a Web site that he/she believes to be his/her bank and receives (or in the background, the browser receives) what is supposed to be his/her banks public key, how can he/she protect against the possibility that someone has intervened and replaced the bank's public key with another key? If indeed this happens, the user will send the session key to the imposter and subsequently, the information sent (could be his/her PIN or credit card number or other identification information) may be compromised. Thus there should be a way for the user to verify that the public key indeed belongs to the bank. This can be achieved via *digital signatures,* which we now discuss. This was introduced by Diffie and Hillman (1976).

DIGITAL SIGNATURE

When two persons/entities enter into a contract over the Internet (an e-contract), it is necessary that they put their *signature* on the contract so that, at a later date, either party cannot repudiate the fact that they had agreed to the terms of the contract. Of course this cannot be achieved by appending a signature at the end of the e-contract as in the paper contract since, in the e-form, parts can be changed, leaving the rest intact.

A digital signature will have to be something that binds the identity of signatory and the contents of the document being signed. The framework discussed earlier about public key cryptography can be used as a framework for the digital signature.

Once again we assume that various individuals (and entities) generate a public key and a private key pair for an agreed algorithm, say RSA. A trusted third party stores identities of individuals, along with their public keys. Suppose A wants to send a digitally signed document to B. Now A encrypts the document using his/her private key, and then the resulting ciphertext is now taken as the digitally signed document and sent to B. On receiving the document, B can obtain A's public key and recover the document and, at the same time, be assured that it was signed by A. Moreover, in case of a dispute later, B can prove to a third party, say a court, that indeed A had signed the document; all he/she has to do is to preserve the ciphertext received and produce that as evidence.

Let us closely examine the last statement. The argument goes as follows: The ciphertext received is such that when decrypted using the public key of A it yields a meaningful text. Thus, it could only have been generated by someone who has access to A's private key. Since it is assumed that only A knows his/her private key, it must have been signed by A.

At the outset it looks to be a weak argument. The probability that someone can produce a bit stream that, when decrypted with A's public key, would yield meaningful text is very small, therefore, it must have been generated by A. It may still seem a rather weak argument, but it is accepted and indeed recognized by law in several countries. Indeed, this is the backbone of all e-commerce and e-transactions.

Let us assume that all individuals or entities that may exchange signed messages store their public key with a trusted third party. This trusted third party (TTP) itself generates a private key/public key pair, and its public key is available with each user. For each user, the TTP generates a digital certificate, it generates a document giving the identity of the user and the users public key, and signs it with its own private key. When A sends a digitally signed document to B, A also sends the digital certificate issued by TTP. Then B first decrypts the certificate using TTP's public key, thus recovering the identity as well as the public key (authenticated) of the sender, and can proceed to decrypt the ciphertext using the public key of the sender to recover the signed message.

Note that this system requires that there exists a trusted third party whose public key is known to all users; each user gets a digital certificate

from TTP that binds his/her identity with his/her public key. Beyond this, the system does not require anything else. The individual users communicating with each other may be total strangers to each other.

In the scenario described, anyone who intercepts the communication can recover the document since all it needs is the public key of the sender. However, the same infrastructure can be used to have secret authenticated communication as follows: let KA1, KA2 be the public key and the private key of A ,and KB1 and KB2 be the public key and the private key of B. Suppose A wishes to send B a secret document, say M, signed by A. He/she follows the following steps.

Sign the document (using his/her private key) to generate C1, which is the signed document as described:

pubEncrypt: (M , KA2) $\longrightarrow$ C1.

Encrypt C1 using public key of B to generate signed ciphertext C2:
pubEncrypt: (C1, KB1) $\longrightarrow$ C2.

Transmit C2 to B:
B can now decrypt the transmitted message C2 using his/her private key:
pubDecrypt: (C2, KB2) $\longrightarrow$ C3.

C3 is the same as C1. Thus, B has the digitally signed document C1. Now B can get the document (message) M by decrypting it using the public key of A. Moreover, B can retain the signed document C1 as proof that, indeed, A has signed the document M. Even if an adversary intercepts the transmission C2, (and knows or can find out the public keys of both A and B, but does not know the private key of B), he/she cannot recover C1 from C2, and so he/she cannot decrypt to get M.

Thus, the infrastructure for public key cryptography, a TTP, with a registry of users' identities and their public keys (while the users keep their private keys to themselves), suffices for digital signatures.

We had earlier remarked that if a large document is to be encrypted using a public key cryptoprotocol, say RSA, it needs a huge computational effort for encryption as well as decryption. Instead, it is common to exchange a session key for a symmetric key algorithm via public key algorithm, and then encrypt the message using this session key and symmetric key algorithm.

This does not work for a digital signature. The signature has to be something that depends on the full document and the identity of the person signing the document. However, there is one way to avoid signing the full document; it uses the notion of *hash function,* which we will discuss next.

DATA INTEGRITY

One way to ensure data integrity is to use hash functions. A hash function, or more precisely cryptographic hash function, h, is a function that takes as an input a message M (or a file) of arbitrary length and produces an output h(M) of fixed length, say n bits. The output h(M) is often referred to as the hash or hash value of M. Of course this would mean that there are bound to be collisions, that is, two distinct messages M_1, M_2 can lead to identical hash values ($h(M_1)=h(M_2)$). However, if the size of the output n is large, say 1,024, then the chance that two messages M_1, M_2 (chosen independently) could lead to the same hash value is very small – $2^{\{-1024\}}$. This is smaller than the probability that two randomly chosen persons have the same DNA!

Uses of the hash function: Suppose two users have downloaded a large file from the Web and, at a later date, would like to know if the two files are identical? Suppose they are in two locations far away from each other and can communicate over the Internet. Can this be done without the whole file being transferred from one location

to the other? Yes, the two users can compute the hash value of their files and then compare the hash values. If they are identical, the chances are very high that the two files are the same. Another usage is in the distribution of public domain software, say *FileUtilities,* that is downloadable from several Web sites. How does the distributor and the buyer ensure that someone else is not passing off a malicious software in the name of *FileUtilities* name that has a virus embedded in it? One way to do this is (that along with the executable file) the creator publishes its hash value and before installing the software, the buyer cross-checks the hash value of the downloaded version with the value published by the creator.

In order that a hash function be useful for this purpose given, it is necessary that a change in the input leads to an unpredictable change in the hash value. If not, a malicious person can first change the file and then make other changes that have no effect on the virus, but lead to the same hash value as the original.

Desirable properties of a hash function:

- **Pre-image resistance:** given any y for which a corresponding input is not known, it is computationally infeasible to find any pre-image M such that h(M) =y.
- **2nd-pre-image resistance:** given M, it is computationally infeasible to find a 2nd-pre-image M* different from M such that h(M) = h(M*).
- **Collision resistance:** It is computationally infeasible to find any two distinct inputs M_1, M_2 that hash to the same output, that is, such that $h(M_1) = h(M_2)$. (Note that here there is free choice of both inputs.)

If h is a hash function with these properties, then for a message (or file) M, its hash value h(M) can be taken as a representative for the purposes described. Thus, if we are presented a copy M* of a message M (M* may be sent over an insecure channel, which is subject to tampering) and the hash value y=h(M) (say sent over a secure channel), we can compute the hash value h(M*), and if this equals y, we can conclude that M* is indeed a copy of the message M. This would follow from the 2nd pre-image resistance property.

Hash functions can be used for data integrity checks as follows: as soon as an authorized modification of a database (or some other data source) is carried out, the administrator can generate its hash value and store it securely. Now any modification (addition, deletion) would change the database and would disturb the hash value. Thus, the administrator can periodically compute the hash value and compare it with the value stored with him. If the hash values match, the administrator can be (almost) certain that the data has not been altered.

Hash functions can also be used in the context of a digital signature, where we wish to avoid huge computations involved in digitally signing a large document. Given a hash function h with the properties of pre-image resistance, 2nd-pre-image resistance, and collision resistance, in order to digitally sign a message M, first the hash value y of M is computed (h(M) = y) and y is digitally signed using RSA. Note that irrespective of the size of M, y has a fixed predetermined size. Of course M cannot be recovered from y, but M is sent along with the digitally signed copy of y. Now y is recovered from its signature and the hash value of M is computed. If these two quantities coincide, we can conclude that M was signed by the person who signed y.

We will now see the reasons for demanding the listed properties of the hash function. Here h should be 2nd-pre-image resistance, otherwise, an adversary may observe the signature of A on h(M), then find an $M^{\#}$ such that $h(M) = h(M^{\#})$, and claim that A has signed $M^{\#}$. If the adversary is able to actually choose the message that A signs, then C need only find a collision pair $(M, M^{\#})$ rather than the harder task of finding a second pre-image of $y=h(M^{\#})$, thus collision resistance is also required. This forgery may be of concern if the attacker can

find a pair (M, $M^{\#}$) with same hash such that M seems harmless and so the person agrees to sign it, while $M^{\#}$ is something that the adversary could use (say a promise to pay $10,000!).

Number theory has played an important role in development and analysis of various algorithms described. As the computing power increases, there is need to develop new algorithms that need a lot more computing power to break, especially in defense applications. Elliptic curves are playing an increasing role in developments on the cryptography front. These days, the computing power available on desktops (and other low-cost computing environments such as Linux clusters) is large and is increasing. Thus, elliptic curve-based algorithms would become essential even for commercial algorithms.

AUTHENTICATION

All users of e-mail are used to logging into their accounts, where they choose a user name (or user_id) and then a password at the time of account set up and then, subsequently, when they wish to access their account, they have to identify themselves to the system by providing the user_id and password. The system checks this combination with the stored records and if it matches, then the user is allowed to access the account. In Unix/Linux systems, the password is not stored directly, but its hash value is stored (in some implementations, the password and user_id are concatenated to produce a string whose hash is stored). The same operation is done when a user tries to access the account and if hash values match, the access is allowed. A disadvantage of this system is that once a user has given password, someone having access to his/her system can trap it and, thereafter, impersonate him/her.

There are alternatives to this scheme; these are interactive and are based on challenge-response, Here when a user is trying to identify and authenticate himself, the system sends one or more challenges (a different set each time) and the user is to give an appropriate response. One such protocol, Feige-Fiat-Shamir (FFS) protocol (Feige, Fiat, & Shamir, 1988) relies on the computational difficulty of the problem of finding the square root modulo composite integers n that are the product of two large "RSA like" primes p,q (n=pq). This, and some other identification protocols, are known as *zerokKnowledgepProtocols(ZK)*.

They have a property that, at the end of the identification, they have passed on no other information than the identification itself. Thus, anyone having the full transcript of the communications by one user A over a period of time would still not be able to impersonate A.

We will discuss a simpler version of FFS called Fiat-Shamir protocol (Fiat & Shamir, 1987).

Setup time:

- A trusted third party generates RSA-like modulus n=pq and publishes n, but keeps p, q secret.
- Each user generates a random number s, $0<s<n$ (not equal to p,q) and computes $v=s^2$ modulo n and registers v (the users signature) with the trusted third party. The users signature is public knowledge.

Identification protocol: A user (prover) identifies himself/herself to the trusted party (verifier) and the verifier retrieves the prover's registered signature v. The following steps 1-4 are carried out say 20 times.

1. The user (prover) generates a random number r, $0<r<n$ and computes $x=r^2$ and sends x (called a witness) to the trusted party.
2. The trusted party (verifier) generates e=0 or e=1 with probability ½ each and sends e to the prover. (e is called challenge).
3. If e=0, the prover sends y=r and if e=1 sends y=rs to the verifier.
4. If e=0 and $y^2=x$ or if e=1 and $y^2=xv$ then accept the prover's claim.

If the prover's claim is accepted in all the rounds, then the prover's identity is accepted. At first glance, it would appear strange that in step 2, the verifier may select e=0 and would require the user to send only the random number r. This step is required to prevent forgery: suppose the prover always generates e=1 and a malicious person knows this (and also v, which is public knowledge), then the user can generate a number t, compute $x=t^2/v$ (modulo n) and send as witness and in step 3, send y=t. Without knowing s, the malicious person would have satisfied the condition $y^2=xv$. In this case, if e=0 is allowed, then the malicious person will not be able to send the square root of $x= t^2/v$ (modulo n) in step 3.

This protocol can be thought of as follows: in step 1, the prover sends a witness x and claims to have answers to the questions "what is r?" and "what is y=rs?", but in one instance would answer any of the two questions but not both (as it would reveal his/her secret).

CONCLUDING COMMENTS

In this chapter, we have four pillars of cryptology: confidentiality, data signature, data integrity, and authentication. Thus, we have discussed the problems that can occur when data is transmitted over *insecure* lines.

There are other dangers as well. Starting from someone observing the keystrokes when a person enters a key or password via a keyboard, to someone running a simple program on a person's computer that traps (and record) all keystrokes, an adversary can recover a key or password that was typed. Likewise, unless one is careful, one can recover files that were deleted (even after the recycle bin on Windows operating system has been emptied!). Garfinkel and Shelat (2003) analyzed 158 second-hand hard drives. They found that less than 10% had been sufficiently cleaned, and a wide variety of personal and confidential information was found in the rest. Thus, there are simple measures that most people ignore to secure information. No amount of cryptographic safeguards can save us from human follies.

REFERENCES

Advanced Encryption Standard 1. (n.d.). Retrieved August 16, 2006 from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

Advanced Encryption Standard 2. (n.d.). Retrieved August 16, 2006 from http://csrc.nist.gov/CryptoToolkit/aes/round1/conf2/papers/biham2.pdf

Advanced Encryption Standard 3. (n.d.). Retrieved August 16, 2006 from http://csrc.ncsl.nist.gov/CryptoToolkit/aes/rijndael/misc/nissc2.pdf

Daemen, J. & Rijmen, V. (2002). *The design of Rijndael: AES—the advanced encryption standard.* Berlin: Springer-Verlag.

Davies, D. W., & Price, W.L. (1989). *Security for computer networks* (2nd ed.). New York: JohnWiley & Sons.

Diffie, W., & Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *IT-22*(6), 644-654.

Feige, U., Fiat, A., & Shamir, A. (1988). Zero-knowledge proofs of identity. *Journal of Cryptography, 1,* 66-94.

Fiat, A., & Shamir, A. (1987). How to prove yourself: Practical solutions to identification and signature problems. *Advances in Cryptology—Crypto '86* (pp.186-194). Berlin: Springer-Verlag.

Garfinkel, S. L., & Shelat, A. (2003). Rememberance of data passed: A study of disk sanitation practise. *IEEE Security and Privacy*, *1*(1), 17-27.

Golomb, S. W. (1982). *Shift register aequences.* San Francisco: Holden-Day. Walnut Creek: Aegean Park Press.

Kahn, D. (1967). *The codebreakers*. New York: Macmillan Publishing Company.

Menezes, A., van Oorschot, P., & Vanstone, S. (1996). *Handbook of applied cryptography*. Boca Raton: CRC Press.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), 120-126.

Schneier, B. (1996). *Applied cryptography: Protocols, algorithms, and source code in C* (2nd ed.). New York: John Wiley & Sons.

This work was previously published in E-Business Process Management: Technologies and Solutions, edited by J. Sounderpandian and T. Sinha, pp. 28-44, copyright 2007 by IGI Publishing, formerly known as Idea Group Publishing (an imprint of IGI Global).