chapter 1 introduction to cryptography

1.1 basic terminology

Cryptography is the art of making and keeping messages secret. In practical terms, this involves the conversion of a **plaintext** message into a cryptic one, called **cyphertext**. The process of conversion, or encoding of the clear text is called **encryption**. The process of converting the cyphertext to the original content of the message, the plaintext, is called **decryption**. Both processes make use (in one way or another) of an encryption procedure, called encryption (decryption) **algorithm**. While most of these algorithms are public, the secrecy is guaranteed by the usage of an encryption (decryption) **key**, which is, in most cases, known only by the legitimate entities at the both ends of the communication channel. **Cryptology** is a branch of mathematics and describes the mathematical foundation of cryptographic methods, while **cryptanalysis** is the art of breaking ciphers.

1.2 cryptography

Cryptography provides the following services:

- authentication
- integrity
- non-repudiation
- secrecy

Let's have a more detailed look at these services.

Authentication allows the recipient of the message to validate the identity of the sender. It prevents an unauthorized entity to masquerade itself as a legitimate sender of the message.

Integrity guarantees that the message sent has not been modified or altered along the communication channel. This is usually accomplished by attaching to the message itself a digest (compressed version) of fixed length of the message, digest which allows verify if the original message was (intentionally or not) altered.

Non-repudiation with proof of origin assures the receiver of the identity of the sender, while non-repudiation with proof of delivery ensures the sender that the message was delivered.

Secrecy prevents unauthorized entities from accessing the real content of a message.

1.3 cryptographic algorithms classification

There are two types of key-based encryption algorithms:

- secret – key, or symmetric key algorithms
- public – key, or asymmetric key algorithms

Symmetric key algorithms rely on the secrecy of the encoding (decoding) key. This key is only known by the sender and the receiver of the message.

These algorithms can be classified further into **stream** ciphers and **block** ciphers. The former ones act on characters as encoding unit while thy later one act upon a block of characters, which is treated as an encoding unit.

The execution of symmetric algorithms is much faster than the execution of asymmetric ones. On the other side, the key exchange implied by the utilization of symmetric algorithms raises new security issues. In practice, it is customary to use an asymmetric encryption for key generation and exchange and the generated key to be used for symmetric encryption of the actual message.

1.4 symmetric key algorithms

Symmetric key encryption algorithms use a single (secret) key to perform both encryption and decryption of the message. Therefore, preserving the secrecy of this common key is crucial in preserving the security of the communication.

- **DES** – Data Encryption Standard – developed in the mid 70's. It is a standard of NIST (US National Institute of Standards and Technology). DES is a block cipher which uses 64-bit blocks and a 56-bit key. The short length of the key makes it susceptible to exhaustion attacks. Specified initially in FIPS 46. The latest variant of DES is called Triple-DES and is based on using DES 3 times, with 3 different, unrelated keys. It is much stronger than DES, but slow compared to the newest algorithms. 3DES is the object of FIPS 46-3 (October 1999)
- **AES** – Advanced Encryption Standard – object of FIPS 197 (nov. 2001). AES is a block cipher which uses 128-bit blocks and a key of size 128 bits. Variations using 192 and 256-bit keys are also specified. What is specific for this algorithm is that it processes data at byte level, as opposed to bit level processing which was used previously. The algorithm is efficient and considered safe.

1.5 secret key distribution

As mentioned before, symmetric key encryption requires a system for secret key exchange between all parties involved in the communication process. Of course, the key itself, being secret, must be encrypted before being sent electronically, or it may be distributed by other means, which make the event of intercepting the key by an unauthorized party unlikely.

There are 2 main standards for automated secret key distribution. The first standard, called X9.17 is defined by the American National standards Institute (ANSI) and the second one is the Diffie-Hellman protocol.

1.6 asymmetric key algorithms

Asymmetric key algorithms rely on two distinct keys for the implementation of the encryption/decryption phases:

- a public key, which may be distributed or made public upon request
- a private (secret) key which corresponds to a particular public key, and which is known only by the authorized entities.

Each of these two keys defines a transformation function. The 2 transformation functions defined by a pair of public/private keys are inverse to each other, and can be used the encryption/decryption of the message. It is irrelevant which of those 2 functions is used for a particular task.

Although asymmetric key algorithms are slower in execution but have the advantage of eliminating the need for key exchange.

Main public algorithms :

- **RSA** – (Rivest-Shamir-Aldeman) is the most used asymmetric (public) key algorithm. Used mainly for private key exchange and digital signatures. All computation are made modulo some big integer n = p*q, where n is public, but p and q are secret prime numbers. The message m is used to create a cyphertext c = m^e (mod n). The recipient uses the multiplicative inverse d = e^(-1) (mod (p-1)*(q-1)). Then c^d = m^(e*d) = m (mod n). The private key is (n, p, q, e, d) (or just p, q, d) while the public key is (n, e). The size of n should be greater than 1024 bits (about 10^300).

- **Rabin** – This cryptosystem is proven to be equivalent to factoring. Although is not the subject of a federal standard (as RSA is), it is explained well in several books. Keys of size greater than 1024 bits are deemed safe.

1.7 hash functions

Hash functions take a message of arbitrary length as input and generate a fixed length digest (checksum). The length of the digest depends on the function used, but in general is between 128 and 512 bits.

The hash functions are used in 3 main areas:

- assure the integrity of a message (or of a downloaded file) by attaching the generated digest to the message itself. The receiver recomputes the digest using the received message and compares it against the digest generated by the sender.
- are part of the creation of the digital signature
- password storage – password are (almost) never stored in their original form. What is stored, in general, is a hash of the password. When a user introduces a password, its hash is computed and is compared with the stored hash.

The most used hash functions are those in the MD and the SHA families – namely MD5 and SHA-1 and the newest ones SHA-2 and SHA-3. Another hash function of interest is RipeMD-160. The MD functions generate a 128 bit digest and were designed by the company RSA Security. While MD5 is still widespread, MD4 has been broken and is deemed insecure. SHA-1 and RipeMD-160 are considered safe for now. While SHA-2 is an extension of SHA-1, SHA_3 features a brand new algorithm for computing the hash.

Starting with the newest function, here is a list of hash functions of practical interest.

- SHA-3 uses the Keccak algorithm, a sponge construction in which message blocks are XORed into a subset of the state, which is then transformed as a whole. In the version used in SHA-3, the state consists of a 5×5 array of 64-bit words, 1600 bits total. The standardization process is not finished yet as of April 2015.
- SHA-2 includes significant changes from its predecessor, SHA-1. The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits: **SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256**.
- SHA-1 – Secure Hash Algorithm. Published by the US Government. Its specification is the object of FIPS 180-1 (April 1995). FIPS stands for Federal Information Processing Standards. Produces a 160 bit digest (5 32-bit words).
- RipeMD-160 – designed as a replacement for the MD series. It produces a digest of 160 bits (or 20 bytes, if you want).
- MD5 – Message Digest Algorithm 5. Developed by RSA Labs. Produces a 128 bit digest. Still in use, especially for message (download) integrity check.
- MD2, MD4 – Older hash algorithms from RSA Data Security. Since they have known flaws, they are only of historic interest.

1.8 digital signature

Some public-key algorithms can be used to generate **digital signatures**. A digital signature is a small amount of data that was created using some private key, and there is a public key that can be used to verify that the signature was really generated using the corresponding private key. The algorithm used to generate the signature must be such that without knowing the private key it is not possible to create a signature that would verify as valid.

Digital signatures are used to verify that a message really comes from the claimed sender (assuming only the sender knows the private key corresponding to the public key). This is called (data origin) authentication. They can also be used to **timestamp** documents: a trusted party **signs** the document and its timestamp with his/her private key, thus testifying that the document existed at the stated time.

Digital signatures can also be used to **certify** that a public key belongs to a particular entity. This is done by signing the combination of the public key and the information about its owner by a trusted key. The resulting data structure is often called a **public-key certificate** (or simply, a **certificate**). Certificates can be thought of as analogous to passports that guarantee the identity of their bearers.

The trusted party who issues certificates to the identified entities is called a **certification authority (CA)**. Certification authorities can be thought of as being analogous to governments issuing passports for their citizens.

A certification authority can be operated by an external certification service provider, or even by a government, or the CA can belong to the same organization as the entities. CAs can also issue certificates to other (sub-)CAs. This leads to a tree-like **certification hierarchy**. The highest trusted CA in the tree is called a **root CA**. The hierarchy of trust formed by end entities, sub-CAs, and root CA is called a **public-key infrastructure (PKI)**.

A public-key infrastructure does not necessarily require an universally accepted hierarchy or roots, and each party may have different trust points. This is the **web of trust** concept used, for example, in PGP.

A digital signature of an arbitrary document is typically created by computing a **message digest** from the document, and concatenating it with information about the signer, a timestamp, etc. This can be done by applying a cryptographic hash function on the data. The resulting string is then encrypted using the private key of the signer using a suitable algorithm. The resulting encrypted block of bits is the signature. It is often distributed together with information about the public key that was used to sign it.

To verify a signature, the recipient first determines whether it trusts that the key belongs to the person it is supposed to belong to (using a certificate or a priori knowledge), and then decrypts the signature using the public key of the person. If the signature decrypts properly and the information matches that of the message (proper message digest etc.), the signature is accepted as valid. In addition to authentication, this technique also provides data integrity, which means that unauthorized alteration of the data during transmission is detected.

Several methods for making and verifying digital signatures are freely available. The most widely known algorithm is RSA.

1.9 cryptographic protocols and standards

We mention only the most important protocols:

- **DNSSEC** – Domain Name Server Security. Protocol for secure distributed name services
- **IPsec** – or Internet Protocol Security, used in Virtual Privet Networks (VPNs)
- **Kerberos –** computer networks authentication protocol
- **PPP (Point to Point)** - data link layer communication protocol between routers
- **PKCS** – Public Key Encryption Standards - developed by RSA Data Security and define safe ways to use RSA.
- **SSL** – Secure Socket Layer – main protocol for secure WWW connections. Increasing importance due to higher sensitive information traffic. The latest version of the protocol is called **TLS** – Transport Security Layer. Was originally developed by Netscape as an open protocol standard.
- **SHTTP** – a newer protocol, more flexible than SSL. Specified by RFC 2660.
- **SSH** – Secure Shell

When it comes to standards, they usually consist of the actual specification of a particular algorithm or architecture.

Encryption standards: DES, AES, RSA, etc.

Hash standards: MD5, SHA-1, SHA-2, SHA-3, HMAC, etc.

Digital signature standards: DSS (based on DSA), RSA, Elliptic Curve DSA

PKI (Public key infrastructure) standard: X.509

Wireless standards: WEP (Wired Equivalent Privacy), WPA, 802.11.i, A5/1, A5/2, etc.

US Federal Information Processing Standards (FIPS): like FIPS 46-3 (DES), FIPS 186-2 (DSS), etc.

1.10 strength of cryptographic algorithms

Good cryptographic systems should always be designed so that they are as difficult to break as possible. It is possible to build systems that cannot be broken in practice (though this cannot usually be proved). This does not significantly increase system implementation effort; however, some care and expertise is required. There is no excuse for a system designer to leave the system breakable. Any mechanisms that can be used to circumvent security must be made explicit, documented, and brought into the attention of the end users.

In theory, any cryptographic method with a key can be broken by trying all possible keys in sequence. If using **brute force** to try all keys is the only option, the required computing power increases exponentially with the length of the key. A *32*-bit key takes 2^{32} (about 10^9) steps. This is something anyone can do on his/her home computer. A system with *56*-bit keys, such as DES, requires a substantial effort, but using massive distributed systems requires only hours of computing. In 1999, a brute-force search using a specially designed supercomputer and a worldwide network of nearly 100,000 PCs on the Internet, found a DES key in 22 hours and 15 minutes. It is currently believed that keys with at least *128* bits (as in AES, for example) will be sufficient against brute-force attacks into the foreseeable future.

However, key length is not the only relevant issue. Many ciphers can be broken without trying all possible keys. In general, it is very difficult to design ciphers that could not be broken more effectively using other methods.

Unpublished or secret algorithms should generally be regarded with suspicion. Quite often the designer is not sure of the security of the algorithm, or its security depends on the secrecy of the algorithm. Generally, no algorithm that depends on the secrecy of the algorithm is secure. For professionals, it is easy to disassemble and reverse-engineer the algorithm. Experience has shown that the vast majority of secret algorithms that have become public knowledge later have been pitifully weak in reality.

The keys used in public-key algorithms are usually much longer than those used in symmetric algorithms. This is caused by the extra structure that is available to the cryptanalyst. There the problem is not that of guessing the right key, but deriving the matching private key from the public key. In the case of RSA, this could be done by factoring a large integer that has two large prime factors. In the case of some other cryptosystems, it is equivalent to computing the discrete logarithm modulo a large integer (which is believed to be roughly comparable to factoring when the moduli is a large prime number). There are public-key cryptosystems based on yet other problems.

To give some idea of the complexity for the RSA cryptosystem, a *256*-bit modulus is easily factored at home, and *512*-bit keys can be broken by university research groups within a few months. Keys with *768* bits are probably not secure in the long term. Keys with *1024* bits and more should be safe for now unless major cryptographical advances are made against RSA. RSA Security claims that *1024*-bit keys are equivalent in strength to *80*-bit symmetric keys. *2048*-bit RSA keys are claimed to be equivalent to *112*-bit symmetric keys and can be used at least up to 2030 and represented the de-facto standard as of feb. 2021.

It should be emphasized that the strength of a cryptographic system is usually equal to its weakest link. No aspect of the system design should be overlooked, from the choice of algorithms to the key distribution and usage policies.

1.11 cryptanalysis and attacks on cryptosystems

Cryptanalysis is the art of deciphering encrypted communications without knowing the proper keys. There are many cryptanalytic techniques. Some of the more important ones for a system implementor are described below.

- **Ciphertext-only attack**: This is the situation where the attacker does not know anything about the contents of the message, and must work from ciphertext only. In practice it is quite often possible to make guesses about the plaintext, as many types of messages have fixed format headers. Even ordinary letters and documents begin in a very predictable way. For example, many classical attacks use frequency analysis of the ciphertext, however, this does not work well against modern ciphers.

  Modern cryptosystems are not weak against ciphertext-only attacks, although sometimes they are considered with the added assumption that the message contains some statistical bias.
- **Known-plaintext attack**: The attacker knows or can guess the plaintext for some parts of the ciphertext. The task is to decrypt the rest of the ciphertext blocks using this information. This may be done by determining the key used to encrypt the data, or via some shortcut.

  One of the best known modern known-plaintext attacks is linear cryptanalysis against block ciphers.
- **Chosen-plaintext attack**: The attacker is able to have any text he likes encrypted with the unknown key. The task is to determine the key used for encryption.

  A good example of this attack is the differential cryptanalysis which can be applied against block ciphers (and in some cases also against hash functions).

  Some cryptosystems, particularly RSA, are vulnerable to chosen-plaintext attacks. When such algorithms are used, care must be taken to design the application (or protocol) so that an attacker can never have chosen plaintext encrypted.
- **Man-in-the-middle attack**: This attack is relevant for cryptographic communication and key exchange protocols. The idea is that when two parties, A and B, are exchanging keys for secure communication (for example, using Diffie-Hellman), an adversary positions himself between A and B on the communication line. The adversary then intercepts the signals that A and B send to each other, and performs a key exchange with A and B separately. A and B will end up using a different key, each of which is known to the adversary. The adversary can then decrypt any communication from A with the key he shares with A, and then resends the communication to B by encrypting it again with the key he shares with B. Both A and B will think that they are communicating securely, but in fact the adversary is hearing everything.

  The usual way to prevent the man-in-the-middle attack is to use a public-key cryptosystem capable of providing digital signatures. For set up, the parties must know each other's public keys in advance. After the shared secret has been generated, the parties send digital signatures of it to each other. The man-in-the-middle fails in his attack, because he is unable to forge these signatures without the knowledge of the private keys used for signing.

  This solution is sufficient if there also exists a way to securely distribute public keys. One such way is a certification hierarchy such as X.509. It is used for example in IPSec.
- **Correlation** between the secret key and the output of the cryptosystem is the main source of information to the cryptanalyst. In the easiest case, the information about the secret key is directly leaked by the cryptosystem. More complicated cases require studying the correlation (basically, any relation that would not be expected on the basis of chance alone) between the observed (or measured) information about the cryptosystem and the guessed key information.

  For example, in linear (resp. differential) attacks against block ciphers the cryptanalyst studies the known (resp. chosen) plaintext and the observed ciphertext. Guessing some of the key bits of the cryptosystem the analyst determines by correlation between the plaintext and the ciphertext whether she guessed correctly. This can be repeated, and has many variations.

  The differential cryptanalysis introduced by Eli Biham and Adi Shamir in late 1980s was the first attack that fully utilized this idea against block ciphers (especially against DES). Later Mitsuru Matsui came up with linear cryptanalysis which was even more effective against DES. More recently, new attacks using similar ideas have been developed.

Perhaps the best introduction to this material is the proceedings of EUROCRYPT and CRYPTO throughout the 1990s. There one can find Mitsuru Matsui's discussion of linear cryptanalysis of DES, and the ideas of truncated differentials by Lars Knudsen (for example, IDEA cryptanalysis). The book by Eli Biham and Adi Shamir about the differential cryptanalysis of DES is the "classical" work on this subject.

The correlation idea is fundamental to cryptography and several researchers have tried to construct cryptosystems which are provably secure against such attacks. For example, Knudsen and Nyberg have studied provable security against differential cryptanalysis.

- **Attack against or using the underlying hardware**: in the last few years as more and more small mobile crypto devices have come into widespread use, a new category of attacks has become relevant which aims directly at the hardware implementation of the cryptosystem.

  The attacks use the data from very fine measurements of the crypto device doing, say, encryption and compute key information from these measurements. The basic ideas are then closely related to those in other correlation attacks. For instance, the attacker guesses some key bits and attempts to verify the correctness of the guess by studying correlation against her measurements.

  Several attacks have been proposed such as using careful timings of the device, fine measurements of the power consumption, and radiation patterns. These measurements can be used to obtain the secret key or other kinds information stored on the device.

  This attack is generally independent of the used cryptographical algorithms and can be applied to any device that is not explicitly protected against it.

  More information about differential power analysis is available at http://www.cryptography.com.

- **Faults in cryptosystems** can lead to cryptanalysis and even the discovery of the secret key. The interest in cryptographical devices lead to the discovery that some algorithms behaved very badly with the introduction of small faults in the internal computation.

  For example, the usual implementation of RSA private-key operations are very suspectible to fault attacks. It has been shown that by causing one bit of error at a suitable point can reveal the factorization of the modulus (i.e. it reveals the private key).

  Similar ideas have been applied to a wide range of algorithms and devices. It is thus necessary that cryptographical devices are designed to be highly resistant against faults (and against malicious introduction of faults by cryptanalysts).

- **Quantum computing**: Peter Shor's paper on polynomial time factoring and discrete logarithm algorithms with quantum computers has caused growing interest in quantum computing. Quantum computing is a recent field of research that uses quantum mechanics to build computers that are, in theory, more powerful than modern serial computers. The power is derived from the inherent parallelism of quantum mechanics. So instead of doing tasks one at a time, as serial machines do, quantum computers can perform them all at once. Thus it is hoped that with quantum computers we can solve problems infeasible with serial machines.

  Shor's results imply that if quantum computers could be implemented effectively then most of public-key cryptography will become history. However, they are much less effective against secret-key cryptography.

  The current state of the art of quantum computing does not appear alarming, as only very small machines have been implemented. The theory of quantum computation gives much promise for better performance than serial computers, however, whether it will be realized in practice is an open question.

  Quantum mechanics is also a source for new ways of data hiding and secure communication with the potential of offering unbreakable security, this is the field of quantum cryptography. Unlike quantum computing, many successful experimental implementations of quantum cryptography have been already achieved. However, quantum cryptography is still some way off from being realized in commercial applications.

For more information, check: https://en.wikipedia.org/wiki/Quantum_cryptography

- **DNA cryptography**: Leonard Adleman (one of the inventors of RSA) came up with the idea of using DNA as computers. DNA molecules could be viewed as a very large computer capable of parallel execution. This parallel nature could give DNA computers exponential speed-up against modern serial computers.

  There are unfortunately problems with DNA computers, one being that the exponential speed-up requires also exponential growth in the volume of the material needed. Thus in practice DNA computers would have limits on their performance. Also, it is not very easy to build one.

  It may be also viewed as hiding data in terms of DNA sequences. See the link: https://www.geeksforgeeks.org/dna-cryptography/ for more details.

chapter 2 classical cryptography

2.1 cryptograms

A **cryptogram** is the combination of the plaintext (PT) and the ciphertext (CT) obtained as result of encrypting the plaintext, using some encryption method.

Cryptograms may be divided into ciphers and codes.

A **cipher** message is one produced by applying a method of cryptography to the individual letters of the plain text taken either singly or in groups of constant length. Practically every cipher message is the result of the joint application of a General System (or Algorithm) or method of treatment, which is invariable and a Specific Key which is variable, at the will of the correspondents and controls the exact steps followed under the general system. It is assumed that the general system is known by the correspondents and the cryptanalyst.

A **code** message is a cryptogram which has been produced by using a code book consisting of arbitrary combinations of letters, entire words, figures substituted for words, partial words, phrases, of PT. Whereas a cipher system acts upon individual letters or definite groups taken as units, a code deals with entire words or phrases or even sentences taken as units.

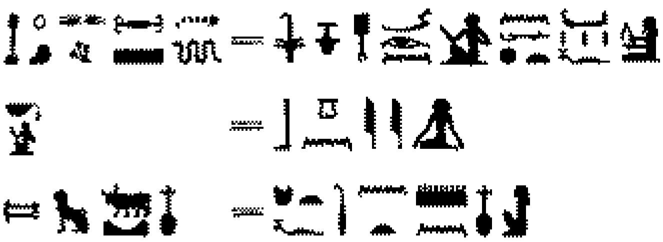
Cipher systems are divided into two classes: **substitution** and **transposition**. A Substitution cipher is a cryptogram in which the original letters of the plain text, taken either singly or in groups of constant length, have been replaced by other letters, figures, signs, or combination of them in accordance with a definite system and key. A transposition cipher is a cryptogram in which the original letters of the plain text have merely been rearranged according to a definite system. Modern cipher systems use both substitution and transposition to create secret messages.

Cipher systems can be further divided into **monoalphabetic** ciphers - those in which only one substitution/transposition is used - and **polyalphabetic** - where several substitutions/ transpositions are used.

2.2 historical developments

2.2.1 ancient ciphers

- have a history of at least 4000 years
- ancient Egyptians enciphered some of their hieroglyphic writing on monuments



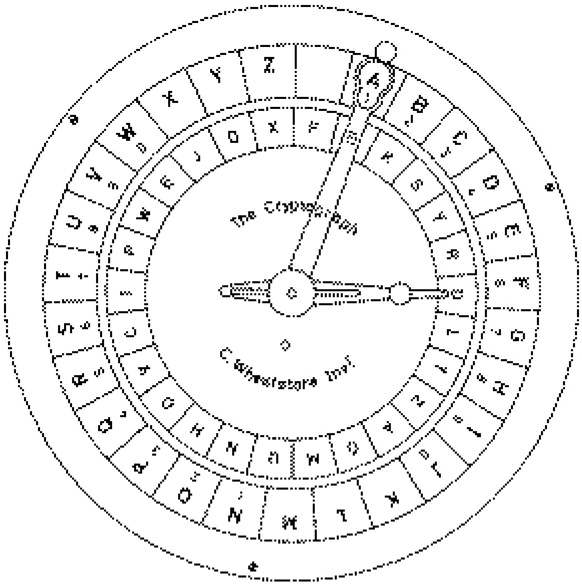
Hieroglyphic encipherments of proper names and titles, with cipher hieroglyphs at left, plain equivalents at right

- ancient Hebrews enciphered certain words in the scriptures
- 2000 years ago Julius Ceasar used a simple substitution cipher, now known as the Caesar cipher
- Roger Bacon described several methods in 1200s
- Geoffrey Chaucer included several ciphers in his works
- Leon Alberti devised a cipher wheel, and described the principles of frequency analysis in the 1460s
- Blaise de Vigenère published a book on cryptology in 1585, & described the polyalphabetic

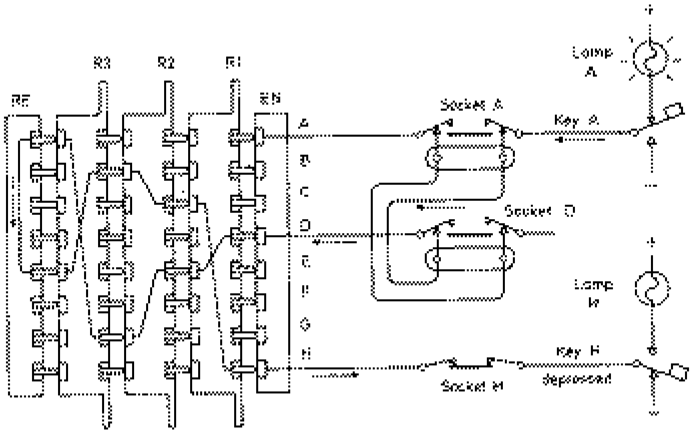
2.2.2 machine ciphers

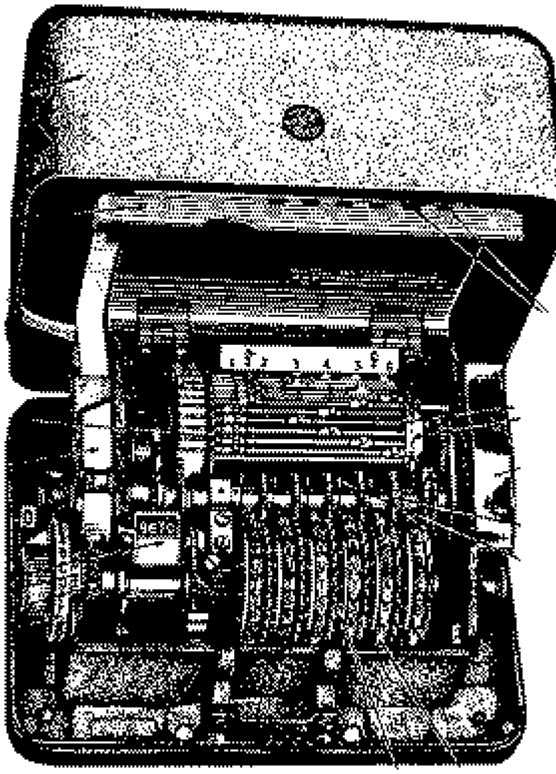
- **Jefferson cylinder**, developed in 1790s, comprised 36 disks, each with a random alphabet, order of disks was key, message was set, then another row became cipher

- **Wheatstone disc**, originally invented by Wadsworth in 1817, but developed by Wheatstone in 1860's, comprised two concentric wheels used to generate a polyalphabetic cipher



- **Enigma Rotor machine**, one of a very important class of cipher machines, heavily used during 2nd world war, comprised a series of rotor wheels with internal cross-connections, providing a substitution using a continuously changing alphabet enciphered by substitution or transposition.



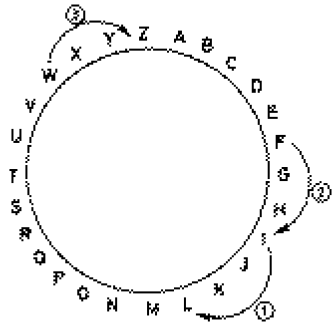


2.3 Caesar cipher - a monoalphabetic cipher

This cipher is a simple substitution, monoalphabetic cipher, used extensively by Caesar in communicating with his field commanders. Each letter of message was replaced by a letter a fixed distance away eg use the 3rd letter on.

For example:

```
L FDPH L VDZ L FRQTXHUHG
I CAME I SAW I CONQUERED
```



In this case the mapping is:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
```

Mathematicaly, one can describe this cipher as:

Encryption E_(k) : i -> i + k mod 26

Decryption D_(k) : i -> i - k mod 26

2.3.1 cryptanalysis of the Caesar cipher

- only have 26 possible ciphers
- could simply try each in turn - **exhaustive key search**

```
                          GDUCUGQFRMPCNJYACJCRRCPQ
                          HEVDVHRGSNQDOKZBDKDSSDQR
Plain  -       IFWEWISHTOREPLACELETTERS
                          JGXFXJTIUPSFQMBDFMFUUFST
                          KHYGYKUJVQTGRNCEGNGVVGTU
Cipher -       LIZHZLVKWRUHSODFHOHWWHUV
                          MJAIAMWLXSVITPEGIPIXXIVW
```

2.4 the Vigenère cipher - a polyalphabetic cipher

What is now known as the Vigenère cipher was originally described by Giovan Battista Bellaso in his 1553 book *La cifra del. Sig. Giovan Battista Bellaso*. He built upon the tabula recta of Trithemius, but added a repeating "countersign" (a key) to switch cipher alphabets every letter.

Blaise de Vigenère published his description of a similar but stronger autokey cipher before the court of Henry III of France, in 1586. Later, in the 19th century, the invention of Bellaso's cipher was misattributed to Vigenère. David Kahn in his book *The Codebreakers* lamented the misattribution by saying that history had "ignored this important contribution and instead named a regressive and elementary cipher for him [Vigenère] though he had nothing to do with it".

The Vigenère cipher gained a reputation for being exceptionally strong. Noted author and mathematician Charles Lutwidge Dodgson (Lewis Carroll) called the Vigenère cipher unbreakable in his 1868 piece "The Alphabet Cipher" in a children's magazine. In 1917, *Scientific American* described the Vigenère cipher as "impossible of translation". This reputation was not deserved since Kasiski entirely broke the cipher in the 19th century and some skilled cryptanalysts could occasionally break the cipher in the 16th century.

The Vigenère cipher is simple enough to be a field cipher if it is used in conjunction with cipher disks. [4] The Confederate States of America, for example, used a brass cipher disk to implement the Vigenère cipher during the American Civil War. The Confederacy's messages were far from secret and the Union regularly cracked their messages. Throughout the war, the Confederate leadership primarily relied upon three keywords, "Manchester Bluff", "Complete Victory" and, as the war came to a close, "Come Retribution".[5]

Gilbert Vernam tried to repair the broken cipher (creating the Vernam-Vigenère cipher in 1918), but, no matter what he did, the cipher was still vulnerable to cryptanalysis. Vernam's work, however, eventually led to the one-time pad, a provably unbreakable cipher.

A reproduction of the Confederacy's cipher disk. Only five originals are known to exist.

2.4.1 description

	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
A	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
B	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	A
C	C	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
D	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
E	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	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	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

To encipher, a table of alphabets can be used, termed a *tabula recta*, *Vigenère square*, or *Vigenère table*. It consists of the alphabet written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar ciphers. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. The alphabet used at each point depends on a repeating keyword.

For example, suppose that the plaintext to be encrypted is:

```
ATTACKATDAWN
```

The person sending the message chooses a keyword and repeats it until it matches the length of the plaintext, for example, the keyword "LEMON":

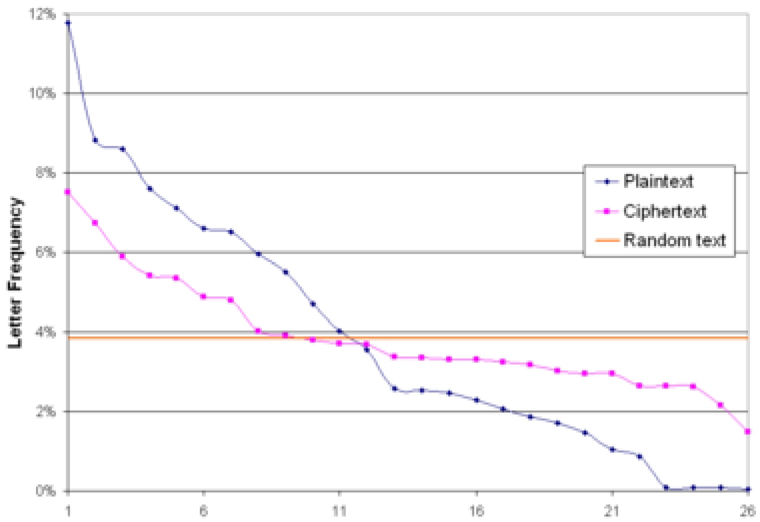
```
LEMONLEMONLE
```

The first letter of the plaintext, A, is enciphered using the alphabet in row L, which is the first letter of the key. This is done by looking at the letter in row L and column A of the Vigenère square, namely L. Similarly, for the second letter of the plaintext, the second letter of the key is used; the letter at row E and column T is X. The rest of the plaintext is enciphered in a similar fashion:

Plaintext:	ATTACKATDAWN
Key:	LEMONLEMONLE
Ciphertext:	LXFOPVEFRNHR

Decryption is performed by finding the position of the ciphertext letter in a row of the table, and then taking the label of the column in which it appears as the plaintext. For example, in row L, the ciphertext L appears in column A, which taken as the first plaintext letter. The second letter is decrypted by looking up X in row E of the table; it appears in column T, which is taken as the plaintext letter.

2.4.2 cryptanalysis



The Vigenère cipher masks the characteristic letter frequencies of English plaintexts, but some patterns remain.

The idea behind the Vigenère cipher, like all polyalphabetic ciphers, is to disguise plaintext letter frequencies, which interferes with a straightforward application of frequency analysis. For instance, if P is the most frequent letter in a ciphertext whose plaintext is in English, one might suspect that P corresponds to E, because E is the most frequently used letter in English. However, using the Vigenère cipher, E can be enciphered as different ciphertext letters at different points in the message, thus defeating simple frequency analysis.

The primary weakness of the Vigenère cipher is the repeating nature of its key. If a cryptanalyst correctly guesses the key's length, then the cipher text can be treated as interwoven Caesar ciphers, which individually are easily broken. The Kasiski and Friedman tests can help determine the key length.

2.4.3 Kasiski examination

In 1863 Friedrich Kasiski was the first to *publish* a successful attack on the Vigenère cipher, but Charles

Babbage had already developed the same test in 1854. Babbage was goaded into breaking the Vigenère cipher when John Hall Brock Thwaites submitted a "new" cipher to the Journal of the Society of the Arts; when Babbage showed that Thwaites' cipher was essentially just another recreation of the Vigenère cipher, Thwaites challenged Babbage to break his cipher. Babbage succeeded in decrypting a sample, which turned out to be the poem "The Vision of Sin", by Alfred Tennyson, encrypted according to the keyword "Emily", the first name of Tennyson's wife.

The Kasiski examination, also called the Kasiski test, takes advantage of the fact that certain common words like "`the`" will, by chance, be encrypted using the same key letters, leading to repeated groups in the ciphertext. For example, a message encrypted with the keyword `ABCDEF` might not encipher "crypto" the same way each time it appears in the plain text:

```
Key:        ABCDEF AB CDEFA BCD EFABCDEFABCD
Plaintext:  CRYPTO IS SHORT FOR CRYPTOGRAPHY
Ciphertext: CSASXT IT UKSWT GQU GWYQVRKWAQJB
```

The encrypted text here will not have repeated sequences that correspond to repeated sequences in the plaintext. However, if the key length is different, as in this example:

```
Key:        ABCDAB CD ABCDA BCD ABCDABCDABCD
Plaintext:  CRYPTO IS SHORT FOR CRYPTOGRAPHY
Ciphertext: CSASTP KV SIQUT GQU CSASTPIUAQJB
```

then the Kasiski test is effective. Longer messages make the test more accurate because they usually contain more repeated ciphertext segments. The following ciphertext has several repeated segments and allows a cryptanalyst to discover its key length:

```
Ciphertext: DYDUXRMHTVDVNQDQNWDYDUXRMHARTJGWNQD
```

The distance between the repeated `DYDUXRMH`s is 18. This, assuming that the repeated segments represent the same plaintext segments, implies that the key is 18, 9, 6, 3, or 2 characters long. The distance between the `NQD`s is 20 characters. This means that the key length could be 20, 10, 5, 4, or 2 characters long (all factors of the distance are possible key lengths – a key of length one is just a simple shift cipher, where cryptanalysis is much easier). By taking the intersection of these sets one could safely conclude that the key length is (almost certainly) 2.

2.4.4 the Friedman test

The Friedman test (sometimes known as the kappa test) was invented during the 1920s by William F. Friedman. Friedman used the index of coincidence, which measures the unevenness of the cipher letter frequencies, to break the cipher. By knowing the probability κ_p that any two randomly chosen source-language letters are the same (around 0.067 for monocase English) and the probability of a coincidence for a uniform random selection from the alphabet κ_r (1/26 = 0.0385 for English), the key length can be estimated as:

$$\frac{\kappa_p - \kappa_r}{\kappa_o - \kappa_r}$$

from the observed coincidence rate:

$$\kappa_o = \frac{\sum_{i=1}^{c} n_i(n_i - 1)}{N(N-1)}$$

where c is the size of the alphabet (26 for English), N is the length of the text, and n_1 through n_c are the observed ciphertext letter frequencies, as integers.

This is, however, only an approximation whose accuracy increases with the size of the text. It would in practice be necessary to try various key lengths close to the estimate.[7] A better approach for repeating-key ciphers is to copy the ciphertext into rows of a matrix having as many columns as an assumed key length, then compute the average index of coincidence with each column considered separately; when this is done for each possible key length, the highest average I.C. then corresponds to the most likely key length.[8] Such tests may be supplemented by information from the Kasiski examination.

2.5 four basic operations of cryptanalysis

William F. Friedman presents the fundamental operations for the solution of practically every cryptogram:

(1) The determination of the language employed in the plain text version.
(2) The determination of the general system of cryptography employed.
(3) The reconstruction of the specific key in the case of a cipher system, or the reconstruction of, partial or complete, of the code book, in the case of a code system or both in the case of an enciphered code system.
(4) The reconstruction or establishment of the plain text.

In some cases, step (2) may proceed step (1). This is the classical approach to cryptanalysis. It may be further reduced to:

1. Arrangement and rearrangement of data to disclose non-random characteristics or manifestations (i.e. frequency counts, repetitions, patterns, symmetrical phenomena)
2. Recognition of the nonrandom characteristics or manifestations when disclosed (via statistics or other techniques)
3. Explanation of nonrandom characteristics when recognized. (by luck, intelligence, or perseverance)

Much of the work is in determining the general system. In the final analysis, the solution of every cryptogram involving a form of substitution depends upon its reduction to mono-alphabetic terms, if it is not originally in those terms.

2.6 outline of the cipher solution – the navy department approach

According to the Navy Department OP-20-G Course in Cryptanalysis, the solution of a substitution cipher generally progresses through the following stages:

(a) Analysis of the cryptogram(s)
- (1) Preparation of a frequency table.
- (2) Search for repetitions.
- (3) Determination of the type of system used.
- (4) Preparation of a work sheet.
- (5) Preparation of individual alphabets (if more than one)
- (6) Tabulation of long repetitions and peculiar letter distributions.

(b) Classification of vowels and consonants by a study of:
- (1) Frequencies
- (2) Spacing
- (3) Letter combinations
- (4) Repetitions

(c) Identification of letters.
- (1) Breaking in or wedge process
- (2) Verification of assumptions.
- (3) Filling in good values throughout messages
- (4) Recovery of new values to complete the solution.

(d) Reconstruction of the system.
- (1) Rebuilding the enciphering table.
- (2) Recovery of the key(s) used in the operation of the system
- (3) Recovery of the key or keyword(s) used to construct the alphabet sequences.

All steps above to be done with orderly reasoning. It is not an exact mechanical process.

2.7 the analysis of a simple substitution example

While reading the newspaper you see the following cryptogram. Train your eye to look for wedges or 'ins' into the cryptogram. Assume that we dealing with English and that we have simple substitution. What do we know? Although short, there are several entries for solution. Number the words. Note that it is a quotation (12, 13 words with * represent a proper name in ACA lingo).

A-1. Elevated thinker. K2 (71) LANAKI

1	2	3	4	5
FYV	Y Z X Y V E F	I T A M G V U X V	ZE	FA ITAM
6	7	8	9	10
FYQF	MV	QDV	E J D D A J T U V U	RO
11	12	13		
H O E F V D O	* Q G R V D F	* E S Y M V Z F P V D		

The analysis of A1

Note words 1 and 6 could be: ' The....That' and words 3 and 5 use the same 4 letters I T A M . Note that there is a flow to this cryptogram The _ _ is? _ _ and? _ _. Titles either help or should be ignored as red herrings. Elevated might mean "high" and the thinker could be the proper person. We also could attack this cipher using pattern words (lists of words with repeated letters put into thesaurus form and referenced by pattern and word length) for words 2, 3, 6, 9, and 11.

Filling in the cryptogram using [The... That] assumption we have:

1	2	3	4	5
THE	H__HE_ _T	_________E	__	T____ _
FYV	YZXYVE F	ITAMGVUXV	ZE	FAITA M
6	7	8	9	10
THAT	_E	A_E	__________ E__	__
FYQF	MV	QDV	E J D D A J T U V U	RO
11	12	13		
_____T E ____	* A____E___T	* ___H___E__T__E__		
H O E F V D O	* Q G R V D F	* E S Y M V Z F P V D		

Not bad for a start. We find the ending e_t might be 'est'. A two letter word starting with t_ is 'to'. Word 8 is 'are'. So we add this part of the puzzle. Note how each wedge leads to the next wedge. Always look for confirmation that your assumptions are correct. Have an eraser ready to start back a step if necessary. Keep a tally on which letters have been placed correctly. Those that are unconfirmed guesses, signify with ? Piece by piece, we build on the opening wedge.

1	2	3	4	5
THE	H__HEST	__O_______E	_S	TO__O _
FYV	YZXYVEF	ITAMGVUXV	ZE	FAITA M
6	7	8	9	10
THAT	_E	ARE	S__R R O_____E__	—
FYQF	MV	QDV	E J D D A J T U V U	RO
11	12	13		
____S T E R _	* A_____E R T	* S__H___E__T__E R		
H O E F V D O	* Q G R V D F	* E S Y M V Z F P V D		

Now we have some bigger wedges. The s_h is a possible 'sch' from German. Word 9 could be 'surrounded.' Z = i. The name could be Albert Schweitzer. Lets try these guesses. Word 2 might be 'highest' which goes with the title.

1	2	3	4	5
THE	HIGHEST	_NOWLEDGE	IS	TO_N OW
FYV	YZXYVEF	ITAMGVUXV	ZE	FAITA M
6	7	8	9	10
THAT	WE	ARE	S U R R O U N D E D	—
FYQF	MV	QDV	E J D D A J T U V U	RO
11	12	13		
____S T E R _	* A L B E R T	* S C H W E I T Z E R		

H O E F V D O	*Q G R V D F	*E S Y M V Z F P V D	

The final message is: The highest knowledge is to know that we are surrounded by mystery. Albert Schweitzer.

Ok that's the message, but what do we know about the keying method.

2.8 keying conventions

Ciphertext alphabets are generally mixed for more security and an easy pneumonic to remember as a translation key. ACA ciphers are keyed in K1, K2, K3, K4 or K()M for mixed variety. K1 means that a keyword is used in the PT alphabet to scramble it. K2 is the most popular for CT alphabet scrambling. K3 uses the same keyword in both PT and CT alphabets, K4 uses different keywords in both PT and CT alphabets. A keyword or phrase is chosen that can easily be remembered. Duplicate letters after the first occurrence are deleted.

Following the keyword, the balance of the letters are written out in normal order. A one-to-one correspondence with the regular alphabet is maintained. A K2M mixed keyword sequence using the word METAL and key DEMOCRAT might look like this:

```
4   2   5   1   3
M   E   T   A   L
=============
D   E   M   O   C
R   A   T   B   F
G   H   I   J   K
L   N   P   Q   S
U   V   W   X   Y
Z
```

The CT alphabet would be taken off by columns and used:

```
CT: OBJQX EAHNV CFKSY DRGLUZ MTIPW
```

Going back to A-1. Since it is keyed aa a K-2, we set up the PT alphabet as a normal sequence and fill in the CT letters below it. Do you see the keyword LIGHT?

```
PT  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
CT  Q R S U V W X Y Z L I G H T A B C D E F J K M N O P
                      -----------
KW = LIGHT
```

In tough ciphers, we use the above key recovery procedure to go back and forth between the cryptogram and keying alphabet to yield additional information.

To summarize the eyeball method:

1. Common letters appear frequently throughout the message but don't expect an exact correspondence in popularity.
2. Look for short, common words (the, and, are, that, is, to) and common endings (tion, ing, ers, ded, ted, ess)
3. Make a guess, try out the substitutions, keep track of your progress. Look for readability.

2.9 general nature of the english language

A working knowledge of the letters, characteristics, relations with each other, and their favorite positions in words is very valuable in solving substitution ciphers.

Friedman was the first to employ the principle that English Letters are mathematically distributed in a unilateral frequency distribution:

```
  13 9 8 8 7 7 7 6 6 4 4 3 3 3 3 2 2 2 1 1 1 - - - - -
   E T A O N I R S H L D C U P F M W Y B G V K Q X J Z
```

That is, in each 100 letters of text, E has a frequency (or number of appearances) of about 13; T, a frequency of about 9; K Q X J Z appear so seldom, that their frequency is a low decimal.

Other important data on English (based on Hitt's Military Text):

```
6 Vowels: A E I O U Y                          =  40 %
20 Consonants:
    5 High Frequency (D N R S T)               =  35 %
   10 Medium Frequency (B C F G H L M P V W)   =  24 %
    5 Low Frequency (J K Q X Z)                =   1 %
                                                 ====
                                                 100.%
```

The four vowels A, E, I, O and the four consonants N, R, S, T form 2/3 of the normal English plain text. [FR1]

Friedman gives a Digraph chart taken from Parker Hitts Manual on p22 of reference. [FR2]

The most frequent English digraphs per 200 letters are:

```
TH--50       AT--25       ST--20
ER--40       EN--25       IO--18
ON--39       ES--25       LE--18
AN--38       OF--25       IS--17
RE--36       OR--25       OU--17
HE--33       NT--24       AR--16
IN--31       EA--22       AS--16
ED--30       TI--22       DE--16
ND--30       TO--22       RT--16
HA--26       IT--20       VE--16
```

The most frequent English trigraphs per 200 letters are:

```
THE--89       TIO--33       EDT--27
AND--54       FOR--33       TIS--25
THA--47       NDE--31       OFT--23
ENT--39       HAS--28       STH--21
ION--36       NCE--27       MEN--20
```

Frequency of Initial and Final Letters:

```
Letters-- 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
Initial-- 9 6 6 5 2 4 2 3 3 1 1 2 4 2 10 2 - 4 5 17 2 - 7 - 3 -
Final   -- 1 -  1017 6 4 2 - - 1 6 1 9 4  1 - 8 9 11 1 - 1 - 8 -
```

Relative Frequencies of Vowels:

```
A 19.5%   E 32.0%   I 16.7%  O 20.2%  U 8.0%  Y 3.6%
```

Average number of vowels per 20 letters, 8.

Becker and Piper partition the English language into 5 groups based on their Table 1.1 [STIN], [BP82]

Table 1.1
Probability Of Occurrence of 26 Letters

Letter	Probability	Letter	Probability
A	.082	N	.067
B	.015	O	.075
C	.028	P	.019
D	.043	Q	.001
E	.127	R	.060
F	.022	S	.063
G	.020	T	.091
H	.061	U	.028
I	.070	V	.010
J	.002	W	.023
K	.008	X	.001
L	.040	Y	.020
M	.024	Z	.001

Groups:

1. E, having a probability of about 0.127
2. T, A, O, I, N, S, H, R, each having probabilities between 0.06 - 0.09
3. D, L, having probabilities around 0.04
4. C, U, M, W, F, G, Y, P, B, each having probabilities between 0.015 - 0.023.
5. V, K, J, X, Q, Z, each having probabilities less 0.01.

2.10 homework problems

Solve these cryptograms, recovery the keywords, and send your solutions to me for credit. Be sure to show how you cracked them. If you used a computer program, please provide "gut" details. Answers do not need to be typed but should be generously spaced and not in RED color. Let me know what part of the problem was the "ah ha", i e. the light of inspiration that brought for the message to you.

A-1. Bad design. K2 (91) AURION

V G S   E U L Z K   W U F G Z   G O N   G M   V D G X Z A J U =

X U V B Z   H B U K N D W   V O N   D K   X D K U H H G D F =

N Z X   U K   Y D K   V G U N   A J U X O U B B S

X D K K G B P Z K   D F   N Y Z   B U L Z .

A-2. Not now. K1 (92) BRASSPOUNDER

K D C Y   L Q Z K T L J Q X   C Y   M D B C Y J Q L :   " T R

H Y D   F K X C ,   F Q   M K X   R L Q Q I Q   H Y D L

M K L   D X C T W   R D C D L Q   J Q M N K X T M B

P T B M Y E Q L   K   F K H   C Y   L Q Z K T L   T C . "

A-3. Ms. Packman really works! K4 (101) APEX DX

* Z D D Y Y D Q T   Q M A R P A C ,   * Q A K C M K

chapter 2

* T D V S V K . B P W V G Q N V O M C M V B : L D X V

K Q A M S P D L V Q U , L D B Z I U V K Q F P O

W A M U X V , E M U V P X Q N V , U A M O Z

N Q K L M O V (S A P Z V O) .

A-4. Money value. K4 (80) PETROUSHKA

D V T U W E F S Y Z C V S H W B D X P U Y T C Q P V

E V Z F D A E S T U W X Q V S P F D B Y P Q Y V D A F S ,

H Y B P Q P F Y V C D Q S F I T X P X B J D H W Y Z .

A-5. Zoology lesson. K4 (78) MICROPOD

A S P D G U L W , J Y C R S K U Q N B H Y Q I X S P I N

O C B Z A Y W N = O G S J Q O S R Y U W , J N Y X U

O B Z A (B C W S D U R B C) T B G A W U Q E S L.

* C B S W

chapter 3 hash functions - MD5

3.1 hash functions

A **cryptographic hash function** is a transformation that takes an input and returns a fixed-size string, which is called either **hash value, checksum** or **digest**. Hash functions with this property are used for a variety of computational purposes, including cryptography. The hash value is a concise representation of the longer message or document from which it was computed. The message digest is a sort of "digital fingerprint" of the larger document. Cryptographic hash functions are used to do message integrity checks and digital signatures in various information security applications, such as authentication and message integrity.

A hash function takes a string (or 'message') of any length as input and produces a fixed length string as output, sometimes termed a **message digest** or a **digital fingerprint**. A hash value (also called a "digest" or a "checksum") is a kind of "signature" for a stream of data that represents the contents. One analogy that explains the role of the hash function would be the "tamper-evident" seals used on a software package.

In various standards and applications, the two most-commonly used hash functions are MD5 and the SHA family. In 2005, security flaws were identified in both algorithms. In 2007 the National Institute of Standards and Technology announced a contest to design a hash function which will be given the name SHA-3 and be the subject of a FIPS standard

For a hash function h with domain D and range R, the following requirements are mandatory:

1. Pre-image resistance – given y in R, it is computationally unfeasible to find x in D such that h(x) = y.
2. Second pre-image resistance – for a given x in D, it is it is computationally unfeasible to find another z in D such that h(x) = h(z).
3. Collision resistance – it is computationally unfeasible to find any x, z in D such that h(x) = h(z)

3.2 applications

A typical use of a cryptographic hash would be as follows: Alice poses a tough math problem to Bob, and claims she has solved it. Bob would like to try it himself, but would yet like to be sure that Alice is not bluffing. Therefore, Alice writes down her solution, appends a random nonce (a number used only once), computes its hash and tells Bob the hash value (whilst keeping the solution and nonce secret). This way, when Bob comes up with the solution himself a few days later, Alice can prove that she had the solution earlier by revealing the nonce to Bob. (This is an example of a **simple commitment scheme**; in actual practice, Alice and Bob will often be computer programs, and the secret would be something less easily spoofed than a claimed puzzle solution).

Another important application of secure hashes is verification of **message integrity**. Determining whether any changes have been made to a message (or a file), for example, can be accomplished by comparing message digests calculated before, and after, transmission (or any other event).

A message digest can also serve as a means of reliably identifying a file; several source code management systems, including Git, Mercurial and Monotone, use the sha1sum of various types of content (file content, directory trees, ancestry information, etc) to uniquely identify them.

A related application is **password verification**. Passwords are usually not stored in cleartext, for obvious reasons, but instead in digest form. To authenticate a user, the password presented by the user is hashed and compared with the stored hash. This is sometimes referred to as one-way encryption.

For both security and performance reasons, most **digital signature** algorithms specify that only the digest of the message be "signed", not the entire message. Hash functions can also be used in the generation of pseudorandom bits.

3.3 MD5 - basics

MD5 is a **block** hash function (the block size is **512 bits**) which has been developed by Rivest in 1991. The input for MD5 is an arbitrary length message or file, while the output is is a fixed length digest. The length of this digest is **128 bits** or 4 words. The formal specification of this hash algorithm is specified in RFC 1321.

3.4 MD5 algorithm description

We begin by supposing that we have a **b-bit message** as input, and that we wish to find its message **digest**. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

m_0 m_1 ... m_{b-1}

The following five steps are performed to compute the message digest of the message.

3.4.1 step 1 - append padding bits

The message is "**padded**" (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512.

Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.

3.4.2 step 2 - append length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. In the unlikely event that b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

At this point the resulting message (after padding with bits and with the length) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let M[0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

3.4.3 step 3 - initialize the MD buffer

A four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are **initialized** to the following values in hexadecimal, low-order bytes first):

```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

3.4.4 step 4 - process message in 16-word blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

```
F(X,Y,Z) = (X&Y)|(~X&Z)
G(X,Y,Z) = (X&Z)|(Y&~Z)
H(X,Y,Z) = X^Y^Z
I(X,Y,Z) = Y^(X|~Z)
```

In each bit position F acts as a conditional: if X then Y else Z. The function F could have been defined using + instead of or since XY and not(X)Z will never have 1's in the same bit position.) It is interesting to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of F(X,Y,Z) will be independent and unbiased.

The functions G, H, and I are similar to the function F, in that they act in "bitwise parallel" to produce their output from the bits of X, Y, and Z, in such a manner that if the corresponding bits of X, Y, and Z are independent and unbiased, then each bit of G(X,Y,Z), H(X,Y,Z), and I(X,Y,Z) will be independent and unbiased. Note that the function H is the bit-wise "xor" or "parity" function of its inputs.

This step uses a 64-element table T[1 ... 64] constructed from the sinus function. Let T[i] denote the i-th element of the table, which is equal to the integer part of 4294967296 times abs(sin(i)), where i is in radians. The elements of the table are given in the appendix.

Below, N is the number of words. Because the last block of 512 bits (16 words) has been padded, N is a multiple of 16 while N/16 is the number of blocks in the padded message.

Do the following:

```
/* Process each 16-word block. */
For i = 0 to N/16 - 1 do

  /* Copy block i into X. */
  For j = 0 to 15 do
    Set X[j] to M[i*16+j].
  end /* of loop on j */

  /* Save A as AA, B as BB, C as CC, and D as DD. */
  AA = A
  BB = B
  CC = C
  DD = D

  /* Round 1. */
  /* Let [abcd k s i] denote the operation
       a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
  /* Do the following 16 operations. */
  [ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
  [ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
  [ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
  [ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]

  /* Round 2. */
  /* Let [abcd k s i] denote the operation
       a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
  /* Do the following 16 operations. */
  [ABCD  1  5 17]  [DABC  6  9 18]  [CDAB 11 14 19]  [BCDA  0 20 20]
  [ABCD  5  5 21]  [DABC 10  9 22]  [CDAB 15 14 23]  [BCDA  4 20 24]
  [ABCD  9  5 25]  [DABC 14  9 26]  [CDAB  3 14 27]  [BCDA  8 20 28]
  [ABCD 13  5 29]  [DABC  2  9 30]  [CDAB  7 14 31]  [BCDA 12 20 32]

  /* Round 3. */
  /* Let [abcd k s t] denote the operation
       a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
  /* Do the following 16 operations. */
  [ABCD  5  4 33]  [DABC  8 11 34]  [CDAB 11 16 35]  [BCDA 14 23 36]
  [ABCD  1  4 37]  [DABC  4 11 38]  [CDAB  7 16 39]  [BCDA 10 23 40]
```

```
   [ABCD 13  4 41]  [DABC  0 11 42]  [CDAB  3 16 43]  [BCDA  6 23 44]
   [ABCD  9  4 45]  [DABC 12 11 46]  [CDAB 15 16 47]  [BCDA  2 23 48]

   /* Round 4. */
   /* Let [abcd k s t] denote the operation
        a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
   /* Do the following 16 operations. */
   [ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
   [ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
   [ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
   [ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]

   /* Then perform the following additions. (That is increment each
      of the four registers by the value it had before this block
      was started.) */
   A = A + AA
   B = B + BB
   C = C + CC
   D = D + DD

end /* of loop on i */
```

3.4.5 step 5 - output

The message digest produced as output is A, B, C, D. That is, we begin with the low-order byte of A, and end with the high-order byte of D.

This completes the description of MD5.

3.5 The test suite for MD5

The hash values of some test strings, as specified In RFC 1321, are the following:

```
MD5("") = d41d8cd9 8f00b204 e9800998 ecf8427e

MD5("a") = 0cc175b9 c0f1b6a8 31c399e2 69772661

MD5("abc") = 90015098 3cd24fb0 d6963f7d 28e17f72

MD5("message digest") = f96b697d 7cb7938d 525a2f31 aaf161d0

MD5("abcdefghijklmnopqrstuvwxyz") = c3fcd3d7 6192e400 7dfb496c ca67e13b

MD5("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789") =

   d174ab98 d277d9f5 a5611c2c 9f419d9f

MD5 ("1234567890123456789012345678901234567890123456789012345678901234567890123456

78901234567890") = 57edf4a2 2be3c955 ac49da2e 2107b67a
```

3.6 MD5 cryptanalysis

In 1993, Den Boer and Bosselaers gave an early, although limited, result of finding a "pseudo-collision" of the MD5 compression function; that is, two different initialization vectors which produce an identical digest.

In 1996, Dobbertin announced a collision of the compression function of MD5 (Dobbertin, 1996). While this was not an attack on the full MD5 hash function, it was close enough for cryptographers to recommend switching to a replacement, such as WHIRLPOOL, SHA-1 or RIPEMD-160.

The size of the hash—128 bits—is small enough to contemplate a birthday attack. MD5CRK was a distributed project started in March 2004 with the aim of demonstrating that MD5 is practically insecure by finding a collision using a birthday attack.

MD5CRK ended shortly after 17 August 2004, when collisions for the full MD5 were announced by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Their analytical attack was reported to take only one hour on an IBM p690 cluster.

On 1 March 2005, Arjen Lenstra, Xiaoyun Wang, and Benne de Weger demonstrated the construction of two X.509 certificates with different public keys and the same MD5 hash, a demonstrably practical collision. The construction included private keys for both public keys. A few days later, Vlastimil Klima described an improved algorithm, able to construct MD5 collisions in a few hours on a single notebook computer. On 18 March 2006, Klima published an algorithm that can find a collision within one minute on a single notebook computer, using a method he calls tunneling.

An actual collision can be found at http://www.mathstat.dal.ca/~selinger/md5collision/ . We reproduce it here, since the two messages are not that long. Moreover, the two messages are almost identical.

```
d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70

d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70
```

The common MD5 digest of these two messages is - 79054025-255fb1a2-6e4bc422-aef54eb4 .

chapter 4 secure hash algorithm

The **SHA hash functions** are a set of cryptographic hash functions designed by the National Security Agency (NSA) and published by the NIST as a U.S. Federal Information Processing Standard. SHA stands for **Secure Hash Algorithm**. The three SHA algorithms are structured differently and are distinguished as *SHA-0*, *SHA-1*, and *SHA-2*. The *SHA-2* family uses an identical algorithm with a variable digest size which is distinguished as *SHA-224*, *SHA-256*, *SHA-384*, and *SHA-512*.

SHA-1 is the best established of the existing SHA hash functions, and is employed in several widely used security applications and protocols. In 2005, security flaws were identified in SHA-1, namely that a possible mathematical weakness might exist, indicating that a stronger hash function would be desirable. Although no attacks have yet been reported on the SHA-2 variants, they are algorithmically similar to SHA-1 and so efforts have been made to develop improved alternatives. A new hash standard, SHA-3, using a brand new algorithm, called **Keccak**, has been selected via an open competition that ran between fall 2008 and 2012.

SHA-3 has been standardized in august 2015 as **FIPS 202**.

4.1 SHA-0 and SHA-1

The original specification of the algorithm was published in 1993 as the *Secure Hash Standard*, FIPS PUB 180, by US government standards agency NIST (National Institute of Standards and Technology). This version is now often referred to as *SHA-0*. It was withdrawn by NSA shortly after publication and was superseded by the revised version, published in 1995 in FIPS PUB 180-1 and commonly referred to as *SHA-1*. SHA-1 differs from SHA-0 only by a single bitwise rotation in the message schedule of its compression function; this was done, according to NSA, to correct a flaw in the original algorithm which reduced its cryptographic security. However, NSA did not provide any further explanation or identify the flaw that was corrected. Weaknesses have subsequently been reported in both SHA-0 and SHA-1. SHA-1 appears to provide greater resistance to attacks, supporting the NSA’s assertion that the change increased the security.

SHA-1 (as well as SHA-0) produces a 160-bit digest from a message with a maximum length of ($2^{64} - 1$) bits. SHA-1 is based on principles similar to those used by Ronald L. Rivest of MIT in the design of the MD4 and MD5 message digest algorithms, but has a more conservative design.

4.2 SHA-2 family

NIST published four additional hash functions in the SHA family, named after their digest lengths (in bits): SHA-224, SHA-256, SHA-384, and SHA-512. The algorithms are collectively known as SHA-2.

The algorithms were first published in 2001 in the draft FIPS PUB 180-2, at which time review and comment were accepted. FIPS PUB 180-2, which also includes SHA-1, was released as an official standard in 2002. In February 2004, a change notice was published for FIPS PUB 180-2, specifying an additional variant, SHA-224, defined to match the key length of two-key Triple DES. These variants are patented in US patent 6829355. The United States has released the patent under a royalty free license.

SHA-256 and SHA-512 are novel hash functions computed with 32- and 64-bit words, respectively. They use different shift amounts and additive constants, but their structures are otherwise virtually identical, differing only in the number of rounds. SHA-224 and SHA-384 are simply truncated versions of the first two, computed with different initial values.

Unlike SHA-1, the SHA-2 functions are not widely used, despite their better security. Reasons might include lack of support for SHA-2 on systems running Windows XP SP2 or older, a lack of perceived urgency since SHA-1 collisions have not yet been found, or a desire to wait until SHA-3 is standardized. SHA-256 is used to authenticate Debian Linux software packages and in the DKIM message signing standard; SHA-512 is part of a system to authenticate archival video from the International Criminal Tribunal of the Rwandan genocide. SHA-256 and SHA-512 are proposed for use in DNSSEC NIST's directive that U.S. government agencies stop most uses of SHA-1 after 2010, and the completion of SHA-3, may accelerate migration away from SHA-1.

Currently, the best public attacks on SHA-2 break 24 of the 64 or 80 rounds.

4.3 SHA-3

An open competition for a new SHA-3 function was formally announced in the *Federal Register* on November 2, 2007. "NIST is initiating an effort to develop one or more additional hash algorithms through a public competition, similar to the development process for the Advanced Encryption Standard (AES)." Submissions were due October 31, 2008 and the proclamation of a winner and publication of the new standard took place in 2012.

NIST selected 51 entries for the Round 1, and 14 of them advanced to Round 2.

4.3.1 accepted for round two

The following hash function submissions have been accepted for Round Two.

- BLAKE
- Blue Midnight Wish
- CubeHash
- ECHO (France Telecom)
- Fugue
- Grøstl (Knudsen et al.)
- Hamsi
- JH
- Keccak (Keccak team, Daemen et al.)
- Luffa
- Shabal
- SHAvite-3
- SIMD
- Skein (Schneier et al.)

4.3.2 and the winner is … Keccak

In October 2012 the Keccak algorithm has been declared the winner..

4.4 applications

SHA-1 is the most widely employed of the SHA family. It forms part of several widely used security applications and protocols, including TLS and SSL, PGP, SSH, S/MIME, and IPsec. Those applications can also use MD5; both MD5 and SHA-1 are descended from MD4. SHA-1 hashing is also used in distributed revision control systems such as Git, Mercurial, and Monotone to identify revisions, and to detect data corruption or tampering.

SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 are the secure hash algorithms required by law for use in certain U. S. Government applications, including use within other cryptographic algorithms and protocols, for the protection of sensitive unclassified information. FIPS PUB 180-1 also encouraged adoption and use of SHA-1 by private and commercial organizations. SHA-1 is being retired for most government uses; the U.S. National Institute of Standards and Technology says, "Federal agencies ***should*** stop using SHA-1 for...applications that require collision resistance as soon as practical, and must use the SHA-2 family of hash functions for these applications after 2010" (emphasis in original).

A prime motivation for the publication of the Secure Hash Algorithm was the Digital Signature Standard, in which it is incorporated.

The SHA hash functions have been used as the basis for the SHACAL block ciphers.

4.5 cryptanalysis and validation

For a hash function for which L is the number of bits in the message digest, finding a message that corresponds to a given message digest can always be done using a brute force search in 2^L evaluations. This is called a **preimage attack** and may or may not be practical depending on L and the particular computing environment. The second criterion, finding two different messages that produce the same message digest, known as a **collision**, requires on average only $2^{L/2}$ evaluations using a **birthday attack**. For the latter reason the strength of a hash function is usually compared to a symmetric cipher of half the message digest length. Thus SHA-1 was originally thought to have 80-bit strength.

Cryptographers have produced collision pairs for SHA-0 and have found algorithms that should produce SHA-1 collisions in far fewer than the originally expected 2^{80} evaluations.

In terms of practical security, a major concern about these new attacks is that they might pave the way to more efficient ones. Whether this is the case has yet to be seen, but a migration to stronger hashes is believed to be prudent. Some of the applications that use cryptographic hashes, such as password storage, are only minimally affected by a collision attack. Constructing a password that works for a given account requires a preimage attack, as well as access to the hash of the original password (typically in the *shadow* file) which may or may not be trivial. Reversing password encryption (e.g. to obtain a password to try against a user's account elsewhere) is not made possible by the attacks. (However, even a secure password hash can't prevent brute-force attacks on weak passwords.)

In the case of document signing, an attacker could not simply fake a signature from an existing document—the attacker would have to produce a pair of documents, one innocuous and one damaging, and get the private key holder to sign the innocuous document. There are practical circumstances in which this is possible; until the end of 2008, it was possible to create forged SSL certificates using an MD5 collision.

4.5.1 SHA-0

At CRYPTO 98, two French researchers, Florent Chabaud and Antoine Joux, presented an attack on SHA-0 (Chabaud and Joux, 1998): collisions can be found with complexity 2^{61}, fewer than the 2^{80} for an ideal hash function of the same size.

In 2004, Biham and Chen found near-collisions for SHA-0—two messages that hash to nearly the same value; in this case, 142 out of the 160 bits are equal. They also found full collisions of SHA-0 reduced to 62 out of its 80 rounds.

Subsequently, on 12 August 2004, a collision for the full SHA-0 algorithm was announced by Joux, Carribault, Lemuet, and Jalby. This was done by using a generalization of the Chabaud and Joux attack. Finding the collision had complexity 2^{51} and took about 80,000 CPU hours on a supercomputer with 256 Itanium 2 processors. (Equivalent to 13 days of full-time use of the computer.)

On 17 August 2004, at the Rump Session of CRYPTO 2004, preliminary results were announced by Wang, Feng, Lai, and Yu, about an attack on MD5, SHA-0 and other hash functions. The complexity of their attack on SHA-0 is 2^{40}, significantly better than the attack by Joux *et al.*

In February 2005, an attack by Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu was announced which could find collisions in SHA-0 in 2^{39} operations.

4.5.2 SHA-1

In light of the results for SHA-0, some experts suggested that plans for the use of SHA-1 in new cryptosystems should be reconsidered. After the CRYPTO 2004 results were published, NIST announced that they planned to phase out the use of SHA-1 by 2010 in favor of the SHA-2 variants.

In early 2005, Rijmen and Oswald published an attack on a reduced version of SHA-1—53 out of 80 rounds—which finds collisions with a computational effort of fewer than 2^{80} operations.

In February 2005, an attack by Xiaoyun Wang, Yiqun Lisa Yin, Bayarjargal, and Hongbo Yu was announced. The attacks can find collisions in the full version of SHA-1, requiring fewer than 2^{69} operations.

(A brute-force search would require 2^{80} operations.)

The authors write: "In particular, our analysis is built upon the original differential attack on SHA-0 [*sic*], the near collision attack on SHA-0, the multiblock collision techniques, as well as the message modification techniques used in the collision search attack on MD5. Breaking SHA-1 would not be possible without these powerful analytical techniques." The authors have presented a collision for 58-round SHA-1, found with 2^{33} hash operations. The paper with the full attack description was published in August 2005 at the CRYPTO conference.

In an interview, Yin states that, "Roughly, we exploit the following two weaknesses: One is that the file preprocessing step is not complicated enough; another is that certain math operations in the first 20 rounds have unexpected security problems."

On 17 August 2005, an improvement on the SHA-1 attack was announced on behalf of Xiaoyun Wang, Andrew Yao and Frances Yao at the CRYPTO 2005 rump session, lowering the complexity required for finding a collision in SHA-1 to 2^{63}. On 18 December 2007 the details of this result were explained and verified by Martin Cochran.

Christophe De Cannière and Christian Rechberger further improved the attack on SHA-1 in "Finding SHA-1 Characteristics: General Results and Applications," receiving the Best Paper Award at ASIACRYPT 2006. A two-block collision for 64-round SHA-1 was presented, found using unoptimized methods with 2^{35} compression function evaluations. As this attack requires the equivalent of about 2^{35} evaluations, it is considered to be a significant theoretical break. In order to find an actual collision in the full 80 rounds of the hash function, however, massive amounts of computer time are required. To that end, a collision search for SHA-1 using the distributed computing platform BOINC began August 8, 2007, organized by the Graz University of Technology. The effort was abandoned May 12, 2009 due to lack of progress.

At the Rump Session of CRYPTO 2006, Christian Rechberger and Christophe De Cannière claimed to have discovered a collision attack on SHA-1 that would allow an attacker to select at least parts of the message.

Cameron McDonald, Philip Hawkes and Josef Pieprzyk presented a hash collision attack with claimed complexity 2^{52} at the Rump session of Eurocrypt 2009. However, the accompanying paper, "Differential Path for SHA-1 with complexity O(2^{52})" has been withdrawn due to the authors' discovery that their estimate was incorrect.

4.5.3 SHA-2

There are two meet-in-the-middle preimage attacks against SHA-2 with a reduced number of rounds. The first one attacks 41-round SHA-256 out of 64 rounds with time complexity of 2^253.5 and space complexity of 2^16, and 46-round SHA-512 out of 80 rounds with time 2^511.5 and space 2^3. The second one attacks 42-round SHA-256 with time complexity of 2^251.7 and space complexity of 2^12, and 42-round SHA-512 with time 2^502 and space 2^22.

4.6 SHA-1 overview

SHA-1 is the object of FIPS 180-1, a NIST document.

4.7 operational prerequisites

4.7.1 bit strings and integers

The following terminology related to bit strings and integers will be used:

a. A hex digit is an element of the set {0, 1, ... , 9, A, ... , F}. A hex digit is the representation of a 4-bit string. **Examples:** 7 = 0111, A = 1010.

b. A word equals a 32-bit string which may be represented as a sequence of 8 hex digits. To convert a word to 8 hex digits each 4-bit string is converted to its hex equivalent as described in (a) above. **Example:**

1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

c. An integer between 0 and 2^{32} - 1 inclusive may be represented as a word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. **Example:** the integer 291 = $2^8+2^5+2^1+2^0$ = 256+32+2+1 is represented by the hex word, 00000123.

If z is an integer, 0 <= z < 2^{64}, then z = 2^{32}x + y where 0 <= x < 2^{32} and 0 <= y < 2^{32}. Since x and y can be represented as words X and Y, respectively, z can be represented as the pair of words (X,Y).

d. block = 512-bit string. A block (e.g., B) may be represented as a sequence of 16 words.

4.7.2 operations on words

The following logical operators will be applied to words:

a. Bitwise logical word operations

X ^ Y = bitwise logical "and" of X and Y.

X \/ Y = bitwise logical "inclusive-or" of X and Y.

X XOR Y = bitwise logical "exclusive-or" of X and Y

~ X = bitwise logical "complement" of X.

Example:

```
        01101100101110011101001001111011
    XOR 01100101110000010110100110110111
        --------------------------------
    =   00001001011110001011101111001100
```

b. The operation X + Y is defined as follows: words X and Y represent integers x and y, where 0 <= x < 2^{32} and 0 <= y < 2^{32}. For positive integers n and m, let n mod m be the remainder upon dividing n by m. Compute

z = (x + y) mod 2^{32}.

Then 0 <= z < 2^{32}. Convert z to a word, Z, and define Z = X + Y.

c. The circular left shift operation S^n(X), where X is a word and n is an integer with 0 <= n ^{32}, is defined by

S^n(X) = (X << n) OR (X >> 32-n).

In the above, X << n is obtained as follows: discard the left-most n bits of X and then pad the result with n zeroes on the right (the result will still be 32 bits). X >> n is obtained by discarding the right-most n bits of X and then padding the result with n zeroes on the left. Thus S^n(X) is equivalent to a circular shift of X by n positions to the left.

4.8 SHA-1 description

4.8.1 message padding

The SHA-1 is used to compute a message digest for a message or data file that is provided as input. The message or data file should be considered to be a bit string. The length of the message is the number of bits in the message (the empty message has length 0). If the number of bits in a message is a multiple of 8, for compactness we can represent the message in hex. The purpose of message padding is to make the total length of a padded message a multiple of 512. The SHA-1 sequentially processes blocks of 512 bits when computing the message digest. The following specifies how this padding shall be performed. As a summary, a "1" followed by m "0"s followed by a 64-bit integer are appended to the end of the message to produce a padded message of length 512 * n. The 64-bit integer is l, the length of the original message. The padded message is then processed by the SHA-1 as n 512-bit blocks.

Suppose a message has length $l < 2^{64}$. Before it is input to the SHA-1, the message is padded on the right as follows:

a. "1" is appended. **Example:** if the original message is "01010000", this is padded to "010100001".

b. "0"s are appended. The number of "0"s will depend on the original length of the message. The last 64 bits of the last 512-bit block are reserved for the length l of the original message.

Example: Suppose the original message is the bit string

01100001 01100010 01100011 01100100 01100101.

After step (a) this gives

01100001 01100010 01100011 01100100 01100101 1.

Since l = 40, the number of bits in the above is 41 and 407 "0"s are appended, making the total now 448. This gives (in hex)

61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000.

c. Obtain the 2-word representation of l, the number of bits in the original message. If $l < 2^{32}$ then the first word is all zeroes. Append these two words to the padded message.

Example: Suppose the original message is as in (b). Then l = 40 (note that l is computed before any padding). The two-word representation of 40 is hex 00000000 00000028. Hence the final padded message is hex

61626364 65800000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000028.

The padded message will contain 16 * n words for some n > 0. The padded message is regarded as a sequence of n blocks M_1 , M_2, ... , M_n, where each M_i contains 16 words and M_1 contains the first characters (or bits) of the message.

4.8.2 functions used

A sequence of logical functions f_0, f_1,..., f_{79} is used in the SHA-1. Each f_t, 0 <= t <= 79, operates on three 32-bit words B, C, D and produces a 32-bit word as output. f_t(B,C,D) is defined as follows: for words B, C, D,

f_t(B,C,D) = (B AND C) OR ((NOT B) AND D) (0 <= t <= 19)

f_t(B,C,D) = B XOR C XOR D (20 <= t <= 39)

$f_t(B,C,D)$ = (B AND C) OR (B AND D) OR (C AND D) (40 <= t <= 59)

$f_t(B,C,D)$ = B XOR C XOR D (60 <= t <= 79).

4.8.3 constants used

A sequence of constant words K(0), K(1), ... , K(79) is used in the SHA-1. In hex these are given by

K = 5A827999 (0 <= t <= 19)

K_t = 6ED9EBA1 (20 <= t <= 39)

K_t = 8F1BBCDC (40 <= t <= 59)

K_t = CA62C1D6 (60 <= t <= 79)

4.9 SHA-1 pseudocode

A **cryptographic hash function** is a transformation that takes an input and returns a fixed-size string, which is called the hash value.

```
// initialize variables
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

// pre-processing:
append the bit '1' to the message
append 0 ≤ k < 512 bits '0', so that the resulting message length (in bits)
   is congruent to 448 ≡ −64 (mod 512)
append length of message (before pre-processing), in bits, as 64-bit big-
endian integer

// Process the message in successive 512-bit chunks:
// break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    // Extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

    // Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f = (b and c) or ((not b) and d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39
            f = b xor c xor d
            k = 0x6ED9EBA1
```

```
        else if 40 ≤ i ≤ 59
            f = (b and c) or (b and d) or (c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k + w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        a = temp

    // Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian):
digest = hash = h0 append h1 append h2 append h3 append h4
```

4.10 a simple message and its digest

This appendix is for informational purposes only and is not required to meet the standard.

Let the message be the ASCII binary-coded form of "abc", i.e.,

01100001 01100010 01100011.

This message has length l = 24. In step (a) of Section 4, we append "1". In step (b) we append 423 "0"s. In step (c) we append hex 00000000 00000018, the 2-word representation of 24. Thus the final padded message consists of one block, so that n = 1 in the notation of Section 4.

The initial hex values of $\{H_i\}$ are

H_0 = 67452301
H_1 = EFCDAB89
H_2 = 98BADCFE
H_3 = 10325476
H_4 = C3D2E1F0.

Start processing block 1. The words of block 1 are

W[0] = 61626380
W[1] = 00000000
W[2] = 00000000
W[3] = 00000000
W[4] = 00000000
W[5] = 00000000
W[6] = 00000000
W[7] = 00000000
W[8] = 00000000

```
W[9] = 00000000
W[10] = 00000000
W[11] = 00000000
W[12] = 00000000
W[13] = 00000000
W[14] = 00000000
W[15] = 00000018.
```

The hex values of A,B,C,D,E after pass t of the "for t = 0 to 79" loop (step (d) of Section 7 or step (c) of Section 8) are

```
             A           B           C           D           E

t =  0: 0116FC33    67452301    7BF36AE2    98BADCFE    10325476
t =  1: 8990536D    0116FC33    59D148C0    7BF36AE2    98BADCFE
t =  2: A1390F08    8990536D    C045BF0C    59D148C0    7BF36AE2
t =  3: CDD8E11B    A1390F08    626414DB    C045BF0C    59D148C0
t =  4: CFD499DE    CDD8E11B    284E43C2    626414DB    C045BF0C
t =  5: 3FC7CA40    CFD499DE    F3763846    284E43C2    626414DB
t =  6: 993E30C1    3FC7CA40    B3F52677    F3763846    284E43C2
t =  7: 9E8C07D4    993E30C1    0FF1F290    B3F52677    F3763846
t =  8: 4B6AE328    9E8C07D4    664F8C30    0FF1F290    B3F52677
t =  9: 8351F929    4B6AE328    27A301F5    664F8C30    0FF1F290
t = 10: FBDA9E89    8351F929    12DAB8CA    27A301F5    664F8C30
t = 11: 63188FE4    FBDA9E89    60D47E4A    12DAB8CA    27A301F5
t = 12: 4607B664    63188FE4    7EF6A7A2    60D47E4A    12DAB8CA
t = 13: 9128F695    4607B664    18C623F9    7EF6A7A2    60D47E4A
t = 14: 196BEE77    9128F695    1181ED99    18C623F9    7EF6A7A2
t = 15: 20BDD62F    196BEE77    644A3DA5    1181ED99    18C623F9
t = 16: 4E925823    20BDD62F    C65AFB9D    644A3DA5    1181ED99
t = 17: 82AA6728    4E925823    C82F758B    C65AFB9D    644A3DA5
t = 18: DC64901D    82AA6728    D3A49608    C82F758B    C65AFB9D
t = 19: FD9E1D7D    DC64901D    20AA99CA    D3A49608    C82F758B
t = 20: 1A37B0CA    FD9E1D7D    77192407    20AA99CA    D3A49608
t = 21: 33A23BFC    1A37B0CA    7F67875F    77192407    20AA99CA
t = 22: 21283486    33A23BFC    868DEC32    7F67875F    77192407
t = 23: D541F12D    21283486    0CE88EFF    868DEC32    7F67875F
t = 24: C7567DC6    D541F12D    884A0D21    0CE88EFF    868DEC32
t = 25: 48413BA4    C7567DC6    75507C4B    884A0D21    0CE88EFF
t = 26: BE35FBD5    48413BA4    B1D59F71    75507C4B    884A0D21
t = 27: 4AA84D97    BE35FBD5    12104EE9    B1D59F71    75507C4B
t = 28: 8370B52E    4AA84D97    6F8D7EF5    12104EE9    B1D59F71
t = 29: C5FBAF5D    8370B52E    D2AA1365    6F8D7EF5    12104EE9
t = 30: 1267B407    C5FBAF5D    A0DC2D4B    D2AA1365    6F8D7EF5
t = 31: 3B845D33    1267B407    717EEBD7    A0DC2D4B    D2AA1365
t = 32: 046FAA0A    3B845D33    C499ED01    717EEBD7    A0DC2D4B
t = 33: 2C0EBC11    046FAA0A    CEE1174C    C499ED01    717EEBD7
t = 34: 21796AD4    2C0EBC11    811BEA82    CEE1174C    C499ED01
t = 35: DCBBB0CB    21796AD4    4B03AF04    811BEA82    CEE1174C
t = 36: 0F511FD8    DCBBB0CB    085E5AB5    4B03AF04    811BEA82
t = 37: DC63973F    0F511FD8    F72EEC32    085E5AB5    4B03AF04
t = 38: 4C986405    DC63973F    03D447F6    F72EEC32    085E5AB5
t = 39: 32DE1CBA    4C986405    F718E5CF    03D447F6    F72EEC32
t = 40: FC87DEDF    32DE1CBA    53261901    F718E5CF    03D447F6
t = 41: 970A0D5C    FC87DEDF    8CB7872E    53261901    F718E5CF
t = 42: 7F193DC5    970A0D5C    FF21F7B7    8CB7872E    53261901
t = 43: EE1B1AAF    7F193DC5    25C28357    FF21F7B7    8CB7872E
t = 44: 40F28E09    EE1B1AAF    5FC64F71    25C28357    FF21F7B7
t = 45: 1C51E1F2    40F28E09    FB86C6AB    5FC64F71    25C28357
```

```
t = 46: A01B846C    1C51E1F2    503CA382    FB86C6AB    5FC64F71
t = 47: BEAD02CA    A01B846C    8714787C    503CA382    FB86C6AB
t = 48: BAF39337    BEAD02CA    2806E11B    8714787C    503CA382
t = 49: 120731C5    BAF39337    AFAB40B2    2806E11B    8714787C
t = 50: 641DB2CE    120731C5    EEBCE4CD    AFAB40B2    2806E11B
t = 51: 3847AD66    641DB2CE    4481CC71    EEBCE4CD    AFAB40B2
t = 52: E490436D    3847AD66    99076CB3    4481CC71    EEBCE4CD
t = 53: 27E9F1D8    E490436D    8E11EB59    99076CB3    4481CC71
t = 54: 7B71F76D    27E9F1D8    792410DB    8E11EB59    99076CB3
t = 55: 5E6456AF    7B71F76D    09FA7C76    792410DB    8E11EB59
t = 56: C846093F    5E6456AF    5EDC7DDB    09FA7C76    792410DB
t = 57: D262FF50    C846093F    D79915AB    5EDC7DDB    09FA7C76
t = 58: 09D785FD    D262FF50    F211824F    D79915AB    5EDC7DDB
t = 59: 3F52DE5A    09D785FD    3498BFD4    F211824F    D79915AB
t = 60: D756C147    3F52DE5A    4275E17F    3498BFD4    F211824F
t = 61: 548C9CB2    D756C147    8FD4B796    4275E17F    3498BFD4
t = 62: B66C020B    548C9CB2    F5D5B051    8FD4B796    4275E17F
t = 63: 6B61C9E1    B66C020B    9523272C    F5D5B051    8FD4B796
t = 64: 19DFA7AC    6B61C9E1    ED9B0082    9523272C    F5D5B051
t = 65: 101655F9    19DFA7AC    5AD87278    ED9B0082    9523272C
t = 66: 0C3DF2B4    101655F9    0677E9EB    5AD87278    ED9B0082
t = 67: 78DD4D2B    0C3DF2B4    4405957E    0677E9EB    5AD87278
t = 68: 497093C0    78DD4D2B    030F7CAD    4405957E    0677E9EB
t = 69: 3F2588C2    497093C0    DE37534A    030F7CAD    4405957E
t = 70: C199F8C7    3F2588C2    125C24F0    DE37534A    030F7CAD
t = 71: 39859DE7    C199F8C7    8FC96230    125C24F0    DE37534A
t = 72: EDB42DE4    39859DE7    F0667E31    8FC96230    125C24F0
t = 73: 11793F6F    EDB42DE4    CE616779    F0667E31    8FC96230
t = 74: 5EE76897    11793F6F    3B6D0B79    CE616779    F0667E31
t = 75: 63F7DAB7    5EE76897    C45E4FDB    3B6D0B79    CE616779
t = 76: A079B7D9    63F7DAB7    D7B9DA25    C45E4FDB    3B6D0B79
t = 77: 860D21CC    A079B7D9    D8FDF6AD    D7B9DA25    C45E4FDB
t = 78: 5738D5E1    860D21CC    681E6DF6    D8FDF6AD    D7B9DA25
t = 79: 42541B35    5738D5E1    21834873    681E6DF6    D8FDF6AD.
```

Block 1 has been processed. The values of $\{H_i\}$ are

H_0 = 67452301 + 42541B35 = A9993E36
H_1 = EFCDAB89 + 5738D5E1 = 4706816A
H_2 = 98BADCFE + 21834873 = BA3E2571
H_3 = 10325476 + 681E6DF6 = 7850C26C
H_4 = C3D2E1F0 + D8FDF6AD = 9CD0D89D.

Message digest = A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D

4.11 the Keccak sponge function family for SHA-3

As explained in [KSF], authored by Gudo Bertoni, Joan Daemen, Michael Peeters and Gilles Van Assche

Keccak (pronounced [kɛtʃak], like “ketchak”) is a family of hash functions that has been adopted as the NIST's hash algorithm for the SHA-3. The text below is a quick description of Keccak using pseudo-code. In no way should this introductory text be considered as a formal and reference description of Keccak. Instead the goal here is to present Keccak with emphasis on readability and clarity.

4.11.1 Structure of Keccak

Keccak is a family of hash functions that is based on the sponge construction, and hence is a sponge function family. In Keccak, the underlying function is a permutation chosen in a set of seven Keccak-f permutations, denoted Keccak-f[b], where b ∈ {25, 50, 100, 200, 400, 800, 1600} is the width of the permutation. The width of the permutation is also the width of the state in the sponge construction.

The state is organized as an array of 5×5 lanes, each of length w ∈ {1, 2, 4, 8, 16, 32, 64} (b=25w). When implemented on a 64-bit processor, a lane of Keccak-f[1600] can be represented as a 64-bit CPU word.

We obtain the Keccak[r,c] sponge function, with parameters capacity c and bitrate r, if we apply the sponge construction to Keccak-f[r+c] and by applying a specific padding to the message input.

4.11.2 Pseudo-code description

We first start with the description of Keccak-f in the pseudo-code below. The number of rounds nr depends on the permutation width, and is given by nr = 12+2l, where 2l = w. This gives 24 rounds for Keccak-f[1600].

```
Keccak-f[b](A) {
  forall i in 0…nr-1
    A = Round[b](A, RC[i])
  return A
}


Round[b](A,RC) {
  // θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   forall x in
0…4
  D[x] = C[x-1] xor rot(C[x+1],1),                             forall x in
0…4
  A[x,y] = A[x,y] xor D[x],                          forall (x,y) in (0…4,0…
4)

  // ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                forall (x,y) in (0…4,0…
4)

  // χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]), forall (x,y) in (0…4,0…
4)

  // ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

In the pseudo-code above, the following conventions are in use. All the operations on the indices are done modulo 5. A denotes the complete permutation state array, and A[x,y] denotes a particular lane in that state. B[x,y], C[x], D[x] are intermediate variables. The constants r[x,y] are the rotation offsets (see Table 2), while RC[i] are the round constants (see Table 1). rot(W,r) is the usual bitwise cyclic shift operation, moving bit at position i into position i+r (modulo the lane size).

Then, we present the pseudo-code for the Keccak[r,c] sponge function, with parameters capacity c and bitrate r. The description below is restricted to the case of messages that span a whole number of bytes. For messages with a number of bits not dividable by 8, we refer to the specifications [1] for more details. Also, we assume for simplicity that r is a multiple of the lane size; this is the case for the SHA-3 candidate parameters.

```
    Keccak[r,c](M) {
      // Initialization and padding
      S[x,y] = 0,                                 forall (x,y) in (0…4,0…4)
      P = M || 0x01 || 0x00 || … || 0x00
      P = P xor (0x00 || … || 0x00 || 0x80)

      // Absorbing phase
      forall block Pi in P
        S[x,y] = S[x,y] xor Pi[x+5*y],          forall (x,y) such that x+5*y <
r/w
        S = Keccak-f[r+c](S)

      // Squeezing phase
      Z = empty string
      while output is requested
        Z = Z || S[x,y],                        forall (x,y) such that x+5*y <
r/w
        S = Keccak-f[r+c](S)

      return Z
    }
```

In the pseudo-code above, S denotes the state as an array of lanes. The padded message P is organized as an array of blocks Pi, themselves organized as arrays of lanes. The || operator denotes the usual byte string concatenation.

chapter 5 digital encryption standard

5.1 history of DES

The origins of DES go back to the early 1970s. In 1972, after concluding a study on the US government's computer security needs, the US standards body NBS (National Bureau of Standards) - now named NIST (National Institute of Standards and Technology) - identified a need for a government-wide standard for encrypting unclassified, sensitive information. Accordingly, on 15 May 1973, after consulting with the NSA, NBS solicited proposals for a cipher that would meet rigorous design criteria. None of the submissions, however, turned out to be suitable. A second request was issued on 27 August 1974. This time, IBM submitted a candidate which was deemed acceptable - a cipher developed during the period 1973–1974 based on an earlier algorithm, Horst Feistel's Lucifer cipher. The team at IBM involved in cipher design and analysis included Feistel, Walter Tuchman, Don Coppersmith, Alan Konheim, Carl Meyer, Mike Matyas, Roy Adler, Edna Grossman, Bill Notz, Lynn Smith, and Bryant Tuckerman.

5.1.1 NSA's involvement in the design

On 17 March 1975, the proposed DES was published in the *Federal Register*. Public comments were requested, and in the following year two open workshops were held to discuss the proposed standard. There was some criticism from various parties, including from public-key cryptography pioneers Martin Hellman and Whitfield Diffie, citing a shortened key length and the mysterious "S-boxes" as evidence of improper interference from the NSA. The suspicion was that the algorithm had been covertly weakened by the intelligence agency so that they - but no-one else - could easily read encrypted messages. Alan Konheim (one of the designers of DES) commented, "We sent the S-boxes off to Washington. They came back and were all different." The United States Senate Select Committee on Intelligence reviewed the NSA's actions to determine whether there had been any improper involvement. In the unclassified summary of their findings, published in 1978, the Committee wrote:

> *"In the development of DES, NSA convinced IBM that a reduced key size was sufficient; indirectly assisted in the development of the S-box structures; and certified that the final DES algorithm was, to the best of their knowledge, free from any statistical or mathematical weakness."*

However, it also found that

> *"NSA did not tamper with the design of the algorithm in any way. IBM invented and designed the algorithm, made all pertinent decisions regarding it, and concurred that the agreed upon key size was more than adequate for all commercial applications for which the DES was intended."*

Another member of the DES team, Walter Tuchman, is quoted as saying, "We developed the DES algorithm entirely within IBM using IBMers. The NSA did not dictate a single wire!" In contrast, a declassified NSA book on cryptologic history states:

> *"In 1973 NBS solicited private industry for a data encryption standard (DES). The first offerings were disappointing, so NSA began working on its own algorithm. Then Howard Rosenblum, deputy director for research and engineering, discovered that Walter Tuchman of IBM was working on a modification to Lucifer for general use. NSA gave Tuchman a clearance and brought him in to work jointly with the Agency on his Lucifer modification."*

Some of the suspicions about hidden weaknesses in the S-boxes were allayed in 1990, with the independent discovery and open publication by Eli Biham and Adi Shamir of differential cryptanalysis, a general method for breaking block ciphers. The S-boxes of DES were much more resistant to the attack than if they had been chosen at random, strongly suggesting that IBM knew about the technique back in the 1970s. This was indeed the case - in 1994, Don Coppersmith published some of the original design criteria for the S-boxes. According to Steven Levy, IBM Watson researchers discovered differential cryptanalytic attacks in 1974 and were asked by the NSA to keep the technique secret. Coppersmith explains IBM's secrecy decision by saying, "that was because [differential cryptanalysis] can be a very powerful tool, used against many schemes, and there was concern that such information in the public domain could adversely affect national security." Levy quotes Walter Tuchman: "They asked us to stamp all our documents confidential... We actually put a number on each one and locked them up in safes, because they were considered U.S. government classified. They said do it. So I did it".

5.1.2 the algorithm as a standard

Despite the criticisms, DES was approved as a federal standard in November 1976, and published on 15 January 1977 as **FIPS PUB 46**, authorized for use on all unclassified data. It was subsequently reaffirmed as the standard in 1983, 1988 (revised as **FIPS-46-1**), 1993 (**FIPS-46-2**), and again in 1999 (**FIPS-46-3**), the latter prescribing "Triple DES" (see below). On 26 May 2002, DES was finally superseded by the Advanced Encryption Standard (AES), following a public competition. On 19 May 2005, FIPS 46-3 was officially withdrawn, but NIST has approved Triple DES through the year 2030 for sensitive government information.

The algorithm is also specified in ANSI X3.92, NIST SP 800-67 and ISO/IEC 18033-3 (as a component of TDEA).

Another theoretical attack, linear cryptanalysis, was published in 1994, but it was a brute force attack in 1998 that demonstrated that DES could be attacked very practically, and highlighted the need for a replacement algorithm.

5.2 description

DES is the archetypal block cipher - an algorithm that takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bitstring of the same length. In the case of DES, the block size is 64 bits. DES also uses a key to customize the transformation, so that decryption can supposedly only be performed by those who know the particular key used to encrypt. The key ostensibly consists of 64 bits; however, only 56 of these are actually used by the algorithm. Eight bits are used solely for checking parity, and are thereafter discarded. Hence the effective key length is 56 bits, and it is usually quoted as such.

Like other block ciphers, DES by itself is not a secure means of encryption but must instead be used in a mode of operation. FIPS-81 specifies several modes for use with DES. Further comments on the usage of DES are contained in FIPS-74.

The algorithm's overall structure is shown in Figure 1: there are 16 identical stages of processing, termed *rounds*. There is also an initial and final permutation, termed *IP* and *FP*, which are inverses (IP "undoes" the action of FP, and vice versa). IP and FP have almost no cryptographic significance, but were apparently included in order to facilitate loading blocks in and out of mid-1970s hardware, as well as to make DES run slower in software.

Before the main rounds, the block is divided into two 32-bit halves and processed alternately; this criss-crossing is known as the Feistel scheme. The Feistel structure ensures that decryption and encryption are very similar processes - the only difference is that the subkeys are applied in the reverse order when decrypting. The rest of the algorithm is identical. This greatly simplifies implementation, particularly in hardware, as there is no need for separate encryption and decryption algorithms.

The ⊕ symbol denotes the exclusive-OR (XOR) operation. The *F-function* scrambles half a block together with some of the key. The output from the F-function is then combined with the other half of the block, and the halves are swapped before the next round. After the final round, the halves are not swapped; this is a feature of the Feistel structure which makes encryption and decryption similar processes.

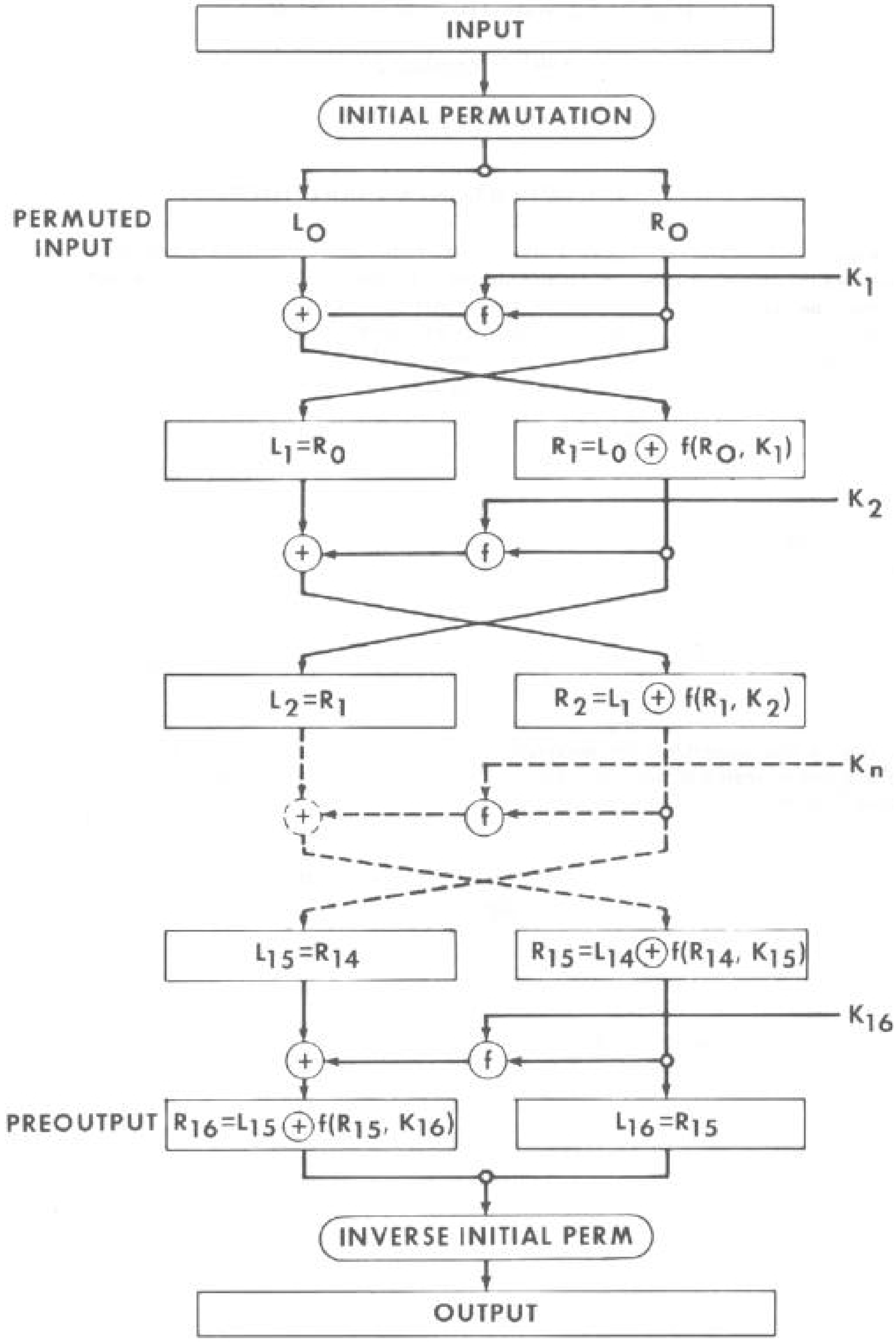


Figure 1 - The general DES algorithm structure

5.3 the Feistel (F) function

The F-function, depicted in Figure 2, operates on half a block (32 bits) at a time and consists of four stages:

1. *Expansion* - the 32-bit half-block is expanded to 48 bits using the *expansion permutation*, denoted *E* in the diagram, by duplicating half of the bits. The output consists of 8 6-bit pieces, each containing a copy of 4 corresponding input bits, plus a copy of the immediately adjacent bit from each of the input pieces to either side.
2. *Key mixing* - the result is combined with a *subkey* using an XOR operation. Sixteen 48-bit subkeys - one for each round - are derived from the main key using the *key schedule* (described below).
3. *Substitution* - after mixing in the subkey, the block is divided into eight 6-bit pieces before processing by the *S-boxes*, or *substitution boxes*. Each of the eight S-boxes replaces its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table. The S-boxes provide the core of the security of DES - without them, the cipher would be linear, and trivially breakable.
4. *Permutation* - finally, the 32 outputs from the S-boxes are rearranged according to a fixed permutation, the *P-box*. This is designed so that, after expansion, each S-box's output bits are spread across 6 different S boxes in the next round.

The alternation of substitution from the S-boxes, and permutation of bits from the P-box and E-expansion provides so-called "confusion and diffusion" respectively, a concept identified by Claude Shannon in the 1940s as a necessary condition for a secure yet practical cipher.

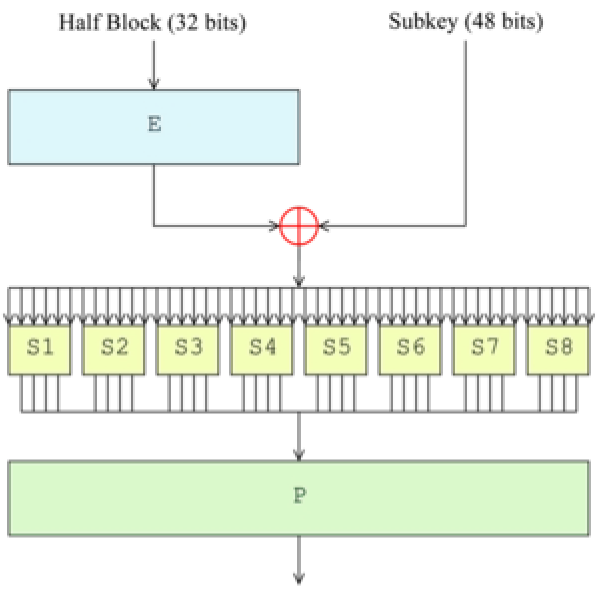


Figure 2 – the Feistel function

5.4 the key schedule

Figure 3 illustrates the *key schedule* for encryption - the algorithm which generates the subkeys. Initially, 56 bits of the key are selected from the initial 64 by *Permuted Choice 1* (*PC-1*) - the remaining eight bits are either discarded or used as parity check bits. The 56 bits are then divided into two 28-bit halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left by one or two bits (specified for each round), and then 48 subkey bits are selected by *Permuted Choice 2* (*PC-2*) - 24 bits from the left half, and 24 from the right. The rotations (denoted by "<<<" in the diagram) mean that a different set of bits is used in each subkey; each bit is used in approximately 14 out of the 16 subkeys.

The key schedule for decryption is similar - the subkeys are in reverse order compared to encryption. Apart from that change, the process is the same as for encryption.

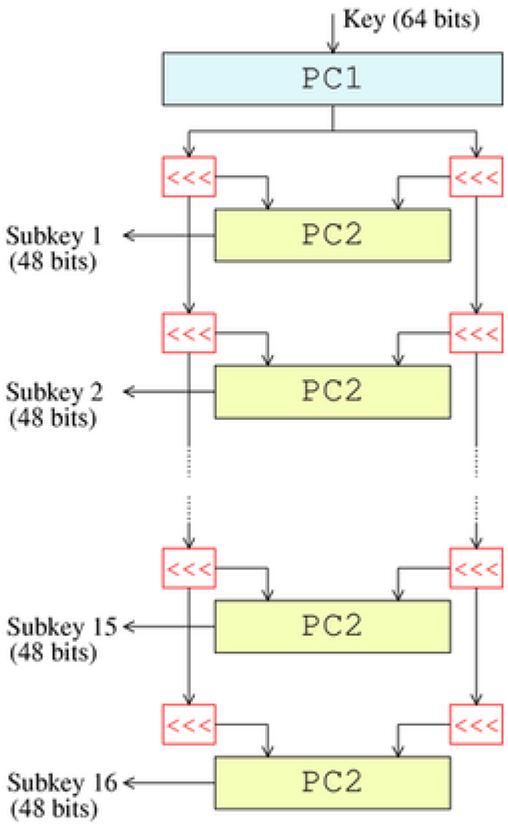


Figure 3

5.5 DES algorithm

The 64 bits of the input block to be enciphered are first subjected to the following permutation, called the initial permutation ***IP***:

IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

That is the permuted input has bit 58 of the input as its first bit, bit 50 as its second bit, and so on with bit 7 as its last bit. The permuted input block is then the input to a complex key-dependent computation described below. The output of that computation, called the preoutput, is then subjected to the following permutation which is the inverse of the initial permutation:

IP^{-1}

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

The computation which uses the permuted input block as its input to produce the preoutput block consists, but for a final interchange of blocks, of 16 iterations of a calculation that is described below in terms of the cipher function ***f*** which operates on two blocks, one of 32 bits and one of 48 bits, and produces a block of 32 bits.

Let the 64 bits of the input block to an iteration consist of a 32 bit block ***L*** followed by a 32 bit block ***R***. Using the notation defined in the introduction, the input block is then ***LR***. Let ***K*** be a block of 48 bits chosen from the 64-bit key. Then the output ***L'R'*** of an iteration with input ***LR*** is defined by:

(1) ***L'*** = ***R***

R' = ***L ^ f(R,K)***

where ^ denotes bit-by-bit addition modulo 2.

As remarked before, the input of the first iteration of the calculation is the permuted input block. If ***L'R'*** is the output of the 16th iteration then ***R'L'*** is the preoutput block. At each iteration a different block ***K*** of key bits is chosen from the 64-bit key designated by ***KEY***.

With more notation we can describe the iterations of the computation in more detail. Let ***KS*** be a function which takes an integer *n* in the range from 1 to 16 and a 64-bit block ***KEY*** as input and yields as output a 48-bit block ***Kn*** which is a permuted selection of bits from ***KEY***. That is

(2) ***Kn*** = ***KS(****n****,KEY)***

with ***Kn*** determined by the bits in 48 distinct bit positions of ***KEY***. ***KS*** is called the key schedule because the block ***K*** used in the *n*'th iteration of (1) is the block ***Kn*** determined by (2).

As before, let the permuted input block be ***LR***. Finally, let ***L()*** and ***R()*** be respectively ***L*** and ***R*** and let ***Ln*** and ***Rn*** be respectively ***L'*** and ***R'*** of (1) when ***L*** and ***R*** are respectively ***Ln-1*** and ***Rn-1*** and ***K*** is ***Kn***; that is, when *n* is in the range from 1 to 16,

(3) ***Ln*** = ***Rn-1***

Rn = ***Ln-1 ^ f(Rn-1,Kn)***

The preoutput block is then $R_{16}L_{16}$.

The key schedule produces the 16 ***Kn*** which are required for the algorithm.

5.6 an example

5.7 security and cryptanalysis

Although more information has been published on the cryptanalysis of DES than any other block cipher, the most practical attack to date is still a brute force approach. Various minor cryptanalytic properties are known, and three theoretical attacks are possible which, while having a theoretical complexity less than a brute force attack, require an unrealistic amount of known or chosen plaintext to carry out, and are not a concern in practice.

5.7.1 brute force attack

For any cipher, the most basic method of attack is brute force - trying every possible key in turn. The length of the key determines the number of possible keys, and hence the feasibility of this approach. For DES, questions were raised about the adequacy of its key size early on, even before it was adopted as a standard, and it was the small key size, rather than theoretical cryptanalysis, which dictated a need for a replacement algorithm. As a result of discussions involving external consultants including the NSA, the key size was reduced from 128 bits to 56 bits to fit on a single chip.

The EFF's US$250,000 DES cracking machine contained 1,856 custom chips and could brute force a DES key in a matter of days - the photo shows a DES Cracker circuit board fitted with several Deep Crack chips.

In academia, various proposals for a DES-cracking machine were advanced. In 1977, Diffie and Hellman proposed a machine costing an estimated US$20 million which could find a DES key in a single day. By 1993, Wiener had proposed a key-search machine costing US$1 million which would find a key within 7 hours. However, none of these early proposals were ever implemented—or, at least, no implementations were publicly acknowledged. The vulnerability of DES was practically demonstrated in the late 1990s. In 1997, RSA Security sponsored a series of contests, offering a $10,000 prize to the first team that broke a message encrypted with DES for the contest. That contest was won by the DESCHALL Project, led by Rocke Verser, Matt Curtin, and Justin Dolske, using idle cycles of thousands of computers across the Internet. The feasibility of cracking DES quickly was demonstrated in 1998 when a custom DES-cracker was built by the Electronic Frontier Foundation (EFF), a cyberspace civil rights group, at the cost of approximately US$250,000 (see EFF DES cracker). Their motivation was to show that DES was breakable in practice as well as in theory: "*There are many people who will not believe a truth until they can see it with their own eyes. Showing them a physical machine that can crack DES in a few days is the only way to convince some people that they really cannot trust their security to DES.*" The machine brute-forced a key in a little more than 2 days search.

The COPACOBANA machine, built in 2006 for US$10,000 by the Universities of Bochum and Kiel, Germany,[16] contains 120 low-cost FPGAs and could perform an exhaustive key search on DES in 9 days on average. The photo shows the backplane of the machine with the FPGAs.

The only other confirmed DES cracker was the COPACOBANA machine built in 2006 by teams of the Universities of Bochum and Kiel, both in Germany. Unlike the EFF machine, COPACOBANA consists of commercially available, reconfigurable integrated circuits. 120 of these Field-programmable gate arrays (FPGAs) of type XILINX Spartan3-1000 run in parallel. They are grouped in 20 DIMM modules, each containing 6 FPGAs. The use of reconfigurable hardware makes the machine applicable to other code breaking tasks as well. The figure shows a full-sized COPACOBANA. One of the more interesting aspects of COPACOBANA is its cost factor. One machine can be built for approximately $10,000. The cost decrease by roughly a factor of 25 over the EFF machine is an impressive example for the continuous improvement of digital hardware. Adjusting for inflation over 8 years yields an even higher improvement of about 30x. Since 2007, SciEngines GmbH, a spin-off company of the two project partners of COPACOBANA has enhanced and developed successors of COPACOBANA. In 2008 their COPACOBANA RIVYERA reduced the time to break DES to less than one day, using 128 Spartan-3 5000's.

5.7.2 attacks faster than brute-force

There are three attacks known that can break the full sixteen rounds of DES with less complexity than a brute-force search: differential cryptanalysis (DC), linear cryptanalysis (LC), and Davies' attack. However, the attacks are theoretical and are unfeasible to mount in practice; these types of attack are sometimes termed certificational weaknesses.

- *Differential cryptanalysis* was rediscovered in the late 1980s by Eli Biham and Adi Shamir; it was known earlier to both IBM and the NSA and kept secret. To break the full 16 rounds, differential cryptanalysis requires 2^{47} chosen plaintexts.[*citation needed*] DES was designed to be resistant to DC.
- *Linear cryptanalysis* was discovered by Mitsuru Matsui, and needs 2^{43} known plaintexts (Matsui, 1993); the method was implemented (Matsui, 1994), and was the first experimental cryptanalysis of DES to be reported. There is no evidence that DES was tailored to be resistant to this type of attack. A generalization of LC - *multiple linear cryptanalysis* - was suggested in 1994 (Kaliski and Robshaw), and was further refined by Biryukov et al. (2004); their analysis suggests that multiple linear approximations could be used to reduce the data requirements of the attack by at least a factor of 4 (i.e. 2^{41} instead of 2^{43}). A similar reduction in data complexity can be obtained in a chosen-plaintext variant of linear cryptanalysis (Knudsen and Mathiassen, 2000). Junod (2001) performed several experiments to determine the actual time complexity of linear cryptanalysis, and reported that it was somewhat faster than predicted, requiring time equivalent to 2^{39}–2^{41} DES evaluations.
- *Improved Davies' attack*: while linear and differential cryptanalysis are general techniques and can be applied to a number of schemes, Davies' attack is a specialised technique for DES, first suggested by Donald Davies in the eighties, and improved by Biham and Biryukov (1997). The most powerful form of the attack requires 2^{50} known plaintexts, has a computational complexity of 2^{50}, and has a 51% success rate.

There have also been attacks proposed against reduced-round versions of the cipher, i.e. versions of DES with fewer than sixteen rounds. Such analysis gives an insight into how many rounds are needed for safety, and how much of a "security margin" the full version retains. Differential-linear cryptanalysis was proposed by Langford and Hellman in 1994, and combines differential and linear cryptanalysis into a single attack. An enhanced version of the attack can break 9-round DES with $2^{15.8}$ known plaintexts and has a $2^{29.2}$ time complexity (Biham et al., 2002).

5.8 triple DES

5.8.1 algorithm

Triple DES uses a "key bundle" which comprises three DES keys, K1, K2 and K3, each of 56 bits (excluding parity bits). The encryption algorithm is:

ciphertext = EK3(DK2(EK1(plaintext)))

I.e., DES encrypt with K1, DES *decrypt* with K2, then DES encrypt with K3.

Decryption is the reverse:

plaintext = DK1(EK2(DK3(ciphertext)))

I.e., decrypt with K3, *encrypt* with K2, then decrypt with K1.

Each triple encryption encrypts one blockof 64 bits of data.

In each case the middle operation is the reverse of the first and last. This improves the strength of the algorithm when using keying option 2, and provides backward compatibility with DES with keying option 3.

5.8.2 keying options

The standards define three keying options:

- Keying option 1: All three keys are independent.
- Keying option 2: K1 and K2 are independent, and K3 = K1.
- Keying option 3: All three keys are identical, i.e. K1 = K2 = K3.

Keying option 1 is the strongest, with 3 × 56 = 168 independent key bits.

Keying option 2 provides less security, with 2 × 56 = 112 key bits. This option is stronger than simply DES encrypting twice, e.g. with K1 and K2, because it protects against meet-in-the-middle attacks.

Keying option 3 is equivalent to DES, with only 56 key bits. This option provides backward compatibility with DES, because the first and second DES operations cancel out. It is no longer recommended by the National Institute of Standards and Technology (NIST) and is not supported by ISO/IEC 18033-3.

chapter 6 advanced encryption standard

6.1 the history of AES

Advanced Encryption Standard (**AES**) is an encryption standard adopted by the U.S. government. The standard comprises three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as **Rijndael.** Each AES cipher has a 128-bit block size, with key sizes of 128, 192 and 256 bits, respectively. The AES ciphers have been analyzed extensively and are now used worldwide, as was the case with its predecessor, the Data Encryption Standard (DES).

AES was announced by National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 (FIPS 197) on November 26, 2001 after a 5-year standardization process in which fifteen competing designs were presented and evaluated before Rijndael was selected as the most suitable (see Advanced Encryption Standard process for more details). It became effective as a Federal government standard on May 26, 2002 after approval by the Secretary of Commerce. It is available in many different encryption packages. AES is the first publicly accessible and open cipher approved by the NSA for top secret information (see Security of AES, below).

The Rijndael cipher was developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and submitted by them to the AES selection process. Rijndael (pronounced [rɛindaːl]) is a portmanteau of the names of the two inventors.

6.2 overall description

AES is based on a design principle known as a Substitution permutation network. It is fast in both software and hardware. Unlike its predecessor, DES, AES does not use a Feistel network.

AES has a fixed block size of 128 bits and a key size of 128, 192, or 256 bits, whereas Rijndael can be specified with block and key sizes in any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits.

AES operates on a 4×4 array of bytes, termed the *state* (versions of Rijndael with a larger block size have additional columns in the state). Most AES calculations are done in a special finite field.

The AES cipher is specified as a number of repetitions of transformation rounds that convert the input plaintext into the final output of ciphertext. Each round consists of several processing steps, including one that depends on the encryption key. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

6.3 algebraic fundamentals

6.3.1 byte representation

The processing unit in AES is the byte. This is quite different from DES, where the basic operations were performed at bit level. A byte is a sequence of eight bits, call them $b_0, b_1, \ldots, b_7$ – b_0 being the least significant bit. To each byte (or sequence of eight bits) we associate a polynomial of degree at most 7, with coefficients in $\mathbf{Z}_2$, the ring of integers modulo 2, as follows. If our byte value is $b_7 * 2^7 + b_6 * 2^6 + b_5 * 2^5 + b_4 * 2^4 + b_3 * 2^3 + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0$ with all coefficients in $\mathbf{Z}_2$, then the associated polynomial is $b_7 * x^7 + b_6 * x^6 + \ldots + b_2 * x^2 + b_1 * x + b_0$.

For example, {01100011} identifies the specific finite field element $x^6 + x^5 + x + 1$. Apart from their binary or polynomial representations, bytes will be also represented in hexadecimal notation, either as {63} (the above byte) or as 0x63.

All bytes in the AES algorithm are interpreted as finite field elements. Finite field elements can be added and multiplied, but these operations are different from those used for numbers. The following subsections introduce the basic mathematical concepts needed for the algorithm description.

6.3.2 addition

The addition of two elements in a finite field is achieved by “adding” the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the XOR operation (denoted by ^) - i.e., modulo 2 - so that 1^1 = 0 , 1^0 = 1, and 0^0 = 0 .

Consequently, subtraction of polynomials is identical to addition of polynomials.

Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ and $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$, the sum is

$\{c_7c_6c_5c_4c_3c_2c_1c_0\}$, where each $c_i = a_i$ ^ b_i (i.e., $c_7 = a_7$ ^ b_7, $c_6 = a_6$ ^ b_6, ...$c_0 = a_0$ ^ b_0).

For example, the following expressions are equivalent to one another:

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2$ (polynomial notation);

{01010111} ^ {10000011} = {11010100} (binary notation);

{57} ^ {83} = {d4} (hexadecimal notation).

6.3.3 multiplication

In the polynomial representation, multiplication in GF(28) (denoted by ·) corresponds with the multiplication of polynomials modulo an **irreducible polynomial** of degree 8. A polynomial is irreducible if its only divisors are one and itself. **For the AES algorithm, this irreducible** polynomial is

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

or {01}{1b} in hexadecimal notation.

For example, {57} · {83} = {c1}, because

$(x^6 + x^4 + x^2 + x + 1)\,(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^7 +$

$x^7 + x^5 + x^3 + x^2 + x +$

$x^6 + x^4 + x^2 + x + 1$

$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$

and

$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$ modulo $(x^8 + x^4 + x^3 + x + 1)$

$= x^7 + x^6 + 1.$

The modular reduction by $m(x)$ ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication.

The multiplication defined above is associative, and the element {01} is the multiplicative identity. For any non-zero binary polynomial $b(x)$ of degree less than 8, the multiplicative inverse of $b(x)$, denoted b-1(x), can be found as follows: the extended Euclidean algorithm is used to compute polynomials $a(x)$ and $c(x)$ such that

$b(x)a(x) + m(x)c(x) = 1$

Hence, $a(x) \cdot b(x)$mod $m(x)$ = 1, which means

b^8-1 $(x) = a(x)$mod $m(x)$

Moreover, for any $a(x)$, $b(x)$ and $c(x)$ in the field, it holds that

$a(x) \cdot (b(x) + c(x)) = a(x) \cdot b(x) + a(x) \cdot c(x)$.

It follows that the set of 256 possible byte values, with XOR used as addition and the multiplication defined as above, has the structure of the finite field GF(2^8).

6.3.4 multiplication by *x*

Multiplying the binary polynomial defined in equation (3.1) with the polynomial x results in The result $x \cdot b(x)$ is obtained by reducing the above result modulo $m(x)$. If $b7 = 0$, the result is already in reduced form. If $b7 = 1$, the reduction is accomplished by subtracting (i.e., XORing) the polynomial $m(x)$. It follows that multiplication by x (i.e., {00000010} or {02}) can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with {1b}. This operation on bytes is denoted by xtime().

Multiplication by higher powers of x can be implemented by repeated application of xtime().

By adding intermediate results, multiplication by any constant can be implemented.

For example, {57} · {13} = {fe} because

{57} · {02} = xtime({57}) = {ae}

{57} · {04} = xtime({ae}) = {47}

{57} · {08} = xtime({47}) = {8e}

{57} · {10} = xtime({8e}) = {07},

thus,

{57} · {13} = {57} · ({01} ^ {02} ^ {10})

= {57} ^ {ae} ^ {07}

= {fe}.

6.3.5 polynomials with coefficients in GF(2^8)

Four-term polynomials can be defined - with coefficients that are finite field elements - as:

$$a(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

which will be denoted as a word in the form $[a_0, a_1, a_2, a_3]$. Note that the polynomials in this section behave somewhat differently than the polynomials used in the definition of finite field elements, even though both types of polynomials use the same indeterminate, x. The coefficients in this section are themselves finite field elements, i.e., bytes, instead of bits; also, the multiplication of four-term polynomials uses a different reduction polynomial, defined below.

The distinction should always be clear from the context.

To illustrate the addition and multiplication operations, let

$$b(x) = b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

define a second four-term polynomial. Addition is performed by adding the finite field coefficients of like powers of x. This addition corresponds to an XOR operation between the corresponding bytes in each of the words – in other words, the XOR of the complete word values.

Thus, using the above two equations,

$$a(x) + b(x) = (a_3 \wedge b_3)x^3 + (a_2 \wedge b_2)x^2 + (a_1 \wedge b_1)x + (a_0 \wedge b_0)$$

Multiplication is achieved in two steps. In the first step, the polynomial product $c(x) = a(x) \cdot b(x)$ is algebraically expanded, and like powers are collected to give

$$c(x) = c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0$$

where

$c_0 = a_0 \cdot b_0$

$c_1 = a_1 \cdot b_0 \wedge a_0 \cdot b_1$

$c_2 = a_2 \cdot b_0 \wedge a_1 \cdot b_1 \wedge a_0 \cdot b_2$

$c_3 = a_3 \cdot b_0 \wedge a_2 \cdot b_1 \wedge a_1 \cdot b_2 \wedge a_0 \cdot b_3$.

$c_4 = a_3 \cdot b_1 \wedge a_2 \cdot b_2 \wedge a_1 \cdot b_3$

$c_5 = a_3 \cdot b_2 \wedge a_2 \cdot b_3$

$c_6 = a_3 \cdot b_3$

The result, $c(x)$, does not represent a four-byte word. Therefore, the second step of the multiplication is to reduce $c(x)$ modulo a polynomial of degree 4; the result can be reduced to a polynomial of degree less than 4. **For the AES algorithm, this is accomplished with the polynomial $x^8$4 + 1**, so that

$x^i \bmod(x^4 + 1) = x^{i \bmod 4}$

The modular product of $a(x)$ and $b(x)$, denoted by $a(x) \otimes b(x)$, is given by the four-term polynomial $d(x)$, defined as follows:

$d(x) = d_3x^3 + d_2x^2 + d_3\, x + d_3$

with

$d_0 = a_0 \cdot b_0 \wedge a_3 \cdot b_1 \wedge a_2 \cdot b_2 \wedge a_1 \cdot b_3$

$d_1 = a_1 \cdot b_0 \wedge a_0 \cdot b_1 \wedge a_3 \cdot b_2 \wedge a_2 \cdot b_3$

$d_2 = a_2 \cdot b_0 \wedge a_1 \cdot b_1 \wedge a_0 \cdot b_2 \wedge a_3 \cdot b_3$

$d_3 = a_3 \cdot b_0 \wedge a_2 \cdot b_1 \wedge a_1 \cdot b_2 \wedge a_0 \cdot b_3$

Because *x*4 +1 is not an irreducible polynomial over GF(28), multiplication by a fixed four-term polynomial is not necessarily invertible. However, the AES algorithm specifies a fixed four-term polynomial that *does* have an inverse.

$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$

$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$

Another polynomial used in the AES algorithm has $a_0 = a_1 = a_2 = \{00\}$ and $a_3 = \{01\}$, which is the polynomial x^3. Its effect is to form the output word by rotating bytes in the input word. This means that [b_0, b_1, b_2, b_3] is transformed into [b_1, b_2, b_3, b_0].

6.4 the key schedule

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word temp
   i = 0

   while (i < Nk)
         w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
         i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
         temp = w[i-1]

         if (i mod Nk = 0)
               temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
         else if (Nk > 6 and i mod Nk = 4)
```

```
            temp = SubWord(temp)
        end if

        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

The key schedule for decryption is similar - the subkeys are in reverse order compared to encryption. Apart from that change, the process is the same as for encryption.

6.5 functions involved

6.5.1 the SubBytes() step

In the `SubBytes` step, each byte in the state is replaced with its entry in a fixed 8-bit lookup table, *S*; $b_{ij} = S(a_{ij})$.

In the `SubBytes` step, each byte in the array is updated using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over **GF**(2^8), known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points (and so is a derangement), and also any opposite fixed points.

6.5.2 the ShiftRows() step

In the `ShiftRows` step, bytes in each row of the state are shifted cyclically to the left. The number of places each byte is shifted differs for each row.

The `ShiftRows` step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For the block of size 128 bits and 192 bits the shifting pattern is the same. In this way, each column of the output state of the `ShiftRows` step is composed of bytes from each column of the input state. (Rijndael variants with a larger block size have slightly different offsets). In the case of the 256-bit block, the first row is unchanged and the shifting for second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively - this change only applies for the Rijndael cipher when used with a 256-bit block, as AES does not use 256-bit blocks.

6.5.3 the MixColumns() step

In the `MixColumns` step, each column of the state is multiplied with a fixed polynomial *c(x)*.

In the `MixColumns` step, the four bytes of each column of the state are combined using an invertible linear transformation. The `MixColumns` function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with `ShiftRows`, `MixColumns` provides diffusion in the cipher. Each column is treated as a polynomial over **GF**(2^8) and is then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = 0x03x^3 + x^2 + x + 0x02$. (The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from **GF**(2)[*x*].) The `MixColumns` step can also be viewed as a multiplication by a particular MDS matrix in Finite field. This process is described further in the article Rijndael mix columns.

6.5.4 the AddRoundKey() step

In the `AddRoundKey` step, each byte of the state is combined with a byte of the round subkey using the

XOR operation (⊕).

In the `AddRoundKey` step, the subkey is combined with the state. For each round, a subkey is derived from the main key using Rijndael's key schedule; each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

6.6 pseudocode

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte state[4,Nb]
   state = in
   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
          SubBytes(state)
          ShiftRows(state)
          MixColumns(state)
          AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
   out = state
end
```

6.7 inverse functions

Figure 3 illustrates the *key schedule* for encryption - the algorithm which generates the subkeys.

Initially, 56 bits of the key are selected from the initial 64 by *Permuted Choice 1* (*PC-1*) - the remaining eight bits are either discarded or used as parity check bits. The 56 bits are then divided into two 28-bit halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left by one or two bits (specified for each round), and then 48 subkey bits are selected by *Permuted Choice 2* (*PC-2*) - 24 bits from the left half, and 24 from the right. The rotations (denoted by "<<<" in the diagram) mean that a different set of bits is used in each subkey; each bit is used in approximately 14 out of the 16 subkeys.

The key schedule for decryption is similar - the subkeys are in reverse order compared to encryption. Apart from that change, the process is the same as for encryption.

6.8 the decryption phase

Figure 3 illustrates the *key schedule* for encryption - the algorithm which generates the subkeys. Initially, 56 bits of the key are selected from the initial 64 by *Permuted Choice 1* (*PC-1*) - the remaining eight bits are either discarded or used as parity check bits. The 56 bits are then divided into two 28-bit halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left by one or two bits (specified for each round), and then 48 subkey bits are selected by *Permuted Choice 2* (*PC-2*) - 24 bits from the left half, and 24 from the right. The rotations (denoted by "<<<" in the diagram) mean that a different set of bits is used in each subkey; each bit is used in approximately 14 out of the 16 subkeys.

The key schedule for decryption is similar - the subkeys are in reverse order compared to encryption. Apart from that change, the process is the same as for encryption.

6.9 an example

```
PLAINTEXT:     00112233445566778899aabbccddeeff
KEY:           000102030405060708090a0b0c0d0e0f

CIPHER (ENCRYPT):

round[ 0].input 00112233445566778899aabbccddeeff
round[ 0].k_sch 000102030405060708090a0b0c0d0e0f
round[ 1].start 00102030405060708090a0b0c0d0e0f0
round[ 1].s_box 63cab7040953d051cd60e0e7ba70e18c
round[ 1].s_row 6353e08c0960e104cd70b751bacad0e7
round[ 1].m_col 5f72641557f5bc92f7be3b291db9f91a
round[ 1].k_sch d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 2].start 89d810e8855ace682d1843d8cb128fe4
round[ 2].s_box a761ca9b97be8b45d8ad1a611fc97369
round[ 2].s_row a7be1a6997ad739bd8c9ca451f618b61
round[ 2].m_col ff87968431d86a51645151fa773ad009
round[ 2].k_sch b692cf0b643dbdf1be9bc5006830b3fe
round[ 3].start 4915598f55e5d7a0daca94fa1f0a63f7
round[ 3].s_box 3b59cb73fcd90ee05774222dc067fb68
round[ 3].s_row 3bd92268fc74fb735767cbe0c0590e2d
round[ 3].m_col 4c9c1e66f771f0762c3f868e534df256
round[ 3].k_sch b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 4].start fa636a2825b339c940668a3157244d17
round[ 4].s_box 2dfb02343f6d12dd09337ec75b36e3f0
round[ 4].s_row 2d6d7ef03f33e334093602dd5bfb12c7
round[ 4].m_col 6385b79ffc538df997be478e7547d691
round[ 4].k_sch 47f7f7bc95353e03f96c32bcfd058dfd
round[ 5].start 247240236966b3fa6ed2753288425b6c
round[ 5].s_box 36400926f9336d2d9fb59d23c42c3950
round[ 5].s_row 36339d50f9b539269f2c092dc4406d23
round[ 5].m_col f4bcd45432e554d075f1d6c51dd03b3c
round[ 5].k_sch 3caaa3e8a99f9deb50f3af57adf622aa
round[ 6].start c81677bc9b7ac93b25027992b0261996
round[ 6].s_box e847f56514dadde23f77b64fe7f7d490
round[ 6].s_row e8dab6901477d4653ff7f5e2e747dd4f
round[ 6].m_col 9816ee7400f87f556b2c049c8e5ad036
round[ 6].k_sch 5e390f7df7a69296a7553dc10aa31f6b
round[ 7].start c62fe109f75eedc3cc79395d84f9cf5d
round[ 7].s_box b415f8016858552e4bb6124c5f998a4c
round[ 7].s_row b458124c68b68a014b99f82e5f15554c
round[ 7].m_col c57e1c159a9bd286f05f4be098c63439
round[ 7].k_sch 14f9701ae35fe28c440adf4d4ea9c026
round[ 8].start d1876c0f79c4300ab45594add66ff41f
round[ 8].s_box 3e175076b61c04678dfc2295f6a8bfc0
round[ 8].s_row 3e1c22c0b6fcbf768da85067f6170495
round[ 8].m_col baa03de7a1f9b56ed5512cba5f414d23
round[ 8].k_sch 47438735a41c65b9e016baf4aebf7ad2
round[ 9].start fde3bad205e5d0d73547964ef1fe37f1
round[ 9].s_box 5411f4b56bd9700e96a0902fa1bb9aa1
round[ 9].s_row 54d990a16ba09ab596bbf40ea111702f
round[ 9].m_col e9f74eec023020f61bf2ccf2353c21c7
round[ 9].k_sch 549932d1f08557681093ed9cbe2c974e
round[10].start bd6e7c3df2b5779e0b61216e8b10b689
round[10].s_box 7a9f102789d5f50b2beffd9f3dca4ea7
round[10].s_row 7ad5fda789ef4e272bca100b3d9ff59f
round[10].k_sch 13111d7fe3944a17f307a78b4d2b30c5
```

```
round[10].output 69c4e0d86a7b0430d8cdb78070b4c55a

INVERSE CIPHER (DECRYPT):

round[ 0].iinput 69c4e0d86a7b0430d8cdb78070b4c55a
round[ 0].ik_sch 13111d7fe3944a17f307a78b4d2b30c5
round[ 1].istart 7ad5fda789ef4e272bca100b3d9ff59f
round[ 1].is_row 7a9f102789d5f50b2beffd9f3dca4ea7
round[ 1].is_box bd6e7c3df2b5779e0b61216e8b10b689
round[ 1].ik_sch 549932d1f08557681093ed9cbe2c974e
round[ 1].ik_add e9f74eec023020f61bf2ccf2353c21c7
round[ 2].istart 54d990a16ba09ab596bbf40ea111702f
round[ 2].is_row 5411f4b56bd9700e96a0902fa1bb9aa1
round[ 2].is_box fde3bad205e5d0d73547964ef1fe37f1
round[ 2].ik_sch 47438735a41c65b9e016baf4aebf7ad2
round[ 2].ik_add baa03de7a1f9b56ed5512cba5f414d23
round[ 3].istart 3e1c22c0b6fcbf768da85067f6170495
round[ 3].is_row 3e175076b61c04678dfc2295f6a8bfc0
round[ 3].is_box d1876c0f79c4300ab45594add66ff41f
round[ 3].ik_sch 14f9701ae35fe28c440adf4d4ea9c026
round[ 3].ik_add c57e1c159a9bd286f05f4be098c63439
round[ 4].istart b458124c68b68a014b99f82e5f15554c
round[ 4].is_row b415f8016858552e4bb6124c5f998a4c
round[ 4].is_box c62fe109f75eedc3cc79395d84f9cf5d
round[ 4].ik_sch 5e390f7df7a69296a7553dc10aa31f6b
round[ 4].ik_add 9816ee7400f87f556b2c049c8e5ad036
round[ 5].istart e8dab6901477d4653ff7f5e2e747dd4f
round[ 5].is_row e847f56514dadde23f77b64fe7f7d490
round[ 5].is_box c81677bc9b7ac93b25027992b0261996
round[ 5].ik_sch 3caaa3e8a99f9deb50f3af57adf622aa
round[ 5].ik_add f4bcd45432e554d075f1d6c51dd03b3c
round[ 6].istart 36339d50f9b539269f2c092dc4406d23
round[ 6].is_row 36400926f9336d2d9fb59d23c42c3950
round[ 6].is_box 247240236966b3fa6ed2753288425b6c
round[ 6].ik_sch 47f7f7bc95353e03f96c32bcfd058dfd
round[ 6].ik_add 6385b79ffc538df997be478e7547d691
round[ 7].istart 2d6d7ef03f33e334093602dd5bfb12c7
round[ 7].is_row 2dfb02343f6d12dd09337ec75b36e3f0
round[ 7].is_box fa636a2825b339c940668a3157244d17
round[ 7].ik_sch b6ff744ed2c2c9bf6c590cbf0469bf41
round[ 7].ik_add 4c9c1e66f771f0762c3f868e534df256
round[ 8].istart 3bd92268fc74fb735767cbe0c0590e2d
round[ 8].is_row 3b59cb73fcd90ee05774222dc067fb68
round[ 8].is_box 4915598f55e5d7a0daca94fa1f0a63f7
round[ 8].ik_sch b692cf0b643dbdf1be9bc5006830b3fe
round[ 8].ik_add ff87968431d86a51645151fa773ad009
round[ 9].istart a7be1a6997ad739bd8c9ca451f618b61
round[ 9].is_row a761ca9b97be8b45d8ad1a611fc97369
round[ 9].is_box 89d810e8855ace682d1843d8cb128fe4
round[ 9].ik_sch d6aa74fdd2af72fadaa678f1d6ab76fe
round[ 9].ik_add 5f72641557f5bc92f7be3b291db9f91a
round[10].istart 6353e08c0960e104cd70b751bacad0e7
round[10].is_row 63cab7040953d051cd60e0e7ba70e18c
round[10].is_box 00102030405060708090a0b0c0d0e0f0
round[10].ik_sch 000102030405060708090a0b0c0d0e0f
round[10].i_outp 00112233445566778899aabbccddeeff
```

6.10 side-channel attacks

Until May 2009, the only successful published attacks against the full AES were side-channel attacks on specific implementations. The National Security Agency (NSA) reviewed all the AES finalists, including Rijndael, and stated that all of them were secure enough for US Government non-classified data. In June 2003, the US Government announced that AES may be used to protect classified information:

> *"The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths. The implementation of AES in products intended to protect national security systems and/or information must be reviewed and certified by NSA prior to their acquisition and use."*

AES has 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. By 2006, the best known attacks were on 7 rounds for 128-bit keys, 8 rounds for 192-bit keys, and 9 rounds for 256-bit keys.

For cryptographers, a cryptographic "break" is anything faster than an exhaustive search. Thus, an XSL attack against a 128-bit-key AES requiring 2^{100} operations (compared to 2^{128} possible keys) would be considered a break. The largest successful publicly-known brute force attack has been against a 64-bit RC5 key by distributed.net.

Unlike most other block ciphers, AES has a very neat algebraic description. In 2002, a theoretical attack, termed the "XSL attack", was announced by Nicolas Courtois and Josef Pieprzyk, purporting to show a weakness in the AES algorithm due to its simple description. Since then, other papers have shown that the attack as originally presented is unworkable; see XSL attack on block ciphers.

During the AES process, developers of competing algorithms wrote of Rijndael, "...we are concerned about [its] use...in security-critical applications." However, at the end of the AES process, Bruce Schneier, a developer of the competing algorithm Twofish, wrote that while he thought successful academic attacks on Rijndael would be developed someday, "I do not believe that anyone will ever discover an attack that will allow someone to read Rijndael traffic."

On July 1, 2009, Bruce Schneier blogged about a related-key attack on the 192-bit and 256-bit versions of AES discovered by Alex Biryukov and Dmitry Khovratovich; the related key attack on the 256-bit version of AES exploits AES' somewhat simple key schedule and has a complexity of 2^{119}. This is a follow-up to an attack discovered earlier in 2009 by Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic, with a complexity of 2^{96} for one out of every 2^{35} keys.

Another attack was blogged by Bruce Schneier on July 30, 2009 and published on August 3, 2009. This new attack, by Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir, is against AES-256 that uses only two related keys and 2^{39} time to recover the complete 256-bit key of a 9-round version, or 2^{45} time for a 10-round version with a stronger type of related subkey attack, or 2^{70} time for a 11-round version. 256-bit AES uses 14 rounds, so these attacks aren't effective against full AES.

In November 2009, there exists the first published attack against the 8-round version of AES-128. This known-key distinguishing attack is an improvement of the rebound or the start-from-the-middle attacks for AES-like permutations, which view two consecutive rounds of permutation as the application of a so-called Super-Box. It works on the 8-round version of AES-128, with a computation complexity of 2^{48}, and a memory complexity of 2^{32}.

6.10.1 side-channel attacks

Side-channel attacks do not attack the underlying cipher and so have nothing to do with its security as described here, but attack implementations of the cipher on systems which inadvertently leak data. There are several such known attacks on certain implementations of AES.

In April 2005, D.J. Bernstein announced a cache-timing attack that he used to break a custom server that used OpenSSL's AES encryption. The custom server was designed to give out as much timing information as possible (the server reports back the number of machine cycles taken by the encryption operation), and the attack required over 200 million chosen plaintexts.

In October 2005, Dag Arne Osvik, Adi Shamir and Eran Tromer presented a paper demonstrating several cache-timing attacks against AES. One attack was able to obtain an entire AES key after only 800 operations triggering encryptions, in a total of 65 milliseconds. This attack requires the attacker to be able to run programs on the same system that is performing AES.

Tadayoshi Kohno wrote a paper entitled "Attacking and Repairing the WinZip Encryption Scheme" showing possible attacks against the WinZip AES implementation (the zip archive's metadata isn't encrypted).

In December 2009 an attack on some hardware implementations was published that used Differential Fault Analysis and allows recovery of key with complexity of 2^{32}.

chapter 7 elements of number theory

chapter 8 the diffie-hellman key exchange algorithm

8.1 history of the protocol

The Diffie -Hellman – Merkle key agreement was invented in 1976 during a collaboration between Whitfield Diffie and Martin Hellman and was the first practical method for establishing a shared secret over an unprotected communications channel. Ralph Merkle's work on public key distribution was an influence. John Gill suggested application of the discrete logarithm problem. It had first been invented by Malcolm Williamson of GCHQ in the UK some years previously, but GCHQ chose not to make it public until 1997, by which time it had no influence on research in academia.

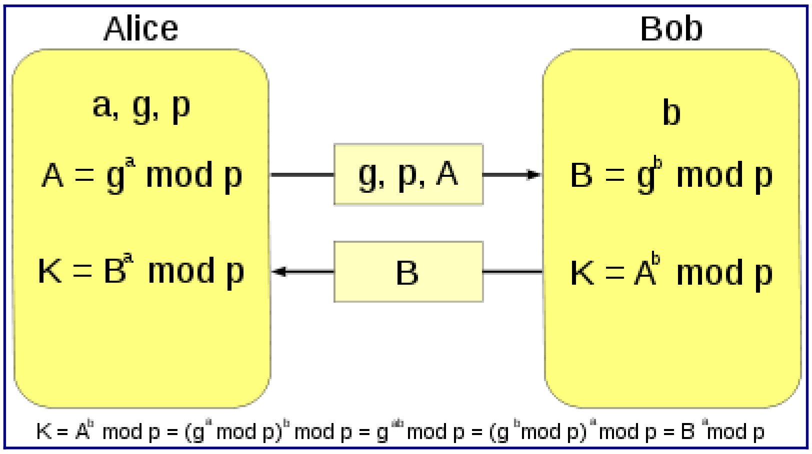
The method was followed shortly afterwards by RSA, another implementation of public key cryptography using asymmetric algorithms.

U.S. Patent 4,200,770, now expired, describes the algorithm and credits Hellman, Diffie, and Merkle as inventors.

8.2 description

Diffie-Hellman establishes a shared secret that can be used for secret communications by exchanging data over a public network.

Here is an explanation which includes the encryption's mathematics:



Diffie-Hellman key exchange

The simplest, and original, implementation of the protocol uses the multiplicative group of integers modulo *p*, where *p* is prime and *g* is primitive root mod *p*. Here is an example of the protocol, with non-secret values in green, and secret values in **boldface red**:

ALICE				BOB		
Secret	Public	Calculus		Calculus	Public	Secret
	p, g				p, g	
a						b
		g^a (mod p)				

				g^b (mod p)		
	$(g^a)^b$(mod p)		$\rightarrow$ $\leftarrow$		$(g^b)^a$(mod p)	

1. Alice and Bob agree to use a prime number p=23 and base g=5.
2. Alice chooses a secret integer ***a*=6**, then sends Bob A = $g^{\boldsymbol{a}}$ mod p
   - A = $5^{\mathbf{6}}$ mod 23 = 8.
3. Bob chooses a secret integer ***b*=15**, then sends Alice B = $g^{\boldsymbol{b}}$ mod p
   - B = $5^{\mathbf{15}}$ mod 23 = 19.
4. Alice computes **s** = $B^{\boldsymbol{a}}$ mod p
   - $19^{\mathbf{6}}$ mod 23 = **2**.
5. Bob computes **s** = $A^{\boldsymbol{b}}$ mod p
   - $8^{\mathbf{15}}$ mod 23 = **2**.

Both Alice and Bob have arrived at the same value, because g^{ab} and g^{ba} are equal mod p. Note that only a, b and g^{ab} = g^{ba} mod p are kept secret. All the other values -- p, g, g^a *mod* p, and g^b *mod* p -- are sent in the clear. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel. Of course, much larger values of a, b, and p would be needed to make this example secure, since it is easy to try all the possible values of g^{ab} mod 23 (there will be, at most, 22 such values, even if a and b are large). If p were a prime of at least 300 digits, and a and b were at least 100 digits long, then even the best algorithms known today could not find a given only g, p, and g^a mod p, even using all of mankind's computing power. The problem is known as the discrete logarithm problem. Note that g need not be large at all, and in practice is usually either 2 or 5.

Here's a more general description of the protocol:

1. Alice and Bob agree on a finite cyclic group G and a generating element g in G. (This is usually done long before the rest of the protocol; g is assumed to be known by all attackers.) We will write the group G multiplicatively.
2. Alice picks a random natural number a and sends g^a to Bob.
3. Bob picks a random natural number b and sends g^b to Alice.
4. Alice computes $(g^b)^a$.
5. Bob computes $(g^a)^b$.

Both Alice and Bob are now in possession of the group element g^{ab}, which can serve as the shared secret key. The values of $(g^b)^a$ and $(g^a)^b$ are the same because groups are power associative. (See also exponentiation.)

8.3 chart

Here is a chart to help simplify who knows what. (Eve is an eavesdropper—she watches what is sent between Alice and Bob, but she does not alter the contents of their communications.)

- Let s = shared secret key. s = 2
- Let a = Alice's private key. a = 6

- Let A = Alice's public key. $A = g^a \bmod p = 8$
- Let b = Bob's private key. b = 15
- Let B = Bob's public key. $B = g^b \bmod p = 19$
- Let g = public base. g=5
- Let p = public (prime) number. p = 23

Alice		***Eve***		***Bob***	
knows	doesn't know	knows	doesn't know	knows	doesn't know
p = 23	b = 15	p = 23	a = 6	p = 23	a = 6
base g = 5		base g = 5	b = 15	base g = 5	
a = 6			s = 2	b = 15	
$A = 5^6 \bmod 23 = 8$		$A = 5^a \bmod 23 = 8$		$B = 5^{15} \bmod 23 = 19$	
$B = 5^b \bmod 23 = 19$		$B = 5^b \bmod 23 = 19$		$A = 5^a \bmod 23 = 8$	
$s = 19^6 \bmod 23 = 2$		$s = 19^a \bmod 23$		$s = 8^{15} \bmod 23 = 2$	
$s = 8^b \bmod 23 = 2$		$s = 8^b \bmod 23$		$s = 19^a \bmod 23 = 2$	
$s = 19^6 \bmod 23 = 8^b \bmod 23$		$s = 19^a \bmod 23 = 8^b \bmod 23$		$s = 8^{15} \bmod 23 = 19^a \bmod 23$	
s = 2				s = 2	

Note: It should be difficult for Alice to solve for Bob's private key or for Bob to solve for Alice's private key. If it isn't difficult for Alice to solve for Bob's private key (or vice versa), Eve may simply substitute her own private / public key pair, plug Bob's public key into her private key, produce a fake shared secret key, and solve for Bob's private key (and use that to solve for the shared secret key. Eve may attempt to choose a public / private key pair that will make it easy for her to solve for Bob's private key).

8.4 security

The protocol is considered secure against eavesdroppers if *G* and *g* are chosen properly. The eavesdropper ("Eve") would have to solve the Diffie-Hellman problem to obtain g^{ab}. This is currently considered difficult. An efficient algorithm to solve the discrete logarithm problem would make it easy to compute *a* or *b* and solve the Diffie-Hellman problem, making this and many other public key cryptosystems insecure.

The order of *G* should be prime or have a large prime factor to prevent use of the Pohlig-Hellman algorithm to obtain *a* or *b*. For this reason, a Sophie Germain prime *q* is sometimes used to calculate $p=2q+1$, called a safe prime, since the order of *G* is then only divisible by 2 and *q*. *g* is then sometimes chosen to generate the order *q* subgroup of *G*, rather than *G*, so that the Legendre symbol of g^a never reveals the low order bit of *a*.

If Alice and Bob use random number generators whose outputs are not completely random and can be predicted to some extent, then Eve's task is much easier.

The secret integers *a* and *b* are discarded at the end of the session. Therefore, Diffie-Hellman key exchange by itself trivially achieves perfect forward secrecy because no long-term private keying material exists to be disclosed.

8.5 authentication

In the original description, the Diffie-Hellman-Merkel exchange by itself does not provide authentication of the communicating parties and is thus vulnerable to a man-in-the-middle attack. A person in the middle may establish two distinct Diffie-Hellman key exchanges, one with Alice and the other with Bob, effectively masquerading as Alice to Bob, and vice versa, allowing the attacker to decrypt (and read or store) then re-encrypt the messages passed between them. A method to authenticate the communicating parties to each other is generally needed to prevent this type of attack.

A variety of cryptographic authentication solutions incorporate a Diffie-Hellman exchange. When Alice and Bob have a public key infrastructure, they may digitally sign the agreed key, or g^a and g^b, as in MQV, STS and the IKE component of the IPsec protocol suite for securing Internet Protocol communications. When Alice and Bob share a password, they may use a password-authenticated key agreement form of Diffie-Hellman, such as the one described in ITU-T Recommendation X.1035, which is used by the G.hn home networking standard.

chapter 9 asymmetric encryption - RSA

In cryptography, RSA is an algorithm for public-key cryptography. It was the first algorithm known to be suitable for signing as well as encryption, and one of the first great advances in public key cryptography. RSA is widely used in electronic commerce protocols, and is believed to be secure given sufficiently long keys and the use of up-to-date implementations.

Public-key cryptography, also known as asymmetric cryptography, is a form of cryptography in which a user has a pair of cryptographic keys—a public key and a private key. The private key is kept secret, while the public key may be widely distributed. The keys are related mathematically, but the private key cannot be practically derived from the public key. A message encrypted with the public key can be decrypted only with the corresponding private key.

Conversely, *secret key cryptography*, also known as symmetric cryptography uses a single secret key for both encryption and decryption.

9.1 history

The algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT; the letters RSA are the initials of their surnames.

Clifford Cocks, a British mathematician working for the UK intelligence agency GCHQ, described an equivalent system in an internal document in 1973, but given the relatively expensive computers needed to implement it at the time, it was mostly considered a curiosity and, as far as is publicly known, was never deployed. His discovery, however, was not revealed until 1997 due to its top-secret classification, and Rivest, Shamir, and Adleman devised RSA independently of Cocks' work.

MIT was granted US patent 4405829 for a "Cryptographic communications system and method" that used the algorithm in 1983. The patent expired on 21 September 2000. Since a paper describing the algorithm had been published in August 1977, prior to the December 1977 filing date of the patent application, regulations in much of the rest of the world precluded patents elsewhere and only the US patent was granted. Had Cocks' work been publicly known, a patent in the US might not have been possible either.

9.2 operation

RSA involves a public key and a private key. The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key. The keys for the RSA algorithm are generated the following way:

- Choose two distinct large random prime numbers *p* and *q*
- Compute [illegible]
- [illegible] is used as the modulus for both the public and private keys
- Compute the totient: $\phi(n) = (p-1)(q-1)$. (The totient $\varphi(n)$ of a positive integer *n* is defined to be the number of positive integers less than or equal to *n* that are coprime to *n*. For example, $\varphi(9) = 6$ since the six numbers 1, 2, 4, 5, 7 and 8 are coprime to 9)
- Choose an integer *e* such that 1 < *e* < φ(*n*), and [illegible] and φ(*n*) share no factors other than 1 (i.e. *e* and φ(*n*) are coprime)
- *e* is released as the public key exponent
- Compute *d* to satisfy the congruence relation $de \equiv 1 \pmod{\phi(n)}$; i.e. *de* = 1 + *k*φ(*n*) for some integer *k*.

- *d* is kept as the private key exponent

Notes on the above steps:

- Step 1: Numbers can be probabilistically tested for primality.
- Step 3: changed in PKCS#1 v2.0 Public Key Cryptography Standards to $\lambda(n) = \mathrm{lcm}(p-1, q-1)$, where lcm is the least common multiple, instead of $\phi(n) = (p-1)(q-1)$.
- Step 4: A popular choice for the public exponents is $e = 2^{16} + 1 = 65537$. Some applications choose smaller values such as $e = 3, 5, 17$ or 257 instead. This is done to make encryption and signature verification faster on small devices like smart cards but small public exponents may lead to greater security risks.
- Steps 4 and 5 can be performed with the extended Euclidean algorithm; see modular arithmetic.

The extended Euclidean algorithm is an extension to the Euclidean algorithm for finding the greatest common divisor (GCD) of integers *a* and *b*: it also finds the integers *x* and *y* in Bézout's identity

$$ax + by = \gcd(a, b).$$

(Typically either x or y is negative).

The extended Euclidean algorithm is particularly useful when *a* and *b* are coprime, since *x* is the modular multiplicative inverse of *a* modulo *b*.

The public key consists of the modulus n and the public (or encryption) exponent e.
The private key consists of the modulus n and the private (or decryption) exponent d which must be kept secret.

- For efficiency a different form of the private key can be stored:
- p and q: the primes from the key generation,
- $d \mod (p-1)$ and $d \mod (q-1)$,
- $q^{-1} \mod (p)$.
- All parts of the private key must be kept secret in this form. p and q are sensitive since they are the factors of n, and allow computation of d given e. If p and q are not stored in this form of the private key then they are securely deleted along with other intermediate values from key generation.
- Although this form allows faster decryption and signing by using the Chinese Remainder Theorem, it is considerably less secure since it enables side channel attacks. This is a particular problem if implemented on smart cards, which benefit most from the improved efficiency. (Start with $y = x^e mod n$ and let the card decrypt that. So it computes $y^d (\mathrm{mod}\ p)$ or $y^d (\mathrm{mod}\ q)$ whose results give some value *z*. Now, induce an error in one of the computations. Then $\gcd(z - x, n)$ will reveal *p* or *q*.)

In cryptography, a side channel attack is any attack based on information gained from the physical implementation of a cryptosystem, rather than theoretical weaknesses in the algorithms (compare cryptanalysis). For example, timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information which can be exploited to break the system. Many side-channel attacks require considerable technical knowledge of the internal operation of the system on which the cryptography is implemented.

Attempts to break a cryptosystem by deceiving or coercing people with legitimate access are not typically called side-channel attacks: see social engineering and rubber-hose cryptanalysis. For attacks on computer systems themselves (which are often used to perform cryptography and thus contain cryptographic keys or plaintexts), see computer security.

9.3 examples

A *timing attack* watches data movement into and out of the CPU, or memory, on the hardware running the cryptosystem or algorithm. Simply by observing how long it takes to transfer key information, it is sometimes possible to determine how long the key is in this instance (or to rule out certain lengths which can also be cryptanalytically useful). Internal operational stages in many cipher implementations provide information (typically partial) about the plaintext, key values and so on, and some of this information can be inferred from observed timings. Alternatively, a timing attack may simply watch for the length of time a cryptographic algorithm requires -- this alone is sometimes enough information to be cryptanalytically useful.

A *power monitoring attack* can provide similar information by observing the power lines to the hardware, especially the CPU. As with a timing attack, considerable information is inferable for some algorithm implementations under some circumstances.

As a fundamental and inevitable fact of electrical life, fluctuations in current generate radio waves, making whatever is producing the currents subject -- at least in principle -- to a *van Eck* (aka, TEMPEST) attack. If the currents concerned are patterned in distinguishable ways, which is typically the case, the radiation can be recorded and used to infer information about the operation of the associated hardware. According to former MI5 officer Peter Wright, the British Security Service analysed emissions from French cipher equipment in the 1960s[1]. In the 1980s, Soviet eavesdroppers were known to plant bugs inside IBM Selectric typewriters to monitor the electrical noise generated as the type ball rotated and pitched to strike the paper; the characteristics of those signals could determine which key was pressed[*citation needed*].

If the relevant currents are those associated with a display device (ie, highly patterned and intended to produce human readable images), the task is greatly eased. CRT displays use substantial currents to steer their electron beams and they have been 'snooped' in real time with minimum cost hardware from considerable distances (hundreds of meters have been demonstrated). LCDs require, and use, smaller currents and are less vulnerable -- which is not to say they are invulnerable. Some LCDs have been proven that they are vulnerable too, see [2].

Also as an inescapable fact of electrical life in actual circuits, flowing currents heat the materials through which they flow. Those materials also continually lose heat to the environment due to other equally fundamental facts of thermodynamic existence, so there is a continually changing thermally induced mechanical stress as a result of these heating and cooling effects. That stress appears to be the most significant contributor to low level acoustic (i.e. *noise*) emissions from operating CPUs (about 10 kHz in some cases). Recent research by Shamir et al. has demonstrated that information about the operation of cryptosystems and algorithms can be obtained in this way as well. This is an acoustic attack; if the surface of the CPU chip, or in some cases the CPU package, can be observed, infrared images can also provide information about the code being executed on the CPU, known as a *thermal imaging attack*.

9.4 encrypting messages

Alice transmits her public key (n,e) to Bob and keeps the private key secret. Bob then wishes to send message M to Alice.

He first turns M into a number m < n by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c corresponding to:

$$c = m^e \mod n$$

This can be done quickly using the method of exponentiation by squaring. Bob then transmits c to Alice.

9.5 decrypting messages

Alice can recover m from c by using her private key exponent d by the following computation:

$$m = c^d \mod n.$$

Given , she can recover the original message M.

The above decryption procedure works because first

$$c^d \equiv (m^e)^d \equiv m^{ed} \pmod{n}.$$

Now, $ed \equiv 1 \pmod{(p-1)(q-1)}$, and hence

$ed \equiv 1 \pmod{p-1}$ and

$ed \equiv 1 \pmod{q-1}$

which can also be written as

$ed = k(p-1)+1$ and

$ed = h(q-1)+1$

for proper values of and . If is not a multiple of then and are coprime because is prime; so by Fermat's little theorem

$$m^{(p-1)} \equiv 1 \pmod{p}$$

and therefore, using the first expression for ,

$$m^{ed} = m^{k(p-1)+1} = (m^{p-1})^k m \equiv 1^k m = m \pmod{p}.$$

If instead is a multiple of , then

$$m^{ed} \equiv 0^{ed} = 0 \equiv m \pmod{p}.$$

Using the second expression for , we similarly conclude that

$$m^{ed} \equiv m \pmod{q}.$$

Since and are distinct prime numbers, applying the Chinese remainder theorem to these two congruences yields

$$m^{ed} \equiv m \pmod{pq}.$$

Thus,

$$c^d \equiv m \pmod{n}.$$

9.6 a worked example

Here is an example of RSA encryption and decryption. The parameters used here are artificially small, but you can also use OpenSSL to generate and examine a real keypair.

- Choose two prime numbers

p = 61 and *q* = 53

- Compute

n = 61 * 53 = 3233

- Compute the totient $\phi(n) = (p-1)(q-1)$

$\phi(n) = (61 - 1)(53 - 1) = 3120$

- Choose e > 1 coprime to 3120

e = 17

- Compute such that $de \equiv 1 \pmod{\phi(n)}$ e.g., by computing the modular multiplicative inverse of e modulo $\phi(n)$:

d = 2753

17 * 2753 = 46801 = 1 + 15 * 3120.

The public key is (n = 3233, e = 17). For a padded message the encryption function is:

$c = m^e \mod n = m^{17} \mod 3233.$

The private key is (n = 3233, d = 2753). The decryption function is:

$m = c^d \mod n = c^{2753} \mod 3233.$

For example, to encrypt m = 123, we calculate

$c = 123^{17} \mod 3233 = 855.$

To decrypt c = 855, we calculate

$m = 855^{2753} \mod 3233 = 123$.

Both of these calculations can be computed efficiently using the square-and-multiply algorithm for modular exponentiation.

9.7 padding schemes

When used in practice, RSA is generally combined with some padding scheme. The goal of the padding scheme is to prevent a number of attacks that potentially work against RSA without padding (In cryptography, padding refers to a number of distinct practices):

- When encrypting with low encryption exponents (e.g., e = 3) and small values of the m, (i.e. $m<n^{1/e}$) the result of m^e is strictly less than the modulus n. In this case, ciphertexts can be easily decrypted by taking the eth root of the ciphertext over the integers.
- If the same clear text message is sent to e or more recipients in an encrypted way, and the receiver's shares the same exponent e, but different p, q, and n, then it is easy to decrypt the original clear text message via the Chinese remainder theorem (Chinese remainder theorem refers to a result about congruences in number theory and its generalizations in abstract algebra). Johan Håstad noticed that this attack is possible even if the cleartexts are not equal, but the attacker knows a linear relation between them. This attack was later improved by Don Coppersmith.
- Because RSA encryption is a deterministic encryption algorithm – i.e., has no random component – an attacker can successfully launch a chosen plaintext attack against the cryptosystem, by encrypting likely plaintexts under the public key and test if they are equal to the ciphertext. A cryptosystem is called semantically secure if an attacker cannot distinguish two encryptions from each other even if the attacker knows (or has chosen) the corresponding plaintexts. As described above, RSA without padding is not semantically secure.
- RSA has the property that the product of two ciphertexts is equal to the encryption of the product of the respective plaintexts. That is $m_1^e m_2^e \equiv (m_1 m_2)^e \pmod{n}.$ Because of this multiplicative property a chosen-ciphertext attack is possible. E.g. an attacker, who wants to know the

decryption of a ciphertext $c=m^e$ mod n may ask the holder of the secret key to decrypt an unsuspicious-looking ciphertext $c' = cr^e$mod n for some value r chosen by the attacker. Because of the multiplicative property c' is the encryption of mr mod n. Hence, if the attacker is successful with the attack, he will learn mrmod n from which he can derive the message m by multiplying mr with the modular inverse of r modulo n.

To avoid these problems, practical RSA implementations typically embed some form of structured, randomized padding into the value m before encrypting it. This padding ensures that m does not fall into the range of insecure plaintexts, and that a given message, once padded, will encrypt to one of a large number of different possible ciphertexts.

Standards such as PKCS have been carefully designed to securely pad messages prior to RSA encryption. Because these schemes pad the plaintext m with some number of additional bits, the size of the un-padded message M must be somewhat smaller. RSA padding schemes must be carefully designed so as to prevent sophisticated attacks which may be facilitated by a predictable message structure. Early versions of the PKCS standard (i.e. PKCS #1 up to version 1.5) used a construction that turned RSA into a semantically secure encryption scheme. This version was later found vulnerable to a practical adaptive chosen ciphertext attack. Later versions of the standard include Optimal Asymmetric Encryption Padding (OAEP), which prevents these attacks. The PKCS standard also incorporates processing schemes designed to provide additional security for RSA signatures, e.g., the Probabilistic Signature Scheme for RSA (RSA-PSS).

9.8 practical considerations

9.8.1 key generation

Finding the large primes p and q is usually done by testing random numbers of the right size with probabilistic primality tests which quickly eliminate virtually all non-primes.

p and q should not be 'too close', lest the Fermat factorization for n be successful, if p-q, for instance is less than $2n^{1/4}$ (which for even small 1024-bit values of n is 3×10^{77}) solving for p and q is trivial. Furthermore, if either p-1 or q-1 has only small prime factors, n can be factored quickly by Pollard's p − 1 algorithm, and these values of p or q should therefore be discarded as well.

It is important that the secret key d be large enough. Michael J. Wiener showed in 1990 that if p is between q and $2q$ (which is quite typical) and $d < n^{1/4}/3$, then d can be computed efficiently from n and e. There is no known attack against small public exponents such as e=3, provided that proper padding is used. However, when no padding is used or when the padding is improperly implemented then small public exponents have a greater risk of leading to an attack, such as for example the unpadded plaintext vulnerability listed above. 65537 is a commonly used value for e. This value can be regarded as a compromise between avoiding potential small exponent attacks and still allowing efficient encryptions (or signature verification). The NIST Special Publication on Computer Security (SP 800-78 Rev 1 of August 2007) does not allow public exponents e smaller than 65537, but does not state a reason for this restriction.

9.8.2 speed

RSA is much slower than DES and other symmetric cryptosystems. In practice, Bob typically encrypts a secret message with a symmetric algorithm, encrypts the (comparatively short) symmetric key with RSA, and transmits both the RSA-encrypted symmetric key and the symmetrically-encrypted message to Alice.

Symmetric-key algorithms are a class of algorithms for cryptography that use trivially related, often identical, cryptographic keys for both decryption and encryption.

The encryption key is trivially related to the decryption key, in that they may be identical or there is a simple transform to go between the two keys. The keys, in practice, represent a shared secret between two or more parties that can be used to maintain a private information link.

Other terms for symmetric-key encryption are secret-key, single-key, one-key and eventually private-key encryption. Use of the latter term does conflict with the term *private key* in public key cryptography.

This procedure raises additional security issues. For instance, it is of utmost importance to use a strong random number generator for the symmetric key, because otherwise Eve (an eavesdropper wanting to see what was sent) could bypass RSA by guessing the symmetric key.

9.8.3 key distribution

As with all ciphers, how RSA public keys are distributed is important to security. Key distribution must be secured against a man-in-the-middle attack. Suppose Eve has some way to give Bob arbitrary keys and make him believe they belong to Alice. Suppose further that Eve can *intercept* transmissions between Alice and Bob. Eve sends Bob her own public key, which Bob believes to be Alice's. Eve can then intercept any ciphertext sent by Bob, decrypt it with her own secret key, keep a copy of the message, encrypt the message with Alice's public key, and send the new ciphertext to Alice. In principle, neither Alice nor Bob would be able to detect Eve's presence. Defenses against such attacks are often based on digital certificates or other components of a public key infrastructure.

In cryptography, a public key infrastructure (PKI) is an arrangement that binds public keys with respective user identities by means of a certificate authority (CA). The user identity must be unique for each CA. The binding is established through the registration and issuance process, which, depending on the level of assurance the binding has, may be carried out by software at a CA, or under human supervision. The PKI role that assures this binding is called the Registration Authority (RA) . For each user, the user identity, the public key, their binding, validity conditions and other attributes are made unforgeable in public key certificates issued by the CA.

The term trusted third party (TTP) may also be used for certificate authority (CA). The term PKI is sometimes erroneously used to denote public key algorithms which, however, do not require the use of a CA.

9.9 security

The security of the RSA cryptosystem is based on two mathematical problems: the problem of factoring large numbers and the RSA problem. Full decryption of an RSA ciphertext is thought to be infeasible on the assumption that both of these problems are hard, i.e., no efficient algorithm exists for solving them. Providing security against *partial* decryption may require the addition of a secure padding scheme.

The RSA problem is defined as the task of taking *e*th roots modulo a composite n: recovering a value m such that $c=m^e$ mod n, where (n, e) is an RSA public key and c is an RSA ciphertext. Currently the most promising approach to solving the RSA problem is to factor the modulus n. With the ability to recover prime factors, an attacker can compute the secret exponent d from a public key (n, e), then decrypt c using the standard procedure. To accomplish this, an attacker factors n into p and q, and computes (p-1)(q-1) which allows the determination of d from e. No polynomial-time method for factoring large integers on a classical computer has yet been found, but it has not been proven that none exists. See integer factorization for a discussion of this problem.

As of 2005, the largest number factored by a general-purpose factoring algorithm was 663 bits long (see RSA-200), using a state-of-the-art distributed implementation. RSA keys are typically 1024–2048 bits long. Some experts believe that 1024-bit keys may become breakable in the near term (though this is disputed); few see any way that 4096-bit keys could be broken in the foreseeable future. Therefore, it is generally presumed that RSA is secure if n is sufficiently large. If n is 256 bits or shorter, it can be factored in a few hours on a personal computer, using software already freely available. Keys of 512 bits (or less) have been shown to be practically breakable in 1999 when RSA-155 was factored by using several hundred computers. A theoretical hardware device named TWIRL and described by Shamir and Tromer in 2003 called into question the security of 1024 bit keys. It is currently recommended that n be at least 2048 bits long.

In 1994, Peter Shor published Shor's algorithm, showing that a quantum computer could in principle perform the factorization in polynomial time. However, quantum computation is still in the early stages of development and may never prove to be practical.

9.9.1 adaptive chosen-ciphertext attack

An adaptive chosen-ciphertext attack (abbreviated as CCA2) is an interactive form of chosen-ciphertext attack in which an attacker sends a number of ciphertexts to be decrypted, then uses the results of these decryptions to select subsequent ciphertexts. It is to be distinguished from an indifferent chosen-ciphertext attack (CCA1).

The goal of this attack is to gradually reveal information about an encrypted message, or about the decryption key itself. For public-key systems, adaptive-chosen-ciphertexts are generally applicable only when they have the property of ciphertext malleability — that is, a ciphertext can be modified in specific ways that will have a predictable effect on the decryption of that message.

9.9.2 preventing the adaptive chosen-ciphertext attack

In order to prevent adaptive-chosen-ciphertext attacks, it is necessary to use an encryption or encoding scheme that limits ciphertext malleability. A number of encoding schemes have been proposed; the most common standard for RSA encryption is Optimal Asymmetric Encryption Padding (OAEP). Unlike ad-hoc schemes such as the padding used in PKCS #1 v1, OAEP has been proven secure under the random oracle model.

9.9.3 signing messages

Suppose Alice uses Bob's public key to send him an encrypted message. In the message, she can claim to be Alice but Bob has no way of verifying that the message was actually from Alice since anyone can use Bob's public key to send him encrypted messages. So, in order to verify the origin of a message, RSA can also be used to sign a message.

Suppose Alice wishes to send a signed message to Bob. She can use her own private key to do so. She produces a hash value of the message, raises it to the power of d mod n (as she does when decrypting a message), and attaches it as a "signature" to the message. When Bob receives the signed message, he uses the same hash algorithm in conjunction with Alice's public key. He raises the signature to the power of e mod n (as he does when encrypting a message), and compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the message was in possession of Alice's secret key, and that the message has not been tampered with since.

Note that secure padding schemes such as RSA-PSS are as essential for the security of message signing as they are for message encryption, and that the same key should never be used for both encryption and signing purposes

9.10 practical attacks

Adaptive-chosen-ciphertext attacks were largely considered to be a theoretical concern until 1998, when Daniel Bleichenbacher of Bell Laboratories demonstrated a practical attack against systems using RSA encryption in concert with the PKCS #1 v1 encoding function, including a version of the Secure Socket Layer (SSL) protocol used by thousands of web servers at the time.

The Bleichenbacher attacks took advantage of flaws within the PKCS #1 function to gradually reveal the content of an RSA encrypted message. Doing this requires sending several million test ciphertexts to the decryption device (eg, SSL-equipped web server.) In practical terms, this means that an SSL session key can be exposed in a reasonable amount of time, perhaps a day or less.

9.10.1 timing attacks

Kocher described a new attack on RSA in 1995: if the attacker *Eve* knows *Alice's* hardware in sufficient detail and is able to measure the decryption times for several known ciphertexts, she can deduce the decryption key d quickly. This attack can also be applied against the RSA signature scheme. In 2003, Boneh and Brumley demonstrated a more practical attack capable of recovering RSA factorizations over a network connection (e.g., from a Secure Socket Layer (SSL)-enabled webserver). This attack takes advantage of information leaked by the Chinese remainder theorem optimization used by many RSA implementations.

One way to thwart these attacks is to ensure that the decryption operation takes a constant amount of time for every ciphertext. However, this approach can significantly reduce performance. Instead, most RSA implementations use an alternate technique known as cryptographic blinding. RSA blinding makes use of the multiplicative property of RSA. Instead of computing $c^d \mod n$, Alice first chooses a secret random value r and computes $(r^e c)^d \mod n$. The result of this computation is $r\, m \mod n$ and so the effect of r can be removed by multiplying by its inverse. A new value of r is chosen for each ciphertext. With blinding applied, the decryption time is no longer correlated to the value of the input ciphertext and so the timing attack fails.

9.10.2 adaptive chosen ciphertext attacks

In 1998, Daniel Bleichenbacher described the first practical adaptive chosen ciphertext attack, against RSA-encrypted messages using the PKCS #1 v1 padding scheme (a padding scheme randomizes and adds structure to an RSA-encrypted message, so it is possible to determine whether a decrypted message is valid.) Due to flaws with the PKCS #1 scheme, Bleichenbacher was able to mount a practical attack against RSA implementations of the Secure Socket Layer protocol, and to recover session keys. As a result of this work, cryptographers now recommend the use of provably secure padding schemes such as Optimal Asymmetric Encryption Padding, and RSA Laboratories has released new versions of PKCS #1 that are not vulnerable to these attacks.

9.10.3 branch prediction analysis (BPA) attacks

Many processors use a branch predictor to determine whether a conditional branch in the instruction flow of a program is likely to be taken or not. Usually these processors also implement simultaneous multithreading (SMT). Branch prediction analysis attacks use a spy process to discover (statistically) the private key when processed with these processors.

Simple Branch Prediction Analysis (SBPA) claims to improve BPA in a non-statistical way. In their paper, "On the Power of Simple Branch Prediction Analysis", the authors of SBPA (Onur Aciicmez and Cetin Kaya Koc) claim to have discovered 508 out of 512 bits of an RSA key in 10 iterations.

chapter 10 elliptic curve cryptography

The Elliptic Curve Cryptography (ECC) is a public-key cryptography method based on the study of elliptic curves over finite fields.

10.1 history

The algorithm was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT; the letters RSA are the initials of their surnames.

Clifford Cocks, a British mathematician working for the UK intelligence agency GCHQ, described an equivalent system in an internal document in 1973, but given the relatively expensive computers needed to implement it at the time, it was mostly considered a curiosity and, as far as is publicly known, was never deployed. His discovery, however, was not revealed until 1997 due to its top-secret classification, and Rivest, Shamir, and Adleman devised RSA independently of Cocks' work.

MIT was granted US patent 4405829 for a "Cryptographic communications system and method" that used the algorithm in 1983. The patent expired on 21 September 2000. Since a paper describing the algorithm had been published in August 1977, prior to the December 1977 filing date of the patent application, regulations in much of the rest of the world precluded patents elsewhere and only the US patent was granted. Had Cocks' work been publicly known, a patent in the US might not have been possible either.

10.2 finite fields

RSA involves a public key and a pr

10.3 elliptic curves over finite fields

RSA involves a public key and a pr

10.4 finite fields

RSA involves a public key and a pr

10.5 finite fields

RSA involves a public key and a pr

10.6 finite fields

RSA involves a public key and a pr

10.7 finite fields

RSA involves a public key and a pr

chapter 11 digital signature standard - DSS

11.1 DSA parameters

The DSA makes use of the following parameters:

1. p = a prime modulus, where 2L-1 < p < 2L for 512 £ L £ 1024 and L a multiple of 64
2. q = a prime divisor of p - 1, where 2159 < q < 2160
3. g = h(p-1)/q mod p, where h is any integer with 1 < h < p - 1 such that h(p-1)/q mod p > 1

(g has order q mod p)

4. x = a randomly or pseudorandomly generated integer with 0 < x < q
5. y = gx mod p
6. k = a randomly or pseudorandomly generated integer with 0 < k < q

The integers p, q, and g can be public and can be common to a group of users. A user's private and public keys are x and y, respectively. They are normally fixed for a period of time. Parameters x and k are used for signature generation only, and must be kept secret. Parameter k must be regenerated for each signature.

Parameters p and q shall be generated as specified in Appendix 2, or using other FIPS approved security methods. Parameters x and k shall be generated as specified in Appendix 3, or using other FIPS approved security methods.

11.2 DSA signature generation

The signature of a message M is the pair of numbers r and s computed according to the equations below:

r = (gk mod p) mod q and

s = (k-1(SHA-1(M) + xr)) mod q.

In the above, k-1 is the multiplicative inverse of k, mod q; i.e., (k-1 k) mod q = 1 and 0 < k-1 < q. The value of SHA-1(M) is a 160-bit string output by the Secure Hash Algorithm specified in FIPS 180-1.

For use in computing s, this string must be converted to an integer. The conversion rule is given in Appendix 2.2.

As an option, one may wish to check if r = 0 or s = 0. If either r = 0 or s = 0, a new value of k should be generated and the signature should be recalculated (it is extremely unlikely that r = 0 or s = 0 if signatures are generated properly).

The signature is transmitted along with the message to the verifier.

11.3 DSA signature verification

Prior to verifying the signature in a signed message, p, q and g plus the sender's public key and identity are made available to the verifier in an authenticated manner.

Let M', r', and s' be the received versions of M, r, and s, respectively, and let y be the public key of the signatory. To verify the signature, the verifier first checks to see that 0 < r' < q and 0 < s' < q; if either condition is violated the signature shall be rejected. If these two conditions are satisfied, the verifier computes w = (s')-1 mod q

u1 = ((SHA-1(M'))w) mod q

u2 = ((r')w) mod q

v = (((g)u1 (y)u2) mod p) mod q.

If v = r', then the signature is verified and the verifier can have high confidence that the received message was sent by the party holding the secret key x corresponding to y. For a proof that v = r' when M' = M, r' = r, and s' = s, see Appendix 1.

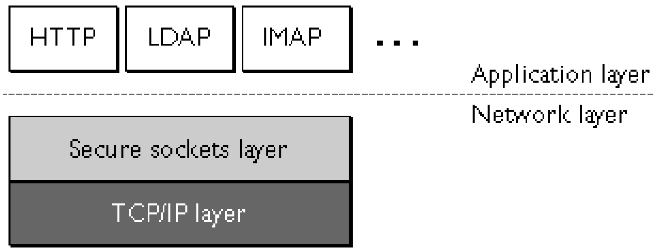
If v does not equal r', then the message may have been modified, the message may have been incorrectly signed by the signatory, or the message may have been signed by an impostor. The message should be considered invalid.

chapter 12 secure socket layer – SSL, TLS

12.1 SSL(Secure Sockets Layer)

- Protocol developed by Netscape for transmitting private documents via the Internet
- It uses a cryptographic system that uses 2 keys to encrypt data : a public key known to everyone and a private or secret key known only to the recipient of the message

12.2 the SSL protocol



The SSL protocol runs above TCP/IP and below higher-level protocols such as HTTP or IMAP. It uses TCP/IP on behalf of the higher-level protocols, and in the process allows an SSL-enabled server to authenticate itself to an SSL-enabled client, allows the client to authenticate itself to the server, and allows both machines to establish an encrypted connection.

- **SSL server authentication** allows a user to confirm a server's identity
- **SSL client authentication** allows a server to confirm a user's identity
- **An encrypted SSL connection** requires all information sent between a client and a server to be encrypted by the sending software and decrypted by the receiving software, thus providing a high degree of confidentiality

12.3 the SSL handshake

1. The client sends the server the client's SSL version number, cipher settings, randomly generated data, and other information the server needs to communicate with the client using SSL.
2. The server sends the client the server's SSL version number, cipher settings, randomly generated data, and other information the client needs to communicate with the server over SSL. The server also sends its own certificate and, if the client is requesting a server resource that requires client authentication, requests the client's certificate.
3. The client uses some of the information sent by the server to authenticate the server (see Server Authentication for details). If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client goes on to Step 4.
4. Using all data generated in the handshake so far, the client (with the cooperation of the server, depending on the cipher being used) creates the **premaster secret** for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in Step 2), and sends the

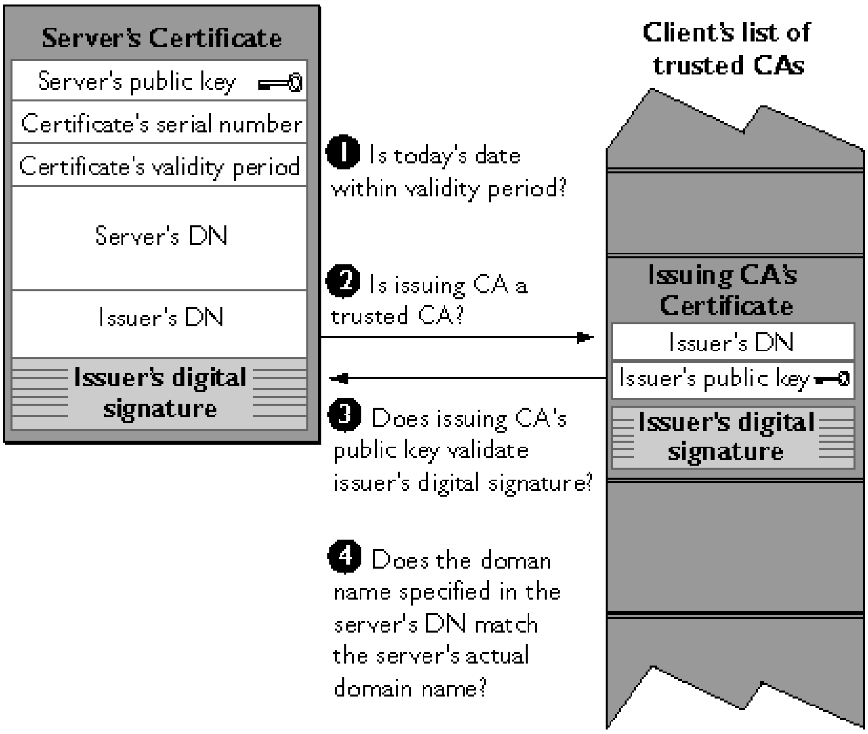
encrypted premaster secret to the server.

5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case the client sends both the signed data and the client's own certificate to the server along with the encrypted premaster secret.
6. If the server has requested client authentication, the server attempts to authenticate the client (see Client Authentication for details). If the client cannot be authenticated, the session is terminated. If the client can be successfully authenticated, the server uses its private key to decrypt the premaster secret, then performs a series of steps (which the client also performs, starting from the same premaster secret) to generate the **master secret.**
7. Both the client and the server use the master secret to generate the **session keys,** which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session and to verify its integrity--that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection.
8. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
9. The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.
10. The SSL handshake is now complete, and the SSL session has begun. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

12.4 server authentication

In the case of server authentication, the client encrypts the premaster secret with the server's public key. Only the corresponding private key can correctly decrypt the secret, so the client has some assurance that the identity associated with the public key is in fact the server with which the client is connected. Otherwise, the server cannot decrypt the premaster secret and cannot generate the symmetric keys required for the session, and the session will be terminated.

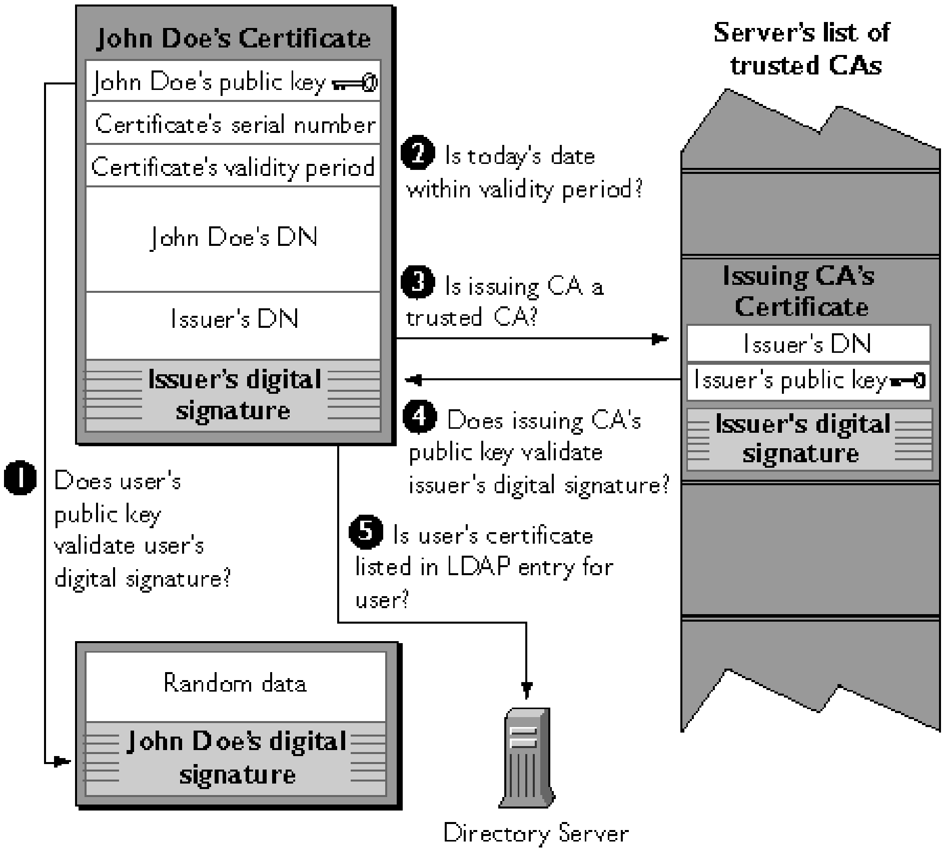
Example: How a Netscape client authenticates a server certificate



12.5 client authentication

In the case of client authentication, the client encrypts some random data with the client's private key--that is, it creates a digital signature. The public key in the client's certificate can correctly validate the digital signature only if the corresponding private key was used. Otherwise, the server cannot validate the digital signature and the session is terminated.

Example: How a Netscape client authenticates a server certificate



12.6 ciphers used with SSL

- **DES.** Data Encryption Standard, an encryption algorithm used by the U.S. Government.
- **DSA.** Digital Signature Algorithm, part of the digital authentication standard used by the U.S. Government.
- **KEA.** Key Exchange Algorithm, an algorithm used for key exchange by the U.S. Government.
- **MD5.** Message Digest algorithm developed by Rivest.
- **RC2 and RC4.** Rivest encryption ciphers developed for RSA Data Security.
- **RSA.** A public-key algorithm for both encryption and authentication. Developed by Rivest, Shamir, and Adleman.
- **RSA key exchange.** A key-exchange algorithm for SSL based on the RSA algorithm.
- **SHA-1.** Secure Hash Algorithm, a hash function used by the U.S. Government.
- **SKIPJACK.** A classified symmetric-key algorithm implemented in FORTEZZA-compliant hardware used by the U.S. Government.
- **TRIPLE DES.** DES applied three times

chapter 13 secure shell

13.1 history and development

13.1.1 version 1.0

In 1995, Tatu Ylönen, a researcher at Helsinki University of Technology, Finland, designed the first version of the protocol (now called **SSH-1**) prompted by a password-sniffing attack at his university network. The goal of SSH was to replace the earlier rlogin, TELNET and rsh protocols, which did not provide strong authentication or guarantee confidentiality. Ylönen released his implementation as freeware in July 1995, and the tool quickly gained in popularity. Towards the end of 1995, the SSH user base had grown to 20,000 users in fifty countries.

In December 1995, Ylönen founded SSH Communications Security to market and develop SSH. The original version of the SSH software used various pieces of free software, such as GNU libgmp, but later versions released by SSH Secure Communications evolved into increasingly proprietary software.

13.1.2 version 2.0

"Secsh" was the official Internet Engineering Task Force's (IETF) name for the IETF working group responsible for version 2 of the SSH protocol. In 1996, a revised version of the protocol, **SSH-2**, was adopted as a standard. This version is incompatible with SSH-1. SSH-2 features both security and feature improvements over SSH-1. Better security, for example, comes through Diffie-Hellman key exchange and strong integrity checking via message authentication codes. New features of SSH-2 include the ability to run any number of shell sessions over a single SSH connection.

13.2 OpenSSH

In 1999, developers wanting a free software version to be available went back to the older 1.2.12 release of the original SSH program, which was the last released under an open source license. Björn Grönvall's OSSH was subsequently developed from this codebase. Shortly thereafter, OpenBSD developers forked Grönvall's code and did extensive work on it, creating OpenSSH, which shipped with the 2.6 release of OpenBSD. From this version, a "portability" branch was formed to port OpenSSH to other operating systems.

It is estimated that, as of 2000, there were 2,000,000 users of SSH.

As of 2005, OpenSSH is the single most popular SSH implementation, coming by default in a large number of operating systems. OSSH meanwhile has become obsolete.

13.3 the SSH-2 internet standard

13.3.1 original publication

In 2006, the aforementioned SSH-2 protocol became a proposed Internet standard with the publication by the IETF "secsh" working group of RFCs. It was first published in January 2006.

- RFC 4250, The Secure Shell (SSH) Protocol Assigned Numbers
- RFC 4251, The Secure Shell (SSH) Protocol Architecture
- RFC 4252, The Secure Shell (SSH) Authentication Protocol
- RFC 4253, The Secure Shell (SSH) Transport Layer Protocol
- RFC 4254, The Secure Shell (SSH) Connection Protocol
- RFC 4255, Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints
- RFC 4256, Generic Message Exchange Authentication for the Secure Shell Protocol (SSH)

- RFC 4335, The Secure Shell (SSH) Session Channel Break Extension
- RFC 4344, The Secure Shell (SSH) Transport Layer Encryption Modes
- RFC 4345, Improved Arcfour Modes for the Secure Shell (SSH) Transport Layer Protocol

13.3.2 later modifications

It was later modified and expanded by the following publications.

- RFC 4419, Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol (March 2006)
- RFC 4432, RSA Key Exchange for the Secure Shell (SSH) Transport Layer Protocol (March 2006)
- RFC 4716, The Secure Shell (SSH) Public Key File Format (Nov 2006)

13.4 uses

Example of tunneling an X11 application over SSH: the user 'josh' has SSHed from the local machine 'foofighter' to the remote machine 'tengwar' to run xeyes.

Logging into OpenWrt via SSH using PuTTY running on Windows.

SSH is a protocol that can be used for many applications across many platforms including UNIX, Microsoft Windows, Apple Mac and Linux. Some of the applications below may require features that are only available or compatible with specific SSH clients or servers. For example, using the SSH protocol to implement a VPN is possible, but presently only with the OpenSSH server and client implementation.

- For login to a shell on a remote host (replacing Telnet and rlogin)
- For executing a single command on a remote host (replacing rsh)
- For copying files from a local server to a remote host. See SCP, as an alternative for rcp
- In combination with SFTP, as a secure alternative to FTP file transfer
- In combination with rsync to backup, copy and mirror files efficiently and securely
- For port forwarding or tunneling a port (not to be confused with a VPN which routes packets between different networks or bridges two broadcast domains into one.).
- For using as a full-fledged encrypted VPN. Note that only OpenSSH server and client supports this feature.
- For forwarding X11 through multiple hosts
- For browsing the web through an encrypted proxy connection with SSH clients that support the SOCKS protocol.
- For securely mounting a directory on a remote server as a filesystem on a local computer using SSHFS.
- For automated remote monitoring and management of servers through one or more of the mechanisms as discussed above.
- For secure collaboration of multiple SSH shell channel users where session transfer, swap, sharing, and recovery of disconnected sessions is possible.

13.5 architecture

Diagram of the SSH-2 binary packet.

The SSH-2 protocol has an internal architecture (defined in RFC 4251) with well-separated layers. These are:

- The *transport* layer (RFC 4253). This layer handles initial key exchange and server authentication and sets up encryption, compression and integrity verification. It exposes to the upper layer an interface for sending and receiving plaintext packets of up to 32,768 bytes each (more can be allowed by the implementation). The transport layer also arranges for key re-exchange, usually after 1 GB of data has been transferred or after 1 hour has passed, whichever is sooner.
- The *user authentication* layer (RFC 4252). This layer handles client authentication and provides a number of authentication methods. Authentication is *client-driven*: when one is prompted for a

password, it may be the SSH client prompting, not the server. The server merely responds to client's authentication requests. Widely used user authentication methods include the following:

- *password*: a method for straightforward password authentication, including a facility allowing a password to be changed. This method is not implemented by all programs.
- *publickey*: a method for public key-based authentication, usually supporting at least DSA or RSA keypairs, with other implementations also supporting X.509 certificates.
- *keyboard-interactive* (RFC 4256): a versatile method where the server sends one or more prompts to enter information and the client displays them and sends back responses keyed-in by the user. Used to provide one-time password authentication such as S/Key or SecurID. Used by some OpenSSH configurations when PAM is the underlying host authentication provider to effectively provide password authentication, sometimes leading to inability to log in with a client that supports just the plain *password* authentication method.
- GSSAPI authentication methods which provide an extensible scheme to perform SSH authentication using external mechanisms such as Kerberos 5 or NTLM, providing single sign on capability to SSH sessions. These methods are usually implemented by commercial SSH implementations for use in organizations, though OpenSSH does have a working GSSAPI implementation.

- The *connection* layer (RFC 4254). This layer defines the concept of channels, channel requests and global requests using which SSH services are provided. A single SSH connection can host multiple channels simultaneously, each transferring data in both directions. Channel requests are used to relay out-of-band channel specific data, such as the changed size of a terminal window or the exit code of a server-side process. The SSH client requests a server-side port to be forwarded using a global request. Standard channel types include:
  - *shell* for terminal shells, SFTP and exec requests (including SCP transfers)
  - *direct-tcpip* for client-to-server forwarded connections
  - *forwarded-tcpip* for server-to-client forwarded connections
- The SSHFP DNS record (RFC 4255) provides the public host key fingerprints in order to aid in verifying the authenticity of the host.

This open architecture provides considerable flexibility, allowing SSH to be used for a variety of purposes beyond secure shell. The functionality of the transport layer alone is comparable to TLS; the user authentication layer is highly extensible with custom authentication methods; and the connection layer provides the ability to multiplex many secondary sessions into a single SSH connection, a feature comparable to BEEP and not available in TLS.

13.6 security issues

Since SSH-1 has inherent design flaws which make it vulnerable (e.g., man-in-the-middle attacks), it is now generally considered obsolete and should be avoided by explicitly disabling fallback to SSH-1. While most modern servers and clients support SSH-2, some organizations still use software with no support for SSH-2, and thus SSH-1 cannot always be avoided.

In all versions of SSH, it is important to verify unknown public keys before accepting them as valid. Accepting an attacker's public key as a valid public key has the effect of disclosing the transmitted password and allowing man-in-the-middle attacks.

chapter 14 data security

Data security is the means of ensuring that data is kept safe from corruption and that access to it is suitably controlled. Thus data security helps to ensure privacy. It also helps in protecting personal data.

In the UK, the Data Protection Act is used to ensure that personal data is accessible to those whom it concerns, and provides redress to individuals if there are inaccuracies. This is particularly important to ensure individuals are treated fairly, for example for credit checking purposes. The Data Protection Act states that only individuals and companies with legitimate and lawful reasons can process personal information and cannot be shared.

The International Standard ISO/IEC 17799 covers data security under the topic of information security, and one of its cardinal principles is that all stored information, i.e. data, should be owned so that it is clear whose responsibility it is to protect and control access to that data.

14.1 data corruption

Data corruption refers to errors in computer data that occur during transmission or retrieval, introducing unintended changes to the original data. Computer storage and transmission systems use a number of measures to provide data integrity, the lack of errors.

Data corruption during transmission has a variety of causes. Interruption of data transmission causes information loss. Environmental conditions can interfere with data transmission, especially when dealing with wireless transmission methods. Heavy clouds can block satellite transmissions. Wireless networks are susceptible to interference from devices such as microwave ovens.

Data loss during storage has two broad causes: hardware and software failure. Head crashes and general wear and tear of media fall into the former category, while software failure typically occurs due to bugs in the code.

When data corruption behaves as a Poisson process, where each bit of data has an independently low probability of being changed, data corruption can generally be detected by the use of checksums, and can often be corrected by the use of error correcting codes.

If an uncorrectable data corruption is detected, procedures such as automatic retransmission or restoration from backups can be applied. RAID disk arrays, store and evaluate parity bits for data across a set of hard disks and can reconstruct corrupted data upon of the failure of a single disk.

If appropriate mechanisms are employed to detect and remedy data corruption, data integrity can be maintained. This is particularly important in banking, where an undetected error can drastically affect an account balance, and in the use of encrypted or compressed data, where a small error can make an extensive dataset unusable.

14.2 data privacy

Data privacy is the relationship between collection and dissemination of data, technology, the public expectation of privacy, and the legal issues surrounding them.

Privacy concerns exist wherever personally identifiable information is collected and stored - in digital form or otherwise. Improper or non-existent disclosure control can be the root cause for privacy issues. Data privacy issues can arise in response to information from a wide range of sources, such as:

- Healthcare records
- Criminal justice investigations and proceedings
- Financial institutions and transactions
- Biological traits, such as genetic material

- Residence and geographic records
- Ethnicity

The challenge in data privacy is to share data while protecting personally identifiable information. The fields of data security and information security design and utilize software, hardware and human resources to address this issue.

14.3 data remanence

Data remanence is the residual representation of data that has been in some way nominally erased or removed. This residue may be due to data being left intact by a nominal delete operation, or through physical properties of the storage medium. Data remanence may make inadvertent disclosure of sensitive information possible, should the storage media be released into an uncontrolled environment (e.g., thrown in the trash, or given to a third-party).

Over time, various techniques have been developed to counter data remanence. Depending on the effectiveness and intent, they are often classified as either clearing or purging/sanitizing. Specific methods include overwriting, degaussing, encryption, and physical destruction.

14.4 data spill

Data spill may include incidents such as theft or loss of digital media such as computer tapes, hard drives, or laptop computers containing such media upon which such information is stored unencrypted, posting such information on the Worldwide web or on a computer otherwise accessible from the Internet without proper information security precautions, transfer of such information to a system which is not completely open but is not appropriately or formally accredited for security at the approved level, such as unencrypted e-mail, or transfer of such information to the Information systems of a possibly hostile agency, such as a competing corporation or a foreign nation, where it may be exposed to more intensive decryption techniques.

14.5 data theft

Data theft is a growing problem primarily perpetrated by office workers with access to technology such as desktop computers and hand-held devices capable of storing digital information such as flash drives, iPods and even digital cameras. Since employees often spend a considerable amount of time developing contacts and confidential and copyrighted information for the company they work for they often feel they have some right to the information and are inclined to copy and/or delete part of it when they leave the company, or misuse it while they are still in employment.

While most organizations have implemented firewalls and intrusion-detection systems very few take into account the threat from the average employee that copies proprietary data for personal gain or use by another company. A common scenario is where a sales person makes a copy of the contact database for use in their next job. Typically this is a clear violation of their terms of employment.

The damage caused by data theft can be considerable with today's ability to transmit very large files via e-mail, web pages, USB devices, DVD storage and other hand-held devices. Removable media devices are getting smaller with increased hard drive capacity, and activities such as podslurping are becoming more and more common. It is now possible to store 80 GB of data on a device that will fit in an employee's pocket, data that could contribute to the downfall of a business.

14.6 separation of protection and security

In Computer sciences the **Separation of protection and security** is a design choice. Wulf et al identified protection as a mechanism and security as a policy, therefore making the protection-security distinction as a particular case of the mechanism-policy distinction principle.

The adoption of this distinction in a computer architecture, usually means that protection is provided as a fault tolerance mechanism by hardware/firmware and kernel, supporting the operating system and applications running on top in implementing their security policies. In this design, security policies rely therefore on the protection mechanisms and on additional cryptography techniques.

The two major hardware approaches for security and/or protection are Hierarchical protection domains (ring architectures with "supervisor mode" and "user mode"), and capability-based addressing.[4] The first approach adopts a policy already at the lower architecture levels (hw/firmware/kernel), restricting the rest of the system to rely on it; therefore, the choice to distinguish between protection and security in the overall architecture design leads to the rejection of the hierarchical approach in favour of capability-based addressing.

The Bell-LaPadula model is an example of a model where protection and security are not separated. In Landwehr 1981 there's a table showing which models for computer security separates protection mechanism and security policy. Those with the separation are: access matrix, UCLA Data Secure Unix, take-grant and filter; those without are: high-water mark, Bell and LaPadula (original and revisited), information flow, strong dependency and constraints.

chapter 15 data access

15.1 access control lists

15.2 file system ACLs

For file systems, the access control lists are operating system specific data structures which specify individual or group rights to certain objects, like files, directories or processes.

15.2.1 the NTFS example

NTFS – New Technology File System is the standard file system for MS based operating systems, starting with Windows NT. Besides support for ACLs, NTFS offers file system journaling, as well.

15.3 network ACLs

15.4 passwords

15.5 password efficiency

15.6 password storing

15.7 passwords over the network

15.8 password breaking

chapter 16 network security

chapter 17 viruses

chapter 18 trojans

18.1 definition

A Trojan horse (sometimes shortened to trojan), is non-self-replicating malware that appears to perform a desirable function for the user but instead **facilitates unauthorized access** to the user's computer system. The term is derived from the Trojan Horse story in Greek mythology.

An apparently innocent program designed to circumvent the security features of a system. The usual method of introducing a Trojan horse is by donating a program, or part of a program, to a user of the system whose security is to be breached. The donated code will ostensibly perform a useful function; the recipient will be unaware that the code has other effects, such as writing a copy of his or her username and password into a file whose existence is known only to the donor, and from which the donor will subsequently collect whatever data has been written.

18.2 purpose and operation

Trojan horses are designed to allow a hacker remote access to a target computer system. Once a Trojan horse has been installed on a target computer system, it is possible for a hacker to access it remotely and perform various operations. The operations that a hacker can perform are limited by user privileges on the target computer system and the design of the Trojan horse.

Operations that could be performed by a hacker on a target computer system include:

- Use of the machine as part of a botnet (i.e. to perform spamming or to - perform Distributed Denial-of-service (DDoS) attacks)
- Data theft (e.g. passwords, credit card information, etc.)
- Installation of software (including other malware)
- Downloading or uploading of files
- Modification or deletion of files
- Keystroke logging
- Viewing the user's screen
- Wasting computer storage space

Trojan horses require interaction with a hacker to fulfill their purpose, though the hacker need not be the individual responsible for distributing the Trojan horse. In fact, it is possible for hackers to scan computers on a network using a port scanner in the hope of finding one with a Trojan horse installed, that the hacker can then use to control the target computer.

18.3 installation and distribution

Trojan horses can be installed through the following methods:

1. **Software downloads** (i.e., a Trojan horse included as part of a software application downloaded from a file sharing network)
2. **Websites containing executable content** (i.e., a Trojan horse in the form of an ActiveX control)
3. **Email attachments**
4. **Application exploits** (i.e., flaws in a web browser, media player, messaging client, or other software that can be exploited to allow installation of a Trojan horse). Also, there have been reports of

compilers that are themselves Trojan horses. While compiling code to executable form, they include code that causes the output executable to become a Trojan horse.

Users can be tricked into installing Trojan horses by being enticed or frightened. For example, a Trojan horse might arrive in email described as a computer game. When the user receives the mail, they may be enticed by the description of the game to install it. Although it may in fact be a game, it may also be taking other action that is not readily apparent to the user, such as deleting files or mailing sensitive information to the attacker. As another example, an intruder may forge an advisory from a security organization, such as the CERT Coordination Center, that instructs system administrators to obtain and install a patch.

Other forms of *"social engineering"* can be used to trick users into installing or running Trojan horses. For example, an intruder might telephone a system administrator and pose as a legitimate user of the system who needs assistance of some kind. The system administrator might then be tricked into running a program of the intruder's design.

Software distribution sites can be compromised by intruders who replace legitimate versions of software with Trojan horse versions. If the distribution site is a central distribution site whose contents are mirrored by other distribution sites, the Trojan horse may be downloaded by many sites and spread quickly throughout the Internet community.

Because the Domain Name System (DNS) does not provide strong authentication, users may be tricked into connecting to sites different than the ones they intend to connect to. This could be exploited by an intruder to cause users to download a Trojan horse, or to cause users to expose confidential information.

Intruders may install Trojan horse versions of system utilities after they have compromised a system. Often, collections of Trojan horses are distributed in toolkits that an intruder can use to compromise a system and conceal their activity after the compromise, e.g., a toolkit might include a Trojan horse version of ls which does not list files owned by the intruder. Once an intruder has gained administrative access to your systems, it is very difficult to establish trust in it again without rebuilding the system from known-good software. A Trojan horse may be inserted into a program by a compiler that is itself a Trojan horse.

Finally, a Trojan horse may simply be placed on a web site to which the intruder entices victims. The Trojan horse may be in the form of a Java applet, JavaScript, ActiveX control, or other form of executable content.

18.4 removal

Antivirus software is designed to detect and delete Trojan horses, as well as preventing them from ever being installed. Although it is possible to remove a Trojan horse manually, it requires a full understanding of how that particular Trojan horse operates. In addition, if a Trojan horse has possibly been used by a hacker to access a computer system, it will be difficult to know what damage has been done and what other problems have been introduced. In situations where the security of the computer system is critical, it is advisable to simply erase all data from the hard disk and reinstall the operating system and required software.

18.5 current use

Due to the growing popularity of botnets among hackers, Trojan horses are becoming more common. According to a survey conducted by BitDefender from January to June 2009, "Trojan-type malware is on the rise, accounting for 83-percent of the global malware detected in the wild".

Trojan horses can be particularly effective when offered to systems staff who can run code in highly privileged modes. Two remedies are effective: no code should be run unless its provenance is absolutely certain; no code should be run with a higher level of privilege than is absolutely essential.

18.6 solutions

The best advice with respect to Trojan horses is to ***avoid them*** in the first place. System administrators (including the users of single-user systems) should take care to verify that every piece of software that is installed is from a trusted source and has not been modified in transit. When digital signatures are provided, users are encouraged to validate the signature (as well as validating the public key of the signer). When digital signatures are not available, you may wish to acquire software on tangible media such as CDs, which bear the manufacturer's logo. Of course, this is not foolproof either. Without a way to authenticate software, you may not be able to tell if a given piece of software is legitimate, regardless of the distribution media. Software developers and software distributors are strongly encouraged to use cryptographically strong validation for all software they produce or distribute. Any popular technique based on algorithms that are widely believed to be strong will provide users a strong tool to defeat Trojan horses. Anyone who invests trust in digital signatures must also take care to validate any public keys that may be associated with the signature. It is not enough for code merely to be signed -- it must be signed by a trusted source. Do not execute anything sent to you via unsolicited electronic mail. Use caution when executing content such as Java applets, JavaScript, or Active X controls from web pages. You may wish to configure your browser to disable the automatic execution of web page content. Apply the principle of least privilege in daily activity: do not retain or employ privileges that are not needed to accomplish a given task. For example, do not run with enhanced privilege, such as "root" or "administrator," ordinary tasks such as reading email. Install and configure a tool such as Tripwire® that will allow you to detect changes to system files in a cryptographically strong way.

Educate your users regarding the danger of Trojan horses. Use firewalls and virus products that are aware of popular Trojan horses. Although it is impossible to detect all possible Trojan horses using a firewall or virus product (because a Trojan horse can be arbitrary code), they may aid you in preventing many popular Trojan horses from affecting your systems.

Review the source code to any open source products you choose to install. Open source software has an advantage compared to proprietary software because the source code can be widely reviewed and any obvious Trojan horses will probably be discovered very quickly. However, open source software also tends to be developed by a wide variety of people with little or no central control. This makes it difficult to establish trust in a single entity. Keep in mind that reviewing source code may be impractical at best, and that some Trojan horses may not be evident.

Adopt the use of cryptographically strong mutual authentication systems, such as ssh, for terminal emulation, X.509 public key certificates in web servers, S/MIME or PGPfor electronic mail, and kerberos for a variety of services. Avoid the use of systems that trust the domain name system for authentication, such as telnet, ordinary http (as opposed to https), ftp, or smtp, unless your network is specifically designed to support that trust. Do not rely on timestamps, file sizes, or other file attributes when trying to determine if a file contains a Trojan horse.

Exercise caution when downloading unauthenticated software. If you choose to install software that has not been signed by a trusted source, you may wish to wait for a period of time before installing it in order to see if a Trojan horse is discovered.

chapter 19 software exploits

19.1 definition

"An exploit is a piece of software, a chunk of data, or sequence of commands that take advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized)".

This frequently includes things like violently gaining control of a computer system or allowing privilege escalation or a denial of service attack.

19.2 classification

There are several methods of classifying exploits.

1. The most common one is by how the exploit contacts the vulnerable software.

- A **'remote exploit'** works over a network and exploits the security vulnerability without any prior access to the vulnerable system.
- A **'local exploit'** requires prior access to the vulnerable system and usually increases the privileges of the person running the exploit past those granted by the system administrator.

2. Another classification is by the action against vulnerable system:

- unauthorized data access
- arbitrary code execution
- denial of service.

Many exploits are designed to provide superuser-level access to a computer system. However, it is also possible to use several exploits, first to gain low-level access, then to escalate privileges repeatedly until one reaches root.

Normally a single exploit can only take advantage of a specific software vulnerability. Often, when an exploit is published, the vulnerability is fixed through a patch and the exploit becomes obsolete for newer versions of the software. This is the reason why some blackhat hackers do not publish their exploits but keep them private to themselves or other malicious crackers.

19.3 types

Exploits are commonly categorized and named by these criteria:

1. The type of vulnerability they exploit.
2. Whether they need to be run on the same machine as the program that has the vulnerability (local) or can be run on one machine to attack a program running on another machine (remote).
3. The result of running the exploit

Software exploits are almost always designed to cause harm, but can also be harmless.

Some examples of damages are:

1. Erasing or overwriting data on a computer
2. Re-installing itself after being disabled
3. Encrypting files in a cryptoviral extortion attack
4. Corrupting files in a subtle way
5. Upload and download of files
6. Copying fake links, which lead to false websites, chats, or other account based websites, showing any local account name on the computer falsely engaging in untrue context
7. Falsifying records of downloading software, movies, or games from websites never visited by the victim.
8. Allowing remote access to the victim's computer. This is called a RAT (remote access trojan)
9. Spreading other malware, such as viruses (this type of trojan horse is called a 'dropper' or 'vector')
10. Setting up networks of zombie computers in order to launch DDoS attacks or send spam.
11. Spying on the user of a computer and covertly reporting data like browsing habits to other people (see the article on spyware)
12. Making screenshots
13. Logging keystrokes to steal information such as passwords and credit card numbers
14. Phishing for bank or other account details, which can be used for criminal activities
15. Installing a backdoor on a computer system
16. Opening and closing CD-ROM tray
17. Playing sounds, videos or displaying images
18. Calling using the modem to expensive numbers, thus causing massive phone bills
19. Harvesting e-mail addresses and using them for spam
20. Restarting the computer whenever the infected program is started
21. Deactivating or interfering with anti-virus and firewall programs
22. Deactivating or interfering with other competing forms of malware
23. Randomly shutting off the computer
24. Installing a virus
25. Slowing down your computer

19.4 attacking software dependencies

Applications rely heavily on their environment in order to work properly. They depend on the OS to provide resources like memory and disk space; they rely on the file system to read and write data; they use structures such as the Windows Registry to store and retrieve information; the list goes on and on. These resources all provide input to the software— not as overtly as the human user does—but input nonetheless. Like any input, if the software receives a value outside of its expected range, it can fail. Inducing failure scenarios can allow us to watch an application in its unintended environment and expose critical vulnerabilities.

19.4.1 block access to libraries.

Software depends on libraries from the operating system, third-party vendors, and components bundled with the application. The types of libraries to target depends on your application. Sometimes DLLs have obscure names that give little clue to what they're used for. Others can give you hints to what services they perform for the application. This attack is designed to ensure that the application under test does not behave insecurely if software libraries fail to load.

When environmental failures occur, application error handlers get executed. However, sometimes these situations are not considered during application design, and the result is the dreaded unhandled exception. Even if there is code to handle these types of errors, this code is a fertile breeding ground for bugs, because it is likely that it was subjected to far less testing than the rest of the application.

19.4.2 manipulate the application's registry values.

The Windows registry contains information crucial to the normal operation of the operating system and installed applications. For the OS, the registry keeps track of information like key file locations, directory structure, execution paths, and library version numbers. Applications rely on this and other information stored in the registry to work properly. However, not all information stored in the registry is secured from either users or other installed applications. This attack tests that applications do not store sensitive information in the registry or trust the registry to always behave predictably.

19.4.3 force the application to use corrupt files

Software can only do so much before it needs to store or read persistent data. Data is the fuel that drives an application, so sooner or later all applications will have to interact with the file system. Corrupt files or file names are like putting sugar in your car's gas tank; if you don't catch it before you start the car, the damage may be unavoidable. This attack determines if applications can handle bad data gracefully, without exposing sensitive information or allowing insecure behavior.

A large application may read from and write to hundreds of files in the process of carrying out its prescribed tasks. Every file that an application reads provides input; any of these files can be a potential point of failure and thus a good starting point for an attack. Particularly interesting files to check are those that are used exclusively by the application and not intended for the user to read or alter; they are files where it is least likely that appropriate checks on data integrity will be implemented.

19.4.4 manipulate and replace files that the application creates, reads from, writes to, or executes

Similar to Attack 3, this attack also involves file-system dependencies. In previous attacks, we were trying to get the application to process corrupt data. In this one, we manipulate data, executables, or libraries in ways that force the application to behave insecurely. This attack can be applied any time an application reads or writes to the file system, launches another executable, or accesses functionality from a library. The goal of this attack is to test whether the application allows us to do something we shouldn't be able to do.

19.4.5 force the application to operate in low memory, disk-space and network-availability conditions

An application is a set of instructions for computer hardware to execute. First, the computer will load the application into memory and then give the application additional memory in which to store and manipulate its internal data. Memory is only temporary, though; to really be useful, an application needs to store persistent data. That's where the file system comes in, and with it, the need for disk space. Without sufficient memory disk space, most applications will not be able to perform their intended function.

The objective of this attack is to deprive the application of any of these resources so testers can understand how robust and secure their application is under stress. The decision regarding which failure scenarios to try (and when) can only be determined on a case-by-case basis. A general rule of thumb is to block a resource when an application seems most in need of it.

19.5 attacking the user interface

The user interface is usually the most comfortable bug hunting ground for security testing. It's the way we are accustomed to interacting with our applications, and the way application developers expect us to. The attacks discussed here focus on inputs applied to software through its user interface. Most security bugs result from additional, unintended, and undocumented user behavior. From the UI, this amounts to handling unexpected input from the user in a way that compromises the application, the system it runs on, or its data. The result could be privilege escalation (a normal user acquiring administrative rights) or allowing secret information to be viewed by an unauthorized user.

19.5.1 overflow input buffers

Buffer overflows are by far the most notorious security problems in software. They occur when applications fail to properly constrain input length. Some buffer overflows don't present much of a security threat. Others, however, can allow hackers to take control of a system by sending a well-crafted string to the application. This second type is referred to by the industry as "exploitable," because parts of the string may get executed if they are interpreted as code. What sometimes happens is that a fixed amount of memory is allocated to hold user input. If developers then fail to constrain the length of the input strings entered by the user, data can overwrite application instructions, allowing the user to execute arbitrary code and gain access to the machine.

19.5.2 examine all common switches and options

Some applications are tolerant to varying user input under a default configuration. Most default configurations are chosen by the application developers, and most tests are executed under these conditions, especially if options are obscure or are entered using command-line switches. When these configurations are changed, the software is often forced to use code paths that may be severely under-tested and thus results can be unpredictable. Obviously, to test a wide range of inputs under every possible set of configurations is impossible for large applications; instead, this attack focuses on some of the more obscure configurations, such as those in which switches are set through the command line at startup.

19.5.3 explore escape characters, character sets, and commands

Some applications may treat certain characters as equivalent when they are part of a string. For most purposes, a string with the letter a in a certain position is not likely to be processed any differently from a similar string with the letter z in that same position. With this in mind, the question," Which characters or combinations of characters are treated differently?" naturally follows. This is the driving question behind this attack. By forcing the application to process special characters and commands, we can sometimes force it to behave in ways its designers did not intend. Factors that affect which characters and commands might be interpreted differently include the language the application was written in, the libraries that user data is passed through, and specific words and strings reserved by the underlying operating system.

chapter 20 the hide and seek game

chapter 21 internet specific threats

21.1 generalities

Viruses, hacker attacks and other cyber threats are now a part of daily life. Malware spreading throughout the Internet, hackers stealing confidential data and mailboxes flooded with spam are the price people pay for computing convenience. Any unprotected computer or network is vulnerable.

Home users can lose valuable personal data with one click to the wrong website. Children trading games also exchange viruses unknowingly. You receive an email requesting an update to your payment details, and a hacker gains access to your bank account. A backdoor is installed on your machine, and your PC becomes a zombie, spewing out spam.

The internet has become a critical resource people rely on to get their work done or for entertainment. They use the web to perform research and gather information. They use email and popular instant messaging tools to help them stay in touch with coworkers and customers. And uploading, downloading, and sharing document files and other work products are now everyday activities.

Unfortunately, when people perform these daily tasks, they expose the companies for which they work to serious security risks. Employers must now be concerned with more than simply preventing people from doing things on

the job that they should not be doing – visiting restricted or inappropriate websites, for example. Now people are being exposed to harmful, destructive threats while in the process of simply doing their jobs. Companies should examine their IT security measures and determine whether they are sufficient to protect against these web-borne threats.

Gateway firewalls and antivirus software alone cannot protect against the complex and varied malcode that threatens IT infrastructures. Firewalls can detect web traffic, but most have no means of monitoring the specific information being transferred. Antivirus solutions are reactive, not preventive; they are effective only against very specific threats, and they provide this limited protection only after an attack has already occurred. Organizations need to supplement their existing security systems with a solution that complements these measures with content-level protection.

21.2 exposure to threats

The threats to which people are exposed daily, vary depending on what people are doing on the web. The illustration below summarizes some of these threats based on the task in which people are engaged.

21.2.1 internet access

While browsing on the web, people may unknowingly visit malicious websites – websites that have either been hacked into or designed specifically to distribute malware. When a user visits one of these sites, hackers can exercise control over the user's machine, download files, or install keyloggers or other malcode.

21.2.2 file sharing

When people share files using peer-to-peer networks, they often download spyware and malicious mobile code (MMC) along with the intended work product. Spyware gathers information about the user – often logging keystrokes, web surfing habits, passwords, and email addresses, and transfers that information back to the source site via port 80 back-channel communications. Malicious code can be delivered via web-borne viruses, Trojan horses, worms, or rogue internet code. The acquired MMC distributes itself using web pages or HTML code, including embedded ActiveX or Javascript code, and is embedded in the web pages.

21.2.3 instant messaging

Using instant messaging (IM) applications, people can “talk” and share files effortlessly. IM can help promote communication among team members and reduce the number of face-to-face meetings required. It can also be an invaluable e-commerce tool, with customer service reps supporting new customers by answering product questions, helping to finalize online transactions, and so on. Unfortunately, it can also be used to transmit proprietary company information in unencrypted format and transfer file attachments that completely bypass the existing security infrastructure.

In addition, many IM downloads are infested with viruses, Trojan horses, and worms. In fact, several worms have targeted specific IM clients, sending users IM phishing emails and using IM buddy lists to spread.

21.2.4 e-mail

Even sending and replying to emails can be a risky business. Email gives hackers an easy way to distribute harmful content. Email messages can include file attachments infected with viruses, worms, Trojan horses, or other malware.

Hackers send the infected files and hope that the recipient will open them. Other malicious emails use browser vulnerabilities to spread. One example is the Nimda worm, which ran automatically on computers with a vulnerable (unpatched) version of Internet Explorer or Outlook Express.

21.3 phishing

Phishing is another threat that capitalizes on the popularity of email as a communication tool. In many ploys, phishers send official-looking but phony emails to trick recipients into revealing confidential account or user information. Recipients are encouraged to click links in the emails, leading them to what appear to be customer service pages, complete with links, logos, and all the familiar layout and language of the authentic website. In fact, some fraudulent websites are so convincing that the users’ address bar shows they are connected to a legitimate banking or e-commerce site.

Phishers are considered hackers because they use social-engineering to trick and deceive their targets. For example, a phisher may send an e-mail using the façade of a major bank, credit card or E-money service like PayPal. The email will not only look official, but will also have an official-looking network domain name and return address. The body will contain an innocuous message such as: "Your account information requires updating".

The phisher’s assumption is that people will open the email, read it, and believe the contents. They hope the reader will click on the provided link because it looks official, and be directed to a site that looks exactly like the real thing (PayPal, etc.). In reality, the user has been directed to a mock site, and is about to enter confidential account information that will be recorded and sent back to the attacker. Phishing impacts businesses as well as consumers. Well-known, trusted banks and other online service providers are concerned that fears of identify theft and account-napping will stop consumers from making purchases and processing other financial transactions online. Visa International has joined the first worldwide aggregation service in an effort to combat phishing.

Phishing can also target confidential company information. By targeting people (sending an email to all people at a specific company supposedly from the IT department, for example), phishers may successfully gain access to corporate usernames and passwords. Using this information, hackers may be able to infiltrate and access the corporate network and, in turn, confidential corporate, customer, or user information, which can present not only legal liability issues, but also regulatory compliance problems.

21.4 pharming

Automated malware that lies in wait until a user connects to a target website (primarily banks and other online financial institutions and ecommerce sites) uses a new scheme called “pharming.” Like phishing, this ploy aims to steal confidential account information. Unlike phishing, however, this method does not rely on phony emails to lure unsuspecting victims; in fact, it is nearly undetectable. Pharming uses Trojan horse viruses that change the behavior of web browsers. User attempts to access an online banking site or one of the other target sites actually trigger the browser to redirect to a fraudulent site. Once a machine is infected, a user can type the correct URL and still end up at the fraudulent site.

21.5 hacked websites

Hackers can transform a website into a malicious one. When websites are hacked into, the sites themselves become attack vectors and are used to distribute malicious code. When a company’s web server is compromised, customers (or potential customers) are unwittingly infected with malicious code when they simply visit the site; these infections occur without the customer having to run any programs or open any attachments.

21.6 spoofed websites

Cyber criminals are capitalizing on consumer confidence in certain products and brands, and using this trust to trick users into divulging confidential account information. A typical scenario involves sending users a phishing email, asking them to click a link to update their account information. The HTML in the emails looks convincing and familiar. Many users readily comply with “their bank’s” request, providing sensitive account information at the linked-to websites – sites that appear valid, but are, in fact, fraudulent.

Whether or not users fall victim to these ploys, they are becoming wary and suspicious of any communications from ecommerce or banking sites, and are now less likely to engage in online transactions. These fears – although justified may be impacting global ecommerce.

chapter 22 smart cards

22.1 definition

"An exploit is a piece of software, a chunk of data, or sequence of commands that take advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized)".

This frequently includes things like violently gaining control of a computer system or allowing privilege escalation or a denial of service attack.

22.2 classification

There are several methods of classifying exploits.

1. The most common one is by how the exploit contacts the vulnerable software.

- A **'remote exploit'** works over a network and exploits the security vulnerability without any prior access to the vulnerable system.
- A **'local exploit'** requires prior access to the vulnerable system and usually increases the privileges

chapter 23 biometrics

23.1 definition

"An exploit is a piece of software, a chunk of data, or sequence of commands that take advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized)".

This frequently includes things like violently gaining control of a computer system or allowing privilege escalation or a denial of service attack.

23.2 classification

There are several methods of classifying exploits.

1. The most common one is by how the exploit contacts the vulnerable software.

- A **'remote exploit'** works over a network and exploits the security vulnerability without any prior access to the vulnerable system.
- A **'local exploit'** requires prior access to the vulnerable system and usually increases the privileges

chapter 24 crypto currencies

24.1 definition

"An exploit is a piece of software, a chunk of data, or sequence of commands that take advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized)".

This frequently includes things like violently gaining control of a computer system or allowing privilege escalation or a denial of service attack.

24.2 classification

There are several methods of classifying exploits.

1. The most common one is by how the exploit contacts the vulnerable software.

- A **'remote exploit'** works over a network and exploits the security vulnerability without any prior access to the vulnerable system.
- A **'local exploit'** requires prior access to the vulnerable system and usually increases the privileges
-

Biblography

[KSF] – the keccak sponge family functionhttp://keccak.noekeon.org/specs_summary.html