Information Technology Journal 3 (1): 106-132, 2004
ISSN 1682-6027

Data Security and Cryptographic Techniques–A Review

Kefa Rabah
Department of Physics, Eastern Mediterranean University, Gazimagusa,
North Cyprus, via Mersin 10, Turkey

Abstract: Since almost the beginning of time, it seems, man has had a need to keep information private and, in many situations, needed to decipher information previously made private by others. In our age of high technology these needs have grown exponentially and become more complex. Today, computer technology is on the verge of providing the ability for individuals and groups to communicate and interact with each other in a totally anonymous manner. Two persons may exchange messages, conduct business and negotiate electronic contracts without ever knowing the True Name, or legal identity, of the other. Interactions over networks will be untraceable, via extensive re-routing of encrypted packets and tamper-proof boxes which implement cryptographic protocols with nearly perfect assurance against any tampering. Reputations will be of central importance, far more important in dealings than even the credit ratings of today. These developments will alter completely the nature of government regulations: the ability to tax and control economic interactions, the ability to keep information secret and will even alter the nature of trust and reputation. All this will depend on how secure data can be moved from point-to-point from LAN to Global network data exchange. It is recognized that encryption (otherwise known as scrambling, enciphering or privacy transformation) represents the only means of protecting such data during transmission and, a useful means of protecting the content of data stored on various media, provided encryption of adequate strength can be devised and validated and is inherently integrable into network system architecture.

Key words: Cryptographic, data security, encrypted key exchange

A brief history of cryptography: Cryptography is concerned with methods for ensuring the secrecy and authenticity of messages. A cryptographic algorithm**,** also called a cipher**,** is a mathematical function used for encryption[1]. In most cases, two related functions are employed, one for encryption and the other for decryption. Encryption is the process of transforming information so that it is unintelligible to anyone but the intended recipient. Decryption is the process of transforming encrypted information so that it is intelligible again to the intended recipient. An eavesdropper who intercepts the transmitted message receives only "garbage" (the ciphertext), which makes no sense to him since he does not know how to decrypt it. The large volume of personal and sensitive information currently held in computerized and digital data banks and transmitted over global communication networks runs into billions of dollars making encryption increasingly important. With most modern cryptography, the ability to keep encrypted information secret is based not on the cryptographic algorithm, which is widely known, but on a number called a key that must be used with the algorithm to produce an encrypted result or to decrypt previously encrypted information.

The essence of cryptography is traditionally captured in the following problem: Two parties (the tradition is to call them Bob and Alice) wish to communicate over a public communication channel in the presence of malevolent eavesdropper (the tradition Eve). Bob and Alice could be military jets, e-business or just friend trying to have a private conversation. They can't stop Eve listening to their radio signals (or tapping their phone line, or whatever), so what can they do to keep their communication secret? One solution is for Alice and Bob to exchange a digital key, so they both know it, but it's otherwise secret. Alice uses this key to encrypt messages she sends and Bob reconstructs the original messages by decrypting with the same key. This situation is depicted in Fig. 1, in which we refer to the unencrypted message T as plaintext and to the encrypted message C as ciphertext. Decryption with the correct key is simple. Decryption of the messages by Eve without the correct key is very difficult and in some cases impossible for all practical purposes, but she can always attempt to reconstruct the

Corresponding Author: Kefa Rabah, Department of Physics, Eastern Mediterranean University, Gazimagusa, North Cyprus, via Mersin 10, Turkey

original messages through brute-force or cryptanalysis technique.

Cryptography offers a set of sophisticated security tools for a variety of problems, from protecting data secrecy, through authenticating information and parties, to more complex multi-party security implementation. Yet, the most common attacks on cryptographic security mechanisms are 'system attacks' where the cryptographic keys are directly exposed, rather than cryptoanalytical attacks (e.g., by analyzing the ciphertexts, i.e., the encrypted text) as depicted in Fig. 1. Such 'system attacks' are done by intruders (hackers, or through software trapdoors using viruses or Trojan horses), or by corrupted insiders. Unfortunately, such attacks are very common and frequently quite easy to perform, especially since many existing environments and operating systems are insecure (in particular Windows). In network data communication, the original texts (plaintexts), is usually encrypted into ciphertexts, which is then mailed and is decrypted by the end-user before it can be read.

Several techniques have been used throughout the years to protect data against enemies who would misuse the information[2]. Thousands of years ago, the main use of using encryption was to protect data during war. David Kahn, in his impressive work[3]. The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet, has traced the history of cryptography as far back as ancient Egypt, progressing through India, Mesopotamia, Babylon, World War I, World War II and into modern times, where encryption has taken on new meaning. The extensive use of telegraph and radio waves in modern times has increased the need to encrypt information because sophisticated techniques are available to intercept the information that flows in today's global network environment. Military communication without the use of encryption is worthless. The biggest achievements in cryptography can be attributed to the work done by Alan Turing during World War II. Using the help of Alan Turing in Britain, the allies were able to break the Enigma code used by Germany during the war[4]. Since World War II, cryptographic research and activity was the domain of various governmental National Security Agencies. For years, the use of codes and ciphers was reserved to the governmental and military operations in the west. Civilians had to be content with using envelopes and couriers to protect data on transit or stored.

However, with the advent of computer revolution and the explosion of the information age and especially the Internet, the need for encryption in civilian use was recognized. The manner in which data is disseminated through electronic mail, the Internet and the financial value attached to the information; fueled enough research for civilians to use encryption. In the late 1960s, IBM chairman Thomas Watson, Jr. set up a cryptographic research group. This group, led by Horst Feistel, developed a private key encryption method called Lucifer, which was used by Lloyd's of London to protect a cash-dispensing system[5]. Dr. Walter Tuchman and Dr. Carl Meyer, who tested the cipher and fixed the flaws they found in the method, headed the team formed for this purpose. The success of Lucifer prompted IBM to make it available for commercial use[6]. By 1974, the cipher was ready and available on a silicon chip. However, IBM was not the only company to make ciphers available commercially. Other companies made other codes available, however, there were still some problems associated with all these ciphers technologies: (I) they could not communicate with each other in real time and (ii) there was no way to determine their strength.

Threats to computer and digital systems: Every form of commerce ever invented has been subject to fraud, from rigged scales in a farmers' market to counterfeit currency and including phony invoices. Electronic commerce

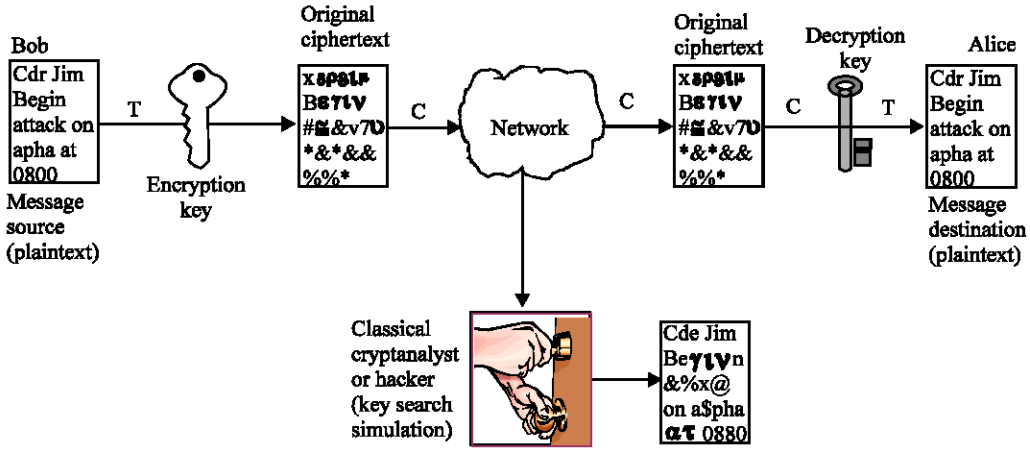

Fig. 1: Shows a schematic classical cryptographic data communication

schemes will also face fraud, through forgery, misrepresentation, denial of service and cheating. In fact, computerization makes the risks even greater, by allowing attacks that are impossible against non-automated systems[7]. Criminal attacks are often opportunistic; a thief can make a living skimming a penny from every Visa cardholder. You can't walk the streets wearing a mask of someone else's face, but in the digital world it is easy to impersonate others.

Privacy violations are another threat. Some attacks on privacy are targeted: a member of the press tries to read a public figure's e-mail, or a company tries to intercept a competitor's communications[8]. Others are broad data-harvesting attacks, searching a sea of data for interesting information: a list of rich widows, AZT users, or people who view a particular Web page. Lawyers sometimes need a system attacked; in order to harvest information to enable them prove their client's innocence. They often use considerable financial resources to buy equipment and hire experts to collect details on the system through the discovery process. And they don't have to defeat the security of a system completely, just enough to convince a jury that the security is flawed.

And of course, vandals and thieves routinely break into networked computer systems. Electronic vandalism is an increasingly serious problem. Even the most secure systems within National Security Agencies are not spared. Computer vandals have already graffitied the CIA's web page, mail-bombed Internet providers and canceled thousands of newsgroup messages[8]. The next biggest threat comes from cyberterrorism. The threat of cyberterrorism is causing much alarm these days. We have been told to expect attacks since 9/11; that cyberterrorists would try to cripple our power system, disable air traffic control and emergency services, open dams, or disrupt banking and communications network. But so far, nothing's happened. But let's not pat our backs and congratulate ourselves yet, it is just the early days of cyberwar. Just imagine for a minute the leadership of al Qaeda sitting in a cave somewhere, plotting the next move in their jihad against the civilized world. One of the leaders jumps up and exclaims: "I have an idea! We'll disable their e-mail...." The closest example we have of this kind of thing comes from Australia in 2000. Vitek Boden broke into the computer network of a sewage treatment plant along Australia's Sunshine Coast. Over the course of two months, he leaked hundreds of thousands of gallons of putrid sludge into nearby rivers and parks. Among the results were black creek water, dead marine life and a stench so unbearable that residents complained. This is the only known case of someone hacking a digital control system with the intent of causing environmental harm.

When security safeguards aren't adequate, trespassers run little risk of getting caught. Attackers don't follow rules; they cheat. They can attack a system using techniques the designers never thought of. Some attackers are motivated by publicity; they usually have significant resources via their research institution or corporation and large amounts of time, but few financial resources. Art thieves have burgled homes by cutting through the walls with a chain saw. Home security systems, no matter how expensive and sophisticated, won't stand a chance against this attack. Computer thieves come through the walls too. They steal technical data, bribe insiders, modify software and collude. They take advantage of technologies newer than the system and even invent new mathematical algorithms to attack the system with[9]. The odds favor the attacker. Bad guys have more to gain by examining a system than good guys. Defenders have to protect themselves against every possible type of vulnerability, but an attacker only has to find one security flaw to compromise the whole system. Only strong cryptography can protect against these attacks.

What cryptography can and can't do: No one can guarantee 100% security. But we can work toward 100% risk acceptance. Fraud exists in current commerce systems: cash can be counterfeited, checks altered, credit card numbers stolen. Yet these systems are still successful because the benefits and conveniences outweigh the losses. Privacy systems - wall safes, door locks and curtains - are not perfect, but they're often good enough. A good cryptographic system strikes a balance between what is possible and what is acceptable.

Strong cryptography can withstand targeted attacks up to a point - the point at which it becomes easier to get the information some other way. A computer encryption program, no matter how good, will not prevent an attacker from going through someone's garbage[10]. But it can prevent data-harvesting attacks absolutely; no attacker can go through enough trash to find every AZT user in the country. And it can protect communications against non-invasive attacks: it's one thing to tap a phone line from the safety of the telephone central office, but quite another to break into someone's house to install a bug.

The good news about cryptography is that we already have the algorithms and protocols we need to secure our systems[1,11]. The bad news is that was the easy part; implementing the protocols successfully requires considerable expertise. The areas of security that interact with public-key management, human/computer interface security and access controls-often defy analysis. Furthermore, the disciplines of public-key infrastructure,

software security, computer security, network security and tamper-resistant hardware design are often very poorly understood/implemented.

Companies often get the easy part wrong and implement insecure algorithms and protocols. But even so, practical cryptography is rarely broken through the mathematics; other parts of systems are much easier to break. The best protocol ever invented can fall to an easy attack if no one pays attention to the more complex and subtle implementation issues[12]. Netscape's security fell to a bug in the random-number generator. Flaws can be anywhere: the threat model, the system design, the software or hardware implementation and the system management. Security is a chain and a single weak link can break the entire system. Fatal bugs may be far removed from the security portion of the software; a design decision that has nothing to do with security can nonetheless create a security flaw.

Once you find a security flaw, you can fix it. But finding the flaws in a product can be incredibly difficult. Security is different from any other design requirement, because functionality does not equal quality. If a word processor prints successfully, you know that the print function works. Security is different; just because a safe recognizes the correct combination does not mean that its contents are secure from a safecracker. No amount of general beta testing will reveal a security flaw and there's no test possible that can prove the absence of flaws.

Threat models: A good design starts with a threat model: what the system is designed to protect, from whom and for how long[13,14]. The threat model must take the entire system into account - not just the data to be protected, but also the people who will use the system and how they will use it. What motivates the attackers? Must attacks be prevented, or can they just be detected? If the worst happens and one of the fundamental security assumptions of a system is broken, what kind of disaster recovery is possible? The answers to these questions can't be standardized; they're different for every system. Too often, designers don't take the time to build accurate threat models or analyze the real risks. Threat models allow both product designers and consumers to determine what security measures they need. Does it make sense to encrypt your hard drive if you don't put your files in a safe? How can someone inside the company defraud the commerce system? Are the audit logs good enough to convince a court of law? Moreover, you can't design a secure system unless you understand what it has to be secure against.

System design: Design work is the mainstay of the science of cryptography and it is very specialized. Cryptography blends several areas of mathematics: number theory, complexity theory, information theory, probability theory, abstract algebra and formal analysis, among others[1,15]. Few can do the science properly and a little knowledge is a dangerous thing: inexperienced cryptographers almost always design flawed systems. Good cryptographers know that nothing substitutes for extensive peer review and years of analysis. Quality systems use published and well-understood algorithms and protocols; using unpublished or unproven elements in a design is risky at best. Cryptographic system design is also an art. A designer must strike a balance between security and accessibility, anonymity and accountability, privacy and availability. Science alone cannot prove security; only experience and the intuition born of experience, can help the cryptographer design secure systems and find flaws in existing designs.

Implementation: There is an enormous difference between a mathematical algorithm and its concrete implementation in hardware or software. Cryptographic system designs are fragile. Just because a protocol is logically secure doesn't mean it will stay secure when a designer starts defining message structures and passing bits around. Close isn't close enough; these systems must be implemented exactly, perfectly, or they will fail. A poorly designed user interface can make a hard-drive encryption program completely insecure. A false reliance on tamper-resistant hardware can render an electronic commerce system all but useless. Since these mistakes aren't apparent in testing, they end up in finished products. Many flaws in implementation cannot be studied in the scientific literature because they are not technically interesting[15]. That's why they crop up in product after product. Under pressure from budgets and deadlines, implementers use bad random-number generators, don't check properly for error conditions and leave secret information in swap files. The only way to learn how to prevent these flaws is to make and break systems, again and again.

Cryptographic systems: Cryptographic systems are generically classified along three independent dimensions:

- The type of operations used for transforming plaintext to ciphertext. All encryption algorithms are based on three general principles: substitution, in which each element in the plaintext (bit, letters, group of letters etc.) is mapped into another element; permutation in which the elements are permuted around and transposition in which elements in the

plaintext are rearranged. Most systems, involve multiple stages of substitutions and transpositions.

- The number of keys used. If both sender and receiver use the same key, the system is referred to as symmetric, single-key, secret-key, private-key or conventional encryption. If the sender and receiver each uses different key, the system is referred to asymmetric, two-key, or public-key encryption.
- The way in which plaintext is processed. A block cipher processes an input block of elements at a time, producing an output block for each input block. A stream cipher processes the input elements continuously, producing output one element at a time, as it goes along.

Simple private-key cryptography–substitution technique: Most successful secret-key encryption techniques use a simple set of functions and procedures to convert the plaintext into cipher text. One such function employs substitution technique, which is very commonly used in cryptographic algorithms. The earliest known use of substitution cipher and the simplest, was by Julius Caesar[4]. The Caesar cipher involves replacing each letter of the alphabet with the letter standing three places further down in the alphabet. For example:

plaintext: COMMANDER JIM COMMENCE ATTACK ON ALPHA
ciphertext: CRPPDQGHU JLP FRPPHQFH DWWDFN RQ DOSKD

Note that alphabet is wrapped around, so that the letter following Z is A. We can write transformation by listing all possibilities, as shown in Table 1.

Table 1: The key of the Caesar algorithm

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

If we assign a numerical value to each letter, then the algorithm can be expressed as follows: For each plaintext letter p, substitute the ciphertext letter C: C = E(p) = (p+3) mod 26.

A shift may be of any amout, so that the general Caesar algorithm is, C = E(p) = (p+k) mod 26, where k (or key) takes on a value in the range 0 to 25. The decryption algorithm is simply, p = D© = (C-k) mod 26, (for software (SW) implementation see Appnd. A1). If it is known that a given ciphertext is a Caeser cipher, then cryptoanalysis is easily performed: simply try all the 25 possible keys. Three important characteristics of this problem enabled us to use a brute-force cryptanalysis: (I) the encryption and decryption algorithms are known; (ii) there are only 25 keys to try; and the language of the plaintext is known and easily recognizable.

Monoalphabetic ciphers: If, instead, the cypher line in Table 1, can be any permutation of the 26 alphabetic characters, then there are 26! possible keys. It would eleminate the brute-force method for crytoanalysis. There is, however, another line of attack. If the cryptoanalyst knows the nature of the plaintext (e.g. noncompressed English text), then the analyst can exploit the regularities of the language. As can be seen, the knowledge of the language and the context of the message give a hacker a lot of information about how to decipher the code, hence, the need for more secure crypto systems with unlimited strength.

Security of cryptosystems – the secure mechanics of cryptosystems: The explosion in the use of computers, the Internet and public-key cryptography has made it possible for individuals to protect their data using a variety of encryption techniques. In this section we will discuss various strategies used by cryptographic algorithms and its implementation in real time. These will include the analyses of some commonly used cryptographic techniques such as substitution; permutation, XOR and other cryptographic functions are also discussed. No matter which technique you choose, you must keep in mind that a desperate cryptanalyst can always decipher the message. Hence, you should always take *all* the necessary precautions to protect your data. Those precautions range from proper choice of cryptographic keys to physically protecting your assets and yourself[16].

Exclusive OR (XOR) encryption function and other alternative algorithms: The exclusive OR is an example of an encryption function, which is very popular method for performing simple block encryption[17]. The XOR-function is used to indicate that if there are two conditions (say condition A and condition B), then either condition A is true or condition B is true, but not both. The complete set of possibilities for two values being XORed and their result is as follows:

XOR(0,0) = 0; XOR(0,1) = 1; XOR(1,0) = 1; XOR(1,1) = 0

The best thing about the XOR-function is that it can be used to reverse itself and can therefore be used for encryption purposes. Suppose that we take the values: A= 10101000 and B= 00111001. Therefore, C= XOR (A,B)= 10010001. Now if we take B and XOR it with C, we will obtain A: XOR(B,C)=XOR(00111001, 10010001)=10101000=A (for SW implementation see Append. A2).

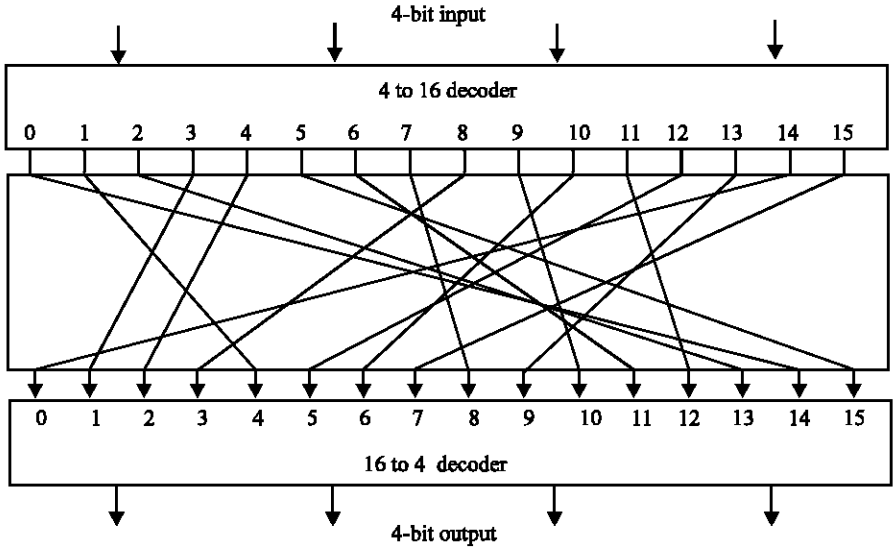

Fig. 2: An implementation of Feistel block cipher encryption algorithm

Substitution techniques usually use some simple strategy such as a lookup table or the XOR-function. The concept of block cipher encryption originally developed by Feistel[18], is very commonly used for this purpose; it involves the use of a block or a group of bytes for encryption process instead of a single byte or character. Each block can be operated on by any combination of several processes. Typical block size of 64 bits is used. Security is obtained by having a one-to-one mapping between blocks of characters of plaintext and blocks of cipher text of the same size - but the relationship between them is not easy to figure out. A block cipher operates on a plaintext block of n-bits to produce a ciphertext block of n-bits. There are 2^n possible different plaintext blocks and, for the encryption to be reversible (i.e. for decryption to be possible), each must produce a unique ciphertext block. Such a transformation is called reversible, or nonsingular. Figure 2 illustrates the logic of a general substitution block cipher for n = 4. A 4-bit input produces one of 16 possible input states (from 0000 to 11111), which is mapped by the substitution cipher into a unique one of 16 possible output states, each of which is represented by 4 ciphertext bits.

An alternative and more popular method of implementing a substitution function is to use a construct referred to as a symmetric (substitution) box, or an S-box. S-box design is one of the most intense areas of research in the field of symmetric block ciphers cryptography. The S-box function takes some bit or set of bits as input and provides some other bit or set of bits as output. It makes use of a replacement table to perform the conversion[19]. These reference tables can map more than one input to the same output. In essence, we would like any change to the input vector to an S-box to result in random-looking changes to the output. The relationship should be nonlinear and difficult to approximate with linear functions. As a result of this truth, a hacker cannot take the output from an S-box and figure out which of the many inputs may have been used to generate the output. One of the obvious characteristics of S-box is its size. An n x m S-box has n input bits and m output bits. DES has 6x4 S-boxes, blowfish and CAST have 8×32 S-boxes. Larger the S-boxes, the more difficult differential and linear cryptanalysis will be. On the other hand, the larger the dimension n, the (exponentially) larger the lookup table. Thus, for practical reasons, a limit of n equal to about 8 to 10 is usually imposed. An n x m S-box typically consists of 2^n rows and of m-bits each. The n-bits of input select one of the rows of the S-box and the m-bits in that row are the output. This type of substitution is not necessarily secure enough; the German Enigma is a complex substitution algorithm that was broken before the advent of digital computers[4].

A further secure implementation of the ciphertext can be achieved via use of permutation technique[20,21]. Permutation technique involves rearrangement of the characters of a plaintext message to convert the message into an anagram that looks like a message with random characters. For example, most messages consist of 7-bit ASCII characters. By scrambling the bits to create a random set of bits, you can get the desired encryption. Permutation techniques are usually used in conjunction with other techniques such as substitution and encryption functions like XOR. Other functions such as binary addition, multiplication and modular arithmetic functions are also common.

Alternatively, the permutation process can include expansion technique. The expansion-permutation approach takes a block of data and expands it into a set of overlapping groups; each group may be small compared to the original block[21]. Suppose that we have a block of 24-bits; we can perform expansion-permutation to convert it into a block of 36 bits as follows: (I) Break the 24 bits into six groups of four bits each; and (ii) To each group, add the bit that precedes it and the bit that follows it. Now we have six groups with six bits each for a total of 36 bits.

The techniques in the preceding paragraphs are just some of the commonly used methods in cryptographic algorithms. You can mix and match them to obtain alternative encryption algorithms, which become more complex and secure by using different encryption techniques one after the other. Popular encryption algorithms make use of 8 or 16 different rounds of encryption techniques. Feistel and co-workers while working at IBM in the early 60s first suggested this systematic approach[5,18]. Feistel proposed that we could approximate the simple substitution cipher by utilizing the concept of a product cipher, which is obtained by combining two or more basic ciphers in sequence in such a way that the final result or product is cryptographically stronger than any of the component ciphers. In particular, Feistel proposed the use of a cipher that alternates substitutions and permutations. In fact, this is a practical application of a proposal by Claude Shannon to develop a product cipher that alternates confusion and diffusion functions[22].

The terms confusion and diffusion were introduced by Claude Shannon to capture the two basic building blocks for any cryptographic system. Shannon's concern was to thwart cryptanalysis based on statistical analysis. On the other hand, confusion seeks to make the relationship between the statistics of the ciphertext and the value of the encryption key as complex as possible, again to thwart cryptanalyst attempts to discover the key. The reasoning is as follows: Assume the attacker has some knowledge of the statistical characteristics of the plaintext. For example, in a human-readable message in some language, the frequency distribution of the various letters may be known. Or there may be words or phrases likely to appear in the message (again with the English alphabet: is, th, the etc.). If these statistics are in any way reflected in the ciphertext, the cryptanalyst may be able to deduce the encryption key, or part of the key, or at least a set of keys likely to contain the exact key.

The confusion process is usually implemented via a design function F, which is the heart of the Feistel block cipher[23]. This function relies on the use of S-boxes. This is also the case for most other symmetric block ciphers[18]. The function F provides the element of confusion in a Feistel cipher. Thus, it must be difficult to "unscramble" the substitution performed by F. One obvious criterion is that F be nonlinear. The more nonlinear F, the more difficult any type cryptanalysis will be. In rough terms, the more difficult it is to approximate F by a set of linear equations, the more nonlinear F is. Several other criteria should be considered in designing F. We would like the algorithm to have good avalanche properties. It means that a change in one bit of the input should produce a change in many bits of the output. Another version of this is strict avalanche criterion (SAC), which states that any output bit j of an S-box should change with probability ½ when any single input bit I is inverted or all I, j.

In diffusion approach, the statistical structure of the plaintext is dissipated into long-range statistics of the ciphertext. This is achieved by having each plaintext digit affect the value of many ciphertext digits, which is equivalent to saying that; each ciphertext digit is affected by many plaintext digits[24]. An example of diffusion is to encrypt a message, $M = m_1, m_2, m_3, \ldots$, of characters with an averaging operation:

$$y_n = \sum_{i=1}^{k} m_{n+i} \pmod{26} \quad (1)$$

Adding k successive letters to get a ciphertext letter y_n. One can show that the statistical structure of the plaintext has been dissipated. Thus, even if the attacker can get some handle on the statistics of the ciphertext, the way in which the key was used to produce that ciphertext is so complex as to make it difficult to deduce the key. So, successful are diffusion and confusion in capturing the essence of the desired attributes of a block cipher that they have become the cornerstone of modern cryptographic systems design and implementation.

The Blowfish Algorithm -Blowfish combines a non-invertible f-function, key-dependent S-boxes and a Feistel network to make a cipher that has not yet been broken[25]. It is relatively simple to implement. The most interesting portion of Blowfish is its non-invertible f-function. This function uses modular arithmetic to generate indexes into the S-boxes. Modular arithmetic is usually used to create non-invertible f-functions. Non-invertability is best explained by example by Table 2: Take the function: $f(x) = x^2 \bmod 9$:

Table 2: An implementation of Blowfish algorithm

X	1	2	3	4	5	6	7	8	9
x^2	1	4	9	16	25	36	49	64	81
$x^2 \bmod 9$	1	4	0	7	7	0	4	1	0

Given an output, there is no function that can generate the specific input to f (x). For example, if you

knew that your function has a value of 4 at some x, there is no way to know if that x is 2, 7, or any other x whose f (x) = 4. Blowfish does its arithmetic over mod 2^{32}, (2^{32} is around 4 billion). This is called arithmetic in a finite field and makes many common mathematical assumptions untrue (1+1 does not equal two if you are in a finite field of size two). During the process of key setup, the key is combined with the S-boxes. The details of this key-setup are relatively uninteresting, but the fact that it combines the key with the S-boxes strengthens the algorithm greatly. Key setup in Blowfish is designed to be relatively slow. This is actually a benefit, as someone doing a brute-force cryptanalysis search of keys will have to go through the slow key setup process for each key tried. However, someone doing encryption and decryption must only go through the key setup process once. Hence, encryption and decryption are relatively fast.

Another important element of Blowfish is the Feistel network. Using the Feistel network gives the cipher two very desirable properties: decryption using the same f function (even if it is non-invertible) and the ability to iterate the function multiple times. These multiple iterations are called rounds. The more rounds, the more secure the algorithm is. The recommended number of rounds depends on the specific algorithm; for Blowfish, it is 16. A Feistel network can be described by the following algorithm[15]: Divide a block of length n into two parts, L (left) and R, (right) of length n/2 such that: $L_i = R_{i+1}$ and $R_i = L_{i-1}$ XOR f (R_{i-1}, K_i) and proceed to implement the appropriate cryptographic algorithm.

Now that we have a sound background and a pretty good knowledge of the possible techniques required for implementation of secure cryptographic systems, we will proceed with the discussion and methodology useful for deployment of secure cryptographic systems for both private-key and public-key cryptosystems.

Conventional (private-key) cryptography: modern techniques: The kind of cryptography used in earlier days and in the code and cipher techniques such as the Caesar Cipher, Hill Cipher, Vernam Cipher, Data Encryption Standard (DES) etc. are called private-key or secret-key cryptography[1,2,15]. The term private-key is used because this technique implies that both the sender and the receiver of the message have a key that must be kept private. Private-key cryptography makes use of the same key on both the sending and the receiving end and is therefore also referred to as symmetric cryptography (Fig. 3). In secrete-key cryptography it is assumed that Bob and Alice both use some piece of information in their encrypting and decrypting algorithms (i.e., the secret-key) that is not known to Eve.

As an example, consider the following protocol based on exponentiation (Fig. 3): For our scenario we suppose that A and B (also known as Alice and Bob) are two users of a private-key cryptosystem. We will distinguish their plaintext and ciphertext procedures with subscripts: T_A, C_A, T_B. C_B. Now Bob and Alice select a large prime number p and next they choose a secret-key, $K_s \in \{1,2,.....,\ p-1\}$ that is relatively prime to, p-1. Now Bob encrypts the plaintext message, T_B, using the function:

$$C_B = f(T_B) = (T_B)^{Ks} \bmod p \qquad (1)$$

Alice decrypts Bob's ciphertext message using:

$$T_B = f^{-1}(C_B) = (C_B)^{I} \bmod p \qquad (2)$$

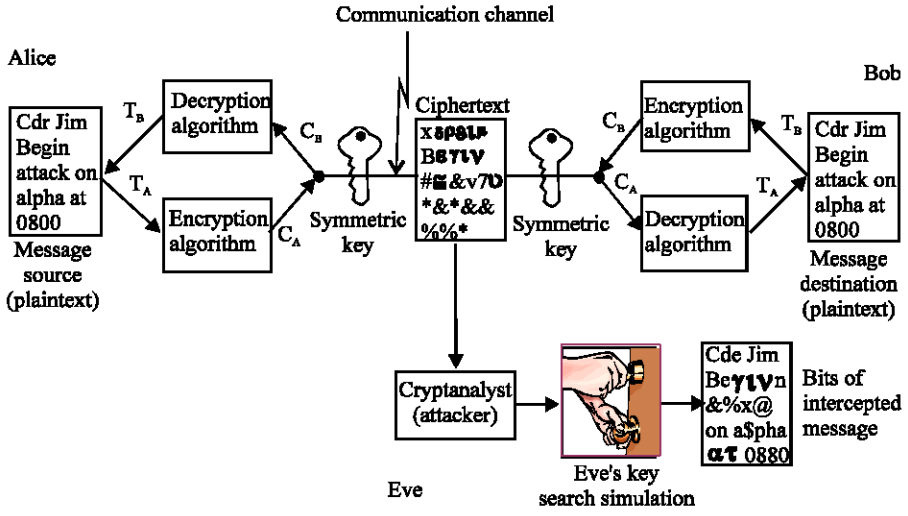

Fig. 3: Shows a simple schematic classical cryptographic algorithm

where K_s modulo p-1 {i.e., $(I.K_s) \bmod (p-1) = 1$}. If K_s is known I can be easily computed using Euclid's algorithm (Appendix B), which can be used to verify that: $f^{-1}(f(T_B)) = T_B$. The secret key K_s can also be computed from C_B (which Eve knows) by solving Eq. (1) for p. This, however, requires the calculation of discrete logarithms modulo p; at present, the best logarithms for performing this operation take time that is exponential in p. Thus, by choosing p large, Bob and Alice make it computationally infeasible for Eve to determine T_A without knowing K_s. This effectively addresses the secrecy issue. As for authenticity of messages, the fact that Eve does not know K_S also makes it impossible for her to hack the system and misrepresent herself. For example, she will not be able to use, Eq. (1) to encrypt a rogue message such as: "Alice, please wire Eve $6077 for the purchase of the laptop. Thanks Bob". Likewise, any ciphertext C_B altered by Eve (without knowing K_s) will most likely be detected by Alice. The alteration will simply lead to garbled plaintext when Alice uses Eq. (2) to decipher C_B.

There are several popular private-key algorithms including the primitive Caesar cipher discussed earlier. In this section we will discus two more private-key encryption algorithms the Data Encryption Standard (DES) which is one of the most widely used modern conventional cryptosystem; and the one-time pad which is the most secure private-key cryptosystem. Although numerous conventional encryption algorithms have been developed since the introduction of DES, it remains the most important such algorithm.

Data encryption standard (DES): The Data Encryption Standard (DES) has been the most popular encryption algorithm of the past twenty-five years. Originally developed at IBM Corporation, it was chosen by the National Bureau of Standards (NBS) as the government-standard encryption algorithm in 1976[25-27]. After becoming a U.S. government standard, DES was adopted by other standards bodies worldwide, including ANSI and ISO. The terms of the standard stipulated that it would be reviewed and recertified every five years. NBS recertified DES for the first time in 1987. NIST (NBS after the name change) recertified DES in 1993. DES was quickly adopted for non-digital media, such as voice-grade public telephone lines. Within a couple of years, for example, International Flavors and Fragrances were using DES to protect its valuable formulas transmitted over the phone[28]. Meanwhile, the banking industry, which is the largest user of encryption outside government, adopted DES as a wholesale banking standard and; was used in many other different applications around the world.

DES is the most important algorithm ever made. Because it had an US National Security Agency (NSA) pedigree, it was widely believed to be secure. However, concerns about its short key length have mostly dogged the algorithm since the beginning. In the late 1990s, it became widely believed that the NSA was able to break DES by trying every possible key, something called "brute force" (cryptanalysis) machine capable of breaking DES. This ability was graphically demonstrated by the Electronic Frontier Foundation in July 1998, when John Gilmore built a machine for $250,000 that could brute-force a DES key in a few days. In 1997 they initiated a program to replace DES: with the Advanced Encryption Standard (AES)[27].

Years before this, more secure applications had already converted to an encryption algorithm called triple-DES (also referred to as 3DES). Triple-DES is the repeated application of three DES encryptions, using two or three different keys[29]. This algorithm leverages all the security of DES while effectively lengthening the key and is in wide use today to protect all kinds of personal, business and financial secrets. It is also the most studied encryption algorithm ever invented and many cryptographers "went to school" on DES. Almost all of the newer encryption algorithms in use today can trace their roots back to DES and research papers analyzing different aspects of DES are still being published today.

Design and implementation of DES: DES was originally designed to be implemented only in hardware and is therefore extremely slow in software applications. The algorithm, although complicated, is pretty straightforward. It uses only simple logical operations on small groups of bits and could be implemented fairly efficiently in the mid-1970s hardware of the time[14]. DES is not very efficient in software, especially the 32-bit architectures that are common today. Its overall structure was something called a Feistel network, also used in another IBM design called Lucifer, which is often considered to be a precursor to DES[6,26]. DES is a block cipher, meaning that it encrypts and decrypts data in blocks: 64-bit blocks. DES is an iterated cipher, meaning that it contains 16 iterations (called rounds) of a simpler cipher. The algorithm's primary strength came from S-box design, a non-linear table-lookup operation.

The specific utilization and the implementation of the DES will be based on many factors particularly to the computer system and its associated components. The cryptographic algorithm specified in this standard transforms a 64-bit binary value into a unique 64-bit binary value based on a 56-bit variable. If the complete 64-bit input is used (i.e., none of the input bits should be predetermined from block to block) and if the 56-bit variable is randomly chosen, no technique other than trying all possible keys using known input and output for the DES will guarantee finding the chosen key. As there are over 70×10^{15} (seventy quadrillion) possible keys of 56

bits, the feasibility of deriving a particular key in this way is extremely unlikely in typical threat environments. Moreover, if the key is changed frequently, the risk of this event is greatly diminished. However, users should be aware that it is theoretically possible to derive the key in fewer trials (with a correspondingly lower probability of success depending on the number of keys tried) and; should be cautioned to change the key as often as practical. Users must also provide the cryptographic key a high level of protection in order to minimize the potential risks of its unauthorized computation or acquisition[16]. The feasibility of computing the correct key may change with advances in cryptographic technology. However, when correctly implemented and properly used, this standard will provide a high level of cryptographic protection to computer data.

Here, we will present an implementation of a simplified version of DES or S-DES using 8-bit block of text. The S-DES encryption algorithm takes an 8-bit block of plaintext (example: 10111101) and a 10-bit key as input and produces an 8-bit block of ciphertext as output. The S-DES decryption algorithm takes an 8-bit block of ciphertext and the same 10-bit key used to produce that ciphertext as input and produces the original 8-bit block of plaintext. (SW implementation A3)

The encryption algorithm involves five functions: An initial permutation (IP); a complex function labeled as f_k, which involves both permutation and substitution operations and depends on a key input; a simple permutation function that switches (SW) the two halves of the data; the function f_k again and finally a permutation function that is the inverse of the initial permutation (IP^{-1}). The use of multiple stages of permutation and substitution results in a more complex algorithm, which increases the difficulty of cryptanalysis.

The function f_k takes as input not only the data passing through the encryption algorithm, but also an 8-bit key. The algorithm could have been designed to work with a 16-bit key, consisting of two 8-bit subkeys, one used for each occurrence of f_k. Alternatively, a single 8-bit key could have been used, with the same key used twice in the algorithm. A compromise is to use a 10-bit key from which two 8-bit subkeys are generated as depicted in the Fig. 4. In this case, the key is first subjected to permutation (P10). Then a shift operation is performed. The output of shift operation then passes through a permutation function that produces an 8-bit output (P8) for the first subkey (K_1). The output of the shift operation also feeds into another shift and another instance of P8 to produce the second subkey (K_2).

S-DES key generation: S-DES depends on the use of a 10-bit key shared between sender and receiver. From this key, two 8-bit subkeys are produced for use in particular stages of the encryption and decryption algorithm. The stages to produce the keys are illustrated below:

First, permute the key in the following fashion. Let the 10-bit key be designed as (k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10}). Then the permutation P10 is defined as:

P10 (k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10}) = (k_3, k_5, k_2, k_7, k_4, k_{10}, k_1, k_9, k_8, k_6).

For example, the key (1010000010) is permuted to (1000001100). Next, perform a circular left shift (LS-1), or rotation, separately on the first five bits and the second five bits. In our example, the result is (00001 11000). Next we apply P8, which picks out and permutes 8 of the 10 bits according to the following rule, P8: (6 3 7 4 8 5 10 9), The result is subkey1 (K_1). In our example, this yields (10100100).

We then go back to the pair of 5-bit strings produced by the two LS-1 functions and perform a circular left shift of 2 bit positions on each string. In our example, the value (00001 11000) becomes (00100 00011). Finally, P8 is applied again to produce K_2. In our example, the result is (01000011).

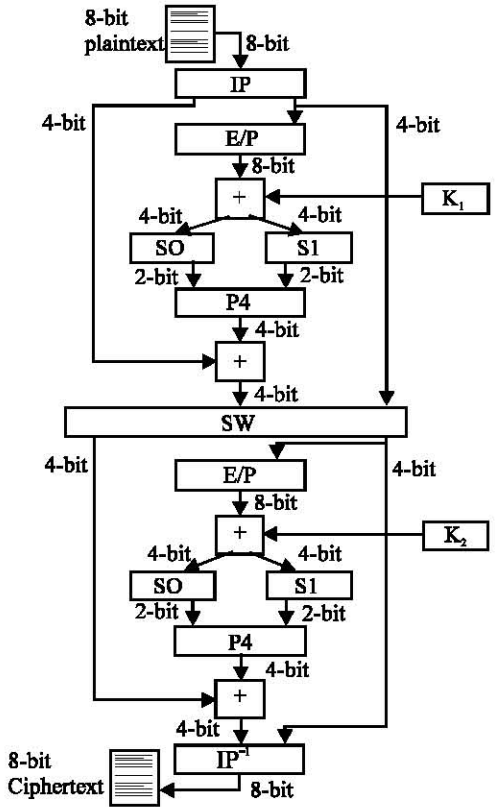

Fig. 4A: Graphical representation of a DES implementation algorithm

S-DES encryption: Involves the sequential application of five functions. We examine each of these as follows:

Initial and final permutations: The input to the algorithm is an 8-bit block of plaintext, which we first permute, using the IP function, IP: 26314857. This retains all 8 bits of the plaintext but mixes them up. At the end of the algorithm, the inverse permutation is used, IP^{-1}: 41357286, Indeed, the second permutation is reverse of the first.

The function f_k: The most complex component of S-DES is the function f_k, which consists of a combination of permutation and substitution functions. The functions can be expressed as follows. Let L and R be the leftmost 4 bits and rightmost 4 bits of the 8-bit input to f_k and let F be a mapping (not necessarily one-to-one) from 4-bit strings to 4-bit strings. Then we let, f_k (L,R) = (L⊕ F(R,SK),R), where SK is a subkey and ⊕ is the bit-by-bit exclusive-OR operation function.

For example, suppose the output of the IP stage is (10111101) and F(1101, SK) = (1110) for some key, SK. Then f_k(10111101) = (01011101) because (1011)⊕(1110) = (0101). We now describe the mapping F. The input is a 4-bit number (n_1 n_2 n_3 n_4). The first operation is an expansion/permutation operation, E/P: 41232341. For what follows, it is clearer to depict the results in this fashion:

$$\begin{array}{|c|cc|c|} \hline n_4 & n_1 & n_2 & n_3 \\ n_2 & n_3 & n_4 & n_1 \\ \hline \end{array}$$

The 8-bit subkey K_1= (k_{11}, k_{12}, k_{13}, k_{14}, k_{15}, k_{16}, k_{17}, k_{18}) is added to this value using exclusive-OR:

$$\begin{array}{|c|cc|c|} \hline n_4+k_{11} & n_1+k_{12} & n_2+k_{16} & n_3+k_{14} \\ n_2+k_{15} & n_3+k_{13} & n_4+k_{17} & n_1+k_{18} \\ \hline \end{array}$$

Let us rename these bits:

$$\begin{array}{|c|cc|c|} \hline p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ \hline \end{array}$$

The first four bits (first row of the precedence matrix) are fed into the S-box, S0, to produce a 2-bit output and the remaining 4 bits (second row) are fed into, S1, to produce another 2-bit output. These two are defined as follows:

$$S0 = \begin{pmatrix} 1 & 0 & 3 & 2 \\ 3 & 2 & 1 & 0 \\ 0 & 2 & 1 & 3 \\ 3 & 1 & 3 & 2 \end{pmatrix} \quad S1 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 0 & 1 & 3 \\ 3 & 0 & 1 & 0 \\ 2 & 1 & 0 & 3 \end{pmatrix}$$

The S-boxes operate as follows: The first and fourth input bits are treated as 2-bit numbers and specify a row of the S-box, while the second and the third input bits specify a column of the S-box. The entry in that row and column, in base 2, is the 2-bit output. For example, if ($p_{0,0}p_{0,3}$) = (00) and ($p_{0,1}p_{0,2}$) = (10), then the output is from row 0, column 2 of S0, which is 3, or (11) in binary. Similarly, ($p_{1,0}p_{1,3}$) and ($p_{1,1}p_{1,2}$) are used to index into a row and column of S1 to produce an additional 2 bits. Next, the 4 bits produced by S0 and S1 undergo a further permutation, P4: 2431. The output of P4 is the output of the function, F.

Switch function: The function f_k only alters the leftmost 4 bits of the input. The switch function (SW) interchanges the left and right 4 bits so that the second instance of f_k operates on a different 4 bits. In the second instance, the E/P, S0, S1 and P4 functions are the same. The key input is K_2. Table 3 shows the implementation of S-DES algorithm (for SW implementation see Append. A3).

As a practical matter, anyone today who wants high security with ultimate strength cryptography uses a more powerful version of DES called Triple-DES. To start encrypting with Triple-DES, two 56-bit keys are selected. Data is encrypted via DES three times, the first time by the first key, the second time by the second key and the third time by the first key once more. This process creates an encrypted data stream that is unbreakable with today's code-breaking techniques and available computing power, while being compatible with DES.

One-time pad: A one-time pad is a very simple yet completely unbreakable symmetric cipher. It was invented in 1917 by Major Joseph Mauborgne and AT and T's Gilbert Vernam[3,30]. As with all symmetric ciphers, the sender must transmit the key to the recipient via some secure and tamper proof channel, otherwise the recipient won't be able to decrypt the ciphertext. The key for a one-time pad cipher is a string of random bits, usually generated by a cryptographically strong pseudo-random number generator (CSPRNG)[31]. It is better to generate the key using the natural randomness of quantum mechanical events (such as those detected by a Geiger counter as used in Experimental Physics), since quantum events are believed scientifically to be the only source of truly random information in the universe. One-time pads that use CSPRNGs are open to attacks, which attempt to compute part or the entire key.

With a one-time pad, there are as many bits in the key as in the plaintext. This is the primary drawback of a one-time pad, but it is also the source of its perfect security. It is essential that no portion of the key may ever be reused for another encryption (hence the name "one-time pad");

Table 3: Implementation of S-DES cryptographic algorithm

Action	Input	Output
S-DES Key Generation		
10-bit key : 1100101001		
P10	1100101001	0111011000
LS-1	0111011000	1110010001
P8	1110010001	11000010 (K_1)
LS-2	1110010001	1001100110
P8	1001100110	00011101 (K_2)
S-DES Encryption		
8-bit plaintext : 10100110		
IP	10100110	01110001
E/P	0001	10000010
Exclusive-OR	10000010, K_1	0100 0000
S0	0100	11
S1	0000	00
P4	1100	1001
Exclusive-OR	0111, 1001	1110
SW	11100001	00011110
E/P	1110	01111101
Exclusive-OR	01111101, K_2	01100000
S0	0110	10
S1	0000	00
P4	1000	0001
Exclusive-OR	0001, 0001	0000
IP^{-1}	00001110	00011001
8-bit ciphertext : 00011001		
S-DES Decryption		
8-bit ciphertext : 00011001		
IP	00011001	00001110
E/P	1110	01111101
Exclusive-OR	01111101, K_2	01100000
S0	0110	10
S1	0000	00
P4	1000	0001
Exclusive-OR	0000, 0001	0001
SW	00011110	11100001
E/P	0001	10000010
Exclusive-OR	10000010, K_1	01000000
S0	0100	11
S1	0000	00
P4	1100	1001
Exclusive-OR	1110, 1001	0111
IP^{-1}	01110001	10100110
8-bit plaintext : 10100110		

otherwise cryptanalysis can break the cipher. The cipher itself is exceedingly simple. To encrypt plaintext, P, with a key, K, producing ciphertext, C, simply compute the bitwise exclusive-or of the key and the plaintext: C = K^P. To decrypt ciphertext, C, the recipient computes: P = K^C. It is that simple and it's perfectly secure, as long as the key is random and is not compromised (for SW implementation see Append. A2). A simple example of one-time pad encryption/decryption algorithm goes like this:

Encryption: CLAUDIUSDIDIT + SYQJOWJQGBOEF = VKRESFEJKKSNZ (L:12 + Y:25 ->K:37)
Decryption: VKRESFEJKKSNZ - SYQJOWJQGBOEF = CLAUDIUSDIDIT

Why are One-Time pads perfectly secure?: If the key is truly random, an XOR-based one-time pad is perfectly secure against ciphertext-only cryptanalysis. This means an attacker can't compute the plaintext from the ciphertext without the knowledge of the key, even via a brute force search of the space of all keys! Trying all possible keys doesn't help you at all, because all possible plaintexts are equally likely decryptions of the ciphertext. This result is true regardless of how few bits the key has or how much you know about the structure of the plaintext. To see this, suppose you intercept a very small, 8-bit, ciphertext. You know it is either the ASCII character 'S' or the ASCII character 'A' encrypted with a one-time pad. You also know that if it's 'S', the enemy will attack by sea and if it's 'A', the enemy will attack by air. That's a lot to know. All you are missing is the key, a silly little 8-bit one-time pad. You assign your crack staff of cryptanalysts to try all 256, 8-bit one-time pads. This is a brute-force search of the keyspace. The results of the brute force search of the keyspace is that your staff finds one 8-bit key that decrypts the ciphertext to 'S' and one that decrypts it to 'A'. and you still don't know which one is the actual plaintext. This argument is easily generalized to keys (and plaintexts) of any arbitrary length.

Private-key distribution: In symmetric cryptography, for both secrecy and authentication, the sender must transmit the key to the recipient via some secure and tamper proof channel, otherwise the recipient won't be able to decrypt the ciphertext (Fig. 3). Hence, the major security issue in secret-key cryptography involves keeping the key K_s secure and these requirements lead to the major shortcoming of symmetric-key cryptography. The process of exchanging the cryptographic key is referred to as key distribution and can be very difficult[16]: It is not difficult to see the impracticality of this approach among a large collection of people. The key is the secret to breaking the cipher text; if there exists a really secure method of communicating the key, why isn't that method used to communicate the message in the first place? For many years, the key distribution method used by the governmental and corporate agencies was to place the keys in a locked briefcase, which was handcuffed to a courier. The courier would board an airplane (or take any specified mode of transport) and would be met at the destination by an official from the end-user agency. The cuffs would be removed at the agency office and the keys were then available to decipher the messages. The courier did not have a way to remove the cuffs or open the briefcase. If the bad guys caught the courier, the end-user would know about it and would not use those particular keys to encrypt messages. The difficulties encountered in private-key cryptography lead to further research leading to the development of public-key cryptography, a system that uses two sets of keys; one for encryption and the other for decryption.

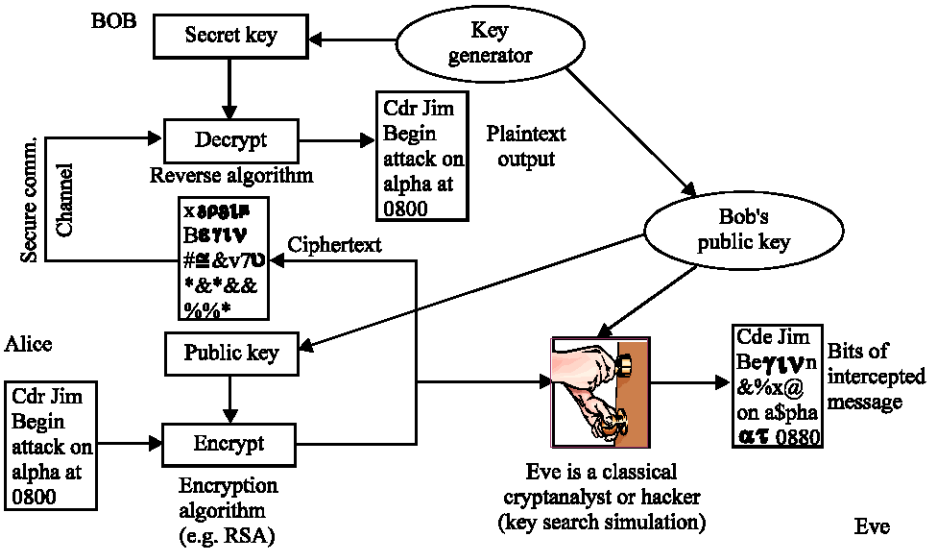

Fig. 5: An implementation of RSA algorithm

Public key cryptography: All classical encryption methods, as mentioned earlier, suffer from the "key distribution problem". The problem is that before a private communication can begin, another private transaction is necessary to distribute corresponding encryption and decryption keys to the sender and receiver, respectively. Typically, a private courier is used to carry a key from the sender to the receiver. Such a practice is not feasible if an electronic mail system is to be rapid and inexpensive. A public-key cryptosystems needs no private couriers; the keys can be distributed over the insecure communication.

Public-key cryptography also referred to as asymmetric cryptography and is the result of a mathematical breakthrough that occurred in the early 1970s[17]. Unlike symmetric key methods that use a single key for encryption and decryption, asymmetric methods make use of two keys: a secret-key and a public-key. In a nutshell, that means that you actually have two keys: a private-key that only you should have access to and a public-key that you give away to anyone you intend to communicate with. When someone wants to send you an encrypted file, they use your public-key to encrypt the file. Having done that, the encrypted file can then only be decrypted by you using your private-key (i.e., the public-key is used to encrypt the message and the secret-key is used to decrypt the message). The receiver has the secret-key that should be protected. A mathematical process can be used to generate the two keys that are mathematically related. The goal of public-key cryptography was to eliminate the biggest problem of private-key cryptography of key distribution.

Public-key cryptography facilitates the following tasks: (I) Encryption and decryption allow two communicating parties to disguise information they send to each other, see Fig. 5. The sender encrypts, or scrambles, information before sending it. The receiver decrypts, or unscrambles, the information after receiving it. While in transit, the encrypted (ciphertext) information is unintelligible to an intruder; (ii) Tamper detection allows the recipient of information to verify that it has not been modified in transit. Any attempt to modify data or substitute a false message for a legitimate one will be detected; (iii) Authentication allows the recipient of information to determine its origin - that is, to confirm the sender's identity and; (iv) Nonrepudiation prevents the sender of information from claiming at a later date that the information was never sent by him. Several techniques have been identified in the domain of public-key cryptography over the years.

Ralph merkle's puzzle technique: Ralph Merkle when he published his work in Communications of the ACM, a premier computer science journal, had indicated that his protocol was processed on "secure communication over insecure channels"[33]. The basis of his communication approach involves the use of puzzles. To understand this method, assume that Bob and Alice want to communicate with each other over a channel that is known to be insecure. Bob first creates a large number of encryption keys - say a million keys. Bob then places the keys in puzzles - one key per puzzle. Each puzzle takes a couple of minutes to solve. Bob sends the puzzles to Alice, who chooses any one of the puzzles and its associated key. Using this key, Alice encrypts a message and sends it to Bob. Bob now figures out the key Alice chose based on his list of keys. Future communications between Bob and

Alice occur using this key. An eavesdropper, say Eve, will be aware of the puzzles going back and forth but will take an extremely long time to figure out the exact key.

Diffie-Hellman multiuser cryptographic techniques: Diffie-Hellman Multiuser Cryptographic Techniques was the first real functional public-key algorithm invented. It first appeared in a paper called "Multiuser Cryptographic Techniques" which was published in 1975 by Whitfield Diffie and Martin Hellman[34]. Their cryptographic techniques used the concept commonly used now in public-key cryptography. The basic idea of this strategy was that it should be possible to encrypt a message using one key and decrypt the message using another key. Several suggestions were made to Diffie and Hellman about how this could be achieved, including the following: (I) Multiplying prime numbers, which can be done easily; but it is difficult to factor the corresponding result and (ii) Using a discrete exponentiation of numbers; the corresponding task of discrete logarithms is difficult (Mathematics Appendix B).

Diffie and Hellman chose the second approach to conduct further research. The Diffie-Hellman exponential key exchange approach was published in their paper "New Directions in Cryptography" in IEEE Transactions on Information Theory[35]. The Diffie-Hellman was further given credence through a suggestion by John Gill, another Stanford colleague: Take the exponents of two numbers and calculate the results modulo some prime number. The method works as follows: (I) Both the active participants must first agree on two numbers, p and q. Numbers p and q can be publicly known; (ii) Each participant must now choose a number, perform a mathematical operation that involves p, q and the chosen number and then transmit the result to the other participant. Suppose that the first participant chooses M1 and the other participant chooses M2. The results of their separate mathematical operations are N1 and N2 and (iii) Using a second mathematical formula, both participants can now compute another number, K, such that K can be computed as a function of the numbers M1 and N2 or the numbers M2 and N1 - but not the numbers N1 and N2. Future communications occur using the session key K. The eavesdropper can have access to p, q, N1 and N2 but neither M1 nor M2. As a result, the eavesdropper cannot calculate K. Thus, K can be used as a session-key for a private-key encryption algorithm such as DES. This method is used for communication between two people and makes use of three keys: two secret-keys (one for each person) and a session key determined by the two people during the course of the conversation. In other words, the conversation starts with the two people using their own keys; they exchange information to determine a session key which is then used for all future messages. It is important to note that Diffie-Hellman algorithm is an excellent tool for key distribution, but cannot be used effectively to encrypt and decrypt messages on the fly independent of the person one communicates with (cf. email communication).

The RSA cryptographic system: The biggest problem with the Diffie-Hellman method is that the two participants must communicate actively. This may not be possible in email communication between two people who are not necessarily actively conversing. In 1976, three professors in the computer science lab at MIT-Ronald Rivest, Adi Shamir and Len Adelman-started working on the proposition made in the Diffie-Hellman paper, "New Directions in Cryptography," to find a practical multi-user cryptography system. After several months of research, they were about to conclude that such a public-key exchange system was not possible. Then, in 1977, they realized a basic fact: It is very easy to multiply two prime numbers to get a large composite number, but it is difficult to take that composite number and find its prime number components. The outcome of this research is the technique simply referred to by the initials of its three inventors: RSA[36]. This method is better than the Diffie-Hellman key exchange technique because it does not rely on active participation between the person performing the encryption and the person performing the decryption. RSA makes use of any publicly available key to encrypt the information, but the decryption can be done only by the person who holds the matching secret-key (Fig. 5). The RSA technique is one of the most powerful encryption methods known to-date. It is also used as the public-key system in PGP (Pretty Good Privacy)[37]. RSA can also be used as a digital signature system as will be seen later.

How does public-key cryptographic functions: In a "public-key cryptosystems" each user places in a public-key server an encryption procedure E. That is, the public-key server is a directory giving the encryption procedure of each user. The user keeps secret the details of his corresponding decryption procedure D. These procedures have the following properties:

(a) Decrypting the encrypted form of plaintext message C = E(T) yields T, i.e.:

$$D© = D(E(T)) = T \quad (4)$$

(b) Both E and D are easy to compute.

(c) By publicly revealing E the user does not reveal an easy way to compute D. This means that in practice only he can decrypt messages encrypted with E, or compute D efficiently.
(d) If a message T is first deciphered and enciphered, T is the result, i.e.:

$$E(D(T)) = T \quad (5)$$

An encryption (or decryption) procedure typically consist of a general method and an encryption key. The general method, under control of the key, encrypts a plaintext message T to obtain the form of the message or ciphertext C. Every one can use the same general method; the security of a given message will rest on the security of the key. Revealing an encryption algorithm then means revealing the key.

When the user reveals E, he reveals a very inefficient method of computing D(C): testing all possible messages T until one such that, E(T) = C, is found. If property © is satisfied the number of such messages to test will be so large that this approach is impractical.

A function E satisfying (a) – (c) is a "trap-door one-way function"; if it also satisfies (d) it is a "trap-door one-way permutation". Diffie-Hellman[35] introduced the concept of trap-door one-way functions but did not present any examples. These functions are called "one-way" because they are easy to compute in one direction but (apparently) very difficult to compute in the other direction. They are called "trap-doors" functions since the inverse functions are in fact easy to compute once certain private "trap-door" information is known. A trap-door one-way function that also satisfies (d) must be a permutation: every message is the ciphertext for some other message and every ciphertext is itself a permissible message. (The mapping is "one-to-one" and "onto"). Property (d) is needed to implement digital "signatures" scheme. The most widespread current use of RSA algorithm is in the Secure Sockets Layer (SSL) protocol for data protection on the Internet.

The mathematics of RSA: To understand how RSA works, (Mathematical Methods, Append. B). The steps to effectively use RSA are as follows:

1. Choose random "large" prime integers p and q (e.g., 1024-bit) and let: n = p.q (ordinary integer multiplication).
2. Let, $\varphi(n) = m = (p-1)(q-1)$
3. Choose a small number e such that e is greater than 1, e is less than n and e and m are relatively prime, which means they have no prime factors in common. However, e does not have to be prime, but it must be odd. But, m, can't be prime because it's an even number.
4. Compute d such that (de-1) is evenly divisible by, m. Mathematicians write this as, gcd (d,(p-1).(q-1))=1 or de = 1{mod (p-1).(q-1) and they call, d, the multiplicative inverse of e. This is easy to do, simply find an integer k, which causes, d = {km+1}/e, to be an integer and then use that value of d.
5. The encryption function is, $C = (T^e) \bmod n$, where C is the ciphertext (a positive integer), T is the plaintext (a positive integer). The message being encrypted, T must be less than the modulus, pq.
6. The decryption function is, $C = (C^d) \bmod n$, where C is the ciphertext (a positive integer), T is the plaintext (a positive integer).
7. The public-key (published) is the pair (n, e).
8. The private-key (kept secret) is the (n, d) (reveal it to no one).
9. Discard securely p and q.

The product, n, is the modulus, e is the public exponent and, d is the secret exponent. You can publish your public-key freely, because there are no known easy methods of calculating d, p, or q given only (n, e) (your public-key). If p and q are each 1024 bits long, the sun will burn out before the most powerful computers presently in existence to factor your modulus into p and q. Authentication on the other hand, is not as easy to guarantee in public-key cryptography. Since everybody knows everybody else's public-key, Eve can easily send message to Alice claiming to be Bob.

The aforementioned method should not be confused with the "exponentiation" technique presented by Diffie and Hellam[35] to solve the key distribution problem. Their technique permits two users to determine a key in common to be used in a normal cryptographic system. It is not based on a trap-door one-way permutation. Pohlig and Hellman[38] have, however, have studied a similar scheme related to RSA, where exponentiation is done modulo a prime number.

In mathematical approach as was performed for secret-key protocol, the public-key protocol can be implemented as follows: Again Bob and Alice are users of public-key cryptosystem. We now distinguish their encryption and decryption procedures with subscripts: E_A, D_A, E_B, D_B. Bob now encrypts his message T_B using Alice's public-key through:

$$f_A(T_B) = E_A(T_B) = (T_B)^e \bmod n \quad (4)$$

to produce C_B. Alice decrypts C_B using her secret-key through:

$$f_A^{-1}(C_B) = D_A(C_B) = (C_B)^d \bmod n \quad (5)$$

An important property of equation (4) and (5) is that they are inverses of one another, i.e., $f_A^{-1}(f_A(T))) = =f_A(f_A^{-1}(T)) = T$ or $D_A(E_A(T)) = E_A(D_A(T)) = T$. Note that encryption does not increase the size of a message; and both the message and the ciphertext are integers in the range 0 to n-1. Observe that no private transaction between Alice and Bob is needed to establish private communication. The only "setup" required is that each user who wishes to receive private communications must place their encryption algorithm in the public-key server. Each user sends his encryption key to the other public-key server. Afterwards all messages are encrypted with encryption key of the recipient, as in the public-key system. Also note that the transactions can take place over an insecure communication channel without consulting the public-key server[33]. An intruder (Eve) listening in on the channel cannot decipher, modify or insert any messages, since it is not possible to derive the decryption keys from the encryption keys.

How to dencryption and decryption efficiently: Computing $T^e \bmod n$ requires at most, $2\cdot \log_2(e)$ multiplications and $2\cdot \log_2(e)$, divisions using the following procedure (decryption can be performed similarly using d instead e):

Step 1: Let $e_k e_{k-1}.....e_1 e_0$ be the binary representation of e.
Step 2: Set the variable C to 1.
Step 3: Repeat steps 3a and 3b for I = k, k-1,....., 0:
Step 3a: Set C to the reminder of C^2 when divided by n.
Step 3b: If e_i, then set C to the reminder of $C\cdot T$ when divided by n.
Step 4: Halt. Now C is the encrypted form of T.

This procedure is called "exponentiation by repeated squaring and multiplication." This procedure is half as good as the best; more efficient procedures are known. Knuth[30] studies this problem in detail. The fact that the enciphering and deciphering are identical leads to simple implementation. The whole operation can be implemented on a few special-purpose integrated circuit chips. A high-speed computer can encrypt a 200-digit message T in a few seconds; special-purpose hardware would be much faster. The encryption time per block increases no faster than the cube of number of digits in n.

A practical example of RSA implementation: Suppose that you choose p = 61 and q = 53. Therefore, n = pq = 61*53 = 3233. Now you have to choose e such that it is relatively prime to (p-1)*(q-1) = 60*52 = 3120. Suppose that you pick e = 17. The decryption key is now calculated using the extended Euclid algorithm and the prime numbers as follows: d = 17-1 (mod 3120) = 2753.

Now a user who wants to encrypt and send some information to us can use (n, e) = (3233, 17) to encrypt the data. Suppose that someone wants to send us the plaintext message (number): T = 123.

- To do so, they'd perform the following calculation: $123^{17} \bmod 3233 = 885$.
- We would receive the number 885 and decrypt it as follows: $885^{2753} \bmod 3233 = 123$.
- To encrypt large number (message), T = 12347656845988954716, first break it into small blocks. Three-digit blocks work nicely in this case. The message will be encrypted in seven blocks, T_i, in which: $T_1 = 123$, $T_2 = 476$, $T_3 = 568$, $T_4 = 459$, $T_5 = 889$, $T_6 = 547$, $T_7 = 16$, which is encrypted into cipher blocks C_i. For example message, $T_1 = 123$, encrypts to, $C_1 = 885$ and follow the same procedure undertaken above for the rest of block of messages (Appendix B4 for detailed implementation procedure).

One may be surprised that RSA just deals with large integers. So how does it represent data? For plaintext encryption scheme, we can code two letters per block, substituting a two-digits number for each letter: blank = 00, A = 01, B = 02,....., Z = 26[39]. Then use a similar scheme presented above for encryption/decryption. For this example, lets have: p = 47 and q = 59, n = pq = 47*59 = 2773, e = 17 and d = 157. Thus the message:

ITS ALL GREEK TO ME

is encoded:

0920 1900 0112 1200 0718 0505 1100 2015 0013 0500

Since, e = 10001, in binary digit, the first block (T = 920) encrypted as:

$$T^{17} = (((((1)^2\cdot T)^2)^2)^2)^2)\cdot T = 948(\text{mod } 2773)$$

The whole message is encrypted as:

0948 2342 1084 1444 2663 2390 0778 0774 0219 1655

For decryption, we perform: $948^{157} = 920 \pmod{2773}$, etc. Alternatively, suppose the value of n is at least

Table 3: Indicates number of operations needed factor n in Schroeppel's method

Digits	Number of operations	Time
50	1.5×10^{10}	3.9 h
75	9.0×10^{12}	104 days
100	2.3×10^{15}	74 years
200	1.2×10^{23}	3.8×10^{9} years
300	1.5×10^{29}	4.9×10^{15} years
500	1.3×10^{39}	4.2×10^{25} years

1024-bits long. This is the same as 128-bytes (1 byte = 8 bits). In principle then, one can just run 128-bytes of ASCII text together and regard the whole as a single RSA plaintext (a single large integer) to be encrypted or signed. In practice, the protocols will demand additional data besides just the raw message, such as a time stamp, but there is room for a lot of data in a single RSA encryption.

Factoring n: Factoring n would enable an enemy cryptanalysis (Eve) to "break" our method. The factors of n enable her to compute $\phi(n) = \phi(p)\phi(q)=(p-1)(q-1)$ and thus d. Fortunately, factoring a number seems to be much more difficult than determining whether it is prime or composite. A large number of factoring algorithms exist. Knuth[30], gives an excellent presentation of many of them. Pollard[40] presents an algorithm, which factors a number n in time $O(n^{1/4})$. The fastest factoring algorithm known to the authors is due to Richard Schroeppel[30]; it can factor n in approximately:

$$p\sqrt{\ln(n)\cdot\ln(\ln(n))} = n^{\sqrt{\ln\ln(n)/\ln(n)}} = (h(n))^{\sqrt{\ln(n)/\ln(\ln($$

steps. Table 3 gives the number of operations needed to factor n with Schroeppel's method and the time required if each operation uses microsecond, for various lengths of numbers n (in decimal digits).

It is recommended that n be about 200 digits long. Longer or shorter lengths can be used depending on the relative importance of encryption speed and security in the application at hand. An 80-digit n provides moderate security against an attack using current technology; using 200 digits provides a margin of safety against future developments. This flexibility to choose a key-length (and thus a level of security) to suit a particular application is a feature not found in many cryptographic schemes.

Digital signature and authentication: If electronic mail systems are to replace the existing paper mail systems for business transactions, "signing" an electronic message must be possible. One way to address the authentication problem encountered in public-key cryptography is to attach digital signature to the end of each message that can be used to verify the sender of the message[17]. The significance of a digital signature is comparable to the significance of a handwritten signature. In some situations, a digital signature may be as legally binding as a handwritten signature. Once you have signed some data, it is difficult to deny doing so later - assuming that the private-key has not been compromised or out of the owner's control. This quality of digital signatures provides a high degree of nonrepudiation - i.e., digital signatures make it difficult for the signer to deny having signed the data. This quality is stronger than mere authentication (where the recipient can verify that the message came from the sender); the recipient can convince a "judge" that the signer sent the message. To do so, he must convince the judge he did not forge the signed message himself! In authentication problem the recipient does not worry about this possibility, since he only wants to satisfy himself that the message came from the sender.

Figure 6 shows two items transferred to the recipient of some signed data: the original data and the digital signature, which is basically a one-way hash (of the original data) that has been encrypted with the signer's private-key[39,41,42]. To validate the integrity of the data, the receiving software first uses the signer's public-key to decrypt the hash. It then uses the same hashing algorithm that generated the original hash to generate a new one-way hash of the same data. (Information about the hashing algorithm used is sent with the digital signature, as described blow.) Finally, the receiving software compares the new hash against the original hash. If the two hashes match, the recipient can be certain that the public-key used to decrypt the digital signature corresponds to the private-key used to create the digital signature. If they don't match, the data may have been tampered with since it was signed, or the signature may have been created with a private-key that doesn't correspond to the public-key presented by the signer. Confirming the identity of the signer, however, also requires some way of confirming that the public-key really belongs to a particular person or other entity and this is achieved via the use of fingerprinting.

In short, an electronic signature must be a message-dependent, as well as signer-dependent. Otherwise the recipient could modify the message before showing the message-signature pair to the judge. Or he could attach the signature to any message whatsoever, since it is not possible to detect electronic "cutting and pasting". To implement signatures the public-key cryptosystem must be implemented with trap-door one-way permutations i.e., have the property (d), since the decryption algorithm will be applied to unenciphered messages. For a discussion of the way this works let's look at the communication between Bob and Alice.

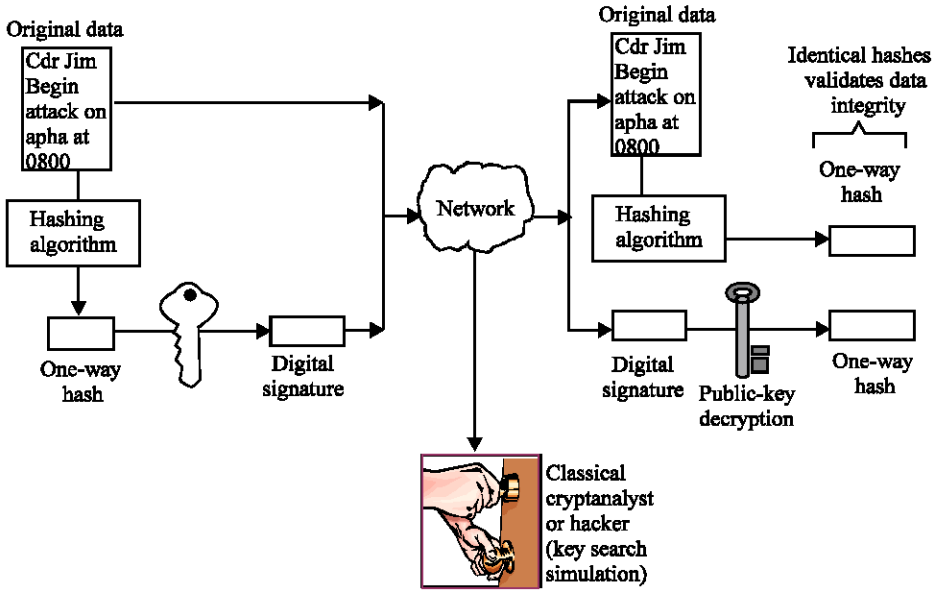

Fig. 6: An implementation of digital signature algorithm

How does digital signature works: How can user Bob send Alice "signed" message T_B in a public-key cryptosystems? He first uses his own secret-key or D_B to encrypt his "digital signature", S_B according to $D_B(S_B)$ given in Eq. (5). (Deciphering the unenciphered message "makes sense" by property (d) of public-key cryptosystem: each message is the ciphertext for some other message). He then appends his encrypted digital signature to his message, T_B, to produce the signed message, $T_B \cdot D_B(S_B)$, where the dot denotes concatenation. Next Bob applies Eq. (4) to his signed message, using Alice's public-key E_A (for privacy) to obtain the ciphertext: $C_B = E_A(T_B \cdot D_B(S_B))$ and transmits it to Alice.

When Alice receives C_B she first applies Eq. (5) using her secret-key to produce, $D_A(C_B) = T_B \cdot D_B(S_B)$. Thus, the message T_B will appear, along with a portion of gibberish at the end of the message. To authenticate T_B, Alice uses Bob's public-key, E_B, (available on the public-key server) to perform, $E_B(D_B(S_B)) = S_B$. If Bob's digital signature appears, she knows the message is authentic.

She now possesses a message-signature pair (T_B, S_B) with properties similar to those of signed paper document. Bob cannot deny having sent Alice this message, since no one else could have created $SB = DB(T^B_{DS})$ (where T^B_{DS} is Bob's "plaintext signature" message). Alice can convince a "judge" that $E_B(S_B) = T^B_{DS}$, so she has proof that Bob signed the document.

Clearly Alice cannot modify T_B to different version T_B', since then she would have to create the corresponding signature, $S_B' = D_B(T_B')$, as well. Therefore, Alice has received a message "signed" by Bob, which she can "prove" that he sent, but which she cannot modify. (Nor can she forge his signature for any other message.)

The only remaining issue involves selecting the digital signature S_B. If Bob uses the same digital signature in every message, Eve will be able to detect this by looking for common string among Bob's transmissions. Even though by doing this Eve will only discover $D_B(S_B)$, this all she needs in order to sign a rogue message and misrepresent herself as Bob to Alice.

Therefore, it is important for Bob to use a different S_B in every message. One strategy is to make S_B depend on the message T_B. Hash functions are commonly used to implement this strategy. In this setting a public hash function h is required to transform a variable-length message into a fixed-length message fingerprint F, i.e.,

$$h: T_b \rightarrow F \qquad (6)$$

Bob's ciphertext message to Alice is encrypted using:

$$C_B = E_A(T_B \cdot D_B(F)) \qquad (7)$$

After applying D_A to this ciphertext, Alice can authenticate it by first computing $E_B(D_B(F)) = F$ and then comparing this result to the result she obtains by applying the hash function h to T_B.

We have already mentioned that there are many instances in which the main security issue is authentication and not secrecy. For example, a financial institution may be content with sending and receiving their transactions unencrypted, as long as they can

guarantee that these transactions are not altered. Specifically, if Bob sends messages, $T_B \cdot D_B(F)$ to Alice, then even though Eve can read T_B, she will not be able to alter it unless she is able to determine F.

In order for a public hash function h to be effective, it must posses at least the following two properties. First, since h is known to all, for any y it must be computationally intractable to find T_B such that, $h(T_B)=y$. In other words, Eve should have great difficulty in trying to invert h in order to obtain T_B. Second, it should be computationally intractable to find messages that collide[43]. To see why, assume we have a hash function h that does not satisfy this property. Now suppose Eve constructs two messages T_B and T_B' such that, $h(T_B) = h(T_B')$ and Bob is perfectly happy to sign T_B but not T_B'. If Eve can convince Bob to sign T_B, then Eve will also be able to achieve her fraudulent goal of signing T_B' with Bob's digital signature.

Many digital hashing schemes are based on the following idea. Let h' be a hash function that maps s-bit keys to k-bit values, for some fixed $s > k$. From h' we construct a public hash function that produces a k-bit messages fingerprint by first breaking the message T_B into blocks, T_{B1}, each containing, T_{B1}, T_{B2},.....,T_{Bt}, each containing, s-k bits. Next let:

$$Fi(T_{Bi}) = h'(F_{i-1} \cdot T_{Bi})$$

where the dot denotes concatenation and F_0 is a k-bit initialization value, often chosen as all zeros. The message fingerprint is then given by F_t.

Message digest: A message digest is a compact digital signature for an arbitrarily long stream of binary data. An ideal message digest algorithm would never generate the same signature for two different sets of input, but achieving such theoretical perfection would require a message digest as long as the input file. Practical message digest algorithms compromise in favor of a digital signature of modest size created with an algorithm designed to make preparation of input text with a given signature computationally infeasible. Message digest algorithms have much in common with techniques used in encryption, but to a different end; verification that data have not been altered since the signature was published.

Many older programs requiring digital signatures employ 16 or 32 bit cyclical redundancy codes (CRC) originally developed to verify correct transmission in data communication protocols, but these short codes, while adequate to detect the kind of transmission errors for which they were intended, are insufficiently secure for applications such as electronic commerce and verification of security related software distributions.

The most commonly used present-day message digest algorithm is the 128-bit MD5 algorithm, developed by Ron Rivest of the MIT Laboratory for Computer Science[44]. Message digest algorithms such as MD5 are not deemed "encryption technology" and are not subject to the export controls some governments impose on other data security products. The MD5 algorithm was originally developed as part of a suite of tools intended to monitor large collections of files (for example, the contents of a Web site) to detect corruption of files and inadvertent (or perhaps malicious) changes.

Other possible application of DS: Electronic checking: - An electronic checking system could be based on signature system such as the above. It is easy to imagine an encryption device in your home terminal allowing you to sign checks that get sent by electronic mail to the payee. It would only be necessary to include a unique check number in each check so that even if the payee copies the check the bank will only honor the first version it sees.

Speech encryption: Another possibility arises if encryption devices can be made fast enough: it will be possible to have a telephone conversation in which every word spoken is signed by the encryption device before transmission.

We have assumed above that each user can always access the public-key server reliably. In a "computer network" this might be difficult; an intruder "Eve" might forge messages purporting to be from the public-key server. The user would like to be sure that he actually obtains the encryption procedure of desire correspondent and not, say, the encryption procedure of the intruder. This danger disappears if the public-key server "signs" each message it sends to a user. The user can check the signature with the public-key server's encryption algorithm E_{PF}. The problem of "looking up" E_{PF} itself in the public-key server is avoided by giving each user a decryption of E_{PF} when he first shows up (in person) to join the public-key cryptosystem and deposit his public encryption procedure. He then stores this description rather than ever looking it up again. The need for a courier between every pair of users has thus been replaced by the requirement for a single secure meeting between each user and the public-key server manager when user joins the system. Another solution is to give each user, when he signs up, a book (like a telephone directory) containing all the encryption keys of users in the system.

In general, the strength of encryption is related to the difficulty of discovering the key, which in turn depends on both the cipher used and the length of the key. For example, the difficulty of discovering the key for the RSA cipher most commonly used for public-key encryption depends on the difficulty of factoring large numbers, a well-known mathematical problem. Encryption strength is often described in terms of the size of the keys used to perform the encryption: in general, longer keys provide stronger encryption. Different ciphers may require different key lengths to achieve the same level of encryption strength. Key length is measured in bits. Thus, a 128-bit key for use with a symmetric-key encryption cipher would provide stronger encryption than a 128-bit key for use with the RSA public-key encryption cipher. This difference explains why the RSA public-key encryption cipher must use a 512-bit key (or longer) to be considered cryptographically strong, whereas symmetric key ciphers can achieve approximately the same level of strength with a 64-bit key. Even this level of strength may be vulnerable to attacks in the near future.

We have also proposed a method for implementing a public-key cryptosystem whose security rests in part on the difficulty of factoring large numbers. If the security of method is appropriately implemented, it permits secure communications to be established without the use of courier to carry keys and it also permits one to "sign" digital documents.

No matter which technique you choose, you must keep in mind that a desperate cryptanalyst can always decipher the message. Hence, you should always take all the necessary precautions to protect your data. Those precautions range from proper choice of cryptographic keys to physically protecting your assets and yourself.

Because the ability to surreptitiously intercept and decrypt encrypted information has historically been a significant military asset, the U.S. Government restricts export of cryptographic software, including most software that permits use of symmetric encryption keys longer than 40 bits. However, novel techniques for confidentiality are interesting in part because of the current debate about cryptographic policy as to whether law enforcement should be given authorized surreptitious access to the plaintext of encrypted messages. The usual technique proposed for such access is "key recovery", where law enforcement has a "back door" that enables them to recover the decryption key. (Obviously, the responsibility for obeying the laws in the jurisdiction in which you reside is entirely your own, but many common Web and Mail utilities use MD5 and I am unaware of any restrictions on their distribution and use.)

Appendix A – Working Source Codes

Appendix A1: Caesar.java: implements the Caesar cipher

```
// This carries out a simple rotation of lower-case letters and does
// nothing to all other characters, making the decryption process even
// easier, because caps and punctuation marks survive unchanged.
// Usage: java Caesar (-d | -e) key (java Caesar –e|-d 3 < message.text)
// Above, option "-d" is for decryption, while "-e" is for encryption

import java.io.*;

public class Caesar {
  private Reader in; // standard input stream for message
  private int key; // (en|de)cryption key

  // Caesar: constructor, opens standard input, passes key
  public Caesar(int k) {
    // open file
    in = new InputStreamReader(System.in);

    key = k;
  }

  // (en|de)crypt: just feed in opposite parameters
  public void encrypt() { translate(key); }
  public void decrypt() { translate(-key); }

  // translate: input message, translate
  private void translate(int k) {
    char c;
    while ((byte)(c = getNextChar()) != -1) {
      if (Character.isLowerCase(c)) {
        c = rotate(c, k);
      }
      System.out.print(c);
    }
  }

  // getNextChar: fetches next char.
  public char getNextChar() {
    char ch = ' '; // = ' ' to keep compiler happy
    try {
      ch = (char)in.read();
    } catch (IOException e) {
      System.out.println("Exception reading character");
    }
    return ch;
  }

  // rotate: translate using rotation, version with table lookup
  public char rotate(char c, int key) { // c must be lowercase
    String s = "abcdefghijklmnopqrstuvwxyz";
    int i = 0;
    while (i < 26) {
        // extra +26 below because key might be negative
      if (c == s.charAt(i)) return s.charAt((i + key + 26)%26);
        i++;
    }
    return c;
  }

  // main: check command, (en|de)crypt, feed in key value
  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: java Caesar (-d | -e) key");
      System.exit(1);
    }
    Caesar cipher = new Caesar(Integer.parseInt(args[1]));
    if (args[0].equals("-e")) cipher.encrypt();
    else if (args[0].equals("-d")) cipher.decrypt();
    else {
```

```
        System.out.println("Usage: java Caesar (-d | -e) key");
        System.exit(1);
      }
    }
}
```

Run Test

```
C:\> java Caesar -e 3 < message.txt //encryption
        commander jim commence attack on alpha          //message

C:\> java Caesar -e 3 < message.text | java Caesar -d 3 //decryption
        crppdqghu jlp frpphqfh dwwdfn rq doskd
```

Appendix A2: XOR.cpp implements Exclusive OR (XOR) to Perform Block Encryption Code

```
// Author: Kefa Rabah
// A Simple implementation of XOR function (XOR.cpp)
// Usage: XOR key input_file output_file
// Purpose: This program takes a "plaintext" file and performs XOR
// between each character of the file and the supplied key
// and places the encrypted result in the "cipher" file.

        #include <stdio.h>
        #include <iostream.h>
        #include <fstream.h>
        #include <math.h>
        #include <ctype.h>
        #include <stdlib.h>
        #include <conio>

         main(int argc, char *argv[])
        {
        FILE *ifp, *ofp;
        char *cp;
        int c;

        if (argc !=4){
        printf("\n%s%s%s\n\n%s\n%s\n\n",
        "Usage: ", argv[0], " key infile outfile",
        "The plaintext letters in infile will be encripted.",
        "The results will be written in outfile.");
        exit(1);
    }
if(cp = argv[1]){
  if ((ifp = fopen(argv[2], "rb"))!=NULL){
  if ((ofp = fopen(argv[3], "wb"))!= NULL){

        if (ifp == NULL || ofp == NULL)
        {
        printf("\n%sSorry. Files cannot be opened\n.");
        return -1;
        }

        while ((c = getc(ifp)) != EOF)
        {
        if (!*cp) cp = argv[1];
        c ^= *(cp++);
        putc(c, ofp);
        }
        fclose(ofp);
           }
        fclose(ifp);
        }
        }
        getch();
        return 0;
        };
```

Appendix A3: - SDES.java implements Simplified DES

```
// simplified DES
//
// Kefa Rabah, June. 2003

public class SDES
{
 // subkeys
 //
  public int K1, K2;

 // permutations
 //
  public static final int P10[] = { 3, 5, 2, 7, 4, 10, 1, 9, 8, 6};
  public static final int P10max = 10;

  public static final int P8[] = { 6, 3, 7, 4, 8, 5, 10, 9};
  public static final int P8max = 10;

  public static final int P4[] = { 2, 4, 3, 1};
  public static final int P4max = 4;

  public static final int IP[] = { 2, 6, 3, 1, 4, 8, 5, 7};
  public static final int IPmax = 8;

  public static final int IPI[] = { 4, 1, 3, 5, 7, 2, 8, 6};
  public static final int IPImax = 8;

  public static final int EP[] = { 4, 1, 2, 3, 2, 3, 4, 1};
  public static final int EPmax = 4;

  public static final int S0[][] = {
    { 1, 0, 3, 2},
    { 3, 2, 1, 0},
    { 0, 2, 1, 3},
    { 3, 1, 3, 2}
  };

  public static final int S1[][] = {
    { 0, 1, 2, 3},
    { 2, 0, 1, 3},
    { 3, 0, 1, 0},
    { 2, 1, 0, 3}
  };

 // permute bits
 //
  public static int permute( int x, int p[], int pmax)
  {
    int y = 0;

    for( int i = 0; i < p.length; ++i) {
      y <<= 1;
      y |= (x >> (pmax - p[i])) & 1;
    }

    return y;
  }

 // F function
 //
  public static int F( int R, int K)
  {
    int t = permute( R, EP, EPmax) ^ K;
    int t0 = (t >> 4) & 0xF;
    int t1 = t & 0xF;

    t0 = S0[ ((t0 & 0x8) >> 2) | (t0 & 1) ][ (t0 >> 1) & 0x3 ];
    t1 = S1[ ((t1 & 0x8) >> 2) | (t1 & 1) ][ (t1 >> 1) & 0x3 ];
```

```
  t = permute( (t0 << 2) | t1, P4, P4max);

  return t;
 }

// fK function
//
 public static int fK( int m, int K)
 {
  int L = (m >> 4) & 0xF;
  int R = m & 0xF;

  return ((L ^ F(R,K)) << 4) | R;
 }

// switch function
//
 public static int SW( int x)
 {
  return ((x & 0xF) << 4) | ((x >> 4) & 0xF);
 }

// encrypt one byte
//
 public byte encrypt( int m)
 {
  m = permute( m, IP, IPmax);
  m = fK( m, K1);
  m = SW( m);
  m = fK( m, K2);
  m = permute( m, IPI, IPImax);

  return (byte) m;
 }

// decrypt one byte
//
 public byte decrypt( int m)
 {
  m = permute( m, IP, IPmax);
  m = fK( m, K2);
  m = SW( m);
  m = fK( m, K1);
  m = permute( m, IPI, IPImax);

  return (byte) m;
 }

// print n bits in binary
//
 public static void printb( int x, int n)
 {
  int mask = 1 << (n-1);

  while( mask > 0) {
   System.out.print( ((x & mask) == 0) ? '0' : '1');
   mask >>= 1;
  }
 }

// constructor - initializes K1 and K2
//
 public SDES( int K)
 {
  K = permute( K, P10, P10max);

  // 5-bit parts of K
  //
  int t1 = (K >> 5) & 0x1F;
  int t2 = K & 0x1F;

  // LS-1
  //
  t1 = ((t1 & 0xF) << 1) | ((t1 & 0x10) >> 4);
  t2 = ((t2 & 0xF) << 1) | ((t2 & 0x10) >> 4);

  K1 = permute( (t1 << 5) | t2, P8, P8max);

  // LS-2
  //
  t1 = ((t1 & 0x7) << 2) | ((t1 & 0x18) >> 3);
  t2 = ((t2 & 0x7) << 2) | ((t2 & 0x18) >> 3);

  K2 = permute( (t1 << 5) | t2, P8, P8max);
 }

}

// SDES Encrypt
// For Decryption process, comment “Class Encrypt” and uncomment
“Class Dencrypt”.
// K. Rabah, July. 2003

//public class Decrypt
public class Encrypt
{
 public static void main( String args[]) throws Exception
 {
  if( args.length != 2) {
   System.err.println( "Usage: Encrypt key plaintext");
   System.exit(1);
  }

  // first command-line argument is 10-bit key in binary
  //
  int K = Integer.parseInt( args[0], 2);

  // create an instance of the SDES Algorithm
  //
  SDES A = new SDES( K);

  // second command-line argument is 8-bit plaintext in binary
  //
  int m = Integer.parseInt( args[1], 2);

  // encrypt or decrypt
  //
  m = A.encrypt( m); //uncomment for encryption

  //m = A.dencrypt( m); //comment for encryption

  // display in binary
  //
  SDES.printb( m, 8);
  System.out.println();
 }
}

/*
```

SAMPLE RUN:

```
java Encrypt 0000011111 11111111
11100001

java Decrypt 0000011111 11100001
11111111

*/
```

Appendix A4: prime.c – Implements Prime Number Generator - Sieve Algorithm

The following code segment shows a simple implementation of the sieve algorithm that can be used to check whether or not a particular number r is a prime number:

```
/ Author: Kefa Rabah
// A Simple implementation of Prime Number Generator.
/*
Simpe Classical way for determinig prime numbers using
Aristocenes arrays solution  (original length 100000).
*/

#include <stdio.h>
#include <conio.h>
#define LENGTH 1000
void main()
{
 long vec[LENGTH],i,j;
 clrscr();
 for (i=0; i<LENGTH; i++)
    vec[i]=i;

 for (i=2; i<LENGTH; i++)
    for (j=i*2; j<LENGTH; j+=i)
       vec[j]=0;

 for (i=0; i<LENGTH; i++)
    if (vec[i])
       printf("%d ",vec[i]);

 getch();
}
```

Appendix B: Mathematical Methods for RSA

Appendix B1: Chinese Remainder Theorem

This theorem provides a way to combine two modular equations that use different moduli.

Theorem: $x = y(\text{mod} p) \Rightarrow x = y(\text{mod } q)$ with p and q coprime, such that: $x = y(\text{mod } pq)$

Proof:

$x = y(\text{mod } p) \Rightarrow x = y + kp \Rightarrow x - y = kp$

That is, p divides (x - y) as p and q are coprime, which we can write as:

$x - y = \lambda(pq) \Rightarrow x = y(\text{mod } pq)$

where λ is an integer.

Appendix B2: Fermat/Euler Theorem

This theorem is a surprising identity that relates the exponent to the modulus.

Theorem: $x^{p-1} = 1(\text{mod } p)$ if p is prime and $x \neq 0(\text{mod } p)$

Proof: Consider the set Q, of numbers 1, 2, ..., p-1

As p is prime, these numbers are coprime to p. But 0 is not coprime to p

⇒ Q includes all the numbers in (mod p) coprime to p

Now consider the set U, obtained by multiplying each element of Q by x(mod p). Both x and each element of Q are coprime to p, implies each element of U is coprime to p.

Also, each element of U is distinct, which we prove by contradiction. We start by assuming two elements are not distinct:

$xQ_i = xQ_j \text{ (mod p) with } i \neq j \Rightarrow Q_i = Q_j \text{ (mod p) as } x \neq 0$

but elements of Q are distinct, so this is a contradiction

⇒ elements of U are distinct

So, U uses all the numbers in (mod p) that are coprime to p, just like Q which implies that U is a permutation of Q. Hence, we can write as:

$U_1 \cdot U_2 \cdot \ldots \cdot U_{p-1} = Q_1 \cdot Q_2 \cdot \ldots \cdot Q_{p-1} (\text{mod } p)$

$xU_1 \cdot xU_2 \cdot \ldots \cdot xU_{p-1} = Q_1 \cdot Q_2 \cdot \ldots \cdot Q_{p-1} (\text{mod } p)$

and if we cancel $Q_1 \cdot Q_2 \cdot \ldots \cdot Q_{p-1}$, we get:

$x^{p-1} = 1(\text{mod } p)$

Appendix B3: RSA Correctness

Here we prove that the combined process of encrypting and decrypting a message correctly results in the original message.

Theorem: $C = M^e (\text{mod } n)$ $M' = C^d (\text{mod } n)$ $M' = M (\text{mod } n)$

where (d, e, n) is a valid RSA key, with $n = pq$ and $0 < M < \min(pq)$

Proof: First we combine the two exponents:

$$M' = M^{ed} (\text{mod } n) \quad (1)$$

where d and e are generated so that: $de = k(p-1)(q-1) + 1$, such that Eq. (1) yields:

$$M' = M^{k(p-1)(q-1)+1} (\text{mod } n)$$

$$= M \cdot M^{k(p-1)(q-1)} (\text{mod } n) \quad (2)$$

Now consider:

$$X = M^{k(p-1)(q-1)} (\text{mod } p)$$

$$= M^{(p-1)k(q-1)} (\text{mod } p)$$

The Fermat/Euler theorem tells us that: $M^{(p-1)} = 1(\text{mod } p)$, hence:

$$X = 1^{k(q-1)} (\text{mod } p) \Rightarrow X = 1(\text{mod } p)$$

By a similar route: $X = 1(\text{mod } n)$

As p and q are distinct primes, we can combine these with the Chinese remainder theorem: $X = 1(\text{mod } pq)$, giving rise to:

$$M^{k(p-1)(q-1)} = 1 (\text{mod } n) \quad (3)$$

Finally, we substitute Eq. (3) back into the Eq. (2)

$$M' = M*1 (\text{mod } n)$$
$$= M (\text{mod } n)$$

Appendix B4: Mathematic Implementation of RSA Algorithm

P = 61 <- first prime number (destroy this after computing E and D)
Q = 53 <- second prime number (destroy this after computing E and D)
PQ = 3233 <- modulus (give this to others)
E = 17 <- public exponent (give this to others)
D = 2753 <- private exponent (keep this secret!)

Your public key is: (E,PQ).
Your private key is: (D,PQ)..

The encryption function is:

encrypt(T) = (T^E) mod PQ
= (T^17) mod 3233

The decryption function is:

decrypt(C) = (C^D) mod PQ
= (C^2753) mod 3233

To encrypt the plaintext value 123, do this:

encrypt(123) = (123^17) mod 3233
= 337587917446653715596592958817679803 mod 3233
= 855

To decrypt the ciphertext value 855, do this:

decrypt(855) = (855^2753) mod 3233
= 123

One way to compute the value of 855^2753 mod 3233 is like this:

2753 = 101011000001 base 2, therefore
2753 = 1 + 2^6 + 2^7 + 2^9 + 2^11
= 1 + 64 + 128 + 512 + 2048

Consider this table of powers of 855:

855^1 = 855 (mod 3233)
855^2 = 367 (mod 3233)
855^4 = 367^2 (mod 3233) = 2136 (mod 3233)
855^8 = 2136^2 (mod 3233) = 733 (mod 3233)
855^16 = 733^2 (mod 3233) = 611 (mod 3233)
855^32 = 611^2 (mod 3233) = 1526 (mod 3233)
855^64 = 1526^2 (mod 3233) = 916 (mod 3233)
855^128 = 916^2 (mod 3233) = 1709 (mod 3233)
855^256 = 1709^2 (mod 3233) = 1282 (mod 3233)
855^512 = 1282^2 (mod 3233) = 1160 (mod 3233)
855^1024 = 1160^2 (mod 3233) = 672 (mod 3233)
855^2048 = 672^2 (mod 3233) = 2197 (mod 3233)

Given the above, we know this:

855^2753 (mod 3233)
= 855^(1 + 64 + 128 + 512 + 2048) (mod 3233)
= 855^1 * 855^64 * 855^128 * 855^512 * 855^2048 (mod 3233)
= 855 * 916 * 1709 * 1160 * 2197 (mod 3233)
= 794 * 1709 * 1160 * 2197 (mod 3233)
= 2319 * 1160 * 2197 (mod 3233)
= 184 * 2197 (mod 3233)
= 123 (mod 3233)
= 123

Appendix B5: An Implementation of RSA Algorithm using Java

```
// A Java implementation of RSA is just a transcription of the algorithm.

  // RSAPublicKey: RSA public key

  import java.math.*; // for BigInteger

  public class RSAPublicKey {
    public BigInteger n; // public modulus
    public BigInteger e = new BigInteger("3"); // encryption exponent
     public String userName; //attach name to each public/private key pair

    public RSAPublicKey(String name) {
      userName = name;
    }

    // setN: to give n a value in case only have public key
    public void setN(BigInteger newN) {
      n = newN;
    }

    // getN: provide n
    public BigInteger getN() {
      return n;
    }

    // RSAEncrypt: just raise m to power e (3) mod n
    public BigInteger RSAEncrypt(BigInteger m) {
      return m.modPow(e, n);
    }

    // RSAVerify: same as encryption, since RSA is symmetric
    public BigInteger RSAVerify(BigInteger s) {
      return s.modPow(e, n);
    }
  }


  // RSAPrivateKeyFast: RSA private key, using fast CRT algorithm

  import java.math.*; // for BigInteger
  import java.util.*; // for Random

  public class RSAPrivateKeyFast extends RSAPublicKey{
    private final BigInteger TWO = new BigInteger("2");
    private final BigInteger THREE = new BigInteger("3");

    private BigInteger p; // first prime
    private BigInteger q; // second prime
    private BigInteger d; // decryption exponent

    private BigInteger p1, pM1, q1, qM1, phiN; // for key generation
    private BigInteger dp, dq, c2; // for fast decryption
       public RSAPrivateKeyFast(int size, Random rnd, String name) {
      super(name); generateKeyPair(size, rnd);
    }
```

```
  public void generateKeyPair(int size, Random rnd) { // size = n in bits
  // want sizes of primes close, but not too close. Here 10-20 bits apart.
    int size1 = size/2;
    int size2 = size1;
    int offset1 = (int)(5.0*(rnd.nextDouble()) + 5.0);
    int offset2 = -offset1;
    if (rnd.nextDouble() < 0.5) {
      offset1 = -offset1; offset2 = -offset2;
    }
    size1 += offset1; size2 += offset2;
    // generate two random primes, so that p*q = n has size bits
    p1  = new BigInteger(size1, rnd); // random int
    p   = nextPrime(p1);
    pM1 = p.subtract(BigInteger.ONE);
    q1  = new BigInteger(size2, rnd);
    q   = nextPrime(q1);
    qM1 = q.subtract(BigInteger.ONE);
    n   = p.multiply(q);
    phiN = pM1.multiply(qM1); // (p-1)*(q-1)
    d   = e.modInverse(phiN);
    // remaining stuff needed for fast CRT decryption
    dp = d.remainder(pM1);
    dq = d.remainder(qM1);
    c2 = p.modInverse(q);
  }

  // nextPrime: next prime p after x, with p-1 and 3 relatively prime
  public BigInteger nextPrime(BigInteger x) {
    if ((x.remainder(TWO)).equals(BigInteger.ZERO))
      x = x.add(BigInteger.ONE);
    while(true) {
      BigInteger xM1 = x.subtract(BigInteger.ONE);
      if (!(xM1.remainder(THREE)).equals(BigInteger.ZERO))
        if (x.isProbablePrime(10)) break;
      x = x.add(TWO);
    }
    return x;
  }

  // RSADecrypt: decryption function, fast CRT version
  public BigInteger RSADecrypt(BigInteger c) {
      // See 14.71 and 14.75 in Handbook of Applied Cryptography, by
    //   Menezes, van Oorschot and Vanstone
    // return c.modPow(d, n);
    BigInteger cDp = c.modPow(dp, p);
    BigInteger cDq = c.modPow(dq, q);
    BigInteger u = ((cDq.subtract(cDp)).multiply(c2)).remainder(q);
    if (u.compareTo(BigInteger.ZERO) < 0) u = u.add(q);
    return cDp.add(u.multiply(p));
  }

   // RSASign: same as decryption for RSA (since it is a symmetric
PKC)
  public BigInteger RSASign(BigInteger m) {
    // return m.modPow(d, n);
    return RSADecrypt(m); // use fast CRT version
  }

   public BigInteger RSASignAndEncrypt(BigInteger m, RSAPublicKey
other) {
    // two ways to go, depending on sizes of n and other.getN()
    if (n.compareTo(other.getN()) > 0)
      return RSASign(other.RSAEncrypt(m));
    else
      return other.RSAEncrypt(RSASign(m));
  }

  public BigInteger RSADecryptAndVerify(BigInteger c,
                          RSAPrivateKeyFast other) {
    // two ways to go, depending on sizes of n and other.getN()
    if (n.compareTo(other.getN()) > 0)
      return other.RSAVerify(RSADecrypt(c));
    else
      return RSADecrypt(other.RSAVerify(c));
  }
}


// RSATestFast: Test Fast RSA Implementation
import java.math.*; // for BigInteger
import java.util.*; // for Random

public class RSATestFast {

  public static void elapsedTime(long startTime) {
    long stopTime = System.currentTimeMillis();
    double elapsedTime = ((double)(stopTime - startTime))/1000.0;
    System.out.println("Elapsed time: " + elapsedTime + " seconds");
  }

  public static void main(String[] args) {
    Random rnd = new Random();
    BigInteger m, m1, m2, m3, c, s, s1;
      RSAPrivateKeyFast alice = new RSAPrivateKeyFast(1024, rnd,
"Alice");
      RSAPrivateKeyFast bob   = new RSAPrivateKeyFast(1024, rnd,
"Bob  ");
    m = new BigInteger(
     "12345678909876543210123456789098765432l0" +
     "12345678909876543210123456789098765432l0" +
     "12345678909876543210123456789098765432l0" +
     "12345678909876543210123456789098765432l0" +
     "12345678909876543210123456789098765432l0" +
     "12345678909876543210123456789098765432l0");
    System.out.println("Message m:\n" + m + "\n");
       System.out.println("ALICE ENCRYPTS m FOR BOB; BOB
DECRYPTS IT:");
    c = bob.RSAEncrypt(m); // Using Bob's public key
     System.out.println("Message encrypted with Bob's public key:\n" +
      c + "\n");
    m1 = bob.RSADecrypt(c); // Using Bob's private key
      System.out.println("Original message back, decrypted:\n" + m1 +
"\n");
     System.out.println("ALICE SIGNS m FOR BOB; BOB VERIFIES
SIGNATURE:");
    s = alice.RSASign(m); // Using Alice's private key
    System.out.println("Message signed with Alice's private key:\n" +
      s + "\n");
    m2 = alice.RSAVerify(s); // Using Alice's public key
       System.out.println("Original message back, verified:\n" + m2 +
"\n");
         System.out.println("BOB SIGNS AND ENCRYPTS m FOR
ALICE;" +
       "\n   ALICE VERIFIES SIGNATURE AND DECRYPTS:");
    c  = bob.RSASignAndEncrypt(m, alice);
    System.out.println("Message signed and encrypted," +
        "\n  using Bob's secret key and Alice's public key:\n" + c + "\n");
    m3 = alice.RSADecryptAndVerify(c, bob);
     System.out.println("Original message back, verified and decrypted,"
+
       "\n  using Alice's secret key and Bob's public key:\n" + m1);
  }
}
```

A Test Run: Here is a run of the above test class, showing simple encryption, signing and a combination of signing and encryption. MS-DOS command line appear in boldface.

```
C:\> javac RSAPublicKey.java
C:\> javac RSAPrivateKeyFast.java
C:\> javac RSATestFast.java
C:\> java RSATestFast

Message m:
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321

ALICE ENCRYPTS m FOR BOB; BOB DECRYPTS IT:
Message encrypted with Bob's public key:

54340581367664807805701276287259968381366767413365992537733576
05567555516424469233387398561035220096421942902314004442496355 3
92009986359056374479092883194576861821720618133177330634846 25
94171529440296314258756692666524438783703841869144887617324529
23241511506638612625965339071681261723119229735067607013528 7

Original message back, decrypted:
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321

ALICE SIGNS m FOR BOB; BOB VERIFIES SIGNATURE:
Message signed with Alice's private key:
23999062709216358693836072721907187585572596559729003884362678
43340567443761018097412829464289935736559871836409863729003566
78910437032277772334474986578993935720568974198358713462782149
86967876889715158405039121980012395643624344524871519902599537
12668674009471364227890694971856927150342941098035705104040

Original message back, verified:
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321

BOB SIGNS AND ENCRYPTS m FOR ALICE;
  ALICE VERIFIES SIGNATURE AND DECRYPTS:
Message signed and encrypted,
  using Bob's secret key and Alice's public key:
55568095448922845163395618641245097592442739125869528222428235
06079933908919391816863062327609127060035395937753704903768704
45903174464182907612502285232696602221467528497111242219800301
03548023484747053340324451311160479401069781901832028916581722
48332837983635709085998517756886150571672421606040611712970

Original message back, verified and decrypted,
      using Alice's secret key and Bob's public key:
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
12345678909876543210123456789098765432101234567890987654321
```

REFERENCES

1. Menezes, A., P. Van Oorschot and S. Vanstone, 1997. Handbook of Applied Cryptography. CRC Press.
2. Smid, M.E. and D.K. Branstead, 1988. The Data Encryption Standard: Past and Future, Proceedings of the IEEE., 76: 550-559.
3. Kahn, D., 1983. The Codebreakers: The story of Secret Writing. New York, Macmillan.
4. Welchman, G., 1982. The Hut Six Story: Breaking the Enigma Codes. McGraw-Hill.
5. Feistel, H., 1973. Cryptography and Computer Privacy. Scientific America, 228: 15-23.
6. Smith, J.L., 1971. The Design of Lucifer, A Cryptographic Device for Data Communications. IBM Research Report RC3326.
7. Andleman, D. and J. Reeds, 1982. On the Cryptanalysis of Rotor Machines and Substitution-Permutation Networks. IEEE Transactions on Information Thoery, v. IT-28: 578-584.
8. Bellovin, S.M. and M. Merritt, 1992. Encrypted Key Exchange: Password-Based Protocols secure Against Dictionary Attacks. Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy, pp: 72-84.
9. Bihan, E., 1992. New Types of Cryptanalytic Attacks Using Related Keys. Technical Report #753, Computer Science Department, Technion-Israel Institute of Technology.
10. Campbell, C.M., 1978. Design and Specification of Cryptographic Capabilities. IEEE Computer Society Magazine, 16: 15-19.
11. Heileman, G.L., 1996. Data Structures, Algorithms and Object Oriented Programing. MGraw-Hill International Edition, New York.
12. Christofferssonm, P., S.A. Ekahll, V. Fak, S. Herda, P. Mattila, W. Price and H.O. Widman, 1988. Crypto Users Handbook, A guide for Implementers of Cryptographic Protection in Computer Systems. North Holland: Elsevier Science Publishers.
13. Kocher, P.C., 1996. Timing Attacks on Implementations of Diffie-Hellman., RSA, DSS and Other Systems. In Crypto'96,LNCS,. Springer-Verlag, New York, pp: 104-113.
14. Davio, M., Y. Desmedt, J. Goubert, F. Hoornaert and J.J. Quisquater, 1985. Efficient Hardware and Software Implementation of DES. Advances in Cryptology–EUROCRYPT'84 Proceedings, Berlin: Springer-Verlag, pp: 62-73.
15. Schneier, B., 1993. Applied Cryptography. John Wwley and Sons New York.
16. Blakley, G.R., 1979. Safeguarding Cryptographic Keys. Proceedings of the National Computer Conference, 1979. American Federation of Imformation Processing Societies, pp: 242-268.
17. Bellare, M., Roch Guérin and Philip Rogaway, 1995. XOR MACs: New methods of message authentication using block cipher. Accepted to Crpto '95.
18. Feistel, H., 1974. Block Cipher Cryptographic System. U.S. Patent #3,798,359,19.
19. Davio, M., Y. Desmedt, M. Fosseprez, R. Govaerts, J. Hulsbrosch, P. Neutjens, P. Piret, J.J. Quisquater, J. Vandewalle and S. Wouters, 1984. Analytical Characteristics of the Data Encryption Standard. Advances in Cryptology: Proceedings of Crypto 83, Plenum Press, pp: 171-202.

20. Branstead, D.K., J. Gait and S. Katzke, 1977. Report on the Workshop on Cryptography in Support of Computer Security. NBSIR 77-1291, National Bureau of Standards, pp: 21-22.
21. Brown, L. and J. Seberry, 1990. On the Deign of Permutation in DES Type Cryptosystems. Advances in Cryptology: Proceedings of EUROCRYPT 89, Berlin, pringer-Velag, pp: 696-705.
22. Shannon, C.E., 1993. Collected Papers: Claude Elmwood Shannon. N. J. A. Sloane and A. D. Wyner, Eds., New York: IEEE Press.
23. Shannon, C.E., 1948. A Mathematical Theory of Communication. Bell System Technical J., 27: 379-423, 623-656.
24. Shannon, C.E., 1949. Communication Theory of Secrete Systems Bell Technical J., 28: 656-715.
25. Schneier, B., 1994. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). Springer-Verlag, pp: 191-204.
26. Lexar Corporation, 1976. An Evaluation of the DES.
27. Davis, R.M., 1978. The Data Security Standard in Perspective. Computer Security and the Data Encryption Standard, National Bureau of Standards Special Publication 500-27.
28. WDT., 1980. With Data Encryption, Scents Are Safe at IFF, Computerworld 14, No. 21, 95.
29. Kroph, T., J. Frosl, W. Beller and T. Giesler, 1990. A Hardware Implementation of Modified DES Algorithm. Microprocessing and Microprogramming, 30: 59-66.
30. Knuth, D.E., 1969. The Art of Computer Programming. Vol. 2, Seminumerical Algorithms Addison-Wesley, Reading, Mass.
31. Maurer, U.M. and J.L. Massey, 1990. Perfect Local Randomness in Pseudo-Random Sequences. Advances in Cryptology – CRYPTO'89 Proceedings, Berlin: Springer-Verlag, pp: 110-112.
32. Bellare, M. and P. Rogaway, 1994. Optimal Asymmetric Encryption. In: A. De Santis, Ed., Proceedings of Crypto'94, vol. 950 of Lecture Notes in Computer Science,. Springer-Verlag, New York, pp: 92-111.
33. Merkle, R. C., 1978. Secure Communication Over Insecure Channels. Communications of the ACM, 21: 294-299.
34. Diffie, W. and M.E. Hellman, 1976. Multi-user Cryptographic Techniques. Proceedings of AFIPS National Computer Conference, pp: 109-112.
35. Diffie, W. and M.E. Hellman, 1966. New Directions in Cryptography. IEEE Transaction on Information Theory, IT-22: 644-654.
36. Rivest, R., 1983. A. Shamir and L.M. Adleman. Cryptographic Communications Systems and Method. U.S. Patent 4,404,829,20.
37. Zimmermann, P., 1992. PGP User's Guide.
38. Pohlig, S.C. and M.E. Hellman, 1978. An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance. To appear in IEEE Trans. Inform, Theory.
39. Rivest, R., 1978. A Shamir and L. Adleman. 1978. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. Comm. Of the ACM, 21: 120-126.
40. Pollard, J.M., 1974. Theorem on Factorization and Primality Testing. Proc. Camb. Phil. Sci., 76: 521-528.
41. Damgård, T.B., 1990. A design principle for hash functions. In G. Brassard, editor, Advances in cryptology. Proceedings of Crypto'89, vol. 435 of Lecture Notes in Computer Science. Springer-Verlag, New York, pp: 416-427.
42. Rabin, M.O., 1979. Digital Signature and Public-Key Functions as Intractable as Factorization, MIT Laboratory of Computer Science, Technical report, MIT/LCS/TR-212.
43. Daemen, J., A. Bosselaers, R. Govaerts and J. Vandewalle, 1999. Collision for Schnorr's Hash Function FFT-Hash Presented at Crypto'91. ASIACRYPT'91, 1991. presented At rump seesion.
44. Rivest, R., 1992. RFC 121: The MD5 Message-Digest Algorithm RSA Data Security, Inc.