An Overview of Cryptography

Gary C. Kessler
30 June 2016

© 1998-2016 — A much shorter, edited version of this paper appears in the 1999 Edition of *Handbook on Local Area Networks*,
published by Auerbach in September 1998. Since that time, this paper has taken on a life of its own...

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

Does increased security provide comfort to paranoid people? Or does security provide some very basic protections that we are naive to believe that we don't need? During this time when the Internet provides essential communication between tens of millions of people and is being increasingly used as a tool for commerce, security becomes a tremendously important issue to deal with.

There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting passwords. One essential aspect for secure communications is that of cryptography. But it is important to note that while cryptography is *necessary* for secure communications, it is not by itself *sufficient*. The reader is advised, then, that the topics covered here only describe the first of many steps necessary for better security in any number of situations.

This paper has two major purposes. The first is to define some of the terms and concepts behind basic cryptographic methods, and to offer a way to compare the myriad cryptographic schemes in use today. The second is to provide some real examples of cryptography in use today.

I would like to say at the outset that this page is very focused on terms, concepts, and schemes in *current* use and is not a treatise of the whole field. No mention is made here about pre-computerized crypto schemes, the difference between a substitution and transposition cipher, cryptanalysis, or other history. Interested readers should check out some of the books in the references section below, a short list of my crypto URLs, or the Learn Cryptography page for detailed — and interesting! — background information.

2. THE PURPOSE OF CRYPTOGRAPHY

Cryptography is the science of writing in secret code and is an ancient art; the first documented use of cryptography in writing dates back to circa 1900 B.C. when an Egyptian scribe used non-standard hieroglyphs in an inscription. Some experts argue that cryptography appeared spontaneously sometime after writing was invented, with applications ranging from diplomatic missives to war-time battle plans. It is no surprise, then, that new forms of cryptography came soon after the widespread development of computer communications. In data and telecommunications, cryptography is necessary when communicating over any untrusted medium, which includes just about *any* network, particularly the Internet.

Within the context of any application-to-application communication, there are some specific security requirements, including:

- *Authentication:* The process of proving one's identity. (The primary forms of host-to-host authentication on the Internet today are name-based or address-based, both of which are notoriously weak.)
- *Privacy/confidentiality:* Ensuring that no one can read the message except the intended receiver.
- *Integrity:* Assuring the receiver that the received message has not been altered in any way from the original.
- *Non-repudiation:* A mechanism to prove that the sender really sent this message.

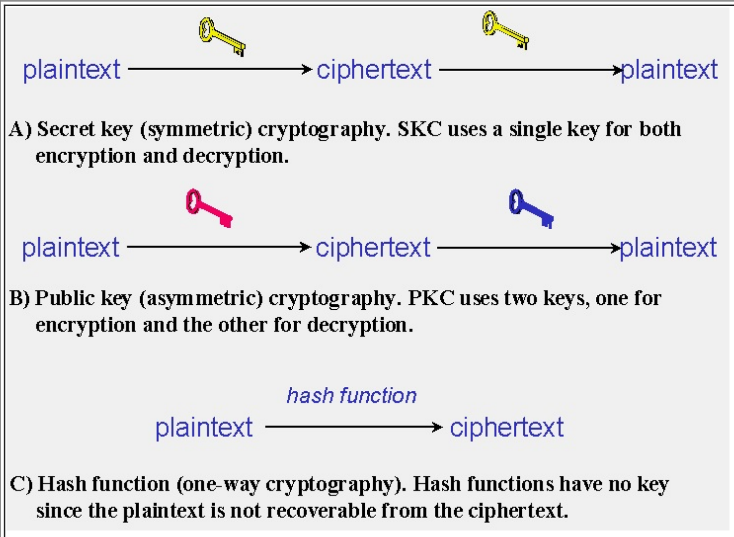
Cryptography, then, not only protects data from theft or alteration, but can also be used for user authentication. There are, in general, three types of cryptographic schemes typically used to accomplish these goals: secret key (or symmetric) cryptography, public-key (or asymmetric) cryptography, and hash functions, each of which is described below. In all cases, the initial unencrypted data is referred to as *plaintext*. It is encrypted into *ciphertext*, which will in turn (usually) be decrypted into usable plaintext.

In many of the descriptions below, two communicating parties will be referred to as Alice and Bob; this is the common nomenclature in the crypto field and literature to make it easier to identify the communicating parties. If there is a third or fourth party to the communication, they will be referred to as Carol and Dave. Mallory is a malicious party, Eve is an eavesdropper, and Trent is a trusted third party.

3. TYPES OF CRYPTOGRAPHIC ALGORITHMS

There are several ways of classifying cryptographic algorithms. For purposes of this paper, they will be categorized based on the number of keys that are employed for encryption and decryption, and further defined by their application and use. The three types of algorithms that will be discussed are (Figure 1):

- Secret Key Cryptography (SKC): Uses a single key for both encryption and decryption
- Public Key Cryptography (PKC): Uses one key for encryption and another for decryption
- Hash Functions: Uses a mathematical transformation to irreversibly "encrypt" information



A) Secret key (symmetric) cryptography. SKC uses a single key for both encryption and decryption.

B) Public key (asymmetric) cryptography. PKC uses two keys, one for encryption and the other for decryption.

C) Hash function (one-way cryptography). Hash functions have no key since the plaintext is not recoverable from the ciphertext.

FIGURE 1: Three types of cryptography: secret-key, public key, and hash function.

3.1. Secret Key Cryptography

With *secret key cryptography*, a single key is used for both encryption and decryption. As shown in Figure 1A, the sender uses the key (or some set of rules) to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key (or ruleset) to decrypt the message and recover the plaintext. Because a single key is used for both functions, secret key cryptography is also called *symmetric encryption*.

With this form of cryptography, it is obvious that the key must be known to both the sender and the receiver; that, in fact, is the secret. The biggest difficulty with this approach, of course, is the distribution of the key.

Secret key cryptography schemes are generally categorized as being either *stream ciphers* or *block ciphers*. Stream ciphers operate on a single bit (byte or computer word) at a time and implement some form of feedback mechanism so that the key is constantly changing. A block cipher is so-called because the scheme encrypts one block of data at a time using the same key on each block. In general, the same plaintext block will always encrypt to the same ciphertext when using the same key in a block cipher whereas the same plaintext will encrypt to different ciphertext in a stream cipher.

Stream ciphers come in several flavors but two are worth mentioning here. *Self-synchronizing stream ciphers* calculate each bit in the keystream as a function of the previous *n* bits in the keystream. It is termed "self-synchronizing" because the decryption process can stay synchronized with the encryption process merely by knowing how far into the *n*-bit keystream it is. One problem is error propagation; a garbled bit in transmission will result in *n* garbled bits at the receiving side. *Synchronous stream ciphers* generate the keystream in a fashion independent of the message stream but by using the same keystream generation function at sender and receiver. While stream ciphers do not propagate transmission errors, they are, by their nature, periodic so that the keystream will eventually repeat.

Block ciphers can operate in one of several modes; the following four are the most important:

- *Electronic Codebook (ECB) mode* is the simplest, most obvious application: the secret key is used to encrypt the plaintext block to form a ciphertext block. Two identical plaintext blocks, then, will always generate the same ciphertext block. Although this is the most common mode of block ciphers, it is susceptible to a variety of brute-force attacks.
- *Cipher Block Chaining (CBC) mode* adds a feedback mechanism to the encryption scheme. In CBC, the plaintext is exclusively-ORed (XORed) with the previous ciphertext block prior to encryption. In this mode, two identical blocks of plaintext never encrypt to the same ciphertext.
- *Cipher Feedback (CFB) mode* is a block cipher implementation as a self-synchronizing stream cipher. CFB mode allows data to be encrypted in units smaller than the block size, which might be useful in some applications such as encrypting interactive terminal input. If we were using 1-byte CFB mode, for example, each incoming character is placed into a shift register the same size as the block, encrypted, and the block transmitted. At the receiving side, the ciphertext is decrypted and the extra bits in the block (i.e., everything above and beyond the one byte) are discarded.
- *Output Feedback (OFB) mode* is a block cipher implementation conceptually similar to a synchronous stream cipher. OFB prevents the same plaintext block from generating the same ciphertext block by using an internal feedback mechanism that is independent of both the plaintext and ciphertext bitstreams.

A nice overview of these different modes can be found at CRYPTO-IT.

Secret key cryptography algorithms in use today — or, at least, important today even if not in use — include:

- *Data Encryption Standard (DES):* The most common SKC scheme used today, DES was designed by IBM in the 1970s and adopted by the National Bureau of Standards (NBS) [now the National Institute for Standards and Technology (NIST)] in 1977 for commercial and unclassified government applications. DES is a block-cipher employing a 56-bit key that operates on 64-bit blocks. DES has a complex set of rules and transformations that were designed specifically to yield fast hardware implementations and slow software implementations, although this latter point is becoming less significant today since the speed of computer processors is several orders of magnitude faster today than twenty years ago. IBM also proposed a 112-bit key for DES, which was rejected at the time by the government; the use of 112-bit keys was considered in the 1990s, however, conversion was never seriously considered.

  DES was defined in American National Standard X3.92 and three Federal Information Processing Standards (FIPS), all withdrawn in 2005:

  - FIPS 46-3: DES (Archived file)
  - FIPS 74: Guidelines for Implementing and Using the NBS Data Encryption Standard
  - FIPS 81: DES Modes of Operation

  Information about vulnerabilities of DES can be obtained from the Electronic Frontier Foundation.

  Two important variants that strengthen DES are:

  - *Triple-DES (3DES):* A variant of DES that employs up to three 56-bit keys and makes three encryption/decryption passes over the block; 3DES is also described in FIPS 46-3 and is the recommended replacement to DES.

  - *DESX:* A variant devised by Ron Rivest. By combining 64 additional key bits to the plaintext prior to encryption, effectively increases the keylength to 120 bits.

  More detail about DES, 3DES, and DESX can be found below in Section 5.4.

- *Advanced Encryption Standard (AES):* In 1997, NIST initiated a very public, 4-1/2 year process to develop a new secure cryptosystem for U.S. government applications. The result, the Advanced Encryption Standard, became the official successor to DES in December 2001. AES uses an SKC scheme called Rijndael, a block cipher designed by Belgian cryptographers Joan Daemen and Vincent Rijmen. The algorithm can use a variable block length and key length; the latest specification allowed any combination of keys lengths of 128, 192, or 256 bits and blocks of length 128, 192, or 256 bits. NIST initially selected Rijndael in October 2000 and formal adoption as the AES standard came in December 2001. FIPS PUB 197 describes a 128-bit block cipher employing a 128-, 192-, or 256-bit key. The AES process and Rijndael algorithm are described in more detail below in Section 5.9.

  > As an aside, the AES selection process managed by NIST was very public. A similar project, the New European Schemes for Signatures, Integrity and Encryption (NESSIE), was designed as an independent project meant to augment the work of NIST by putting out an open call for new cryptographic primitives. NESSIE ran from about 2000-2003. While several new algorithms were found during the NESSIE process, no new stream cipher survived cryptanalysis. As a result, the ECRYPT Stream Cipher Project (eSTREAM) was created, which has approved a number of new stream ciphers for both software and hardware implementation.
  >
  > Similar — but different — is the Japanese Government Cryptography Research and Evaluation Committees (CRYPTREC) efforts to evaluate algorithms submitted for government and industry applications. They, too, have approved a number of cipher suites for various applications.

- *CAST-128/256:* CAST-128, described in Request for Comments (RFC) 2144, is a DES-like substitution-permutation crypto algorithm, employing a 128-bit key operating on a 64-bit block. CAST-256 (RFC 2612) is an extension of CAST-128, using a 128-bit block size and a variable length (128, 160, 192, 224, or 256 bit) key. CAST is named for its developers, Carlisle Adams and Stafford Tavares, and is available internationally. CAST-256 was one of the Round 1 algorithms in the AES process.

- *International Data Encryption Algorithm (IDEA):* Secret-key cryptosystem written by Xuejia Lai and James Massey, in 1992 and patented by Ascom; a 64-bit SKC block cipher using a 128-bit key. Also available internationally.

- *Rivest Ciphers (*aka *Ron's Code):* Named for Ron Rivest, a series of SKC algorithms.

  - *RC1:* Designed on paper but never implemented.

  - *RC2:* A 64-bit block cipher using variable-sized keys designed to replace DES. It's code has not been made public although many companies have licensed RC2 for use in their products. Described in RFC 2268.

  - *RC3:* Found to be breakable during development.

  - *RC4:* A stream cipher using variable-sized keys; it is widely used in commercial cryptography products. An update to RC4, called Spritz (see also), was designed by Rivest and Jacob Schuldt. More detail about RC4 (and a little about Spritz) can be found below in Section 5.13.

  - *RC5:* A block-cipher supporting a variety of block sizes (32, 64, or 128 bits), key sizes, and number of encryption passes over the data. Described in RFC 2040.

  - *RC6:* A 128-bit block cipher based upon, and an improvement over, RC5; RC6 was one of the AES Round 2 algorithms.

- *Blowfish:* A symmetric 64-bit block cipher invented by Bruce Schneier; optimized for 32-bit processors with large data caches, it is significantly faster than DES on a Pentium/PowerPC-class machine. Key lengths can vary from 32 to 448 bits in length. Blowfish, available freely and intended as a substitute for DES or IDEA, is in use in a large number of products.

- *Twofish:* A 128-bit block cipher using 128-, 192-, or 256-bit keys. Designed to be highly secure and highly flexible, well-suited for large microprocessors, 8-bit smart card microprocessors, and dedicated hardware. Designed by a team led by Bruce Schneier and was one of the Round 2 algorithms in the AES process.

- *Camellia:* A secret-key, block-cipher crypto algorithm developed jointly by Nippon Telegraph and Telephone (NTT) Corp. and Mitsubishi Electric Corporation (MEC) in 2000. Camellia has some characteristics in common with AES: a 128-bit block size, support for 128-, 192-, and 256-bit key lengths, and suitability for both software and hardware implementations on common 32-bit processors as well as 8-bit processors (e.g., smart cards, cryptographic hardware, and embedded systems). Also described in RFC 3713. Camellia's application in IPsec is described in RFC 4312 and application in OpenPGP in RFC 5581.

- *MISTY1:* Developed at Mitsubishi Electric Corp., a block cipher using a 128-bit key and 64-bit blocks, and a variable number of rounds. Designed for hardware and software implementations, and is resistant to differential and linear cryptanalysis. Described in RFC 2994.

- *Secure and Fast Encryption Routine (SAFER):* Secret-key crypto scheme designed for implementation in software. Versions have been defined for 40-, 64-, and 128-bit keys.

- *KASUMI:* A block cipher using a 128-bit key that is part of the Third-Generation Partnership Project (3gpp), formerly known as the Universal Mobile Telecommunications System (UMTS). KASUMI is the intended confidentiality and integrity algorithm for both message content and signaling data for emerging mobile communications systems.

- *SEED:* A block cipher using 128-bit blocks and 128-bit keys. Developed by the Korea Information Security Agency (KISA) and adopted as a national standard encryption algorithm in South Korea. Also described in RFC 4269.

- *ARIA:* A 128-bit block cipher employing 128-, 192-, and 256-bit keys. Developed by large group of researchers from academic institutions, research institutes, and federal agencies in South Korea in 2003, and subsequently named a national standard. Described in RFC 5794.

- *CLEFIA:* Described in RFC 6114, CLEFIA is a 128-bit block cipher employing key lengths of 128, 192, and 256 bits (which is compatible with AES). The CLEFIA algorithm was first published in 2007 by Sony Corporation. CLEFIA is one of the new-generation lightweight blockcipher algorithms designed after AES, offering high performance in software and hardware as well as a lightweight implementation in hardware.

- *SMS4:* SMS4 is a 128-bit block cipher using 128-bit keys and 32 rounds to process a block. Declassified in 2006, SMS4 is used in the Chinese National Standard for Wireless Local Area Network (LAN) Authentication and Privacy Infrastructure (WAPI). SMS4 had been a proposed cipher for the Institute of Electrical and Electronics Engineers (IEEE) 802.11i standard on security mechanisms for wireless LANs, but has yet to be accepted by the IEEE or International Organization for Standardization (ISO). SMS4 is described in *SMS4 Encryption Algorithm for Wireless Networks* (translated and typeset by Whitfield Diffie and George Ledin, 2008) or in the original Chinese.

- *Skipjack:* SKC scheme proposed, along with the Clipper chip, as part of the never-implemented Capstone project. Although the details of the algorithm were never made public, Skipjack was a block cipher using an 80-bit key and 32 iteration cycles per 64-bit block. Capstone, proposed by NIST and the NSA as a standard for public and government use, met with great resistance by the crypto community laregly because the design of Skipjack was classified (coupled with the key escrow requirement of the Clipper chip).

- *GSM (Global System for Mobile Communications, originally Groupe Spécial Mobile) encryption:* GSM mobile phone systems use several stream ciphers for over-the-air communication privacy. A5/1 was developed in 1987 for use in Europe and the U.S. A5/2, developed in 1989, is a weaker algorithm and intended for use outside of Europe and the U.S. Significant flaws were found in both ciphers after the "secret" specifications were leaked in 1994, however, and A5/2 has been withdrawn from use. The newest version, A5/3, employs the KASUMI block cipher. **NOTE:** Unfortunately, although A5/1 has been repeatedly "broken" (e.g., see "Secret code protecting cellphone calls set loose" [2009] and "Cellphone snooping now easier and cheaper than ever" [2011]), this encryption scheme remains in widespread use, even in 3G and 4G mobile phone networks. Use of this scheme is reportedly one of the reasons that the National Security Agency (NSA) can easily decode voice and data calls over mobile phone networks.

- *GPRS (General Packet Radio Service) encryption:* GSM mobile phone systems use GPRS for data applications, and GPRS uses a number of encryption methods, offering different levels of data protection. GEA/0 offers no encryption at all. GEA/1 and GEA/2 are proprietary stream ciphers, employing a 64-bit key and a 96-bit or 128-bit state, respectively. GEA/1 and GEA/2 are most widely used by network service providers today although both have been reportedly broken. GEA/3 is a 128-bit block cipher employing a 64-bit key that is used by some carriers; GEA/4 is a 128-bit clock cipher with a 128-bit key, but is not yet deployed.

- *KCipher-2:* Described in RFC 7008, KCipher-2 is a stream cipher with a 128-bit key and a 128-bit initialization vector. Using simple arithmetic operations, the algorithms offers fast encryption and decryption by use of efficient implementations. KCipher-2 has been used for industrial applications, especially for mobile health monitoring and diagnostic services in Japan.

- *Salsa* and *ChaCha*: Salsa20 is a stream cipher proposed for the eSTREAM project by Daniel Bernstein. Salsa20 uses a pseudorandom function based on 32-bit (whole word) addition, bitwise addition (XOR), and rotation operations, aka *add-rotate-xor (ARX)* operations. Salsa20 uses a 256-bit key although a 128-bit key variant also exists. In 2008, Bernstein published ChaCha, a new family of ciphers related to Salsa20. ChaCha20, defined in RFC 7539, is employed (with the Poly1305 authenticator) in IETF protocols, most notably for IPsec and Internet Key Exchange (IKE), per RFC 7634, and Transaction Layer Security (TLS), per RFC 7905. In 2014, Google adopted ChaCha20/Poly1305 for use in OpenSSL, and they are also a part of OpenSSH.

There are several other references that describe interesting algorithms and even SKC codes dating back decades. Two that leap to mind are the Crypto Museum's Crypto List and John J.G. Savard's (albeit old) A Cryptographic Compendium page.

3.2. Public-Key Cryptography

Public-key cryptography has been said to be the most significant new development in cryptography in the last 300-400 years. Modern PKC was first described publicly by Stanford University professor Martin Hellman and graduate student Whitfield Diffie in 1976. Their paper described a two-key crypto system in which two parties could engage in a secure communication over a non-secure communications channel without having to share a secret key.

PKC depends upon the existence of so-called *one-way functions*, or mathematical functions that are easy to compute whereas their inverse function is relatively difficult to compute. Let me give you two simple examples:

1. *Multiplication vs. factorization:* Suppose you have two prime numbers, 3 and 7, and you need to calculate the product; it should take almost no time to calculate that value, which is 21. Now suppose, instead, that you have a number that is a product of two primes, 21, and you need to determine those prime factors. You will eventually come up with the solution but whereas calculating the product took milliseconds, factoring will take longer. The problem becomes much harder if we start with primes that have 400 digits or so, because the product will have ~800 digits.
2. *Exponentiation vs. logarithms:* Suppose you take the number 3 to the 6th power; again, it is relatively easy to calculate $3^6 = 729$. But if if you start with the number 729 and need to determine the two integers, *x* and *y* so that $\log_x 729 = y$, it will take longer to find the two values.

While the examples above are trivial, they do represent two of the functional pairs that are used with PKC; namely, the ease of multiplication and exponentiation versus the relative difficulty of factoring and calculating logarithms, respectively. The mathematical "trick" in PKC is to find a *trap door* in the one-way function so that the inverse calculation becomes easy given knowledge of some item of information.

Generic PKC employs two keys that are mathematically related although knowledge of one key does not allow someone to easily determine the other key. One key is used to encrypt the plaintext and the other key is used to decrypt the ciphertext. The important point here is that it **does not matter which key is applied first**, but that both keys are required for the process to work (Figure 1B). Because a pair of keys are required, this approach is also called *asymmetric cryptography*.

In PKC, one of the keys is designated the *public key* and may be advertised as widely as the owner wants. The other key is designated the *private key* and is never revealed to another party. It is straight forward to send messages under this scheme. Suppose Alice wants to send Bob a message. Alice encrypts some information using Bob's public key; Bob decrypts the ciphertext using his private key. This method could be also used to prove who sent a message; Alice, for example, could encrypt some plaintext with her private key; when Bob decrypts using Alice's public key, he knows that Alice sent the message and Alice cannot deny having sent the message (*non-repudiation*).

Public-key cryptography algorithms that are in use today for key exchange or digital signatures include:

- *RSA:* The first, and still most common, PKC implementation, named for the three MIT mathematicians who developed it — Ronald Rivest, Adi Shamir, and Leonard Adleman. RSA today is used in hundreds of software products and can be used for key exchange, digital signatures, or encryption of small blocks of data. RSA uses a variable size encryption block and a variable size key. The key-pair is derived from a very large number, *n*, that is the product of two prime numbers chosen according to special rules; these primes may be 100 or more digits in length each, yielding an *n* with roughly twice as many digits as the prime factors. The public key information includes *n* and a derivative of one of the factors of *n*; an attacker cannot determine the prime factors of *n* (and, therefore, the private key) from this information alone and that is what makes the RSA algorithm so secure. (Some descriptions of PKC erroneously state that RSA's safety is due to the difficulty in *factoring* large prime numbers. In fact, large prime numbers, like small prime numbers, only have two factors!) The ability for computers to factor large numbers, and therefore attack schemes such as RSA, is rapidly improving and systems today can find the prime factors of numbers with more than 200 digits. Nevertheless, if a large number is created from two prime factors that are roughly the same size, there is no known factorization algorithm that will solve the problem in a reasonable amount of time; a 2005 test to factor a 200-digit number took 1.5 years and over 50 years of compute time (see the Wikipedia article on integer factorization.) Regardless, one presumed protection of RSA is that users can easily increase the key size to always stay ahead of the computer processing curve. As an aside, the patent for RSA expired in September 2000 which does not appear to have affected RSA's popularity one way or the other. A detailed example of RSA is presented below in Section 5.3.

- *Diffie-Hellman:* After the RSA algorithm was published, Diffie and Hellman came up with their own algorithm. D-H is used for secret-key key exchange only, and not for authentication or digital signatures. More detail about Diffie-Hellman can be found below in Section 5.2.

- *Digital Signature Algorithm (DSA):* The algorithm specified in NIST's Digital Signature Standard (DSS), provides digital signature capability for the authentication of messages. Described in FIPS 186-4.

- *ElGamal:* Designed by Taher Elgamal, a PKC system similar to Diffie-Hellman and used for key exchange.

- *Elliptic Curve Cryptography (ECC):* A PKC algorithm based upon elliptic curves. ECC can offer levels of security with small keys comparable to RSA and other PKC methods. It was designed for devices with limited compute power and/or memory, such as smartcards and PDAs. More detail about ECC can be found below in Section 5.8. Other references include the Elliptic Curve Cryptography page and the Online ECC Tutorial page, both from Certicom. See also RFC 6090 for a review of fundamental ECC algorithms and *The Elliptic Curve Digital Signature Algorithm (ECDSA)* for details about the use of ECC for digital signatures.

- *Public-Key Cryptography Standards (PKCS):* A set of interoperable standards and guidelines for public-key cryptography, designed by RSA Data Security Inc.

  - PKCS #1: RSA Cryptography Standard (Also RFC 3447)
  - PKCS #2: *Incorporated into PKCS #1.*
  - PKCS #3: Diffie-Hellman Key-Agreement Standard
  - PKCS #4: *Incorporated into PKCS #1.*
  - PKCS #5: Password-Based Cryptography Standard (PKCS #5 V2.0 is also RFC 2898)
  - PKCS #6: Extended-Certificate Syntax Standard (being phased out in favor of X.509v3)
  - PKCS #7: Cryptographic Message Syntax Standard (Also RFC 2315)

    - PKCS #8: Private-Key Information Syntax Standard (Also RFC 5208)
    - PKCS #9: Selected Attribute Types (Also RFC 2985)
    - PKCS #10: Certification Request Syntax Standard (Also RFC 2986)
    - PKCS #11: Cryptographic Token Interface Standard
    - PKCS #12: Personal Information Exchange Syntax Standard (Also RFC 7292)
    - PKCS #13: Elliptic Curve Cryptography Standard
    - PKCS #14: *Pseudorandom Number Generation Standard is no longer available*
    - PKCS #15: Cryptographic Token Information Format Standard

- *Cramer-Shoup:* A public-key cryptosystem proposed by R. Cramer and V. Shoup of IBM in 1998.

- *Key Exchange Algorithm (KEA):* A variation on Diffie-Hellman; proposed as the key exchange method for the NIST/NSA Capstone project.

- *LUC:* A public-key cryptosystem designed by P.J. Smith and based on Lucas sequences. Can be used for encryption and signatures, using integer factoring.

- *McEliece:* A public-key cryptosystem based on algebraic coding theory.

For additional information on PKC algorithms, see "Public-Key Encryption" (Chapter 8) in *Handbook of Applied Cryptography*, by A. Menezes, P. van Oorschot, and S. Vanstone (CRC Press, 1996).

> **A digression: Who invented PKC?** I tried to be careful in the first paragraph of this section to state that Diffie and Hellman "first described publicly" a PKC scheme. Although I have categorized PKC as a two-key system, that has been merely for convenience; the real criteria for a PKC scheme is that it allows two parties to exchange a secret even though the communication with the shared secret might be overheard. There seems to be no question that Diffie and Hellman were first to publish; their method is described in the classic paper, "New Directions in Cryptography," published in the November 1976 issue of *IEEE Transactions on Information Theory* (*IT-22*(6), 644-654). As shown in Section 5.2, Diffie-Hellman uses the idea that finding logarithms is relatively harder than performing exponentiation. And, indeed, it is the precursor to modern PKC which does employ two keys. Rivest, Shamir, and Adleman described an implementation that extended this idea in their paper, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," published in the February 1978 issue of the *Communications of the ACM (CACM)* (*21*92), 120-126). Their method, of course, is based upon the relative ease of finding the product of two large prime numbers compared to finding the prime factors of a large number.
>
> Some sources, though, credit Ralph Merkle with first describing a system that allows two parties to share a secret although it was not a two-key system, per se. A *Merkle Puzzle* works where Alice creates a large number of encrypted keys, sends them all to Bob so that Bob chooses one at random and then lets Alice know which he has selected. An eavesdropper (Eve) will see all of the keys but can't learn which key Bob has selected (because he has encrypted the response with the chosen key). In this case, Eve's effort to break in is the square of the effort of Bob to choose a key. While this difference may be small it is often sufficient. Merkle apparently took a computer science course at UC Berkeley in 1974 and described his method, but had difficulty making people understand it; frustrated, he dropped the course. Meanwhile, he submitted the paper "Secure Communication Over Insecure Channels," which was published in the *CACM* in April 1978; Rivest et al.'s paper even makes reference to it. Merkle's method certainly wasn't published first, but did he have the idea first?
>
> An interesting question, maybe, but who really knows? For some time, it was a quiet secret that a team at the UK's Government Communications Headquarters (GCHQ) had first developed PKC in the early 1970s. Because of the nature of the work, GCHQ kept the original memos classified. In 1997, however, the GCHQ changed their posture when they realized that there was nothing to gain by continued silence. Documents show that a GCHQ mathematician named James Ellis started research into the key distribution problem in 1969 and that by 1975, James Ellis, Clifford Cocks, and Malcolm Williamson had worked out all of the fundamental details of PKC, yet couldn't talk about their work. (They were, of course, barred from challenging the RSA patent!) By 1999, Ellis, Cocks, and Williamson began to get their due credit in a break-through article in *WIRED Magazine*.
>
> And the National Security Agency (NSA) claims to have knowledge of this type of algorithm as early as 1966 but there is no supporting documentation... yet. So this really was a digression...

3.3. Hash Functions

Hash functions, also called *message digests* and *one-way encryption*, are algorithms that, in some sense, use no key (Figure 1C). Instead, a fixed-length hash value is computed based upon the plaintext that makes it impossible for either the contents or length of the plaintext to be recovered. Hash algorithms are typically used to provide a *digital fingerprint* of a file's contents, often used to ensure that the file has not been altered by an intruder or virus. Hash functions are also commonly employed by many operating systems to encrypt passwords. Hash functions, then, provide a measure of the integrity of a file.

Hash algorithms that are in common use today include:

- *Message Digest (MD) algorithms:* A series of byte-oriented algorithms that produce a 128-bit hash value from an arbitrary-length message.

    - *MD2 (RFC 1319):* Designed for systems with limited memory, such as smart cards. (MD2 has been relegated to historical status, per RFC 6149.)

    - *MD4 (RFC 1320):* Developed by Rivest, similar to MD2 but designed specifically for fast processing in software. (MD4 has been relegated to historical status, per RFC 6150.)

    - *MD5 (RFC 1321):* Also developed by Rivest after potential weaknesses were reported in MD4; this scheme is similar to MD4 but is slower because more manipulation is made to the original data. MD5 has been implemented in a large number of products although several weaknesses in the algorithm were demonstrated by German cryptographer Hans Dobbertin in 1996 ("Cryptanalysis of MD5 Compress").

- *Secure Hash Algorithm (SHA):* Algorithm for NIST's Secure Hash Standard (SHS), described in FIPS 180-4.

    - SHA-1 produces a 160-bit hash value and was originally published as FIPS PUB 180-1 and RFC 3174. It was deprecated by NIST as of the end of 2013 although it is still widely used. In October 2015, the SHA-1 Freestart Collision was announced; see a report by Bruce Schneier and the developers of the attack.

    - SHA-2, originally described in FIPS PUB 180-2 and eventually replaced by FIPS PUB 180-3 (and FIPS PUB 180-4), comprises five algorithms in the SHS: SHA-1 plus SHA-224, SHA-256, SHA-384, and SHA-512 which can produce hash values that are 224, 256, 384, or 512 bits in length, respectively. SHA-2 recommends use of SHA-1, SHA-224, and SHA-256 for messages less than 2^{64} bits in length, and employs a 512 bit block size; SHA-384 and SHA-512 are recommended for messages less than 2^{128} bits in length, and employs a 1,024 bit block size. FIPS PUB 180-4 also introduces the concept of a truncated hash in SHA-512/*t*, a generic name referring to a hash value based upon the SHA-512 algorithm that has been truncated to *t* bits; SHA-512/224 and SHA-512/256 are specifically described. SHA-224, -256, -384, and -512 are also described in RFC 4634.

    - SHA-3 is the current SHS algorithm. Although there had not been any successful attacks on SHA-2, NIST decided that having an alternative to SHA-2 using a different algorithm would be prudent. In 2007, they launched a SHA-3 Competition to find that alternative; a list of submissions can be found at The SHA-3 Zoo. In 2012, NIST announced that after reviewing 64 submissions, the winner was Keccak (pronounced "catch-ack"), a family of hash algorithms based upon sponge functions. The NIST version can support hash output sizes of 256 and 512 bits.

- *RIPEMD:* A series of message digests that initially came from the RIPE (RACE Integrity Primitives Evaluation) project. RIPEMD-160 was designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel, and optimized for 32-bit processors to replace the then-current 128-bit hash functions. Other versions include RIPEMD-256, RIPEMD-320, and RIPEMD-128.

- *HAVAL (HAsh of VAriable Length):* Designed by Y. Zheng, J. Pieprzyk and J. Seberry, a hash algorithm with many levels of security. HAVAL can create hash values that are 128, 160, 192, 224, or 256 bits in length. More details can be found in a *AUSCRYPT '92* paper.

- *Whirlpool:* Designed by V. Rijmen (co-inventor of Rijndael) and P.S.L.M. Barreto, Whirlpool is one of two hash functions endorsed by the New European Schemes for Signatures, Integrity, and Encryption (NESSIE) competition (the other being SHA). Whirlpool operates on messages less than 2^{256} bits in length and produces a message digest of 512 bits. The design of this hash function is very different than that of MD5 and SHA-1, making it immune to the same attacks as on those hashes.

- *Tiger:* Designed by Ross Anderson and Eli Biham, Tiger is designed to be secure, run efficiently on 64-bit processors, and easily replace MD4, MD5, SHA and SHA-1 in other applications. Tiger/192 produces a 192-bit output and is compatible with 64-bit architectures; Tiger/128 and Tiger/160 produce a hash of length 128 and 160 bits, respectively, to provide compatibility with the other hash functions mentioned above.

- *eD2k:* Named for the EDonkey2000 Network (eD2K), the eD2k hash is a *root hash* of an MD4 hash list of a given file. A root hash is used on peer-to-peer file transfer networks, where a file is broken into chunks; each chunk has its own MD4 hash associated with it and the server maintains a file that contains the hash list of all of the chunks. The root hash is the hash of the hash list file.

(Readers might be interested in HashCalc, a Windows-based program that calculates hash values using a dozen algorithms, including MD5, SHA-1 and several variants, RIPEMD-160, and Tiger. Command line utilities that calculate hash values include sha_verify by Dan Mares [Windows; supports MD5, SHA-1, SHA-2] and md5deep [cross-platform; supports MD5, SHA-1, SHA-256, Tiger, and Whirlpool].)

Hash functions are sometimes misunderstood and some sources claim that no two files can have the same hash value. This is, in fact, not correct. Consider a hash function that provides a 128-bit hash value. There are, obviously, 2^{128} possible hash values. But there are an infinite number of possible files and $\infty >> 2;^{128}$. Therefore, there have to be multiple files — in fact, there have to be an infinite number of files! — that have the same 128-bit hash value.

The difficulty is not necessarily in finding two files with the same hash, but in finding a second file that has the same hash value as a given first file. Consider this example. A human head has, generally, no more than ~150,000 hairs. Since there are more than 7 billion people on earth, we know that there are a lot of people with the same number of hairs on their heads. Finding two people with the same number of hairs, then, would be relatively simple. The harder problem is choosing one person (say, you, the reader) and then finding another person who has the same number of hairs on their head.

This is somewhat similar to the Birthday Problem. We know from probability that if you choose a random group of ~23 people, the probability is about 50% that two will share a birthday (the probability goes up to 99.9% with a group of 70 people). However, if you select one person in the group of 23 and try to find a match to that person, the probability is only about 6% of finding a match; you'd need a group of 253 for a 50% probability of a shared birthday (and a group of more than 4,000 to obtain a 99.9% probability).

What is hard to do is to try to create a file that has a given hash value so as to force a hash value collision — which is the reason that hash functions are used extensively for information security and computer forensics applications. Alas, researchers in 2004 found that *practical* collision attacks could be launched on MD5, SHA-1, and other hash algorithms. Readers interested in this problem should read the following:

- AccessData. (2006, April). *MD5 Collisions: The Effect on Computer Forensics*. AccessData White Paper.
- Burr, W. (2006, March/April). Cryptographic hash standards: Where do we go from here? *IEEE Security & Privacy*, *4*(2), 88-91.
- Dwyer, D. (2009, June 3). SHA-1 Collision Attacks Now 2^{52}. *SecureWorks Research blog*.
- Gutman, P., Naccache, D., & Palmer, C.C. (2005, May/June). When hashes collide. *IEEE Security & Privacy*, *3*(3), 68-71.
- Klima, V. (March 2005). Finding MD5 Collisions - a Toy For a Notebook.
- Lee, R. (2009, January 7). Law Is Not A Science: Admissibility of Computer Evidence and MD5 Hashes. *SANS Computer Forensics blog.*
- Stevens, M., Karpman, P., & Peyrin, T. (2015, October 8). Freestart collision on full SHA-1. Cryptology ePrint Archive, Report 2015/967.
- Thompson, E. (2005, February). MD5 collisions and the impact on computer forensics. *Digital Investigation*, *2*(1), 36-40.
- Wang, X., Feng, D., Lai, X., & Yu, H. (2004, August). Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD.
- Wang, X., Yin, Y.L., & Yu, H. (2005, February 13). Collision Search Attacks on SHA1.

Readers are also referred to the Eindhoven University of Technology HashClash Project Web site. An excellent overview of the situation with hash collisions (circa 2005) can be found in RFC 4270 (by P. Hoffman and B. Schneier, November 2005). And for additional information on hash functions, see David Hopwood's MessageDigest Algorithms page and Peter Selinger's MD5 Collision Demo page. Finally, for an interesting twist on this discussion, read about the *Nostradamus* attack reported at Predicting the winner of the 2008 US Presidential Elections using a Sony PlayStation 3 (by M. Stevens, A.K. Lenstra, and B. de Weger, November 2007).

Certain extensions of hash functions are used for a variety of information security and digital forensics applications, such as:

- *Hash libraries* are sets of hash values corresponding to known files. A hash library of known good files, for example, might be a set of files known to be a part of an operating system, while a hash library of known bad files might be of a set of known child pornographic images.
- *Rolling hashes* refer to a set of hash values that are computed based upon a fixed-length "sliding window" through the input. As an example, a hash value might be computed on bytes 1-10 of a file, then on bytes 2-11, 3-12, 4-13, etc.
- *Fuzzy hashes* are an area of intense research and represent hash values that represent two inputs that are similar. Fuzzy hashes are used to detect documents, images, or other files that are close to each other with respect to content. See "Fuzzy Hashing" (PDF) by Jesse Kornblum for a good treatment of this topic.

3.4. Why Three Encryption Techniques?

So, why are there so many different types of cryptographic schemes? Why can't we do everything we need with just one?

The answer is that each scheme is optimized for some specific application(s). Hash functions, for example, are well-suited for ensuring data integrity because any change made to the contents of a message will result in the receiver calculating a different hash value than the one placed in the transmission by the sender. Since it is highly unlikely that two different messages will yield the same hash value, data integrity is ensured to a high degree of confidence.

Secret key cryptography, on the other hand, is ideally suited to encrypting messages, thus providing privacy and confidentiality. The sender can generate a *session key* on a per-message basis to encrypt the message; the receiver, of course, needs the same session key to decrypt the message.

Key exchange, of course, is a key application of public-key cryptography (no pun intended). Asymmetric schemes can also be used for non-repudiation and user authentication; if the receiver can obtain the session key encrypted with the sender's private key, then only this sender could have sent the message. Public-key cryptography could, theoretically, also be used to encrypt messages although this is rarely done because secret-key cryptography operates about 1000 times faster than public-key cryptography.

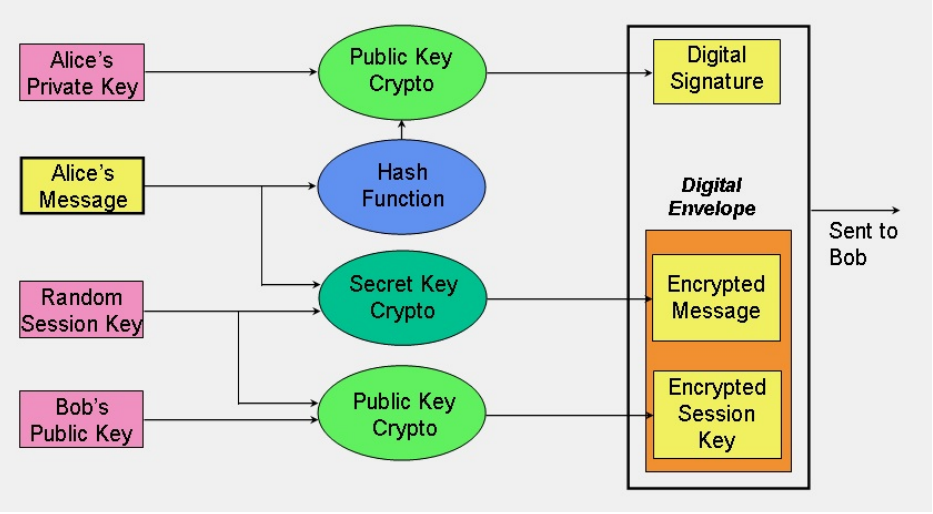


FIGURE 2: Use of the three cryptographic techniques for secure communication.

Figure 2 puts all of this together and shows how a *hybrid cryptographic* scheme combines all of these functions to form a secure transmission comprising *digital signature* and *digital envelope*. In this example, the sender of the message is Alice and the receiver is Bob.

A digital envelope comprises an encrypted message and an encrypted session key. Alice uses secret key cryptography to encrypt her message using the *session key*, which she generates at random with each session. Alice then encrypts the session key using Bob's public key. The encrypted message and encrypted session key together form the digital envelope. Upon receipt, Bob recovers the session secret key using his private key and then decrypts the encrypted message.

The digital signature is formed in two steps. First, Alice computes the hash value of her message; next, she encrypts the hash value with her private key. Upon receipt of the digital signature, Bob recovers the hash value calculated by Alice by decrypting the digital signature with Alice's public key. Bob can then apply the hash function to Alice's original message, which he has already decrypted (see previous paragraph). If the resultant hash value is not the same as the value supplied by Alice, then Bob knows that the message has been altered; if the hash values are the same, Bob should believe that the message he received is identical to the one that Alice sent.

This scheme also provides nonrepudiation since it proves that Alice sent the message; if the hash value recovered by Bob using Alice's public key proves that the message has not been altered, then only Alice could have created the digital signature. Bob also has proof that he is the intended receiver; if he can correctly decrypt the message, then he must have correctly decrypted the session key meaning that his is the correct private key.

This diagram purposely suggests a cryptosystem where the session key is used for just a single session. Even if this session key is somehow broken, only this session will be compromised; the session key for the next session is in no way based upon the key for this session, just as this session's key is not dependent on the key from the previous session. This is known as Perfect Forward Secrecy; you might lose one session key due to a compromise but you won't lose all of them. (This was an issue in the 2014 OpenSSL vulnerability known as Heartbleed.)

3.5. The Significance of Key Length

In a 1998 article in the industry literature, a writer made the claim that 56-bit keys did not provide as adequate protection for DES at that time as they did in 1975 because computers were 1000 times faster in 1998 than in 1975. Therefore, the writer went on, we needed 56,000-bit keys in 1998 instead of 56-bit keys to provide adequate protection. The conclusion was then drawn that because 56,000-bit keys are infeasible (*true*), we should accept the fact that we have to live with weak cryptography (*false!*). The major error here is that the writer did not take into account that the number of possible key values double whenever a single bit is added to the key length; thus, a 57-bit key has twice as many values as a 56-bit key (because 2^{57} is two times 2^{56}). In fact, a 66-bit key would have 1024 times more values than a 56-bit key.

But this does bring up the issue, what is the precise significance of key length as it affects the level of protection?

In cryptography, size does matter. The larger the key, the harder it is to crack a block of encrypted data. The reason that large keys offer more protection is almost obvious; computers have made it easier to attack ciphertext by using brute force methods rather than by attacking the mathematics (which are generally well-known anyway). With a brute force attack, the attacker merely generates every possible key and applies it to the ciphertext. Any resulting plaintext that makes sense offers a candidate for a legitimate key. This was the basis, of course, of the EFF's attack on DES.

Until the mid-1990s or so, brute force attacks were beyond the capabilities of computers that were within the budget of the attacker community. By that time, however, significant compute power was typically available and accessible. General-purpose computers such as PCs were already being used for brute force attacks. For serious attackers with money to spend, such as some large companies or governments, Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuits (ASIC) technology offered the ability to build specialized chips that could provide even faster and cheaper solutions than a PC. (As an example, the AT&T; Optimized Reconfigurable Cell Array (ORCA) FPGA chip cost about $200 and could test 30 million DES keys per second, while a $10 ASIC chip could test 200 million DES keys per second; compare that to a PC which might be able to test 40,000 keys per second.) Distributed attacks, harnessing the power of between tens and tens of thousands of powerful CPUs, are now commonly employed to try to brute-force crypto keys.

The table below — from a 1995 article discussing both why exporting 40-bit keys was, in essence, no crypto at all *and* why DES' days were numbered — shows what DES key sizes were needed to protect data from attackers with different time and financial resources. This information was not merely academic; one of the basic tenets of any security system is to have an idea of *what* you are protecting and *from who* are you protecting it! The table clearly shows that a 40-bit key was essentially worthless against even the most unsophisticated attacker. On the other hand, 56-bit keys were fairly strong unless you might be subject to some pretty serious corporate or government espionage. But note that even 56-bit keys were clearly on the decline in their value and that the times in the table were worst cases.

TABLE 1. Minimum Key Lengths for Symmetric Ciphers (1995).

Type of Attacker	Budget	Tool	Time and Cost Per Key Recovered		Key Length Needed For Protection In Late-1995
			40 bits	56 bits	
Pedestrian Hacker	Tiny	Scavenged computer time	1 week	Infeasible	45
	$400	FPGA	5 hours ($0.08)	38 years ($5,000)	50
Small Business	$10,000	FPGA	12 minutes ($0.08)	18 months ($5,000)	55
Corporate Department	$300K	FPGA	24 seconds ($0.08)	19 days ($5,000)	60
		ASIC	0.18 seconds ($0.001)	3 hours ($38)	
Big Company	$10M	FPGA	7 seconds ($0.08)	13 hours ($5,000)	70
		ASIC	0.005 seconds ($0.001)	6 minutes ($38)	
Intelligence Agency	$300M	ASIC	0.0002 seconds ($0.001)	12 seconds ($38)	75

So, how big is big enough? DES, invented in 1975, was still in use at the turn of the century, nearly 25 years later. If we take that to be a design criteria (i.e., a 20-plus year lifetime) and we believe Moore's Law ("computing power doubles every 18 months"), then a key size extension of 14 bits (i.e., a factor of more than 16,000) should be adequate. The 1975 DES proposal suggested 56-bit keys; by 1995, a 70-bit key would have been required to offer equal protection and an 85-bit key necessary by 2015.

A 256- or 512-bit SKC key will probably suffice for some time because that length keeps us ahead of the brute force capabilities of the attackers. Note that while a large key is good, a huge key may not always be better; for example, expanding PKC keys beyond the current 2048- or 4096-bit lengths doesn't add any necessary protection at this time. Weaknesses in cryptosystems are largely based upon key management rather than weak keys.

Much of the discussion above, including the table, is based on the paper "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security" by M. Blaze, W. Diffie, R.L. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener.

The most effective large-number factoring methods today use a mathematical Number Field Sieve to find a certain number of relationships and then uses a matrix operation to solve a linear equation to produce the two prime factors. The sieve step actually involves a large number of operations that can be performed in parallel; solving the linear equation, however, requires a supercomputer. Indeed, finding the solution to the RSA-140 challenge in February 1999 — factoring a 140-digit (465-bit) prime number — required 200 computers across the Internet about 4 weeks for the first step and a Cray computer 100 hours and 810 MB of memory to do the second step.

In early 1999, Shamir (of RSA fame) described a new machine that could increase factorization speed by 2-3 orders of magnitude. Although no detailed plans were provided nor is one known to have been built, the concepts of TWINKLE (The Weizmann Institute Key Locating Engine) could result in a specialized piece of hardware that would cost about $5000 and have the processing power of 100-1000 PCs. There still appear to be many engineering details that have to be worked out before such a machine could be built. Furthermore, the hardware improves the sieve step only; the matrix operation is not optimized at all by this design and the complexity of this step grows rapidly with key length, both in terms of processing time and memory requirements. Nevertheless, this plan conceptually puts 512-bit keys within reach of being factored. Although most PKC schemes allow keys that are 1024 bits and longer, Shamir claims that 512-bit RSA keys "protect 95% of today's E-commerce on the Internet." (See Bruce Schneier's Crypto-Gram (May 15, 1999) for more information, as well as the comments from RSA Labs.)

It is also interesting to note that while cryptography is good and strong cryptography is better, long keys may disrupt the nature of the randomness of data files. Shamir and van Someren ("Playing hide and seek with stored keys") have noted that a new generation of viruses can be written that will find files encrypted with long keys, making them easier to find by intruders and, therefore, more prone to attack.

Finally, U.S. government policy has tightly controlled the export of crypto products since World War II. Until the mid-1990s, export outside of North America of cryptographic products using keys greater than 40 bits in length was prohibited, which made those products essentially worthless in the marketplace, particularly for electronic commerce; today, crypto products are widely available on the Internet without restriction. The U.S. Department of Commerce Bureau of Industry and Security maintains an Encryption FAQ web page with more information about the current state of encryption registration.

On a related topic, public key crypto schemes can be used for several purposes, including key exchange, digital signatures, authentication, and more. In those PKC systems used for SKC key exchange, the PKC key lengths are chosen so to be resistant to some selected level of attack. The length of the secret keys exchanged via that system have to have at least the same level of attack resistance. Thus, the three parameters of such a system — system strength, secret key strength, and public key strength — must be matched. This topic is explored in more detail in *Determining Strengths For Public Keys Used For Exchanging Symmetric Keys* (RFC 3766).

4. TRUST MODELS

Secure use of cryptography requires trust. While secret key cryptography can ensure message confidentiality and hash codes can ensure integrity, none of this works without trust. In SKC, Alice and Bob had to share a secret key. PKC solved the secret distribution problem, but how does Alice really know that Bob is who he says he is? Just because Bob has a public and private key, and purports to be "Bob," how does Alice know that a malicious person (Mallory) is not pretending to be Bob?

There are a number of *trust models* employed by various cryptographic schemes. This section will explore three of them:

- The web of trust employed by Pretty Good Privacy (PGP) users, who hold their own set of trusted public keys.
- Kerberos, a secret key distribution scheme using a trusted third party.
- Certificates, which allow a set of trusted third parties to authenticate each other and, by implication, each other's users.

Each of these trust models differs in complexity, general applicability, scope, and scalability.

4.1. PGP Web of Trust

Pretty Good Privacy (described more below in Section 5.5) is a widely used private e-mail scheme based on public key methods. A PGP user maintains a local keyring of all their known and trusted public keys. The user makes their own determination about the trustworthiness of a key using what is called a "web of trust."

If Alice needs Bob's public key, Alice can ask Bob for it in another e-mail or, in many cases, download the public key from an advertised server; this server might a well-known PGP key repository or a site that Bob maintains himself. In fact, Bob's public key might be stored or listed in many places. (The author's public key, for example, can be found at *http://www.garykessler.net/pubkey.html*.) Alice is prepared to believe that Bob's public key, as stored at these locations, is valid.

Suppose Carol claims to hold Bob's public key and offers to give the key to Alice. How does Alice know that Carol's version of Bob's key is valid or if Carol is actually giving Alice a key that will allow Mallory access to messages? The answer is, "It depends." If Alice trusts Carol and Carol says that she thinks that her version of Bob's key is valid, then Alice *may* — at *her* option — trust that key. And trust is not necessarily transitive; if Dave has a copy of Bob's key and Carol trusts Dave, it does not necessarily follow that Alice trusts Dave even if she does trust Carol.

The point here is that who Alice trusts and how she makes that determination is strictly up to Alice. PGP makes no statement and has no protocol about how one user determines whether they trust another user or not. In any case, encryption and signatures based on public keys can only be used when the appropriate public key is on the user's keyring.

4.2. Kerberos

Kerberos is a commonly used authentication scheme on the Internet. Developed by MIT's Project Athena, Kerberos is named for the three-headed dog who, according to Greek mythology, guards the entrance of Hades (rather than the exit, for some reason!).

Kerberos employs a client/server architecture and provides user-to-server authentication rather than host-to-host authentication. In this model, security and authentication will be based on secret key technology where every host on the network has its own secret key. It would clearly be unmanageable if every host had to know the keys of all other hosts so a secure, trusted host somewhere on the network, known as a Key Distribution Center (KDC), knows the keys for all of the hosts (or at least some of the hosts within a portion of the network, called a *realm*). In this way, when a new node is brought online, only the KDC and the new node need to be configured with the node's key; keys can be distributed physically or by some other secure means.

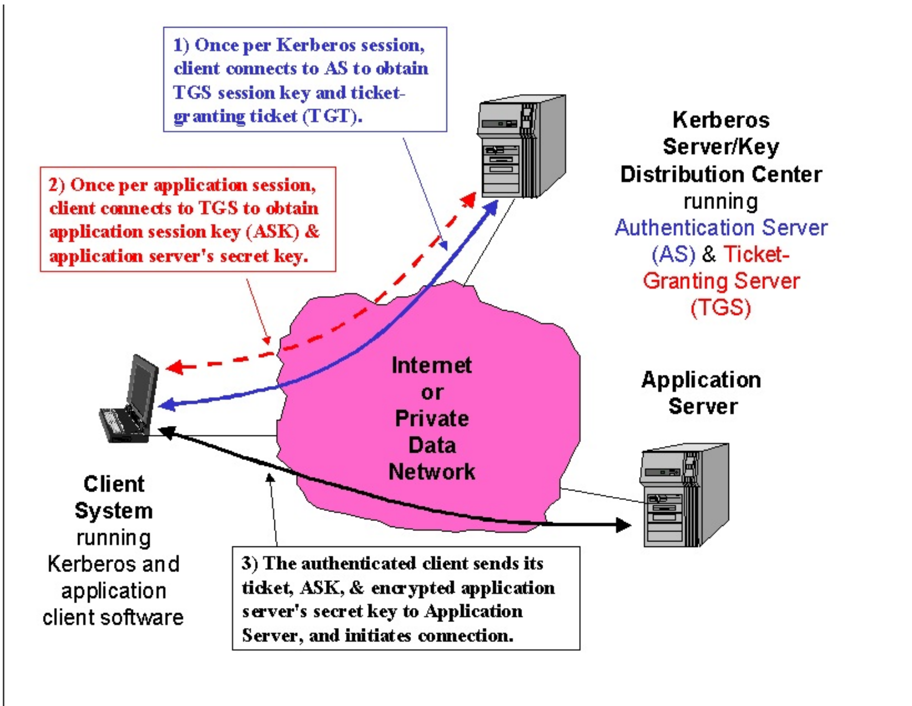

1) Once per Kerberos session, client connects to AS to obtain TGS session key and ticket-granting ticket (TGT).
2) Once per application session, client connects to TGS to obtain application session key (ASK) & application server's secret key.
Kerberos Server/Key Distribution Center running Authentication Server (AS) & Ticket-Granting Server (TGS)
Internet or Private Data Network
Application Server
Client System running Kerberos and application client software
3) The authenticated client sends its ticket, ASK, & encrypted application server's secret key to Application Server, and initiates connection.

FIGURE 3: Kerberos architecture.

The Kerberos Server/KDC has two main functions (Figure 3), known as the Authentication Server (AS) and Ticket-Granting Server (TGS). The steps in establishing an authenticated session between an application client and the application server are:

1. The Kerberos client software establishes a connection with the Kerberos server's AS function. The AS first authenticates that the client is who it purports to be. The AS then provides the client with a secret key for this login session (the *TGS session key*) and a ticket-granting ticket (TGT), which gives the client permission to talk to the TGS. The ticket has a finite lifetime so that the authentication process is repeated periodically.
2. The client now communicates with the TGS to obtain the Application Server's key so that it (the client) can establish a connection to the service it wants. The client supplies the TGS with the TGS session key and TGT; the TGS responds with an application session key (ASK) and an encrypted form of the Application Server's secret key; this secret key is *never* sent on the network in any other form.
3. The client has now authenticated itself *and* can prove its identity to the Application Server by supplying the Kerberos ticket, application session key, and encrypted Application Server secret key. The Application Server responds with similarly encrypted information to authenticate itself to the client. At this point, the client can initiate the intended service requests (e.g., Telnet, FTP, HTTP, or e-commerce transaction session establishment).

The current version of this protocol is Kerberos V5 (described in RFC 1510). While the details of their operation, functional capabilities, and message formats are different, the conceptual overview above pretty much holds for both. One primary difference is that Kerberos V4 uses only DES to generate keys and encrypt messages, while V5 allows other schemes to be employed (although DES is still the most widely algorithm used).

4.3. Public Key Certificates and Certificate Authorities

Certificates and *Certificate Authorities (CA)* are necessary for widespread use of cryptography for e-commerce applications. While a combination of secret and public key cryptography can solve the business issues discussed above, crypto cannot alone address the trust issues that must exist between a customer and vendor in the very fluid, very dynamic e-commerce relationship. How, for example, does one site obtain another party's public key? How does a recipient determine if a public key really belongs to the sender? How does the recipient know that the sender is using their public key for a legitimate purpose for which they are authorized? When does a public key expire? How can a key be revoked in case of compromise or loss?

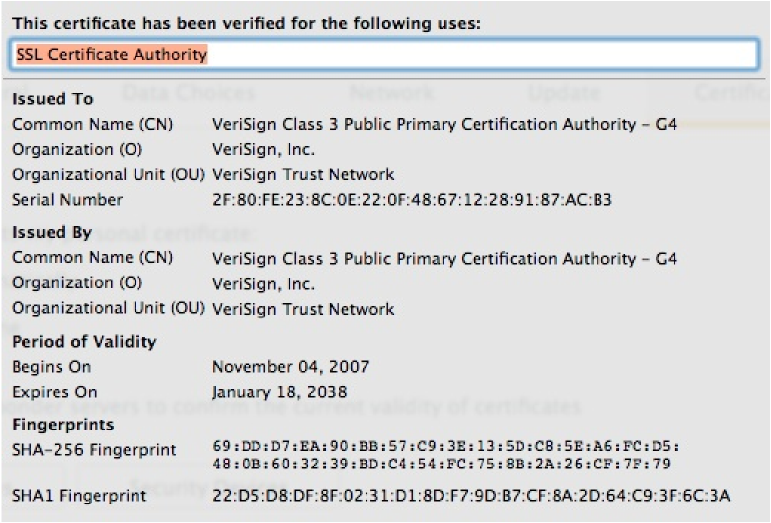
The basic concept of a certificate is one that is familiar to all of us. A driver's license, credit card, or SCUBA certification, for example, identify us to others, indicate something that we are authorized to do, have an expiration date, and identify the authority that granted the certificate.

As complicated as this may sound, it really isn't! Consider driver's licenses. I have one issued by the State of Florida. The license establishes my identity, indicates the type of vehicles that I can operate and the fact that I must wear corrective lenses while doing so, identifies the issuing authority, and notes that I am an organ donor. When I drive in other states, the other jurisdictions throughout the U.S. recognize the authority of Florida to issue this "certificate" and they trust the information it contains. When I leave the U.S., everything changes. When I am in Aruba, Australia, Canada, Israel, and many other countries, they will accept not the Florida license, per se, but *any* license issued in the U.S. This analogy represents the certificate trust chain, where even certificates carry certificates.

For purposes of electronic transactions, certificates are digital documents. The specific functions of the certificate include:

- *Establish identity:* Associate, or *bind*, a public key to an individual, organization, corporate position, or other entity.
- *Assign authority:* Establish what actions the holder may or may not take based upon this certificate.
- *Secure confidential information* (e.g., encrypting the session's symmetric key for data confidentiality).

Typically, a certificate contains a public key, a name, an expiration date, the name of the authority that issued the certificate (and, therefore, is vouching for the identity of the user), a serial number, any pertinent policies describing how the certificate was issued and/or how the certificate may be used, the digital signature of the certificate issuer, and perhaps other information.



This certificate has been verified for the following uses:

SSL Certificate Authority

Issued To

Common Name (CN)	VeriSign Class 3 Public Primary Certification Authority – G4
Organization (O)	VeriSign, Inc.
Organizational Unit (OU)	VeriSign Trust Network
Serial Number	2F:80:FE:23:8C:0E:22:0F:48:67:12:28:91:87:AC:B3

Issued By

Common Name (CN)	VeriSign Class 3 Public Primary Certification Authority – G4
Organization (O)	VeriSign, Inc.
Organizational Unit (OU)	VeriSign Trust Network

Period of Validity

Begins On	November 04, 2007
Expires On	January 18, 2038

Fingerprints

SHA-256 Fingerprint	69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5: 48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:79
SHA1 Fingerprint	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

FIGURE 4: VeriSign Class 3 certificate.

A sample abbreviated certificate is shown in Figure 4. This is a typical certificate found in a browser, in this case, Mozilla Firefox (Mac OS X). While this is a certificate issued by VeriSign, many root-level certificates can be found shipped with browsers. When the browser makes a connection to a secure Web site, the Web server sends its public key certificate to the browser. The browser then checks the certificate's signature against the public key that it has stored; if there is a match, the certificate is taken as valid and the Web site verified by this certificate is considered to be "trusted."

TABLE 2. Contents of an X.509 V3 Certificate.

version number certificate serial number signature algorithm identifier issuer's name and unique identifier validity (or operational) period subject's name and unique identifier subject public key information standard extensions   certificate appropriate use definition   key usage limitation definition   certificate policy information other extensions   Application-specific   CA-specific

The most widely accepted certificate format is the one defined in International Telecommunication Union Telecommunication Standardization Sector (ITU-T) Recommendation X.509. Rec. X.509 is a specification used around the world and any applications complying with X.509 can share certificates. Most certificates today comply with X.509 Version 3 and contain the information listed in Table 2.

Certificate authorities are the repositories for public-keys and can be any agency that issues certificates. A company, for example, may issue certificates to its employees, a college/university to its students, a store to its customers, an Internet service provider to its users, or a government to its constituents.

When a sender needs an intended receiver's public key, the sender must get that key from the receiver's CA. That scheme is straight-forward if the sender and receiver have certificates issued by the same CA. If not, how does the sender know to *trust* the foreign CA? One industry wag has noted, about trust: "You are either born with it or have it granted upon you." Thus, some CAs will be trusted because they are known to be reputable, such as the CAs operated by AT&T; Services, Comodo, DigiNet (formerly GTE Cybertrust), EnTrust, Symantec (formerly VeriSign), and Thawte. CAs, in turn, form trust relationships with other CAs. Thus, if a user queries a foreign CA for information, the user may ask to see a list of CAs that establish a "chain of trust" back to the user.

One major feature to look for in a CA is their identification policies and procedures. When a user generates a key pair and forwards the public key to a CA, the CA has to check the sender's identification and takes any steps necessary to assure itself that the request is really coming from the advertised sender. Different CAs have different identification policies and will, therefore, be trusted differently by other CAs. Verification of identity is just one of many issues that are part of a CA's Certification Practice Statement (CPS) and policies; other issues include how the CA protects the public keys in its care, how lost or compromised keys are revoked, and how the CA protects its own private keys.

4.4. Summary

The paragraphs above describe three very different trust models. It is hard to say that any one is better than the others; it depend upon your application. One of the biggest and fastest growing applications of cryptography today, though, is electronic commerce (e-commerce), a term that itself begs for a formal definition.

PGP's web of trust is easy to maintain and very much based on the reality of users as people. The model, however, is limited; just how many public keys can a single user reliably store and maintain? And what if you are using the "wrong" computer when you want to send a message and can't access your keyring? How easy it is to revoke a key if it is compromised? PGP may also not scale well to an e-commerce scenario of secure communication between total strangers on short-notice.

Kerberos overcomes many of the problems of PGP's web of trust, in that it is scalable and its scope can be very large. However, it also requires that the Kerberos server have *a priori* knowledge of all client systems prior to any transactions, which makes it unfeasible for "hit-and-run" client/server relationships as seen in e-commerce.

Certificates and the collection of CAs will form a Public Key Infrastructure (PKI). In the early days of the Internet, every host had to maintain a list of every other host; the Domain Name System (DNS) introduced the idea of a distributed database for this purpose and the DNS is one of the key reasons that the Internet has grown as it has. A PKI will fill a similar void in the e-commerce and PKC realm.

While certificates and the benefits of a PKI are most often associated with electronic commerce, the applications for PKI are much broader and include secure electronic mail, payments and electronic checks, Electronic Data Interchange (EDI), secure transfer of Domain Name System (DNS) and routing information, electronic forms, and digitally signed documents. A single "global PKI" is still many years away, that is the ultimate goal of today's work as international electronic commerce changes the way in which we do business in a similar way in which the Internet has changed the way in which we communicate.

5. CRYPTOGRAPHIC ALGORITHMS IN ACTION

The paragraphs above have provided an overview of the different types of cryptographic algorithms, as well as some examples of some available protocols and schemes. Table 3 provides a list of some other noteworthy schemes employed — or proposed — for a variety of functions, most notably electronic commerce and secure communication. The paragraphs below will show several real cryptographic applications that many of us employ (knowingly or not) everyday for password protection and private communication. Some of the schemes described below never were widely deployed but are still historically interesting, thus remain included here.

TABLE 3. Other Crypto Algorithms and Systems of Note.

Capstone	A now-defunct U.S. National Institute of Standards and Technology (NIST) and National Security Agency (NSA) project under the Bush Sr. and Clinton administrations for publicly available strong cryptography with keys escrowed by the government (NIST and the Treasury Dept.). Capstone included in one or more tamper-proof computer chips for implementation (Clipper), a secret key encryption algorithm (Skipjack), digital signature algorithm (DSA), key exchange algorithm (KEA), and hash algorithm (SHA).
Challenge-Handshake Authentication Protocol (CHAP)	An authentication scheme that allows one party to prove who they are to a second party by demonstrating knowledge of a shared secret without actually divulging that shared secret to a third party who might be listening. Described in RFC 1994.
Clipper	The computer chip that would implement the Skipjack

	encryption scheme. The Clipper chip was to have had a deliberate backdoor so that material encrypted with this device would not be beyond the government's reach. Described in 1993, Clipper was dead by 1996. See also EPIC's The Clipper Chip Web page.
Derived Unique Key Per Transaction (DUKPT)	A key management scheme used for debit and credit card verification with point-of-sale (POS) transaction systems, automated teller machines (ATMs), and other financial applications. In DUKPT, a unique key is derived for each transaction based upon a fixed, shared key in such a way that knowledge of one derived key does not easily yield knowledge of other keys (including the fixed key). Therefore, if one of the derived keys is compromised, neither past nor subsequent transactions are endangered. DUKPT is specified in American National Standard (ANS) ANSI X9.24-1:2009 *Retail Financial Services Symmetric Key Management Part 1: Using Symmetric Techniques*) and can be purchased at the ANSI X9.24 Web page.
Escrowed Encryption Standard (EES)	Largely unused, a controversial crypto scheme employing the SKIPJACK secret key crypto algorithm and a Law Enforcement Access Field (LEAF) creation method. LEAF was one part of the key escrow system and allowed for decryption of ciphertext messages that had been intercepted by law enforcement agencies. Described more in FIPS 185 (archived; no longer in force).
Federal Information Processing Standards (FIPS)	These computer security- and crypto-related FIPS are produced by the U.S. National Institute of Standards and Technology (NIST) as standards for the U.S. Government.
Fortezza	A PCMCIA card developed by NSA that implements the Capstone algorithms, intended for use with the Defense Messaging Service (DMS). Originally called Tessera.
GOST	GOST is a family of algorithms that is defined in the Russian cryptographic standards. Although most of the specifications are written in Russian, a series of RFCs describe some of the aspects so that the algorithms can be used effectively in Internet applications: • RFC 4357: Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms • RFC 5830: GOST 28147-89: Encryption, Decryption, and Message Authentication Code (MAC) Algorithms • RFC 6986: GOST R 34.11-2012: Hash Function Algorithm • RFC 7091: GOST R 34.10-2012: Digital Signature Algorithm (Updates RFC 5832: GOST R 34.10-2001) • RFC 7801: GOST R 34.12-2015: Block Cipher "Kuznyechik" • RFC 7836: Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012
Identity-Based Encryption (IBE)	Identity-Based Encryption was first proposed by Adi Shamir in 1984 and is a key authentication system where the public key can be derived from some unique information based upon the user's identity. In 2001, Dan Boneh (Stanford) and Matt Franklin (U.C., Davis) developed a practical implementation of IBE based on elliptic curves and a mathematical construct called the Weil Pairing. In that year, Clifford Cocks (GCHQ) also described another IBE solution based on quadratic residues in composite groups. RFC 5091: Identity-Based Cryptography Standard (IBCS) #1: Describes an implementation of IBE using Boneh-Franklin (BF) and Boneh-Boyen (BB1) Identity-based Encryption.
IP Security Protocol (IPsec)	The IPsec protocol suite is used to provide privacy and authentication services at the IP layer. An overview of the protocol suite and of the documents comprising IPsec can be found in RFC 2411. Other documents include: • RFC 4301: IP security architecture. • RFC 4302: IP Authentication Header (AH), one of the two primary IPsec functions; AH provides connectionless integrity and data origin authentication for IP datagrams and protects against replay attacks. • RFC 4303: IP Encapsulating Security Payload (ESP), the

	other primary IPsec function; ESP provides a variety of security services within IPsec. • RFC 4304: Extended Sequence Number (ESN) Addendum, allows for negotiation of a 32- or 64- bit sequence number, used to detect replay attacks. • RFC 4305: Cryptographic algorithm implementation requirements for ESP and AH. • RFC 5996: The Internet Key Exchange (IKE) protocol, version 2, providing for mutual authentication and establishing and maintaining security associations. ◦ IKE v1 was described in three separate documents, RFC 2407 (application of ISAKMP to IPsec), RFC 2408 (ISAKMP, a framework for key management and security associations), and RFC 2409 (IKE, using part of Oakley and part of SKEME in conjunction with ISAKMP to obtain authenticated keying material for use with ISAKMP, and for other security associations such as AH and ESP). IKE v1 is obsoleted with the introdcution of IKEv2. • RFC 4307: Cryptographic algoritms used with IKEv2. • RFC 4308: Crypto suites for IPsec, IKE, and IKEv2. • RFC 4309: The use of AES in CBC-MAC mode with IPsec ESP. • RFC 4312: The use of the Camellia cipher algorithm in IPsec. • RFC 4359: The Use of RSA/SHA-1 Signatures within Encapsulating Security Payload (ESP) and Authentication Header (AH). • RFC 4434: Describes AES-XCBC-PRF-128, a pseudo-random function derived from the AES for use with IKE. • RFC 2403: Describes use of the HMAC with MD5 algorithm for data origin authentication and integrity protection in both AH and ESP. • RFC 2405: Describes use of DES-CBC (DES in Cipher Block Chaining Mode) for confidentiality in ESP. • RFC 2410: Defines use of the NULL encryption algorithm (i.e., provides authentication and integrity without confidentiality) in ESP. • RFC 2412: Describes OAKLEY, a key determination and distribution protocol. • RFC 2451: Describes use of Cipher Block Chaining (CBC) mode cipher algorithms with ESP. • RFCs 2522 and 2523: Description of Photuris, a session-key management protocol for IPsec. In addition, RFC 6379 describes Suite B Cryptographic Suites for IPsec and RFC 6380 describes the Suite B profile for IPsec. IPsec was first proposed for use with IP version 6 (IPv6), but can also be employed with the current IP version, IPv4. (See more detail about IPsec below in Section 5.6.)
Internet Security Association and Key Management Protocol (ISAKMP/OAKLEY)	ISAKMP/OAKLEY provide an infrastructure for Internet secure communications. ISAKMP, designed by the National Security Agency (NSA) and described in RFC 2408, is a framework for key management and security associations, independent of the key generation and cryptographic algorithms actually employed. The OAKLEY Key Determination Protocol, described in RFC 2412, is a key determination and distribution protocol using a variation of Diffie-Hellman.
Kerberos	A secret-key encryption and authentication system, designed to authenticate requests for network resources within a user domain rather than to authenticate messages. Kerberos also uses a trusted third-party approach; a client communications with the Kerberos server to obtain "credentials" so that it may access services at the application server. Kerberos V4 used DES to generate keys and encrypt messages; Kerberos V5 uses DES and other schemes for key generation. Microsoft added support for Kerberos V5 — with some proprietary extensions — in Windows 2000 Active Directory. There are many Kerberos articles posted at Microsoft's Knowledge Base, notably "Kerberos Explained."
Keyed-Hash Message Authentication Code	A message authentication scheme based upon secret key cryptography and the secret key shared between two parties

(HMAC)	rather than public key methods. Described in FIPS 198 and RFC 2104. (See Section 5.6 below for details on HMAC operation.)
Message Digest Cipher (MDC)	Invented by Peter Gutman, MDC turns a one-way hash function into a block cipher.
MIME Object Security Services (MOSS)	Designed as a successor to PEM to provide PEM-based security services to MIME messages. Described in RFC 1848. Never widely implemented and now defunct.
NSA Suite B Cryptography	An NSA standard for securing information at the SECRET level. Defines use of: - Advanced Encryption Standard (AES) with key sizes of 128 and 256 bits, per FIPS PUB 197 for encryption - The Ephemeral Unified Model and the One-Pass Diffie Hellman (referred to as ECDH) using the curves with 256 and 384-bit prime moduli, per NIST Special Publication 800-56A for key exchange - Elliptic Curve Digital Signature Algorithm (ECDSA) using the curves with 256 and 384-bit prime moduli, per FIPS PUB 186-3 for digital signatures - Secure Hash Algorithm (SHA) using 256 and 384 bits, per FIPS PUB 180-3 for hashing RFC 6239 describes Suite B Cryptographic Suites for Secure Shell (SSH) and RFC 6379 describes Suite B Cryptographic Suites for Secure IP (IPsec).
Pretty Good Privacy (PGP)	A family of cryptographic routines for e-mail, file, and disk encryption developed by Philip Zimmermann. PGP 2.6.x uses RSA for key management and digital signatures, IDEA for message encryption, and MD5 for computing the message's hash value; more information can also be found in RFC 1991. PGP 5.x (formerly known as "PGP 3") uses Diffie-Hellman/DSS for key management and digital signatures; IDEA, CAST, or 3DES for message encryption; and MD5 or SHA for computing the message's hash value. OpenPGP, described in RFC 2440, is an open definition of security software based on PGP 5.x. (See more detail about PGP below in Section 5.5.)
Privacy Enhanced Mail (PEM)	An IETF standard for secure electronic mail over the Internet, including provisions for encryption (DES), authentication, and key management (DES, RSA). Developed by the IETF but never widely used. Described in the following RFCs: - RFC 1421: Part I, Message Encryption and Authentication Procedures - RFC 1422: Part II, Certificate-Based Key Management - RFC 1423: Part III, Algorithms, Modes, and Identifiers - RFC 1424: Part IV, Key Certification and Related Services
Private Communication Technology (PCT)	Developed by Microsoft for secure communication on the Internet. PCT supported Diffie-Hellman, Fortezza, and RSA for key establishment; DES, RC2, RC4, and triple-DES for encryption; and DSA and RSA message signatures. Never widely used; superceded by SSL and TLS.
Secure Electronic Transaction (SET)	A communications protocol for securing credit card transactions, developed by MasterCard and VISA, in cooperation with IBM, Microsoft, RSA, and other companies. Merged two other protocols: Secure Electronic Payment Protocol (SEPP), an open specification for secure bank card transactions over the Internet developed by CyberCash, GTE, IBM, MasterCard, and Netscape; and Secure Transaction Technology (STT), a secure payment protocol developed by Microsoft and Visa International. Supports DES and RC4 for encryption, and RSA for signatures, key exchange, and public-key encryption of bank card numbers. SET V1.0 is described in Book 1, Book 2, and Book 3. SET has been superceded by SSL and TLS.
Secure Hypertext Transfer Protocol (S-HTTP)	An extension to HTTP to provide secure exchange of documents over the World Wide Web. Supported algorithms include RSA and Kerberos for key exchange, DES, IDEA, RC2, and Triple-DES for encryption. Described in RFC 2660. S-HTTP was never as widely used as HTTP over SSL (https).
Secure Multipurpose	An IETF secure e-mail scheme intended to supercede PEM.

Internet Mail Extensions (S/MIME)	S/MIME, described in RFCs 2311 and 2312, adds digital signature and encryption capability to Internet MIME messages.
Secure Sockets Layer (SSL)	Developed by Netscape Communications to provide application-independent security and privacy over the Internet. SSL is designed so that protocols such as HTTP, FTP (File Transfer Protocol), and Telnet can operate over it transparently. SSL allows both server authentication (mandatory) and client authentication (optional). RSA is used during negotiation to exchange keys and identify the actual cryptographic algorithm (DES, IDEA, RC2, RC4, or 3DES) to use for the session. SSL also uses MD5 for message digests and X.509 public-key certificates. SSL was found to be breakable soon after the IETF announced formation of group to work on TLS and RFC 6176 specifically prohibits the use of SSL v2.0 by TLS clients. SSL version 3.0 is described in RFC 6101. All versions of SSL are now deprecated in favor of TLS; TLS v1.0 is sometimes referred to as "SSL v3.1." (More detail about SSL can be found below in Section 5.7.)
Server Gated Cryptography (SGC)	Microsoft extension to SSL that provides strong encryption for online banking and other financial applications using RC2 (128-bit key), RC4 (128-bit key), DES (56-bit key), or 3DES (equivalent of 168-bit key). Use of SGC requires a Windows NT Server running Internet Information Server (IIS) 4.0 with a valid SGC certificate. SGC is available in 32-bit Windows versions of Internet Explorer (IE) 4.0, and support for Mac, Unix, and 16-bit Windows versions of IE is expected in the future.
Simple Authentication and Security Layer (SASL)	A framework for providing authentication and data security services in connection-oriented protocols (a la TCP), described in RFC 4422. It provides a structured interface and allows new protocols to reuse existing authentication mechanisms and allows old protocols to make use of new mechanisms. It has been common practice on the Internet to permit anonymous access to various services, employing a plain-text password using a user name of "anonymous" and a password of an email address or some other identifying information. New IETF protocols disallow plain-text logins. The Anonymous SASL Mechanism (RFC 4505) provides a method for anonymous logins within the SASL framework.
Simple Key-Management for Internet Protocol (SKIP)	Key management scheme for secure IP communication, specifically for IPsec, and designed by Aziz and Diffie. SKIP essentially defines a public key infrastructure for the Internet and even uses X.509 certificates. Most public key cryptosystems assign keys on a per-session basis, which is inconvenient for the Internet since IP is connectionless. Instead, SKIP provides a basis for secure communication between any pair of Internet hosts. SKIP can employ DES, 3DES, IDEA, RC2, RC5, MD5, and SHA-1. As it happened, SKIP was not adopted for IPsec; IKE was selected instead.
Transport Layer Security (TLS)	TLS v1.0 is an IETF specification (RFC 2246) intended to replace SSL v3.0. TLS v1.0 employs Triple-DES (secret key cryptography), SHA (hash), Diffie-Hellman (key exchange), and DSS (digital signatures). TLS v1.0 was vulnerable to attack and updated by v1.1 (RFC 4346) and v1.2 (RFC 5246); v1.3 is the most current working draft specification. TLS is designed to operate over TCP. The IETF developed the Datagram Transport Layer Security (DTLS) protocol to operate over UDP. DTLS v1.2 is described in RFC 6347. (See more detail about TLS below in Section 5.7.)
TrueCrypt	Open source, multi-platform cryptography software that can be used to encrypt a file, partition, or entire disk. One of TrueCrypt's more interesting features is that of *plausible deniability* with hidden volumes or hidden operating systems. The original Web site, *truecrypt.org*, suddenly went dark in May 2014; alternative sites have popped up, including CipherShed, TCnext, and VeraCrypt. (See more detail about TrueCrypt below in Section 5.11.)
X.509	ITU-T recommendation for the format of certificates for the

	public key infrastructure. Certificates map (bind) a user identity to a public key. The IETF application of X.509 certificates is documented in RFC 5280. An Internet X.509 Public Key Infrastructure is further defined in RFC 4210 (Certificate Management Protocols) and RFC 3647 (Certificate Policy and Certification Practices Framework).

5.1. Password Protection

Nearly all modern multiuser computer and network operating systems employ passwords at the very least to protect and authenticate users accessing computer and/or network resources. But passwords are *not* typically kept on a host or server in plaintext, but are generally encrypted using some sort of hash scheme.

A) /etc/passwd file

```
 root:Jbw6BwE4XoUHo:0:0:root:/root:/bin/bash
 carol:FM5ikbQt1K052:502:100:Carol Monaghan:/home/carol:/bin/bash
 alex:LqAi7Mdyg/HcQ:503:100:Alex Insley:/home/alex:/bin/bash
 gary:FkJXupRyFqY4s:501:100:Gary Kessler:/home/gary:/bin/bash
 todd:edGqQUAaGv7g6:506:101:Todd Pritsky:/home/todd:/bin/bash
 josh:FiH0ONcjPut1g:505:101:Joshua Kessler:/home/webroot:/bin/bash
```

B.1) /etc/passwd file (with shadow passwords)

```
 root:x:0:0:root:/root:/bin/bash
 carol:x:502:100:Carol Monaghan:/home/carol:/bin/bash
 alex:x:503:100:Alex Insley:/home/alex:/bin/bash
 gary:x:501:100:Gary Kessler:/home/gary:/bin/bash
 todd:x:506:101:Todd Pritsky:/home/todd:/bin/bash
 josh:x:505:101:Joshua Kessler:/home/webroot:/bin/bash
```

B.2) /etc/shadow file

```
 root:AGFw$1$P4u/uhLK$l2.HP35rlu65WlfCzq:11449:0:99999:7:::
 carol:kjHaN%35a8xMM8a/0kMl1?fwtLAM.K&kw.;:11449:0:99999:7:::
 alex:1$1KKmfTy0a7#3.LL9a8H71lkwn/.hH22a:11449:0:99999:7:::
 gary:9ajlknknKJHjhnu7298ypnAIJKL$Jh.hnk:11449:0:99999:7:::
 todd:798POJ90uab6.k$klPqMt%alMlprWqu6$.:11492:0:99999:7:::
 josh:Awmqpsui*787pjnsnJJK%aappaMpQo07.8:11492:0:99999:7:::
```

FIGURE 5: Sample entries in Unix/Linux password files.

Unix/Linux, for example, uses a well-known hash via its *crypt()* function. Passwords are stored in the */etc/passwd* file (Figure 5A); each record in the file contains the username, hashed password, user's individual and group numbers, user's name, home directory, and shell program; these fields are separated by colons (:). Note that each password is stored as a 13-byte string. The first two characters are actually a *salt*, randomness added to each password so that if two users have the same password, they will still be encrypted differently; the salt, in fact, provides a means so that a single password might have 4096 different encryptions. The remaining 11 bytes are the password hash, calculated using DES.

As it happens, the */etc/passwd* file is world-readable on Unix systems. This fact, coupled with the weak encryption of the passwords, resulted in the development of the *shadow password* system where passwords are kept in a separate, non-world-readable file used in conjunction with the normal password file. When shadow passwords are used, the password entry in */etc/passwd* is replaced with a "*" or "x" (Figure 5B.1) and the MD5 hash of the passwords are stored in */etc/shadow* along with some other account information (Figure 5B.2).

Windows NT uses a similar scheme to store passwords in the Security Access Manager (SAM) file. In the NT case, all passwords are hashed using the MD4 algorithm, resulting in a 128-bit (16-byte) hash value (they are then *obscured* using an undocumented mathematical transformation that was a secret until distributed on the Internet). The password password, for example, might be stored as the hash value (in hexadecimal) 60771b22d73c34bd4a290a79c8b09f18.

Passwords are not saved in plaintext on computer systems precisely so they cannot be easily compromised. For similar reasons, we don't want passwords sent in plaintext across a network. But for remote logon applications, how does a client system identify itself or a user to the server? One mechanism, of course, is to send the password as a hash value and that, indeed, may be done. A weakness of that approach, however, is that an intruder can grab the password off of the network and use an off-line attack (such as a *dictionary attack* where an attacker takes every known word and encrypts it with the network's encryption algorithm, hoping eventually to find a match with a purloined password hash). In some situations, an attacker only has to copy the hashed password value and use it later on to gain unauthorized entry without ever learning the actual password.

An even stronger authentication method uses the password to modify a shared secret between the client and server, but never allows the password in any form to go across the network. This is the basis for the Challenge Handshake Authentication Protocol (CHAP), the remote logon process used by Windows NT.

As suggested above, Windows NT passwords are stored in a security file on a server as a 16-byte hash value. In truth, Windows NT stores *two* hashes; a weak hash based upon the old LAN Manager (LanMan) scheme and the newer NT hash. When a user logs on to a server from a remote workstation, the user is identified by the username, sent across the network in plaintext (no worries here; it's not a secret anyway!). The server then generates a 64-bit random number and sends it to the client (also in plaintext). This number is the *challenge*.

Using the LanMan scheme, the client system then encrypts the challenge using DES. Recall that DES employs a 56-bit key, acts on a 64-bit block of data, and produces a 64-bit output. In this case, the 64-bit data block is the random number. The client actually uses three different DES keys to encrypt the random number, producing three different 64-bit outputs. The first key is the first seven bytes (56 bits) of the password's hash value, the second key is the next seven bytes in the password's hash, and the third key is the remaining two bytes of the password's hash concatenated with five zero-filled bytes. (So, for the example above, the three DES keys would be 60771b22d73c34, bd4a290a79c8b0, and 9f180000000000.) Each key is applied to the random number resulting in three 64-bit outputs, which comprise the *response*. Thus, the server's 8-byte challenge yields a 24-byte response from the client and this is all that would be seen on the network. The server, for its part, does the same calculation to ensure that the values match.

There is, however, a significant weakness to this system. Specifically, the response is generated in such a way as to effectively reduce 16-byte hash to three smaller hashes, of length seven, seven, and two. Thus, a password cracker has to break at most a 7-byte hash. One Windows NT vulnerability test program that I have used in the past will report passwords that are "too short," defined as "less than 8 characters." When I asked how the program knew that passwords were too short, the software's salespeople suggested to me that the program broke the passwords to determine their length. This is undoubtedly not true; all the software really has to do is look at the second 7-byte block and some known value indicates that it is empty, which would indicate a password of seven or less characters.

Consider the following example, showing the LanMan hash of two different short passwords (take a close look at the last 8 bytes):

AA: 89D42A44E77140AAAAD3B435B51404EE
AAA: 1C3A2B6D939A1021AAD3B435B51404EE

Note that the NT hash provides no such clue:

AA: C5663434F963BE79C8FD99F535E7AAD8
AAA: 6B6E0FB2ED246885B98586C73B5BFB77

It is worth noting that the discussion above describes the Microsoft version of CHAP, or MS-CHAP (MS-CHAPv2 is described in RFC 2759). MS-CHAP assumes that it is working with hashed values of the password as the key to encrypting the challenge. More traditional CHAP (RFC 1994) assumes that it is starting with passwords in plaintext. The relevance of this observation is that a CHAP client, for example, cannot be authenticated by an MS-CHAP server; both client and server must use the same CHAP version.

5.2. Some of the Finer Details of Diffie-Hellman

Diffie and Hellman introduced the concept of public-key cryptography. The mathematical "trick" of Diffie-Hellman key exchange is that it is relatively easy to compute exponents compared to computing discrete logarithms. Diffie-Hellman allows two parties — the ubiquitous Alice and Bob — to generate a secret key; they need to exchange some information over an unsecure communications channel to perform the calculation but an eavesdropper cannot determine the shared secret key based upon this information.

Diffie-Hellman works like this. Alice and Bob start by agreeing on a large prime number, N. They also have to choose some number G so that G<N.

There is actually another constraint on G, namely that it must be primitive with respect to N. *Primitive* is a definition that is a little beyond the scope of our discussion but basically G is primitive to N if the set of N-1 values of G^i mod N for i = (1,N-1) are all different. As an example, 2 is not primitive to 7 because the set of powers of 2 from 1 to 6, mod 7 (i.e., 2^1 mod 7, 2^2 mod 7, ..., 2^6 mod 7) = {2,4,1,2,4,1}. On the other hand, 3 is primitive to 7 because the set of powers of 3 from 1 to 6, mod 7 = {3,2,6,4,5,1}.

(The definition of primitive introduced a new term to some readers, namely *mod*. The phrase *x mod y* (and read as written!) means "take the remainder after dividing x by y." Thus, 1 mod 7 = 1, 9 mod 6 = 3, and 8 mod 8 = 0. Read more about the *modulo* function in the appendix.)

Anyway, either Alice or Bob selects N and G; they then tell the other party what the values are. Alice and Bob then work independently:

Alice...

1. Choose a large random number, X_A < N. This is Alice's private key.
2. Compute $Y_A = G^{X_A}$ mod N. This is Alice's public key.
3. Exchange public key with Bob.

Bob...

1. Choose a large random number, X_B < N. This is Bob's private key.
2. Compute $Y_B = G^{X_B}$ mod N. This is Bob's public key.
3. Exchange public key with Alice.

4. Compute $K_A = Y_B^{X_A}$ mod N

4. Compute $K_B = Y_A^{X_B}$ mod N

Note that X_A and X_B are kept secret while Y_A and Y_B are openly shared; these are the private and public keys, respectively. Based on their own private key and the public key learned from the other party, Alice and Bob have computed their secret keys, K_A and K_B, respectively, which are equal to $G^{X_A X_B}$ mod N.

Perhaps a small example will help here. Although Alice and Bob will really choose large values for N and G, I will use small values for example only; let's use N=7 and G=3.

Alice...

1. Choose $X_A = 2$
2. Calculate $Y_A = 3^2 \bmod 7 = 2$
3. Exchange public keys with Bob
4. $K_A = 6^2 \bmod 7 = 1$

Bob...

1. Choose $X_B = 3$
2. Calculate $Y_B = 3^3 \bmod 7 = 6$
3. Exchange public keys with Alice
4. $K_B = 2^3 \bmod 7 = 1$

In this example, then, Alice and Bob will both find the secret key 1 which is, indeed, 3^6 mod 7 (i.e., $G^{X_A X_B} = 3^{2x3}$). If an eavesdropper (Mallory) was listening in on the information exchange between Alice and Bob, he would learn G, N, Y_A, and Y_B which is a lot of information but insufficient to compromise the key; as long as X_A and X_B remain unknown, K is safe. As said above, calculating $Y = G^X$ is a lot easier than finding $X = \log_G Y$.

> **A short digression on modulo arithmetic.** In the paragraph above, we noted that 3^6 mod 7 = 1. This can be confirmed, of course, by noting that:
>
> $$3^6 = 729 = 104*7 + 1$$
>
> There is a nice property of modulo arithmetic, however, that makes this determination a little easier, namely: (a mod x)(b mod x) = (ab mod x). Therefore, one possible shortcut is to note that $3^6 = (3^3)(3^3)$. Therefore, 3^6 mod 7 = (3^3 mod 7)(3^3 mod 7) = (27 mod 7)(27 mod 7) = 6*6 mod 7 = 36 mod 7 = 1.

Diffie-Hellman can also be used to allow key sharing amongst multiple users. Note again that the Diffie-Hellman algorithm is used to generate secret keys, not to encrypt and decrypt messages.

5.3. Some of the Finer Details of RSA Public-Key Cryptography

Unlike Diffie-Hellman, RSA can be used for key exchange as well as digital signatures and the encryption of small blocks of data. Today, RSA is primarily used to encrypt the session key used for secret key encryption (message integrity) or the message's hash value (digital signature). RSA's mathematical hardness comes from the ease in calculating large numbers and the difficulty in finding the prime factors of those large numbers. Although employed with numbers using hundreds of digits, the math behind RSA is relatively straight-forward.

To create an RSA public/private key pair, here are the basic steps:

1. Choose two prime numbers, p and q. From these numbers you can calculate the modulus, n = pq.
2. Select a third number, e, that is relatively prime to (i.e., it does not divide evenly into) the product (p-1)(q-1). The number e is the public exponent.
3. Calculate an integer d from the quotient (ed-1)/[(p-1)(q-1)]. The number d is the private exponent.

The public key is the number pair (n,e). Although these values are publicly known, it is computationally infeasible to determine d from n and e if p and q are large enough.

To encrypt a message, M, with the public key, create the ciphertext, C, using the equation:

$$C = M^e \bmod n$$

The receiver then decrypts the ciphertext with the private key using the equation:

$$M = C^d \bmod n$$

Now, this might look a bit complex and, indeed, the mathematics does take a lot of computer power given the large size of the numbers; since p and q may be 100 digits (decimal) or more, d and e will be about the same size and n may be over 200 digits. Nevertheless, a simple example may help. In this example, the values for p, q, e, and d are purposely chosen to be very small and the reader will see exactly how badly these values perform, but hopefully the algorithm will be adequately demonstrated:

1. Select p=3 and q=5.
2. The modulus n = pq = 15.

3. The value e must be relatively prime to (p-1)(q-1) = (2)(4) = 8. Select e=11
4. The value d must be chosen so that (ed-1)/[(p-1)(q-1)] is an integer. Thus, the value (11d-1)/[(2)(4)] = (11d-1)/8 must be an integer. Calculate one possible value, d=3.
5. Let's say we wish to send the string **SECRET**. For this example, we will convert the string to the decimal representation of the ASCII values of the characters, which would be **83 69 67 82 69 84**.
6. The sender encrypts each digit one at a time (we have to because the modulus is so small) using the public key value (e,n)=(11,15). Thus, each ciphertext character $C_i = M_i^{11}$ mod 15. The input digit string **0x836967826984** will be transmitted as **0x2c696d286924**.
7. The receiver decrypts each digit using the private key value (d,n)=(3,15). Thus, each plaintext character $M_i = C_i^3$ mod 15. The input digit string **0x2c696d286924** will be converted to **0x836967826984** and, presumably, reassembled as the plaintext string **SECRET**.

Again, the example above uses small values for simplicity and, in fact, shows the weakness of small values; note that 4, 6, and 9 do not change when encrypted, and that the values 2 and 8 encrypt to 8 and 2, respectively. Nevertheless, this simple example demonstrates how RSA can be used to exchange information.

RSA keylengths of 512 and 768 bits are considered to be pretty weak. The minimum suggested RSA key is 1024 bits; 2048 and 3072 bits are even better.

As an aside, Adam Back (http://www.cypherspace.org/~adam/) wrote a two-line Perl script to implement RSA. It employs dc, an arbitrary precision arithmetic package that ships with most UNIX systems:

```
print pack"C*",split/\D+/,`echo "16iII*o\U@{$/=$z;[(pop,pop,unpack"H*",<>
)]}\EsMsKsN0[lN*1lK[d2%Sa2/d0<X+d*lMLa^*lN%0]dsXx++lMlN/dsM0<J]dsJxp"|dc`
```

5.4. Some of the Finer Details of DES, Breaking DES, and DES Variants

The Data Encryption Standard (DES) started life in the mid-1970s, adopted by the National Bureau of Standards (NBS) [now the National Institute for Standards and Technology (NIST)] as Federal Information Processing Standard 46 (FIPS 46-3) and by the American National Standards Institute (ANSI) as X3.92.

As mentioned earlier, DES uses the Data Encryption Algorithm (DEA), a secret key block-cipher employing a 56-bit key operating on 64-bit blocks. FIPS 81 describes four modes of DES operation: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB). Despite all of these options, ECB is the most commonly deployed mode of operation.

NIST finally declared DES obsolete in 2004, and withdrew FIPS 46-3, 74, and 81 (*Federal Register*, July 26, 2004, *69*(142), 44509-44510). Although other block ciphers have replaced DES, it is still interesting to see how DES encryption is performed; not only is it sort of neat, but DES was the first crypto scheme commonly seen in non-governmental applications and was the catalyst for modern "public" cryptography and the first public Feistel cipher. DES still remains in many products — and cryptography students and cryptographers will continue to study DES for years to come.

DES Operational Overview

DES uses a 56-bit key. In fact, the 56-bit key is divided into eight 7-bit blocks and an 8th odd parity bit is added to each block (i.e., a "0" or "1" is added to the block so that there are an odd number of 1 bits in each 8-bit block). By using the 8 parity bits for rudimentary error detection, a DES key is actually 64 bits in length for computational purposes although it only has 56 bits worth of randomness, or *entropy* (See Section A.3 for a brief discussion of entropy and information theory).

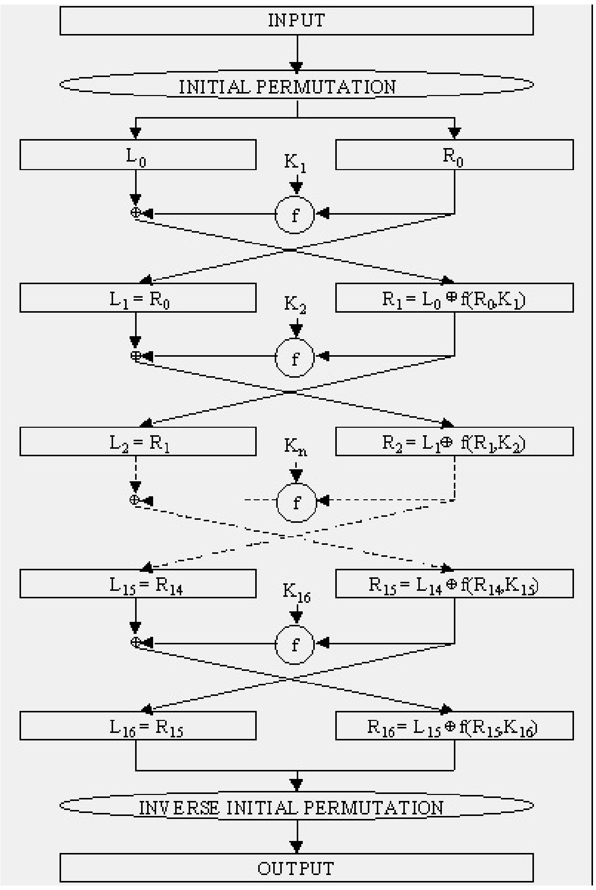


FIGURE 6: DES enciphering algorithm.

DES then acts on 64-bit blocks of the plaintext, invoking 16 rounds of permutations, swaps, and substitutes, as shown in Figure 6. The standard includes tables describing all of the selection, permutation, and expansion operations mentioned below; these aspects of the algorithm are not secrets. The basic DES steps are:

1. The 64-bit block to be encrypted undergoes an initial permutation (IP), where each bit is moved to a new bit position; e.g., the 1st, 2nd, and 3rd bits are moved to the 58th, 50th, and 42nd position, respectively.

2. The 64-bit permuted input is divided into two 32-bit blocks, called *left* and *right*, respectively. The initial values of the left and right blocks are denoted L_0 and R_0.

3. There are then 16 rounds of operation on the L and R blocks. During each iteration (where *n* ranges from 1 to 16), the following formulae apply:

   $L_n = R_{n-1}$
   $R_n = L_{n-1}$ XOR $f(R_{n-1},K_n)$

   At any given step in the process, then, the new L block value is merely taken from the prior R block value. The new R block is calculated by taking the bit-by-bit exclusive-OR (XOR) of the prior L block with the results of applying the DES cipher function, *f*, to the prior R block and K_n. (K_n is a 48-bit value derived from the 64-bit DES key. Each round uses a different 48 bits according to the standard's Key Schedule algorithm.)

   The cipher function, f, combines the 32-bit R block value and the 48-bit subkey in the following way. First, the 32 bits in the R block are expanded to 48 bits by an expansion function (E); the extra 16 bits are found by repeating the bits in 16 predefined positions. The 48-bit expanded R-block is then ORed with the 48-bit subkey. The result is a 48-bit value that is then divided into eight 6-bit blocks. These are fed as input into 8 selection (S) boxes, denoted S_1,...,S_8. Each 6-bit input yields a 4-bit output using a table lookup based on the 64 possible inputs; this results in a 32-bit output from the S-box. The 32 bits are then rearranged by a permutation function (P), producing the results from the cipher function.

4. The results from the final DES round — i.e., L_{16} and R_{16} — are recombined into a 64-bit value and fed into an inverse initial permutation (IP^{-1}). At this step, the bits are rearranged into their original positions, so that the 58th, 50th, and 42nd bits, for example, are moved back into the 1st, 2nd, and 3rd positions, respectively. The output from IP^{-1} is the 64-bit ciphertext block.

Consider this example with the given 56-bit key and input:

Key: **1100101 0100100 1001001 0011101 0110101 0101011 1101100 0011010**

Input character string: **GoAggies**

Input bit string: **11100010 11110110 10000010 11100110 11100110 10010110 10100110 11001110**

Output bit string: **10011111 11110010 10000000 10000001 01011011 00101001 00000011 00101111**
Output character string: **ùOÚ”Àô**

Breaking DES

The mainstream cryptographic community has long held that DES's 56-bit key was too short to withstand a brute-force attack from modern computers. Remember Moore's Law: computer power doubles every 18 months. Given that increase in power, a key that could withstand a brute-force guessing attack in 1975 could hardly be expected to withstand the same attack a quarter century later.

DES is even more vulnerable to a brute-force attack because it is often used to encrypt words, meaning that the entropy of the 64-bit block is, effectively, greatly reduced. That is, if we are encrypting random bit streams, then a given byte might contain any one of 2^8 (256) possible values and the entire 64-bit block has 2^{64}, or about 18.5 quintillion, possible values. If we are encrypting words, however, we are most likely to find a limited set of bit patterns; perhaps 70 or so if we account for upper and lower case letters, the numbers, space, and some punctuation. This means that only about ¼ of the bit combinations of a given byte are likely to occur.

Despite this criticism, the U.S. government insisted throughout the mid-1990s that 56-bit DES was secure and virtually unbreakable if appropriate precautions were taken. In response, RSA Laboratories sponsored a series of cryptographic challenges to prove that DES was no longer appropriate for use.

DES Challenge I was launched in March 1997. It was completed in 84 days by R. Verser in a collaborative effort using thousands of computers on the Internet.

The first DES II challenge lasted 40 days in early 1998. This problem was solved by distributed.net, a worldwide distributed computing network using the spare CPU cycles of computers around the Internet (participants in distributed.net's activities load a client program that runs in the background, conceptually similar to the SETI @Home "Search for Extraterrestrial Intelligence" project). The distributed.net systems were checking 28 *billion* keys per second by the end of the project.

The second DES II challenge lasted less than 3 days. On July 17, 1998, the Electronic Frontier Foundation (EFF) announced the construction of hardware that could brute-force a DES key in an average of 4.5 days. Called Deep Crack, the device could check 90 billion keys per second and cost only about $220,000 including design (it was erroneously and widely reported that subsequent devices could be built for as little as $50,000). Since the design is scalable, this suggests that an organization could build a DES cracker that could break 56-bit keys in an average of a day for as little as $1,000,000. Information about the hardware design and all software can be obtained from the EFF.

The DES III challenge, launched in January 1999, was broken is less than a day by the combined efforts of Deep Crack and distributed.net. This is widely considered to have been the final nail in DES's coffin.

The Deep Crack algorithm is actually quite interesting. The general approach that the DES Cracker Project took was not to break the algorithm mathematically but instead to launch a brute-force attack by guessing every possible key. A 56-bit key yields 2^{56}, or about 72 quadrillion, possible values. So the DES cracker team looked for any shortcuts they could find! First, they assumed that *some* recognizable plaintext would appear in the decrypted string even though they didn't have a specific known plaintext block. They then applied all 2^{56} possible key values to the 64-bit block (I don't mean to make this sound simple!). The system checked to see if the decrypted value of the block was "interesting," which they defined as bytes containing one of the alphanumeric characters, space, or some punctuation. Since the likelihood of a single byte being "interesting" is about ¼, then the likelihood of the entire 8-byte stream being "interesting" is about $¼^8$, or 1/65536 ($½^{16}$). This dropped the number of possible keys that might yield positive results to about 2^{40}, or about a trillion.

They then made the assumption that an "interesting" 8-byte block would be followed by another "interesting" block. So, if the first block of ciphertext decrypted to something interesting, they decrypted the next block; otherwise, they abandoned this key. Only if the second block was also "interesting" did they examine the key closer. Looking for 16 consecutive bytes that were "interesting" meant that only 2^{24}, or 16 million, keys needed to be examined further. This further examination was primarily to see if the text made any sense. Note that possible "interesting" blocks might be 1hJ5&aB7; or DEPOSITS; the latter is more likely to produce a better result. And even a slow laptop today can search through lists of only a few million items in a relatively short period of time. (Interested readers are urged to read *Cracking DES* and EFF's Cracking DES page.)

It is well beyond the scope of this paper to discuss other forms of breaking DES and other codes. Nevertheless, it is worth mentioning a couple of forms of cryptanalysis that have been shown to be effective against DES. *Differential cryptanalysis*, invented in 1990 by E. Biham and A. Shamir (of RSA fame), is a chosen-plaintext attack. By selecting pairs of plaintext with particular differences, the cryptanalyst examines the differences in the resultant ciphertext pairs. *Linear plaintext*, invented by M. Matsui, uses a linear approximation to analyze the actions of a block cipher (including DES). Both of these attacks can be more efficient than brute force.

DES Variants

Once DES was "officially" broken, several variants appeared. But none of them came overnight; work at hardening DES had already been underway. In the early 1990s, there was a proposal to increase the security of DES by effectively increasing the key length by using multiple keys with multiple passes. But for this scheme to work, it had to first be shown that the DES function is **not** a *group*, as defined in mathematics. If DES was a group, then we could show that for two DES keys, X1 and X2, applied to some plaintext (P), we can find a single equivalent key, X3, that would provide the same result; i.e.,

$$E_{X2}(E_{X1}(P)) = E_{X3}(P)$$

where $E_X(P)$ represents DES encryption of some plaintext *P* using DES key *X*. If DES were a group, it wouldn't matter how many keys and passes we applied to some plaintext; we could always find a single 56-bit key that would provide the same result.

As it happens, DES was proven to not be a group so that as we apply additional keys and passes, the effective key length increases. One obvious choice, then, might be to use two keys and two passes, yielding an effective key length of 112 bits. Let's call this Double-DES. The two keys, Y1 and Y2, might be applied as follows:

$$C = E_{Y2}(E_{Y1}(P))$$
$$P = D_{Y1}(D_{Y2}(C))$$

where $E_Y(P)$ and $D_Y(C)$ represent DES encryption and decryption, respectively, of some plaintext *P* and ciphertext *C*, respectively, using DES key *Y*.

So far, so good. But there's an interesting attack that can be launched against this "Double-DES" scheme. First, notice that the applications of the formula above can be thought of with the following individual steps (where C' and P' are intermediate results):

$$C' = E_{Y1}(P) \text{ and } C = E_{Y2}(C')$$
$$P' = D_{Y2}(C) \text{ and } P = D_{Y1}(P')$$

Unfortunately, C'=P'. That leaves us vulnerable to a simple *known plaintext* attack (sometimes called "Meet-in-the-middle") where the attacker knows some plaintext (P) and its matching ciphertext (C). To obtain C', the attacker needs to try all 2^{56} possible values of Y1 applied to P; to obtain P', the attacker needs to try all 2^{56} possible values of Y2 applied to C. Since C'=P', the attacker knows when a match has been achieved — after only $2^{56} + 2^{56} = 2^{57}$ key searches, only twice the work of brute-forcing DES. So "Double-DES" is not a good solution.

Triple-DES (3DES), based upon the Triple Data Encryption Algorithm (TDEA), is described in FIPS 46-3. 3DES, which is not susceptible to a meet-in-the-middle attack, employs three DES passes and one, two, or three keys called K1, K2, and K3. Generation of the ciphertext (C) from a block of plaintext (P) is accomplished by:

$$C = E_{K3}(D_{K2}(E_{K1}(P)))$$

where $E_K(P)$ and $D_K(P)$ represent DES encryption and decryption, respectively, of some plaintext *P* using DES key *K*. (For obvious reasons, this is sometimes referred to as an *encrypt-decrypt-encrypt mode* operation.)

Decryption of the ciphertext into plaintext is accomplished by:

$$P = D_{K1}(E_{K2}(D_{K3}(C)))$$

The use of three, independent 56-bit keys provides 3DES with an effective key length of 168 bits. The specification also defines use of two keys where, in the operations above, K3 = K1; this provides an effective key length of 112 bits. Finally, a third keying option is to use a single key, so that K3 = K2 = K1 (in this case, the effective key length is 56 bits and 3DES applied to some plaintext, P, will yield the same ciphertext, C, as normal DES would with that same key). Given the relatively low cost of key storage and the modest increase in processing due to the use of longer keys, the best recommended practices are that 3DES be employed with three keys.

Another variant of DES, called DESX, is due to Ron Rivest. Developed in 1996, DESX is a very simple algorithm that greatly increases DES's resistance to brute-force attacks without increasing its computational complexity. In DESX, the plaintext input is XORed with 64 additional key bits prior to encryption and the output is likewise XORed with the 64 key bits. By adding just two XOR operations, DESX has an effective keylength of 120 bits against an exhaustive key-search attack. As it happens, DESX is no more immune to other types of more sophisticated attacks, such as differential or linear cryptanalysis, but brute-force is the primary attack vector on DES.

Closing Comments

Although DES has been deprecated and replaced by the Advanced Encryption Standard (AES) because of its vulnerability to a modestly-priced brute-force attack, many applications continue to rely on DES for security, and many software designers and implementers continue to include DES in new applications. In some cases, use of DES is wholly appropriate but, in general, DES should not continue to be promulgated in production software and hardware. RFC 4772 discusses the security implications of employing DES.

On a final note, readers may be interested in seeing an Excel implementation of DES or J.O. Grabbe's The DES Algorithm Illustrated.

5.5. Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) is one of today's most widely used public key cryptography programs. Developed by Philip Zimmermann in the early 1990s and long the subject of controversy, PGP is available as a plug-in for many e-mail clients, such as Claris Emailer, Microsoft Outlook/Outlook Express, and Qualcomm Eudora.

PGP can be used to sign or encrypt e-mail messages with the mere click of the mouse. Depending upon the version of PGP, the software uses SHA or MD5 for calculating the message hash; CAST, Triple-DES, or IDEA for encryption; and RSA or DSS/Diffie-Hellman for key exchange and digital signatures.

When PGP is first installed, the user has to create a key-pair. One key, the public key, can be advertised and widely circulated. The private key is protected by use of a *passphrase*. The passphrase has to be entered every time the user accesses their private key.

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Hi Carol.

What was that pithy Groucho Marx quote?

/kess

-----BEGIN PGP SIGNATURE-----
Version: PGP for Personal Privacy 5.0
Charset: noconv

iQA/AwUBNFUdO5WOcz5SFtuEEQJx/ACaAgR97+vvDU6XWELV/GANjAAgBtUAnjG3
Sdfw2JgmZIOLNjFe7jP0Y8/M
=jUAU
-----END PGP SIGNATURE-----
```

FIGURE 7: A PGP signed message. The sender uses their private key; at the destination, the sender's e-mail address yields the public key from the receiver's keyring.

Figure 7 shows a PGP signed message. This message will not be kept secret from an eavesdropper, but a recipient can be assured that the message has not been altered from what the sender transmitted. In this instance, the sender signs the message using their own private key. The receiver uses the sender's public key to verify the signature; the public key is taken from the receiver's keyring based on the sender's e-mail address. Note that the signature process does not work unless the sender's public key is on the receiver's keyring.

```
-----BEGIN PGP MESSAGE-----
Version: PGP for Personal Privacy 5.0
MessageID: DAdVB3wzpBr3YRunZwYvhK5gBKBXOb/m

qANQR1DBwU4D/TlT68XXuiUQCADfj2o4b4aFYBcWumA7hR1Wvz9rbv2BR6WbEUsy
ZBIEFtjyqCd96qF38sp9IQiJIKlNaZfx2GLRWikPZwchUXxB+AA5+lqsG/ELBvRa
c9XefaYpbbAZ6z6LkOQ+eE0XASe7aEEPfdxvZZT37dVyiyxuBBRYNLN8Bphdr2zv
z/9Ak4/OLnLiJRk05/2UNE5Z0a+3lcvITMmfGajvRhkXqocavPOKiin3hv7+Vx88
uLLem2/fQHZhGcQvkqZVqXx8SmNw5gzuvwjV1WHj9muDGBY0MkjiZIRI7azWnoU9
3KCnmpR60VO4rDRAS5uGl9fioSvze+q8XqxubaNsgdKkoD+tB/4u4c4tznLfw1L2
YBS+dzFDw5desMFSo7JkecAS4NB9jAu9K+f7PTAsesCBNETDd49BTOFFTWWavAfE
gLYcPrcn4s3EriUgvL3OzPR4P1chNu6sa3ZJkTBbriDoA3VpnqG3hxqfNyOlqAka

mJJuQ53Ob9ThaFH8YcE/VqUFdw+bQtrAJ6NpjIxi/x0FfOInhC/bBw7pDLXBFNaX
HdlLQRPQdrmnWskKznOSarxq4GjpRTQo4hpCRJJ5aU7tZO9HPTZXFG6iRIT0wa47


AR5nvkEKoIAjW5HaDKiJriuWLdtN4OXecWvxFsjR32ebz76U8aLpAK87GZEyTzBx
dV+lH0hwyT/y1cZQ/E5USePP4oKWF4uqquPee1OPeFMBo4CvuGyhZXD/18Ft/53Y
WIebvdiCqsOoabK3jEfdGExce63zDI0=
=MpRf
-----END PGP MESSAGE-----
```

FIGURE 8: A PGP encrypted message. The receiver's e-mail address is the pointer to the public key in the sender's keyring. At the destination side, the receiver uses their own private key.

Figure 8 shows a PGP encrypted message (PGP compresses the file, where practical, prior to encryption because encrypted files have a high degree of randomness and, therefore, cannot be efficiently compressed). In this example, public key methods are used to exchange the session key for the actual message encryption that employs secret-key cryptography. In this case, the receiver's e-mail address is the pointer to the public key in the sender's keyring; in fact, the same message can be sent to multiple recipients and the message will not be significantly longer since all that needs to be added is the session key encrypted by each receiver's public key. When the message is received, the recipient will use their private key to extract the session secret key to successfully decrypt the message (Figure 9).

```
Hi Gary,

"Outside of a dog, a book is man's best friend.
Inside of a dog, it's too dark to read."
```

```
Carol
```

FIGURE 9: The decrypted message.

It is worth noting that PGP was one of the first so-called "hybrid cryptosystems" that combined aspects of SKC and PKC. When Zimmermann was first designing PGP in the late-1980s, he wanted to use RSA to encrypt the entire message. The PCs of the days, however, suffered significant performance degradation when executing RSA so he hit upon the idea of using SKC to encrypt the message and PKC to encrypt the SKC key.

PGP went into a state of flux in 2002. Zimmermann sold PGP to Network Associates, Inc. (NAI) in 1997 and himself resigned from NAI in early 2001. In March 2002, NAI announced that they were dropping support for the commercial version of PGP having failed to find a buyer for the product willing to pay what NAI wanted. In August 2002, PGP was purchased from NAI by PGP Corp. which, in turn, was purchased by Symantec. Meanwhile, there are many freeware versions of PGP available through the International PGP Page and the OpenPGP Alliance. Also check out the GNU Privacy Guard (GnuPG), a GNU project implementation of OpenPGP (defined in RFC 2440).

5.6. IP Security (IPsec) Protocol

> **NOTE:** The information in this section assumes that the reader is familiar with the Internet Protocol (IP), at least to the extent of the packet format and header contents. More information about IP can be found in *An Overview of TCP/IP Protocols and the Internet*. More information about IPv6 can be found in IPv6: The Next Generation Internet Protocol.

The Internet and the TCP/IP protocol suite were not built with security in mind. This statement is not meant as a criticism; the baseline UDP, TCP, IP, and ICMP protocols were written in 1980 and built for the relatively closed ARPANET community. TCP/IP wasn't designed for the commercial-grade financial transactions that they now see nor for virtual private networks (VPNs) on the Internet. To bring TCP/IP up to today's security necessities, the Internet Engineering Task Force (IETF) formed the IP Security Protocol Working Group which, in turn, developed the IP Security (IPsec) protocol. IPsec is not a single protocol, in fact, but a suite of protocols providing a mechanism to provide data integrity, authentication, privacy, and nonrepudiation for the classic Internet Protocol (IP). Although intended primarily for IP version 6 (IPv6), IPsec can also be employed by the current version of IP, namely IP version 4 (IPv4).

As shown in Table 3, IPsec is described in nearly a dozen RFCs. RFC 4301, in particular, describes the overall IP security architecture and RFC 2411 provides an overview of the IPsec protocol suite and the documents describing it.

IPsec can provide either message authentication and/or encryption. The latter requires more processing than the former, but will probably end up being the preferred usage for applications such as VPNs and secure electronic commerce.

Central to IPsec is the concept of a *security association (SA)*. Authentication and confidentiality using AH or ESP use SAs and a primary role of IPsec key exchange it to establish and maintain SAs. An SA is a simplex (one-way or unidirectional) logical connection between two communicating IP endpoints that provides security services to the traffic carried by it using either AH or ESP procedures. The endpoint of an SA can be an IP host or IP security gateway (e.g., a proxy server, VPN server, etc.). Providing security to the more typical scenario of two-way (bi-directional) communication between two endpoints requires the establishment of two SAs (one in each direction).

An SA is uniquely identified by a 3-tuple composed of:

- Security Parameter Index (SPI), a 32-bit identifier of the connection
- IP Destination Address
- security protocol (AH or ESP) identifier

The IP Authentication Header (AH), described in RFC 4302, provides a mechanism for data integrity and data origin authentication for IP packets using HMAC with MD5 (RFC 2403), HMAC with SHA-1 (RFC 2404), or HMAC with RIPEMD (RFC 2857). See also RFC 4305.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Next Header   |  Payload Len  |          RESERVED             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Security Parameters Index (SPI)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Sequence Number Field                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                Integrity Check Value-ICV (variable)           |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

FIGURE 10: IPsec Authentication Header format. ***(From RFC 4302)***

Figure 10 shows the format of the IPsec AH. The AH is merely an additional header in a packet, more or less representing another protocol layer above IP (this is shown in Figure 12 below). Use of the IP AH is indicated by placing the value 51 (0x33) in the IPv4 Protocol or IPv6 Next Header field in the IP packet header. The AH follows mandatory IPv4/IPv6 header fields and precedes higher layer protocol (e.g., TCP, UDP) information. The contents of the AH are:

- *Next Header:* An 8-bit field that identifies the type of the next payload after the Authentication Header.
- *Payload Length:* An 8-bit field that indicates the length of AH in 32-bit words (4-byte blocks), minus "2". [The rationale for this is somewhat counter intuitive but technically important. All IPv6 extension headers encode the header extension length (Hdr Ext Len) field by first subtracting 1 from the header length, which is measured in 64-bit words. Since AH was originally developed for IPv6, it is an IPv6 extension header. Since its length is measured in 32-bit words, however, the Payload Length is calculated by subtracting 2 (32 bit words) to maintain consistency with IPv6 coding rules.] In the default case, the three 32-bit word fixed portion of the AH is followed by a 96-bit authentication value, so the Payload Length field value would be 4.
- *Reserved:* This 16-bit field is reserved for future use and always filled with zeros.
- *Security Parameters Index (SPI):* An arbitrary 32-bit value that, in combination with the destination IP address and security protocol, uniquely identifies the Security Association for this datagram. The value 0 is reserved for local, implementation-specific uses and values between 1-255 are reserved by the Internet Assigned Numbers Authority (IANA) for future use.
- *Sequence Number:* A 32-bit field containing a sequence number for each datagram; initially set to 0 at the establishment of an SA. AH uses sequence numbers as an anti-replay mechanism, to prevent a "person-in-the-middle" attack. If anti-replay is enabled (the default), the transmitted Sequence Number is never allowed to cycle back to 0; therefore, the sequence number must be reset to 0 by establishing a new SA prior to the transmission of the 2^{32}nd packet.
- *Authentication Data:* A variable-length, 32-bit aligned field containing the Integrity Check Value (ICV) for this packet (default length = 96 bits). The ICV is computed using the authentication algorithm specified by the SA, such as DES, MD5, or SHA-1. Other algorithms *may* also be supported.

The IP Encapsulating Security Payload (ESP), described in RFC 4303, provides message integrity and privacy mechanisms in addition to authentication. As in AH, ESP uses HMAC with MD5, SHA-1, or RIPEMD authentication (RFC 2403/RFC 2404/RFC 2857); privacy is provided using DES-CBC encryption (RFC 2405), NULL encryption (RFC 2410), other CBC-mode algorithms (RFC 2451), or AES (RFC 3686). See also RFC 4305 and RFC 4308.

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ ----
 |               Security Parameters Index (SPI)                 | ^Int.
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |Cov-
 |                      Sequence Number                          | |ered
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ | ----
 |                    Payload Data* (variable)                   | |   ^
 ~                                                               ~ |   |
 |                                                               | |Conf.
 +               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |Cov-
 |               |     Padding (0-255 bytes)                     | |ered*
 +-+-+-+-+-+-+-+-+               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ |   |
 |                               |  Pad Length   | Next Header   | v   v

 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+ ------
 |         Integrity Check Value-ICV   (variable)                |
 ~                                                               ~
 |                                                               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

     * If included in the Payload field, cryptographic synchronization
       data, e.g., an Initialization Vector (IV), usually is not
       encrypted per se, although it often is referred to as being
       being part of the ciphertext.
```

FIGURE 11: IPsec Encapsulating Security Payload format. ***(From RFC 4303)***

Figure 11 shows the format of the IPsec ESP information. Use of the IP ESP format is indicated by placing the value 50 (0x32) in the IPv4 Protocol or IPv6 Next Header field in the IP packet header. The ESP header (i.e., SPI and sequence number) follows mandatory IPv4/IPv6 header fields and precedes higher layer protocol (e.g., TCP, UDP) information. The contents of the ESP packet are:

- *Security Parameters Index:* (see description for this field in the AH, above.)
- *Sequence Number:* (see description for this field in the AH, above.)
- *Payload Data:* A variable-length field containing data as described by the Next Header field. The contents of this field could be encrypted higher layer data or an encrypted IP packet.
- *Padding:* Between 0 and 255 octets of padding may be added to the ESP packet. There are several applications that might use the padding field. First, the encryption algorithm that is used may require that the plaintext be a multiple of some number of bytes, such as the block size of a block cipher; in this case, the Padding field is used to fill the plaintext to the size required by the algorithm. Second, padding may be

required to ensure that the ESP packet and resulting ciphertext terminate on a 4-byte boundary. Third, padding may be used to conceal the actual length of the payload. Unless another value is specified by the encryption algorithm, the Padding octets take on the value 1, 2, 3, ... starting with the first Padding octet. This scheme is used because, in addition to being simple to implement, it provides some protection against certain forms of "cut and paste" attacks.

- *Pad Length:* An 8-bit field indicating the number of bytes in the Padding field; contains a value between 0-255.
- *Next Header:* An 8-bit field that identifies the type of data in the Payload Data field, such as an IPv6 extension header or a higher layer protocol identifier.
- *Authentication Data:* (see description for this field in the AH, above.)

Two types of SAs are defined in IPsec, regardless of whether AH or ESP is employed. A *transport mode SA* is a security association between two hosts. Transport mode provides the authentication and/or encryption service to the higher layer protocol. This mode of operation is only supported by IPsec hosts. A *tunnel mode SA* is a security association applied to an IP tunnel. In this mode, there is an "outer" IP header that specifies the IPsec destination and an "inner" IP header that specifies the destination for the IP packet. This mode of operation is supported by both hosts and security gateways.

```
ORIGINAL PACKET BEFORE APPLYING AH

       ----------------------------
 IPv4  |orig IP hdr  |     |      |
       |(any options)| TCP | Data |
       ----------------------------

       ---------------------------------------
 IPv6  |             | ext hdrs |     |      |
       | orig IP hdr |if present| TCP | Data |
       ---------------------------------------

AFTER APPLYING AH (TRANSPORT MODE)

        -------------------------------------------------------

  IPv4  |original IP hdr (any options) | AH | TCP |    Data   |
        -------------------------------------------------------
        |<- mutable field processing ->|<- immutable fields ->|
        |<----- authenticated except for mutable fields ----->|

       ------------------------------------------------------------

 IPv6  |             |hop-by-hop, dest*, |    | dest |     |      |
       |orig IP hdr  |routing, fragment. | AH | opt* | TCP | Data |
       ------------------------------------------------------------
       |<--- mutable field processing -->|<-- immutable fields -->|
       |<---- authenticated except for mutable fields ----------->|

             * = if present, could be before AH, after AH, or both

AFTER APPLYING AH (TUNNEL MODE)

      ----------------------------------------------------------------
 IPv4 |                              |    | orig IP hdr*  |   |      |
      |new IP header * (any options) | AH | (any options) |TCP| Data |
      ----------------------------------------------------------------
      |<- mutable field processing ->|<------ immutable fields ----->|
      |<- authenticated except for mutable fields in the new IP hdr->|

      --------------------------------------------------------------
 IPv6 |           | ext hdrs*|    |            | ext hdrs*|   |    |
      |new IP hdr*|if present| AH |orig IP hdr*|if present|TCP|Data|
      --------------------------------------------------------------
      |<--- mutable field -->|<--------- immutable fields -------->|
      |       processing     |
      |<-- authenticated except for mutable fields in new IP hdr ->|

        * = if present, construction of outer IP hdr/extensions and
            modification of inner IP hdr/extensions is discussed in
            the Security Architecture document.
```

FIGURE 12: IPsec tunnel and transport modes for AH. ***(Adapted from RFC 4302)***

Figure 12 show the IPv4 and IPv6 packet formats when using AH in both transport and tunnel modes. Initially, an IPv4 packet contains a normal IPv4 header (which may contain IP options), followed by the higher layer protocol header (e.g., TCP or UDP), followed by the higher layer data itself. An IPv6 packet is similar except that the packet starts with the mandatory IPv6 header followed by any IPv6 extension headers, and then followed by the higher layer data.

Note that in both transport and tunnel modes, the **entire** IP packet is covered by the authentication *except for the mutable fields.* A field is *mutable* if its value might change during transit in the network; IPv4 mutable fields include the fragment offset, time to live, and checksum fields. Note, in particular, that the address fields are *not* mutable.

```
   ORIGINAL PACKET BEFORE APPLYING ESP

            ----------------------------
      IPv4  |orig IP hdr  |     |      |
            |(any options)| TCP | Data |
            ----------------------------


            ---------------------------------------
      IPv6  |             | ext hdrs |     |      |
            | orig IP hdr |if present| TCP | Data |
            ---------------------------------------


   AFTER APPLYING ESP (TRANSPORT MODE)


            -------------------------------------------------
      IPv4  |orig IP hdr  | ESP |     |      |   ESP   | ESP|
            |(any options)| Hdr | TCP | Data | Trailer | ICV|

            -------------------------------------------------
                                |<---- encryption ---->|
                          |<-------- integrity ------->|

            ---------------------------------------------------------
      IPv6  | orig |hop-by-hop,dest*,|   |dest|   |    | ESP   | ESP|

            |IP hdr|routing,fragment.|ESP|opt*|TCP|Data|Trailer| ICV|
            ---------------------------------------------------------
                                         |<--- encryption ---->|
                                     |<------ integrity ------>|

                * = if present, could be before ESP, after ESP, or both

   AFTER APPLYING ESP (TUNNEL MODE)

            -----------------------------------------------------------
      IPv4  | new IP hdr+ |     | orig IP hdr+  |   |    | ESP   | ESP|
            |(any options)| ESP | (any options) |TCP|Data|Trailer| ICV|

            -----------------------------------------------------------
                                |<--------- encryption --------->|
                          |<------------- integrity ----------->|

            ------------------------------------------------------------
      IPv6  | new+ |new ext |   | orig+|orig ext |   |    | ESP   | ESP|
            |IP hdr| hdrs+  |ESP|IP hdr| hdrs+   |TCP|Data|Trailer| ICV|
            ------------------------------------------------------------
                                |<--------- encryption ---------->|
                            |<------------ integrity ------------>|


            + = if present, construction of outer IP hdr/extensions and
                modification of inner IP hdr/extensions is discussed in
                the Security Architecture document.
```

FIGURE 13: IPsec tunnel and transport modes for ESP. ***(Adapted from RFC 4303)***

Figure 13 shows the IPv4 and IPv6 packet formats when using ESP in both transport and tunnel modes.

- As with AH, we start with a standard IPv4 or IPv6 packet.
- In transport mode, the higher layer header and data, as well as ESP trailer information, is encrypted and the entire ESP packet is authenticated. In the case of IPv6, some of the IPv6 extension options can precede or follow the ESP header.
- In tunnel mode, the original IP packet is encrypted and placed inside of an "outer" IP packet, while the entire ESP packet is authenticated.

Note a significant difference in the scope of ESP and AH. AH authenticates the entire packet transmitted on the network whereas ESP only covers a portion of the packet transmitted on the network (the higher layer data in transport mode and the entire original packet in tunnel mode). The reason for this is straight-forward; in AH, the authentication data for the transmission fits neatly into an additional header whereas ESP creates an entirely new packet which is the one encrypted and/or authenticated. But the ramifications are significant. ESP transport mode as well as AH in both modes protect the IP address fields of the original transmissions. Thus, using IPsec in conjunction with network address translation (NAT) *might* be problematic because NAT changes the values of these fields *after* IPsec processing.

The third component of IPsec is the establishment of security associations and key management. These tasks can be accomplished in one of two ways.

The simplest form of SA and key management is manual management. In this method, a security administer or other individual manually configures each system with the key and SA management data necessary for secure communication with other systems. Manual techniques are practical for small, reasonably static environments but they do not scale well.

For successful deployment of IPsec, however, a scalable, automated SA/key management scheme is necessary. Several protocols have defined for these functions:

- The Internet Security Association and Key Management Protocol (ISAKMP) defines procedures and packet formats to establish, negotiate, modify and delete security associations, and provides the framework for exchanging information about authentication and key management (RFC 2407/RFC 2408). ISAKMP's security association and key management is totally separate from key exchange.
- The OAKLEY Key Determination Protocol (RFC 2412) describes a scheme by which two authenticated parties can exchange key information. OAKLEY uses the Diffie-Hellman key exchange algorithm.
- The Internet Key Exchange (IKE) algorithm (RFC 2409) is the default automated key management protocol for IPsec.
- An alternative to IKE is Photuris (RFC 2522/RFC 2523), a scheme for establishing short-lived session-keys between two authenticated parties without passing the session-keys across the Internet. IKE typically creates keys that may have very long lifetimes.

On a final note, IPsec authentication for both AH and ESP uses a scheme called *HMAC*, a keyed-hashing message authentication code described in FIPS 198 and RFC 2104. HMAC uses a shared secret key between two parties rather than public key methods for message authentication. The generic HMAC procedure can be used with just about any hash algorithm, although IPsec specifies support for at least MD5 and SHA-1 because of their widespread use.

In HMAC, both parties share a secret key. The secret key will be employed with the hash algorithm in a way that provides mutual authentication without transmitting the key on the line. IPsec key management procedures will be used to manage key exchange between the two parties.

Recall that hash functions operate on a fixed-size block of input at one time; MD5 and SHA-1, for example, work on 64 byte blocks. These functions then generate a fixed-size hash value; MD5 and SHA-1, in particular, produce 16 byte (128 bit) and 20 byte (160 bit) output strings, respectively. For use with HMAC, the secret key (K) should be at least as long as the hash output.

The following steps provide a simplified, although reasonably accurate, description of how the HMAC scheme would work with a particular plaintext MESSAGE:

1. Alice pads K so that it is as long as an input block; call this padded key K_p. Alice computes the hash of the padded key followed by the message, i.e., HASH (K_p:MESSAGE).
2. Alice transmits MESSAGE and the hash value.
3. Bob has also padded K to create K_p. He computes HASH (K_p:MESSAGE) on the incoming message.
4. Bob compares the computed hash value with the received hash value. If they match, then the sender — Alice — must know the secret key and her identity is, thus, authenticated.

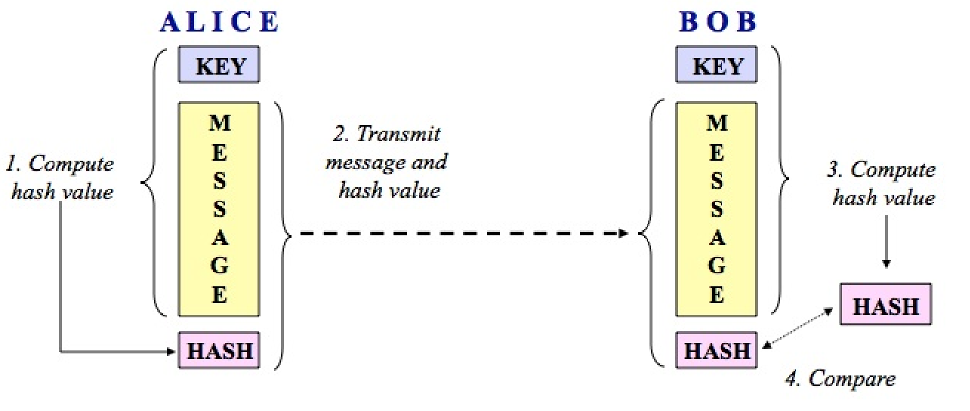
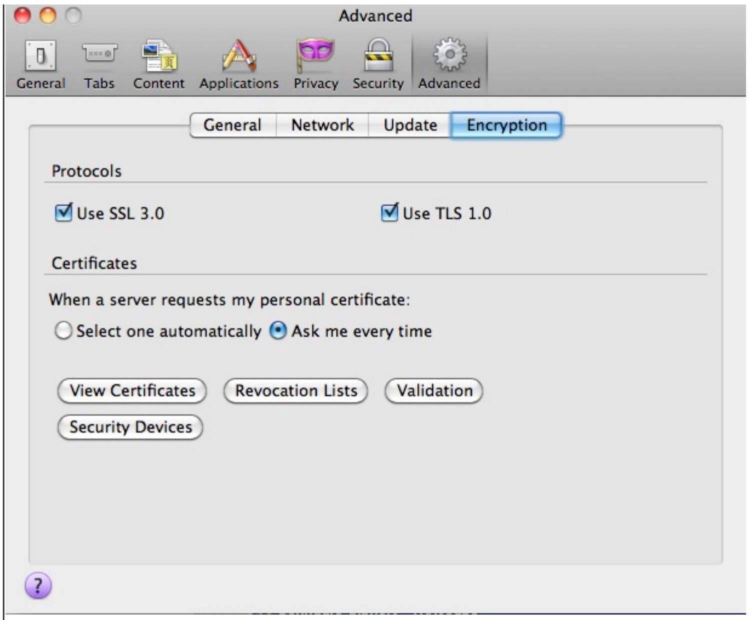


FIGURE 14: Keyed-hash MAC operation.

5.7. The SSL Family of Secure Transaction Protocols for the World Wide Web

The Secure Sockets Layer (SSL) protocol was developed by Netscape Communications to provide application-independent secure communication over the Internet for protocols such as the Hypertext Transfer Protocol (HTTP). SSL employs RSA and X.509 certificates during an initial handshake used to authenticate the server (client authentication is optional). The client and server then agree upon an encryption scheme. SSL v2.0 (1995), the first version publicly released, supported RC2 and RC4 with 40-bit keys. SSL v3.0 (1996) added support for DES, RC4 with a 128-bit key, and 3DES with a 168-bit key, all along with either MD5 or SHA-1 message hashes; this protocol is described in RFC 6101.



Protocols

☑ Use SSL 3.0 ☑ Use TLS 1.0

Certificates

When a server requests my personal certificate:

○ Select one automatically ● Ask me every time

FIGURE 15: Browser encryption configuration screen (Firefox).

In 1997, SSL v3 was found to be breakable. By this time, the Internet Engineering Task Force (IETF) had already started work on a new, non-proprietary protocol called Transport Layer Security (TLS), described in RFC 2246 (1999). TLS extends SSL and supports additional crypto schemes, such as Diffie-Hellman key exchange and DSS digital signatures; RFC 4279 describes the pre-shared key crypto schemes supported by TLS. TLS is backward compatible with SSL (and, in fact, is recognized as SSL v3.1). SSL v3.0 and TLS v1.0 are the commonly supported versions on servers and browsers today (Figure 15); SSL v2.0 is rarely found today and, in fact, RFC 6176-compliant clients and servers that support TLS will never negotiate the use of SSL v2.

In 2002, a cipher block chaining (CBC) vulnerability was described for TLS v1.0. In 2011, the theoretical became practical when a CBC proof-of-concept exploit was released. Meanwhile, TLS v1.1 was defined in 2006 (RFC 4346), adding protection against v1.0's CBC vulnerability. In 2008, TLS v1.2 was defined (RFC 5246), adding several additional cryptographic options. Today, users are urged to use TLS v1.2 or v1.1 in lieu of any earlier versions, and v1.3 is available in draft form.

```
                      CLIENT       SERVER
 (using URL of form https://)      (listening on port 443)

                 ClientHello ---->

                                   ServerHello
                                   Certificate*
                                   ServerKeyExchange*
                                   CertificateRequest*
                             <---- ServerHelloDone

                Certificate*
           ClientKeyExchange

           CertifcateVerify*
          [ChangeCipherSpec]
                    Finished ---->

                                   [ChangeCipherSpec]
                             <---- Finished

            Application Data <---> Application Data


* Optional or situation-dependent messages;
  not always sent

                                    Adapted from RFC 2246
```

FIGURE 16: SSL/TLS protocol handshake.

Figure 16 shows the basic TLS (and SSL) message exchanges:

1. URLs specifying the protocol *https://* are directed to HTTP servers secured using SSL/TLS. The client will automatically try to make a TCP connection to the server at port 443. The client initiates the secure

connection by sending a ClientHello message containing a Session identifier, highest SSL version number supported by the client, and lists of supported crypto and compression schemes (in preference order).

2. The server examines the Session ID and if it is still in the server's cache, it will attempt to re-establish a previous session with this client. If the Session ID is not recognized, the server will continue with the handshake to establish a secure session by responding with a ServerHello message. The ServerHello repeats the Session ID, indicates the SSL version to use for this connection (which will be the highest SSL version supported by the server and client), and specifies which encryption method and compression method to be used for this connection.
3. There are a number of other optional messages that the server might send, including:
   - Certificate, which carries the server's X.509 public key certificate (and, generally, the server's public key). This message will always be sent unless the client and server have already agreed upon some form of anonymous key exchange. (This message is normally sent.)
   - ServerKeyExchange, which will carry a premaster secret when the server's Certificate message does not contain enough data for this purpose; used in some key exchange schemes.
   - CertificateRequest, used to request the client's certificate in those scenarios where client authentication is performed.
   - ServerHelloDone, indicating that the server has completed its portion of the key exchange handshake.
4. The client now responds with a series of mandatory and optional messages:
   - Certificate, contains the client's public key certificate when it has been requested by the server.
   - ClientKeyExchange, which usually carries the secret key to be used with the secret key crypto scheme.
   - CertificateVerify, used to provide explicit verification of a client's certificate if the server is authenticating the client.
5. TLS includes the change cipher spec protocol to indicate changes in the encryption method. This protocol contains a single message, ChangeCipherSpec, which is encrypted and compressed using the current (rather than the new) encryption and compression schemes. The ChangeCipherSpec message is sent by both client and server to notify the other station that all following information will employ the newly negotiated cipher spec and keys.
6. The Finished message is sent after a ChangeCipherSpec message to confirm that the key exchange and authentication processes were successful.
7. At this point, both client and server can exchange application data using the session encryption and compression schemes.

Side Note: It would probably be helpful to make some mention of SSL (or, more properly, TLS) as it is used today. Most of us have used SSL to engage in a secure, private transaction with some vendor. The steps are something like this. During the SSL exchange with the vendor's secure server, the server sends its certificate to our client software. The certificate includes the vendor's public key and a signature from the CA that issued the vendor's certificate. Our browser software is shipped with the major CAs' certificates which contains their public key; in that way we authenticate the server. Note that the server does *not* use a certificate to authenticate us! Instead, we are generally authenticated when we provide our credit card number; the server checks to see if the card purchase will be authorized by the credit card company and, if so, considers us valid and authenticated! While bidirectional authentication is certainly supported by SSL, this form of asymmetric authentication is more commonly employed today since most users don't have certificates.

Microsoft's Server Gated Cryptography (SGC) protocol is another, albeit now defunct, extension to SSL/TLS. For several decades, it has been illegal to generally export products from the U.S. that employed secret-key cryptography with keys longer than 40 bits. For that reason, SSL/TLS has an exportable version with weak (40-bit) keys and a domestic (North American) version with strong (128-bit) keys. Within the last several years, however, use of strong SKC has been approved for the worldwide financial community. SGC is an extension to SSL that allows financial institutions using Windows NT servers to employ strong cryptography. Both the client and server must implement SGC and the bank must have a valid SGC certificate. During the initial handshake, the server will indicate support of SGC and supply its SGC certificate; if the client wishes to use SGC and validates the server's SGC certificate, the session can employ 128-bit RC2, 128-bit RC4, 56-bit DES, or 168-bit 3DES. Microsoft supports SGC in the Windows 95/98/NT versions of Internet Explorer 4.0, Internet Information Server (IIS) 4.0, and Money 98.

As mentioned above, SSL was designed to provide application-independent transaction security for the Internet. Although the discussion above has focused on HTTP over SSL (https/TCP port 443), SSL is also applicable to:

Protocol	TCP Port Name/Number
File Transfer Protocol (FTP)	ftps-data/989 & ftps/990
Internet Message Access Protocol v4 (IMAP4)	imaps/993
Lightweight Directory Access Protocol (LDAP)	ldaps/636
Network News Transport Protocol (NNTP)	nntps/563
Post Office Protocol v3 (POP3)	pop3s/995
Telnet	telnets/992

TLS was originally designed to operate over TCP. The IETF developed the Datagram Transport Layer Security (DTLS) protocol, based upon TLS, to operate over UDP. DTLS v1.2 is described in RFC 6347. (DTLS v1.0 can be found in RFC 4347.) RFC 6655 describes a suite of AES in Counter with Cipher Block Chaining - Message Authentication Code (CBC-MAC) Mode (CCM) ciphers for use with TLS and DTLS. An interesting analysis of the TLS protocol can be found in the paper "Analysis and Processing of Cryptographic Protocols" by Cowie.

Vulnerabilities: A vulnerability in the OpenSSL Library was discovered in 2014. Known as Heartbleed, this vulnerability had apparently been introduced into OpenSSL in late 2011 with the introduction of a feature called *heartbeat*. Heartbleed exploited an implementation flaw in order to exfiltrate keying material from an SSL server (or some SSL clients, in what is known at *reverse Heartbleed*); the flaw allowed an attacker to grab 64 KB blocks from RAM. Heartbleed is known to only affect OpenSSL v1.0.1 through v1.0.1f; the exploit was patched in v1.0.1g.

In addition, the OpenSSL 0.9.8 and 1.0.0 families are not vulnerable. Note also that Heartbleed affects some versions of the Android operating system, notably v4.1.0 and v4.1.1 (and some, possibly custom, implementations of v4.2.2). *Note that Heartbleed did **not** exploit a flaw in the SSL protocol, but rather a flaw in the OpenSSL implementation.*

But that wasn't the only problem with SSL. In October 2014, a new vulnerability was found called POODLE (Padding Oracle On Downgraded Legacy Encryption), a man-in-the-middle attack that exploited another SSL vulnerability that had unknowingly been in place for many years. Weeks later, an SSL vunerability in the bash Unix command shell was discovered, aptly named Shellshock. (Here's a nice overview of the 2014 SSL problems!) In March 2015, the Bar Mitzvah Attack was exposed, exploiting a 13-year old vulnerability in the Rivest Cipher 4 (RC4) encryption algorithm. Then there was the FREAK (Factoring Attack on RSA-EXPORT Keys CVE-2015-0204) SSL/TLS Vulnerabilty that affected some SSL/TLS implementations, including Android OS and Chrome browser for OS X later that month.

In March 2016, the SSL DROWN (Decrypting RSA with Obsolete and Weakened eNcryption) attack was announced. DROWN works by exploiting the presence of SSLv2 to crack encrypted communications and steal information from Web servers, email servers, or VPN sessions. You might have read above that SSLv2 fell out of use by the early 2000s and was formally deprecated in 2011. This is true. But backward compatibility often causes old software to remain dormant and it seems that up to one-third of all HTTPS sites are vulnerable to DROWN because SSLv2 has not been removed or disabled.

5.8. Elliptic Curve Cryptography (ECC)

In general, public-key cryptography systems use hard-to-solve problems as the basis of the algorithm. The most predominant algorithm today for public-key cryptography is RSA, based on the prime factors of very large integers. While RSA can be successfully attacked, the mathematics of the algorithm have not been comprised, per se; instead, computational brute-force has broken the keys. The defense is "simple" — keep the size of the integer to be factored ahead of the computational curve!

In 1985, Elliptic Curve Cryptography (ECC) was proposed independently by cryptographers Victor Miller (IBM) and Neal Koblitz (University of Washington). ECC is based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP). Like the prime factorization problem, ECDLP is another "hard" problem that is deceptively simple to state: Given two points, P and Q, on an elliptic curve, find the integer *n*, if it exists, such that P = nQ.

Elliptic curves combine number theory and algebraic geometry. These curves can be defined over any field of numbers (i.e., real, integer, complex) although we generally see them used over finite fields for applications in cryptography. An elliptic curve consists of the set of real numbers (x,y) that satisfies the equation:

$$y^2 = x^3 + ax + b$$

The set of all of the solutions to the equation forms the elliptic curve. Changing *a* and *b* changes the shape of the curve, and small changes in these parameters can result in major changes in the set of (x,y) solutions.

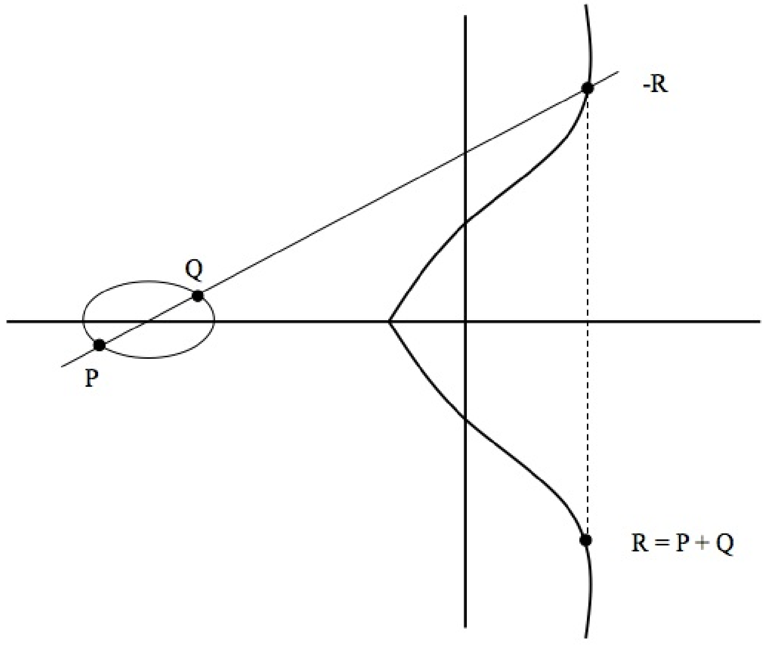


FIGURE 17: Elliptic curve addition.

Figure 17 shows the addition of two points on an elliptic curve. Elliptic curves have the interesting property that adding two points on the elliptic curve yields a third point on the curve. Therefore, adding two points, P and Q, gets us to point R, also on the curve. Small changes in P or Q can cause a large change in the position of R.

So let's go back to the original problem statement from above. The point Q is calculated as a multiple of the starting point, P, *or*, Q = nP. An attacker might know P and Q but finding the integer, *n*, is a difficult problem to solve. Q (i.e., nP) is the public key and *n* is the private key.

ECC may be employed with many Internet standards, including CCITT X.509 certificates and certificate revocation lists (CRLs), Internet Key Exchange (IKE), Transport Layer Security (TLS), XML signatures, and applications or protocols based on the cryptographic message syntax (CMS). RFC 5639 proposes a set of elliptic curve domain parameters over finite prime fields for use in these cryptographic applications and RFC 6637 proposes additional elliptic curves for use with OpenPGP.

RSA had been the mainstay of PKC for over a quarter-century. ECC, however, is emerging as a replacement in some environments because it provides similar levels of security compared to RSA but with significantly reduced key sizes. NIST use the following table to demonstrate the key size relationship between ECC and RSA, and the appropriate choice of AES key size:

TABLE 4. ECC and RSA Key Comparison.

ECC Key Size	RSA Key Size	Key-Size Ratio	AES Key Size
163	1,024	1:6	n/a
256	3,072	1:12	128
384	7,680	1:20	192
512	15,360	1:30	256
Key sizes in bits.		*Source: Certicom, NIST*	

Since the ECC key sizes are so much shorter than comparable RSA keys, the length of the public key and private key is much shorter in elliptic curve cryptosystems. This results into faster processing times, and lower demands on memory and bandwidth; some studies have found that ECC is faster than RSA for signing and decryption, but slower for signature verification and encryption.

ECC is particularly useful in applications where memory, bandwidth, and/or computational power is limited (e.g., a smartcard) and it is in this area that ECC use is expected to grow. A major champion of ECC today is Certicom; readers are urged to see their ECC tutorial.

5.9. The Advanced Encryption Standard (AES) and Rijndael

The search for a replacement to DES started in January 1997 when NIST announced that it was looking for an Advanced Encryption Standard. In September of that year, they put out a formal Call for Algorithms and in August 1998 announced that 15 candidate algorithms were being considered (Round 1). In April 1999, NIST announced that the 15 had been whittled down to five finalists (Round 2): *MARS* (multiplication, addition, rotation and substitution) from IBM; Ronald Rivest's *RC6*; *Rijndael* from a Belgian team; *Serpent*, developed jointly by a team from England, Israel, and Norway; and *Twofish*, developed by Bruce Schneier. In October 2000, NIST announced their selection: Rijndael.

The remarkable thing about this entire process has been the openness as well as the international nature of the "competition." NIST maintained an excellent Web site devoted to keeping the public fully informed, at *http://csrc.nist.gov/archive/aes/*, which is now available as an archive site. Their Overview of the AES Development Effort has full details of the process, algorithms, and comments so I will not repeat everything here.

In October 2000, NIST released the Report on the Development of the Advanced Encryption Standard (AES) that compared the five Round 2 algorithms in a number of categories. The table below summarizes the relative scores of the five schemes (1=low, 3=high):

	Algorithm				
Category	**MARS**	**RC6**	**Rijndael**	**Serpent**	**Twofish**
General security	3	2	2	3	3
Implementation of security	1	1	3	3	2
Software performance	2	2	3	1	1
Smart card performance	1	1	3	3	2
Hardware performance	1	2	3	3	2
Design features	2	1	2	1	3

With the report came the recommendation that Rijndael be named as the AES standard. In February 2001, NIST released the Draft Federal Information Processing Standard (FIPS) AES Specification for public review and comment. AES contains a subset of Rijndael's capabilities (e.g., AES only supports a 128-bit block size) and uses some slightly different nomenclature and terminology, but to understand one is to understand both. The 90-day comment period ended on May 29, 2001 and the U.S. Department of Commerce officially adopted AES in December 2001, published as FIPS PUB 197.

AES (Rijndael) Overview

Rijndael (pronounced as in "rain doll" or "rhine dahl") is a block cipher designed by Joan Daemen and Vincent Rijmen, both cryptographers in Belgium. Rijndael can operate over a variable-length block using variable-length keys; the specification submitted to NIST describes use of a 128-, 192-, or 256-bit key to encrypt data blocks that are 128, 192, or 256 bits long; note that all nine combinations of key length and block length are possible. The algorithm is written in such a way that block length and/or key length can easily be extended in multiples of 32 bits and it is specifically designed for efficient implementation in hardware or software on a range of processors. The design of Rijndael was strongly influenced by the block cipher called *Square*, also designed by Daemen and Rijmen. See

- The Rijndael page for a lot more information.

Rijndael is an iterated block cipher, meaning that the initial input block and cipher key undergoes multiple rounds of transformation before producing the output. Each intermediate cipher result is called a *State*.

For ease of description, the block and cipher key are often represented as an array of columns where each array has 4 rows and each column represents a single byte (8 bits). The number of columns in an array representing the state or cipher key, then, can be calculated as the block or key length divided by 32 (32 bits = 4 bytes). An array representing a State will have **Nb** columns, where **Nb** values of 4, 6, and 8 correspond to a 128-, 192-, and 256-bit block, respectively. Similarly, an array representing a Cipher Key will have **Nk** columns, where **Nk** values of 4, 6, and 8 correspond to a 128-, 192-, and 256-bit key, respectively. An example of a 128-bit State (**Nb**=4) and 192-bit Cipher Key (**Nk**=6) is shown below:

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	$k_{0,4}$	$k_{0,5}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$	$k_{1,4}$	$k_{1,5}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$	$k_{2,4}$	$k_{2,5}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$k_{3,4}$	$k_{3,5}$

The number of transformation rounds (**Nr**) in Rijndael is a function of the block length and key length, and is given by the table below:

No. of Rounds Nr		**Block Size**		
		128 bits Nb = 4	**192 bits Nb = 6**	**256 bits Nb = 8**
Key Size	**128 bits Nk = 4**	10	12	14
	192 bits Nk = 6	12	12	14
	256 bits Nk = 8	14	14	14

Now, having said all of this, the AES version of Rijndael does not support all nine combinations of block and key lengths, but only the subset using a 128-bit block size. NIST calls these supported variants AES-128, AES-192, and AES-256 where the number refers to the key size. The **Nb**, **Nk**, and **Nr** values supported in AES are:

	Parameters		
Variant	**Nb**	**Nk**	**Nr**
AES-128	4	4	10
AES-192	4	6	12
AES-256	4	8	14

The AES/Rijndael cipher itself has three operational stages:

- AddRound Key transformation
- **Nr**-1 Rounds comprising:
  - SubBytes transformation
  - ShiftRows transformation
  - MixColumns transformation
  - AddRoundKey transformation
- A final Round comprising:
  - SubBytes transformation
  - ShiftRows transformation
  - AddRoundKey transformation

The paragraphs below will describe the operations mentioned above. The nomenclature used below is taken from the AES specification although references to the Rijndael specification are made for completeness. The arrays *s* and *s'* refer to the State before and after a transformation, respectively (**NOTE:** The Rijndael specification uses the array nomenclature a and b to refer to the before and after States, respectively). The subscripts *i* and *j* are used to indicate byte locations within the State (or Cipher Key) array.

The SubBytes transformation

The substitute bytes (called *ByteSub* in Rijndael) transformation operates on each of the State bytes independently and changes the byte value. An S-box, or *substitution table*, controls the transformation. The characteristics of the

S-box transformation as well as a compliant S-box table are provided in the AES specification; as an example, an input State byte value of 107 (0x6b) will be replaced with a 127 (0x7f) in the output State and an input value of 8 (0x08) would be replaced with a 48 (0x30).

One way to think of the SubBytes transformation is that a given byte in State s is given a new value in State s' according to the S-box. The S-box, then, is a function on a byte in State s so that:

$$s'_{i,j} = \text{S-box}\ (s_{i,j})$$

The more general depiction of this transformation is shown by:

$$\begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} \quad \text{====>} \quad \boxed{\text{S-box}} \quad \text{====>} \quad \begin{array}{|c|c|c|c|}\hline s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ \hline s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ \hline s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ \hline s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \\ \hline \end{array}$$

The ShiftRows transformation

The shift rows (called *ShiftRow* in Rijndael) transformation cyclically shifts the bytes in the bottom three rows of the State array. According to the more general Rijndael specification, rows 2, 3, and 4 are cyclically left-shifted by C1, C2, and C3 bytes, respectively, per the table below:

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

The current version of AES, of course, only allows a block size of 128 bits (**Nb** = 4) so that C1=1, C2=2, and C3=3. The diagram below shows the effect of the ShiftRows transformation on State s:

$$\begin{array}{c c c}
\text{State s} & & \text{State s'} \\
\begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ \hline s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ \hline s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \\ \hline \end{array} &
\begin{array}{c} \text{----------- no shift ----------->} \\ \text{----> left-shift by C1 (1) ---->} \\ \text{----> left-shift by C2 (2) ---->} \\ \text{----> left-shift by C3 (3) ---->} \end{array} &
\begin{array}{|c|c|c|c|}\hline s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ \hline s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ \hline s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ \hline s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \\ \hline \end{array}
\end{array}$$

The MixColumns transformation

The mix columns (called *MixColumn* in Rijndael) transformation uses a mathematical function to transform the values of a given column within a State, acting on the four values at one time as if they represented a four-term polynomial. In essence, if you think of MixColumns as a function, this could be written:

$$s'_{i,c} = \text{MixColumns}\ (s_{i,c})$$

for $0 \leq i \leq 3$ for some column, c. The column position doesn't change, merely the values within the column.

Round Key generation and the AddRoundKey transformation

The AES Cipher Key can be 128, 192, or 256 bits in length. The Cipher Key is used to derive a different key to be applied to the block during each round of the encryption operation. These keys are called the Round Keys and each will be the same length as the block, i.e., **Nb** 32-bit words (words will be denoted W).

The AES specification defines a key schedule by which the original Cipher Key (of length **Nk** 32-bit words) is used to form an *Expanded Key*. The Expanded Key size is equal to the block size times the number of encryption rounds plus 1, which will provide **Nr**+1 different keys. (Note that there are **Nr** encipherment rounds but **Nr**+1 AddRoundKey transformations.)

Consider that AES uses a 128-bit block and either 10, 12, or 14 iterative rounds depending upon key length. With a 128-bit key, for example, we would need 1408 bits of key material (128x11=1408), or an Expanded Key size of 44 32-bit words (44x32=1408). Similarly, a 192-bit key would require 1664 bits of key material (128x13), or 52 32-bit words, while a 256-bit key would require 1920 bits of key material (128x15), or 60 32-bit words. The key expansion mechanism, then, starts with the 128-, 192-, or 256-bit Cipher Key and produces a 1408-, 1664-, or 1920-bit Expanded Key, respectively. The original Cipher Key occupies the first portion of the Expanded Key and is used to produce the remaining new key material.

The result is an Expanded Key that can be thought of and used as 11, 13, or 15 separate keys, each used for one AddRoundKey operation. These, then, are the *Round Keys*. The diagram below shows an example using a 192-bit Cipher Key (**Nk**=6), shown in ***magenta italics***:

<table>
<tr><th>Expanded Key:</th><td>***W_0***</td><td>***W_1***</td><td>***W_2***</td><td>***W_3***</td><td>***W_4***</td><td>***W_5***</td><td>W_6</td><td>W_7</td><td>W_8</td><td>W_9</td><td>W_{10}</td><td>W_{11}</td><td>W_{12}</td><td>W_{13}</td><td>W_{14}</td><td>W_{15}</td><td>...</td><td>W_{44}</td><td>W_{45}</td><td>W_{46}</td><td>W_{47}</td><td>W_{48}</td><td>W_{49}</td><td>W_{50}</td><td>W_{51}</td></tr>
<tr><th>Round keys:</th><td colspan="4">Round key 0</td><td colspan="4">Round key 1</td><td colspan="4">Round key 2</td><td colspan="4">Round key 3</td><td>...</td><td colspan="4">Round key 11</td><td colspan="4">Round key 12</td></tr>
</table>

The AddRoundKey (called *Round Key addition* in Rijndael) transformation merely applies each Round Key, in turn, to the State by a simple bit-wise exclusive OR operation. Recall that each Round Key is the same length as the block.

Summary

Ok, I hope that you've enjoyed reading this as much as I've enjoyed writing it — and now let me guide you out of the microdetail! Recall from the beginning of the AES overview that the cipher itself comprises a number of rounds of just a few functions:

- *SubBytes* takes the value of a word within a State and substitutes it with another value by a predefined S-box
- *ShiftRows* circularly shifts each row in the State by some number of predefined bytes
- *MixColumns* takes the value of a 4-word column within the State and changes the four values using a predefined mathematical function
- *AddRoundKey* XORs a key that is the same length as the block, using an Expanded Key derived from the original Cipher Key

```
Cipher (byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])

begin
  byte state[4,Nb]

  state = in

  AddRoundKey(state, w)

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

  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, w+Nr*Nb)

  out = state
end
```

FIGURE 18: AES pseudocode.

As a last and final demonstration of the operation of AES, Figure 18 is a pseudocode listing for the operation of the AES cipher. In the code:

- *in[]* and *out[]* are 16-byte arrays with the plaintext and cipher text, respectively. (According to the specification, both of these arrays are actually 4***Nb** bytes in length but **Nb**=4 in AES.)
- *state[]* is a 2-dimensional array containing bytes in 4 rows and 4 columns. (According to the specification, this arrays is 4 rows by **Nb** columns.)
- *w[]* is an array containing the key material and is 4*(**Nr**+1) words in length. (Again, according to the specification, the multiplier is actually **Nb**.)
- *AddRoundKey()*, *SubBytes()*, *ShiftRows()*, and *MixColumns()* are functions representing the individual transformations.

5.10. Cisco's Stream Cipher

Stream ciphers take advantage of the fact that:

x XOR y XOR y = x

One of the encryption schemes employed by Cisco routers to encrypt passwords is a stream cipher. It uses the following fixed keystream (thanks also to Jason Fossen for independently extending and confirming this string):

dsfd;kfoA,.iyewrkldJKDHSUBsgvca69834ncx

When a password is to be encrypted, the password function chooses a number between 0 and 15, and that becomes the offset into the keystream. Password characters are then XORed byte-by-byte with the keystream according to:

$$C_i = P_i \text{ XOR } K_{(\text{offset}+i)}$$

where K is the keystream, P is the plaintext password, and C is the ciphertext password.

Consider the following example. Suppose we have the password *abcdefgh*. Converting the ASCII characters yields the hex string 0x6162636465666768.

The keystream characters and hex code that supports an offset from 0 to 15 bytes and a password length up to 24 bytes is:

d s f d ; k f o A , . i y e w r k l d J K D H S U B s g v c a 6 9 8 3 4 n c x
0x647366643b6b666f412c2e69796577726b6c644a4b4448535542736776636136393833346e6378

Let's say that the function decides upon a keystream offset of 6 bytes. We then start with byte 6 of the keystream (start counting the offset at 0) and XOR with the password:

```
    0x666f412c2e697965
XOR 0x6162636465666768
------------------
    0x070D22484B0F1E0D
```

The password would now be displayed in the router configuration as:

```
password 7 06070D22484B0F1E0D
```

where the "7" indicates the encryption type, the leading "06" indicates the offset into the keystream, and the remaining bytes are the encrypted password characters.

(Decryption is pretty trivial so that exercise is left to the reader. If you need some help with byte-wise XORing, see *http://www.garykessler.net/library/byte_logic_table.html*. If you'd like some programs that do this, see *http://www.garykessler.net/software/index.html#cisco7*.)

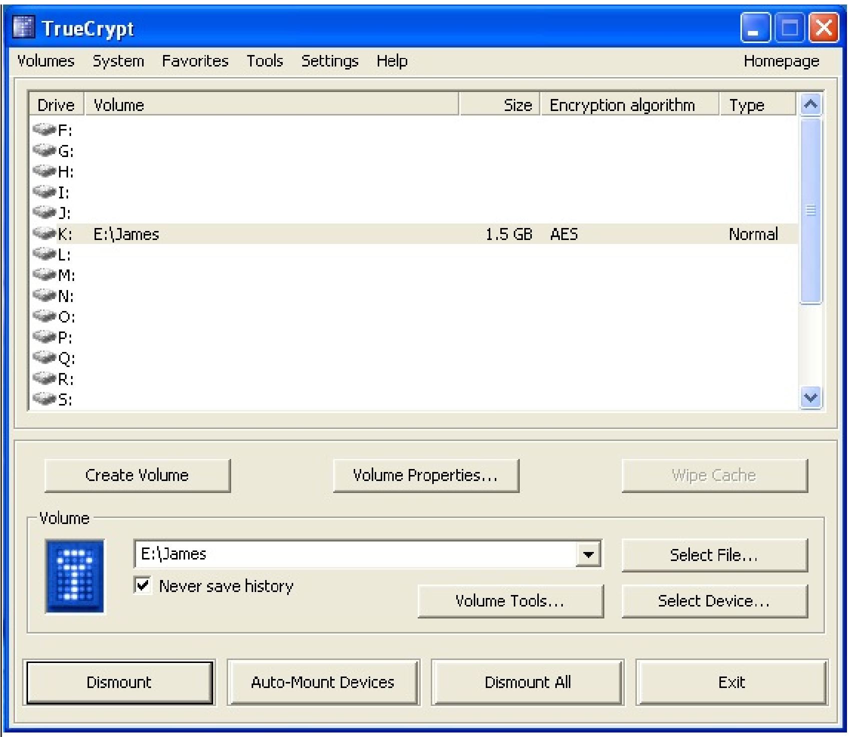
5.11. TrueCrypt

TrueCrypt is an open source, on-the-fly crypto system that can be used on devices supports by Linux, MacOS, and Windows. First released in 2004, TrueCrypt can be employed to encrypt a partition on a disk or an entire disk.

> On May 28, 2014, the TrueCrypt.org Web site was suddenly taken down and redirected to the SourceForge page. Although this paper is intended as a crypto tutorial and not a news source about crypto controversy, the sudden withdrawal of TrueCrypt cannot go without notice. Readers interested in using TrueCrypt should know that the last stable release of the product is v7.1a (February 2012); v7.2, released on May 28, 2014, only decrypts TrueCrypt volumes, ostensibly so that users can migrate to another solution. The current TrueCrypt Web page — TCnext — is TrueCrypt.ch. The TrueCrypt Wikipedia page and accompanying references have some good information about the "end" of TrueCrypt as we knew it.
>
> While there does not appear to be any rush to abandon TrueCrypt at the time of this writing, it is also the case that you don't want to use old, unsupported software for too long. A replacement for TrueCrypt called CipherShed is currently under development. See also "TrueCrypt may live on after all as CipherShed." To date, CipherShed has not produced a product; another — working — fork of TrueCrypt is VeraCrypt.
>
> One final editorial comment. TrueCrypt was ***not*** broken or otherwise compromised! It was withdrawn by its developers for reasons that have not yet been made public but there is no evidence to assume that TrueCrypt has been damaged in any way; on the contrary, two audits, completed in April 2014 and April 2015, found no evidence of backdoors or malicious code. See Steve Gibson's TrueCrypt: Final Release Repository page for more information!

TrueCrypt uses a variety of encryption schemes, including AES, Serpent, and Twofish. A TrueCrypt volume is stored as a file that appears to be filled with random data, thus has no specific file signature. (It is true that a TrueCrypt container will pass a chi-square (X^2) randomness test, but that is merely a general indicator of possibly encrypted content. An additional clue is that a TrueCrypt container will also appear on a disk as a file that is some increment of 512 bytes in size. While these indicators might raise a red flag, they don't rise to the level of clearly indentifying a TrueCrypt volume.)

When a user creates a TrueCrypt volume, a number of parameters need to be defined, such as the size of the volume and the password. To access the volume, the TrueCrypt program is employed to find the TrueCrypt encrypted file, which is then mounted as a new drive on the host system.

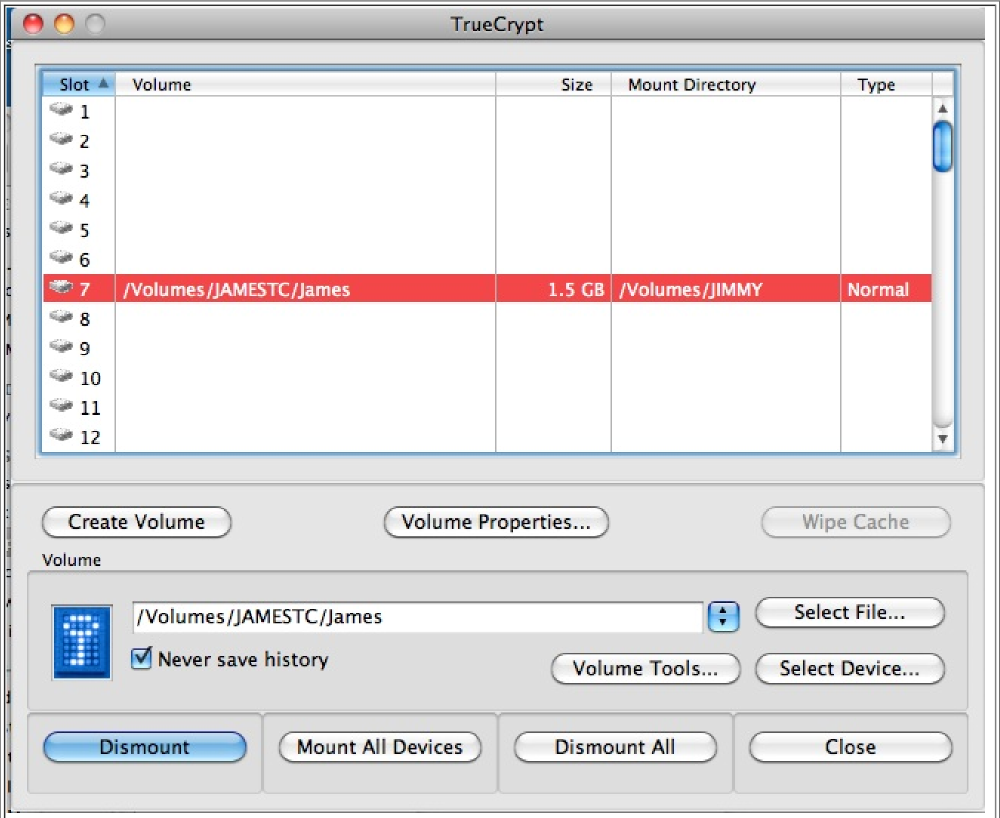


Drive	Volume	Size	Encryption algorithm	Type
F:				
G:				
H:				
I:				
J:				
K:	E:\James	1.5 GB	AES	Normal
L:				
M:				
N:				
O:				
P:				
Q:				
R:				
S:				

Volume: E:\James

☑ Never save history

FIGURE 19: TrueCrypt screen shot (Windows).



Slot ▲	Volume	Size	Mount Directory	Type
1				
2				
3				
4				
5				
6				
7	/Volumes/JAMESTC/James	1.5 GB	/Volumes/JIMMY	Normal
8				
9				
10				
11				
12				

Volume: /Volumes/JAMESTC/James

☑ Never save history

FIGURE 20: TrueCrypt screen shot (MacOS).

Consider this example where an encrypted TrueCrypt volume is stored as a file named *James* on a thumb drive. On a Windows system, this thumb drive has been mounted as device *E:*. If one were to view the *E:* device, any number of files might be found. The TrueCrypt application is used to mount the TrueCrypt file; in this case, the user has chosen to mount the TrueCrypt volume as device *K:* (Figure 19). Alternatively, the thumb drive could be used with a Mac system, where it has been mounted as the */Volumes/JIMMY* volume. TrueCrypt mounts the encrypted file, *James*, and it is now accessible to the system (Figure 20).

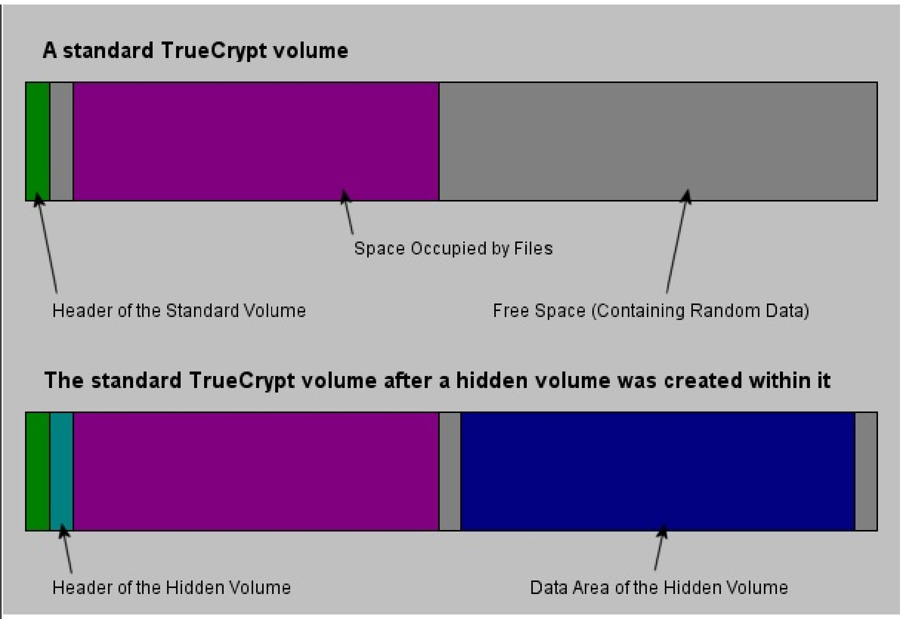

A standard TrueCrypt volume
Space Occupied by Files
Header of the Standard Volume
Free Space (Containing Random Data)
The standard TrueCrypt volume after a hidden volume was created within it
Header of the Hidden Volume
Data Area of the Hidden Volume

FIGURE 21: TrueCrypt hidden encrypted volume within an encrypted volume (from *http://www.truecrypt.org/images/docs/hidden-volume.gif*).

One of the most interesting — certainly one of the most controversial — features of TrueCrypt is called *plausible deniability*, protection in case a user is "compelled" to turn over the encrypted volume's password. When the user creates a TrueCrypt volume, he/she chooses whether to create a standard or hidden volume. A standard volume has a single password, while a hidden volume is created within a standard volume and uses a second password. As shown in Figure 21, the unallocated (free) space in a TrueCrypt volume is always filled with random data, thus it is impossible to differentiate a hidden encrypted volume from a standard volume's free space.

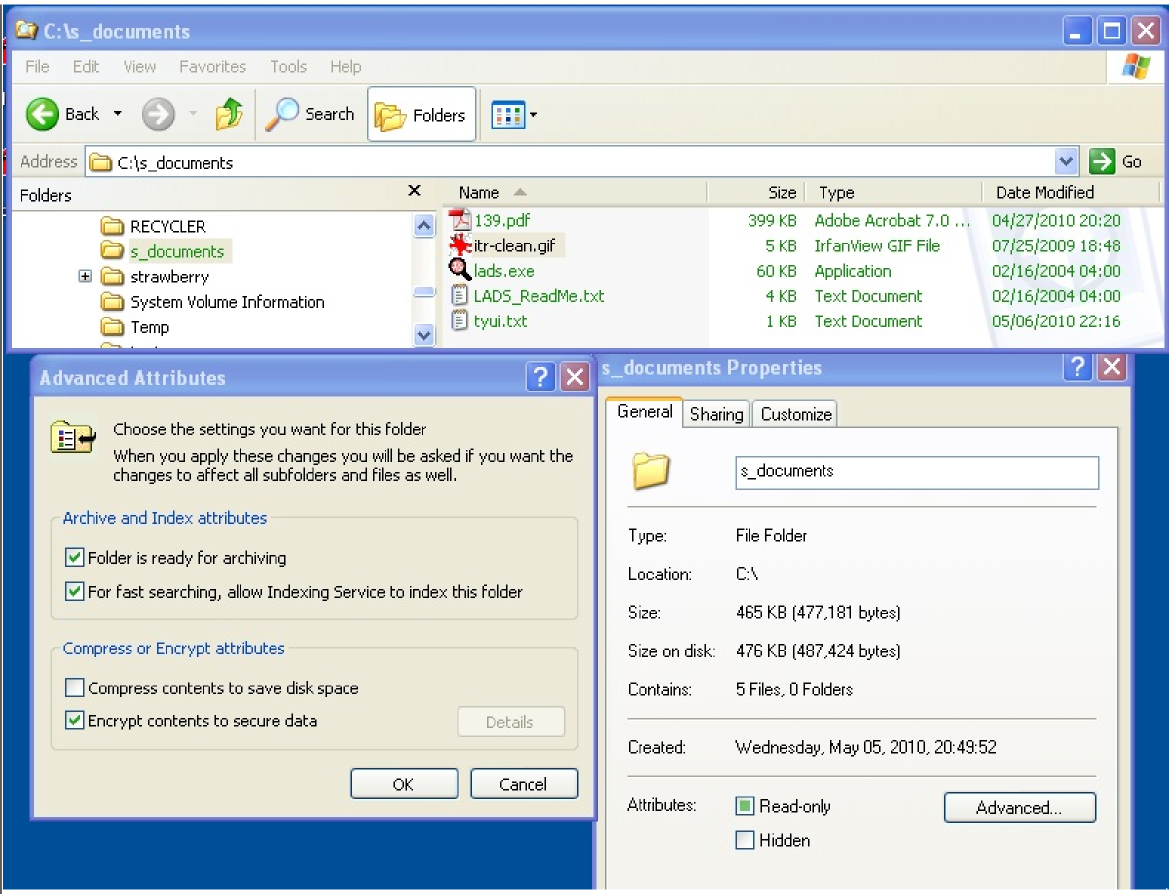
To access the hidden volume, the file is mounted as shown above and the user enters the hidden volume's password. When under duress, the user would merely enter the password of the standard (i.e., non-hidden) TrueCrypt volume.

More information about TrueCrypt can be found at the TCnext Web Site or in the *TrueCrypt User's Guide (v7.1a)*.

> An active area of research in the digital forensics community is to find methods with which to detect hidden TrueCrypt volumes. Most of the methods do not detect the presence of a hidden volume, per se, but infer the presence by forensic remnants left over. As an example, both Mac and Windows system usually have a file or registry entry somewhere containing a cached list of the names of mounted volumes. This list would, naturally, include the name of TrueCrypt volumes, both standard and hidden. If the user gives a name to the hidden volume, it would appear in such a list. If an investigator were somehow able to determine that there were two TrueCrypt volume names but only one TrueCrypt device, the inference would be that there was a hidden volume. A good summary paper that also describes ways to infer the presence of hidden volumes — at least on some Windows systems — can be found in " Detecting Hidden Encrypted Volumes" (Hargreaves & Chivers).
>
> Having nothing to do with TrueCrypt, but having something to do related to plausible deniability and devious crypto schemes, is a new approach to holding password cracking at bay dubbed *Honey Encryption*. With most of today's crypto systems, decrypting with a wrong key produces digital gibberish while a correct key produces something recognizable, making it easy to know when a correct key has been found. Honey Encryption produces fake data that resembles real data for every key that is attempted, making it significantly harder for an attacker to determine whether they have the correct key or not; thus, if an attacker has a credit card file and tries thousands of keys to crack it, they will obtain thousands of possibly legitimate credit card numbers. See "'Honey Encryption' Will Bamboozle Attackers with Fake Secrets" (Simonite) for some general information or "Honey Encryption: Security Beyond the Brute-Force Bound" (Juels & Ristenpart) for a detailed paper.

5.12. Encrypting File System (EFS)

Microsoft introduced the Encrypting File System (EFS) into the NTFS v3.0 file system and has supported EFS since Windows 2000 and XP (although EFS is not supported in all variations of all Windows OSes). EFS can be used to encrypt individual files, directories, or entire volumes. While off by default, EFS encryption can be easily enabled via File Explorer (aka Windows Explorer) by right-clicking on the file, directory, or volume to be encrypted, selecting Properties, Advanced, and *Encrypt contents to secure data* (Figure 22). Note that encrypted files and directories are displayed in green in Windows Explorer.



C:\s_documents
File Edit View Favorites Tools Help
Back Search Folders
Address C:\s_documents Go
Folders
RECYCLER
s_documents
strawberry
System Volume Information
Temp

Name	Size	Type	Date Modified
139.pdf	399 KB	Adobe Acrobat 7.0 ...	04/27/2010 20:20
itr-clean.gif	5 KB	IrfanView GIF File	07/25/2009 18:48
lads.exe	60 KB	Application	02/16/2004 04:00
LADS_ReadMe.txt	4 KB	Text Document	02/16/2004 04:00
tyui.txt	1 KB	Text Document	05/06/2010 22:16

Advanced Attributes

Choose the settings you want for this folder

When you apply these changes you will be asked if you want the changes to affect all subfolders and files as well.

Archive and Index attributes

☑ Folder is ready for archiving

☑ For fast searching, allow Indexing Service to index this folder

Compress or Encrypt attributes

□ Compress contents to save disk space

☑ Encrypt contents to secure data

s_documents Properties

s_documents

Type: File Folder

Location: C:\

Size: 465 KB (477,181 bytes)

Size on disk: 476 KB (487,424 bytes)

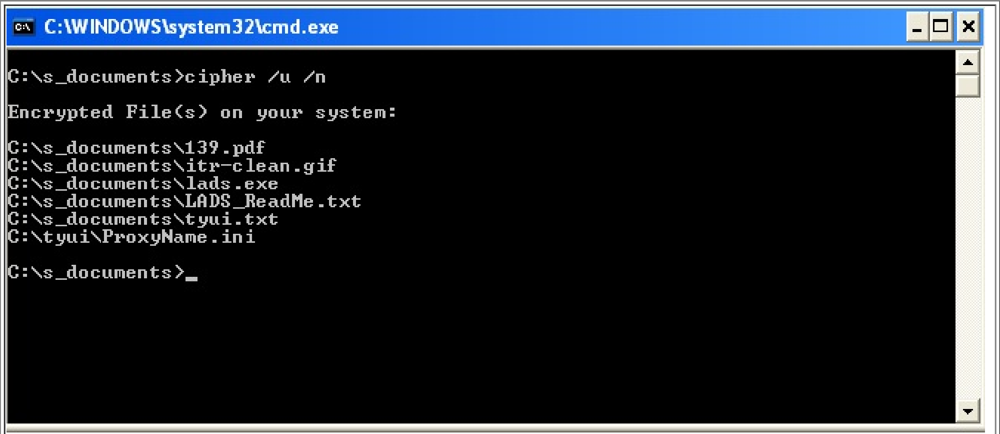
Contains: 5 Files, 0 Folders

Created: Wednesday, May 05, 2010, 20:49:52

Attributes: ■ Read-only □ Hidden

FIGURE 22: EFS and Windows (File) Explorer.

The Windows command prompt provides an easy tool with which to detect EFS-encrypted files on a disk. The cipher command has a number of options, but the /u/n switches can be used to list all encrypted files on a drive (Figure 23).



```
C:\s_documents>cipher /u /n

Encrypted File(s) on your system:

C:\s_documents\139.pdf
C:\s_documents\itr-clean.gif
C:\s_documents\lads.exe
C:\s_documents\LADS_ReadMe.txt
C:\s_documents\tyui.txt
C:\tyui\ProxyName.ini

C:\s_documents>_
```

FIGURE 23: The cipher command.

EFS supports a variety of secret key encryption schemes, including DES, DESX, and AES, as well as RSA public-key encryption. The operation of EFS — at least at the theoretical level — is clever and simple.

When a file is saved to disk:

- A random File Encryption Key (FEK) is generated by the operating system.
- The file contents are encrypted using one of the SKC schemes and the FEK.
- The FEK is stored with the file, encrypted with the user's RSA public key. In addition, the FEK is encrypted with the RSA public key of any other authorized users and, optionally, a recovery agent's RSA public key.

When the file is opened:

- The FEK is recovered using the RSA private key of the user, other authorized user, or the recovery agent.

- The FEK is used to decrypt the file's contents.

There are weaknesses with the system, most of which are related to key management. As an example, the RSA private key can be stored on an external device such as a floppy disk (yes, really!), thumb drive, or smart card. In practice, however, this is rarely done; the user's private RSA key is often stored on the hard drive. In addition, early EFS implementations (prior to Windows XP SP2) tied the key to the username; later implementations employ the user's password.

A more serious implementation issue is that a backup file named *esf0.tmp* is created prior to a file being encrypted. After the encryption operation, the backup file is deleted — not wiped — leaving an unencrypted version of the file available to be undeleted. For this reason, it is best to use encrypted directories because the temporary backup file is protected by being in an encrypted directory.

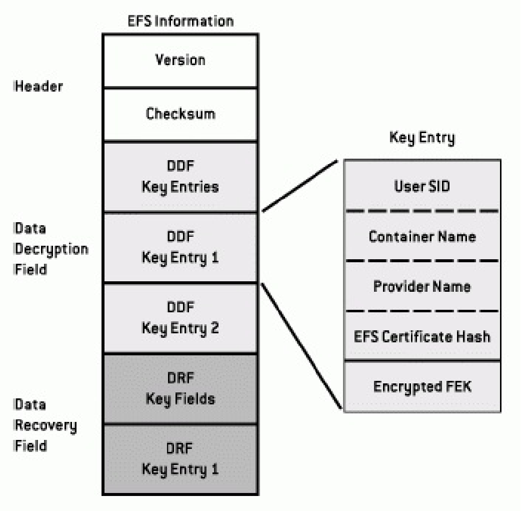


FIGURE 24: EFS key storage. (Source: NTFS.com)

The EFS information is stored as a named stream in the $LOGGED_UTILITY_STREAM Attribute (attribute type 256 [0x100]). This information includes (Figure 24):

- A Data Decryption Field (DDF) for every user authorized to decrypt the file, containing the user's Security Identifier (SID), the FEK encrypted with the user's RSA public key, and other information.
- A Data Recovery Field (DRF) with the encrypted FEK for every method of data recovery

Files in an NTFS file system maintain a number of attributes that contain the system metadata (e.g., the $STANDARD_INFORMATION attribute maintains the file timestamps and the $FILE_NAME attribute contains the file name). Files encrypted with EFS store the keys, as stated above, in a data stream named $EFS within the $LOGGED_UTILITY_STREAM attribute. Figure 25 shows the partial contents of the Master File Table (MFT) attributes for an EFS encrypted file.

```
Master File Table (MFT) Parser V1.4 - Gary C. Kessler (7 June 2012)
   :
   :
0056-0059  Attribute type: 0x10-00-00-00 [$STANDARD_INFORMATION]
0060-0063  Attribute length: 0x60-00-00-00 [96 bytes]
0064       Non-resident flag: 0x00 [Attribute is resident]
   :
   :
0152-0155  Attribute type: 0x30-00-00-00 [$FILE_NAME]
0156-0159  Attribute length: 0x78-00-00-00 [120 bytes]
0160       Non-resident flag: 0x00 [Attribute is resident]
   :
   :
0392-0395  Attribute type: 0x40-00-00-00 [$VOLUME_VERSION/$OBJECT_ID]
0396-0399  Attribute length: 0x28-00-00-00 [40 bytes]
0400       Non-resident flag: 0x00 [Attribute is resident]
   :
   :
0432-0435  Attribute type: 0x80-00-00-00 [$DATA]
0436-0439  Attribute length: 0x48-00-00-00 [72 bytes]
0440       Non-resident flag: 0x01 [Attribute is non-resident]
   :
   :
0504-0507  Attribute type: 0x00-01-00-00 [$LOGGED_UTILITY_STREAM]
0508-0511  Attribute length: 0x50-00-2E-00 [80 bytes (ignore two high-order bytes)]
0512       Non-resident flag: 0x01 [Attribute is non-resident]
   :
0568-0575  Name: 0x24-00-45-00-46-00-53-00 [$EFS]
```

FIGURE 25: The $LOGGED_UTILITY_STREAM Attribute.

5.13. Some of the Finer Details of RC4

RC4 is a variable key-sized stream cipher developed by Ron Rivest in 1987. RC4 works in output-feedback (OFB) mode, so that the key stream is independent of the plaintext. The algorithm is described in detail in Schneier's "Applied Cryptography," 2/e, pg. 397-398 or the Wikipedia *RC4* article.

RC4 employs an 8x8 substitution box (S-box). The S-box is initialized so that S[i] = i, for i=(0,255).

A permutation of the S-box is then performed as a function of the key. The K array is a 256-byte structure that holds the key, repeating itself as necessary so as to be 256 bytes in length (obviously, a longer key results in less repetition). **[[NOTE: All arithmetic below is assumed to be on a byte basis and so is implied to be *modulo 256*.]]**

```
j = 0
for i = 0 to 255
  j = j + S[i] + K[i]
  swap (S[i], S[j])
```

Encryption and decryption are performed by XORing a byte of plaintext/ciphertext with a random byte from the S-box in order to produce the ciphertext/plaintext, as follows:

```
Initialize i and j to zero
```

For each byte of plaintext (or ciphertext):

```
i = i + 1
j = j + S[i]
swap (S[i], S[j])
z = S[i] + S[j]

Decryption: plaintext [i] = S[z] XOR ciphertext [i]
Encryption: ciphertext [i] = S[z] XOR plaintext [i]
```

A Perl implementation of RC4 (fine for academic, but not production, purposes) can be found at *http://www.garykessler.net/software/index.html#RC4*.

In 2014, Rivest and Schuldt developed a redesign of RC4 called *Spritz*. The main operation of Spritz is similar to the main operation of RC4, except that a new variable, *w*, is added:

```
i = i + w
j = k + S [j + S[i]]
k = i + k + S[j]
swap (S[i], S[j])
z = (S[j + S[i + S[z+k]]]

Decryption: plaintext [i] = S[z] XOR ciphertext [i]
Encryption: ciphertext [i] = S[z] XOR plaintext [i]
```

As seen above, RC4 has two pointers into the S-box, namely, *i* and *j*; Spritz adds a third pointer, *k*.

Pointer *i* move slowly through the S-box; note that it is incremented by 1 in RC4 and by a constant, *w*, in Spritz. Spritz allows *w* to take on any odd value, ensuring that it is always relatively prime to 256. (In essence, RC4 sets *w* to a value of 1.)

In both ciphers, the other pointer(s) — *j* in RC4 or *j* and *k* in Spritz — move pseudorandomly through the S-box. Both ciphers have a single swap of entries in the S-box. Both also produce an output byte, *z*, as a function of the other parameters. Spritz, additionally, includes the previous value of *z* as part of the calculation of the new value of *z*.

5.14. Challenge-Handshake Authentication Protocol (CHAP)

CHAP, originally described in RFC 1994, and its variants (e.g., Microsoft's MS-CHAP) are authentication schemes that allow two parties to demonstrate knowledge of a shared secret without actually divulging that shared secret to a third party who might be eavesdropping.

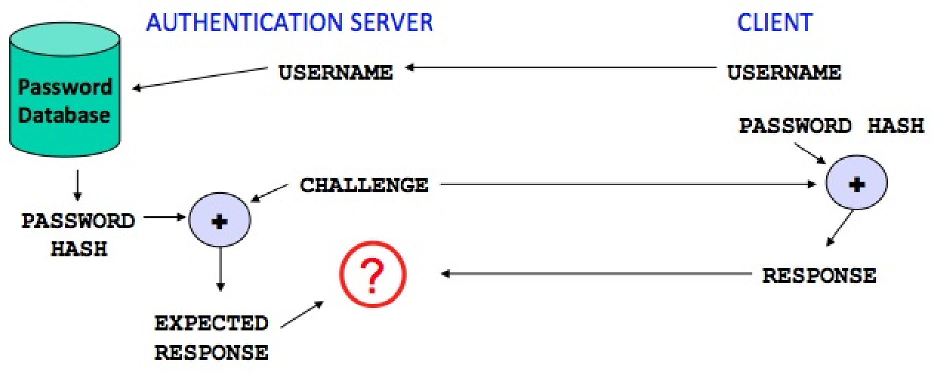


FIGURE 26: CHAP Handshake.

The operation of CHAP is relatively straight-forward. Assume that the Client (Alice) is logging on to a remote Server (Bob) across the Internet. The Client needs to prove to the Server that it knows the password but doesn't want to reveal the password in any form that an eavesdropper (Eve) can decrypt. In CHAP:

1. The User sends the password (in plaintext) to the Server.
2. The Server sends some random *challenge* string (i.e., some number of octets) to the User.
   Based upon the password and some algorithm, the User generates an encrypted *response* string (the same length as the *challenge*) and sends it to the Server.
3. The Server looks up the User's password in it's database and, using the same algorithm, generates an expected *response* string.
4. The Server compares its expected *response* to the actual *response* sent by the User. If the two match, the User is authenticated.

Since the password is never revealed to a third-party, why can't we then just keep the same password forever? Note that CHAP is potentially vulnerable to a known plaintext attack; the *challenge* is plaintext and the *response* is encrypted using the password and a known CHAP algorithm. If Eve has enough *challenge/response* pairs, she might well be able to determine the password.

6. CONCLUSION... OF SORTS

This paper has briefly described how cryptography works. The reader must beware, however, that there are a number of ways to attack every one of these systems; cryptanalysis and attacks on cryptosystems, however, are well beyond the scope of this paper. In the words of Sherlock Holmes (ok, Arthur Conan Doyle, really), "What one man can invent, another can discover" ("The Adventure of the Dancing Men").

Cryptography is a particularly interesting field because of the amount of work that is, by necessity, done in secret. The irony is that secrecy is *not* the key to the goodness of a cryptographic algorithm. Regardless of the mathematical theory behind an algorithm, the best algorithms are those that are well-known and well-documented because they are also well-tested and well-studied! In fact, *time* is the only true test of good cryptography; any cryptographic scheme that stays in use year after year is most likely a good one. The strength of cryptography lies in the choice (and management) of the keys; longer keys will resist attack better than shorter keys.

The corollary to this is that consumers should run, not walk, away from any product that uses a proprietary cryptography scheme, ostensibly because the algorithm's secrecy is an advantage. The observation that a cryptosystem should be secure even if everything about the system — except the key — is known by your adversary has been a fundamental tenet of cryptography for over 125 years. It was first stated by Dutch linguist Auguste Kerckhoffs von Nieuwenhoff in his 1883 (yes, **18**83) papers titled *La Cryptographie militaire*, and has therefore become known as "Kerckhoffs' Principle."

> *Getting a new crypto scheme accepted, marketed, and, commercially viable is always an interesting challenge. Back in ~2011, for example, a $10,000 challenge page for a new cipher called DioCipher was posted and scheduled to expire on 1 January 2013 — which it did. And that was the last that I heard of DioCipher. I leave it to the reader to consider the validity and usefulness of the public challenge process.*

7. REFERENCES AND FURTHER READING

- Bamford, J. (1983). *The Puzzle Palace: Inside the National Security Agency, America's most secret intelligence organization.* New York: Penguin Books.
- Bamford, J. (2001). *Body of Secrets : Anatomy of the Ultra-Secret National Security Agency from the Cold War Through the Dawn of a New Century.* New York: Doubleday.
- Barr, T.H. (2002). *Invitation to Cryptology.* Upper Saddle River, NJ: Prentice Hall.
- Basin, D., Cremers, C., Miyazaki, K., Radomirovic, S., & Watanabe, D. (2015, May/June). Improving the Security of Cryptographic Protocol Standards. *IEEE Security & Privacy*, *13*(3), 24:31.
- Bauer, F.L. (2002). *Decrypted Secrets: Methods and Maxims of Cryptology,* 2nd ed. New York: Springer Verlag.

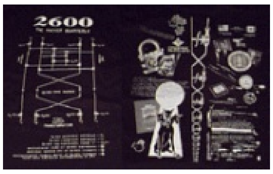
- Belfield, R. (2007). *The Six Unsolved Ciphers: Inside the Mysterious Codes That Have Confounded the World's Greatest Cryptographers.* Berkeley, CA: Ulysses Press.
- Denning, D.E. (1982). *Cryptography and Data Security.* Reading, MA: Addison-Wesley.
- Diffie, W., & Landau, S. (1998). *Privacy on the Line.* Boston: MIT Press.
- Electronic Frontier Foundation. (1998). *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. Sebastopol, CA: O'Reilly & Associates.
- Federal Information Processing Standards (FIPS) 140-2. (2001, May 25). *Security Requirements for Cryptographic Modules.* Gaithersburg, MD: National Intitute of Standards and Technology (NIST). Retrieved from http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf
- Ferguson, N., & Schneier, B. (2003). *Practical Cryptography.* New York: John Wiley & Sons.
- Ferguson, N., Schneier, B., & Kohno, T. (2010). *Cryptography Engineering: Design Principles and Practical Applications.* New York: John Wiley & Sons.
- Flannery, S. with Flannery, D. (2001). *In Code: A Mathematical Journey.* New York: Workman Publishing Company.
- Ford, W., & Baum, M.S. (2001). *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption,* 2nd ed. Englewood Cliffs, NJ: Prentice Hall.
- Garfinkel, S. (1995). *PGP: Pretty Good Privacy.* Sebastopol, CA: O'Reilly & Associates.
- Grant, G.L. (1997). *Understanding Digital Signatures: Establishing Trust over the Internet and Other Networks.* New York: Computing McGraw-Hill.
- Grabbe, J.O. (1997, October 10). Cryptography and Number Theory for Digital Cash. Retrieved from http://www-swiss.ai.mit.edu/6.805/articles/money/cryptnum.htm
- Kahn, D. (1983). *Kahn on Codes: Secrets of the New Cryptology.* New York: Macmillan.
- Kahn, D. (1996). *The Codebreakers: The Story of Secret Writing*, revised ed. New York: Scribner.
- Kaufman, C., Perlman, R., & Speciner, M. (1995). *Network Security: Private Communication in a Public World*. Englewood Cliffs, NJ): Prentice Hall.
- Koblitz, N. (1994). *A Course in Number Theory and Cryptography,* 2nd ed. New York: Springer-Verlag.
- Levy, S. (1999, April). The Open Secret. *WIRED Magazine*, *7*(4). Retrieved from http://www.wired.com/wired/archive/7.04/crypto.html
- Levy, S. (2001). *Crypto: When the Code Rebels Beat the Government — Saving Privacy in the Digital Age.* New York: Viking Press.
- Mao, W. (2004). *Modern Cryptography: Theory & Practice*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference.
- Marks, L. (1998). *Between Silk and Cyanide: A Codemaker's War, 1941-1945*. New York: The Free Press (Simon & Schuster).
- Schneier, B. (1996). *Applied Cryptography*, 2nd ed. New York: John Wiley & Sons.
- Schneier, B. (2000). *Secrets & Lies: Digital Security in a Networked World*. New York: John Wiley & Sons.
- Simion, E. (2015, January/February). The Relevance of Statistical Tests in Cryptography. *IEEE Security & Privacy*, *13*(1), 66:70.
- Singh, S. (1999). *The Code Book: The Evolution of Secrecy from Mary Queen of Scots to Quantum Cryptography.* New York: Doubleday.
- Smart, N. (2014). *Cryptography: An Introduction*, 3rd ed. Retrieved from https://www.cs.umd.edu/~waa/414-F11/IntroToCrypto.pdf
- Smith, L.D. (1943). *Cryptography: The Science of Secret Writing.* New York: Dover Publications.
- Spillman, R.J. (2005). *Classical and Contemporary Cryptology*. Upper Saddle River, NJ: Pearson Prentice-Hall.
- Stallings, W. (2006). *Cryptography and Network Security: Principles and Practice*, 4th ed. Englewood Cliffs, NJ: Prentice Hall.
- Trappe, W., & Washington, L.C. (2006). *Introduction to Cryptography with Coding Theory*, 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall.
- Young, A., & Yung, M. (2004). *Malicious Cryptography: Exposing Cryptovirology*. New York: John Wiley & Sons.

- On the Web:
  - Bob Lord's Online Crypto Museum
  - Crypto Museum
  - *Crypto-Gram* Newsletter
  - Cypherpunk -- A history
  - Internet Engineering Task Force (IETF) Security Area
    - An Open Specification for Pretty Good Privacy (openpgp)
    - Common Authentication Technology (cat)
    - IP Security Protocol (ipsec)
    - One Time Password Authentication (otp)
    - Public-Key Infrastructure (X.509) (pkix)
    - S/MIME Mail Security (smime)
    - Simple Public Key Infrastructure (spki)
    - Transport Layer Security (tls)
    - Web Transaction Security (wts)
    - Web Security (websec)
    - XML Digital Signatures (xmldsig)
  - Kerberos: The Network Authentication Protocol (MIT)
  - The MIT Kerberos & Internet trust (MIT-KIT) Consortium (MIT)
  - Peter Gutman's godzilla crypto tutorial
  - Pretty Good Privacy (PGP):
    - The GNU Privacy Guard (GPG)
    - GPGTools
    - The International PGP Home Page
    - The OpenPGP Alliance

    - RSA's Cryptography FAQ (v4.1, 2000)
    - Interspersed in RSA's Public-Key Cryptography Standards (PKCS) pages are a very good set of chapters about cryptography.
    - Ron Rivest's "Cryptography and Security" Page
    - "List of Cryptographers" from U.C. Berkeley

- Software:
    - Wei Dai's Crypto++, a free C++ class library of cryptographic primitives
    - Peter Gutman's cryptlib security toolkit
    - A Perl implementation of RC4 (for academic but not production purposes) can be found at *http://www.garykessler.net/software/index.html#RC4*.
    - A Perl program to decode Cisco type 7 passwords can be found at *http://www.garykessler.net/software/index.html#cisco7*.
    - The Rijndael page

And for a purely enjoyable fiction book that combines cryptography and history, check out Neal Stephenson's *Crytonomicon* (published May 1999). You will also find in it a new secure crypto scheme based upon an ordinary deck of cards (ok, you need the jokers...) called the Solitaire Encryption Algorithm, developed by Bruce Schneier.



Finally, I am not in the clothing business although I do have an impressive t-shirt collection (over 350 and counting!). I still proudly wear the DES (well, actually the IDEA) encryption algorithm t-shirt from *2600 Magazine* which, sadly, appears to be no longer available (left). (It was always ironic to me that *The Hacker Quarterly* got the algorithm wrong but...) A t-shirt with Adam Back's RSA Perl code can be found at *http://www.cypherspace.org/~adam/uk-shirt.html* (right).

APPENDIX. SOME MATH NOTES

A number of readers over time have asked for some rudimentary background on a few of the less well-known mathematical functions mentioned in this paper. Although this is purposely **not** a mathematical treatise, some of the math functions mentioned here are essential to grasping how modern crypto functions work. To that end, some of the mathematical functions mentioned in this paper are defined in greater detail below.

A.1. The Exclusive-OR (XOR) Function

Exclusive OR (XOR) is one of the fundamental mathematical operations used in cryptography (and many other applications). George Boole, a mathematician in the late 1800s, invented a new form of "algebra" that provides the basis for building electronic computers and microprocessor chips. Boole defined a bunch of primitive logical operations where there are one or two inputs and a single output depending upon the operation; the input and output are either TRUE or FALSE. The most elemental Boolean operations are:

- NOT: The output value is the inverse of the input value (i.e., the output is TRUE if the input is false, FALSE if the input is true)
- AND: The output is TRUE if all inputs are true, otherwise FALSE. (E.g., "the sky is blue AND the world is flat" is FALSE while "the sky is blue AND security is a process" is TRUE.)
- OR: The output is TRUE if either or both inputs are true, otherwise FALSE. (E.g., "the sky is blue OR the world is flat" is TRUE and "the sky is blue OR security is a process" is TRUE.)
- XOR (Exclusive OR): The output is TRUE if exactly one of the inputs is TRUE, otherwise FALSE. (E.g., "the sky is blue XOR the world is flat" is TRUE while "the sky is blue XOR security is a process" is FALSE.)

I'll only discuss XOR for now and demonstrate its function by the use of a so-called *truth tables*. In computers, Boolean logic is implemented in *logic gates*; for design purposes, XOR has two inputs (black) and a single output (red), and its logic diagram looks like this:

XOR		Input #1	
		0	**1**
Input #2	**0**	**0**	**1**
	1	**1**	**0**

So, in an XOR operation, the output will be a 1 if one input is a 1; otherwise, the output is 0. The real significance of this is to look at the "identity properties" of XOR. In particular, any value XORed with itself is 0 and any value XORed with 0 is just itself. Why does this matter? Well, if I take my plaintext and XOR it with a key, I get a jumble of bits. If I then take that jumble and XOR it with the same key, I return to the original plaintext.

NOTE: Boolean truth tables usually show the inputs and output as a single bit because they are based on single bit inputs, namely, TRUE and FALSE. In addition, we tend to apply Boolean operations bit-by-bit. For convenience, I have created Boolean logic tables when operating on bytes.

A.2. The *modulo* Function

The *modulo* function is, simply, the remainder function. It is commonly used in programming and is critical to the operation of any mathematical function using digital computers.

To calculate *X modulo Y* (usually written *X mod Y*), you merely determine the remainder after removing all multiples of Y from X. Clearly, the value X mod Y will be in the range from 0 to Y-1.

Some examples should clear up any remaining confusion:

- 15 mod 7 = 1
- 25 mod 5 = 0
- 33 mod 12 = 9
- 203 mod 256 = 203

Modulo arithmetic is useful in crypto because it allows us to set the size of an operation and be sure that we will never get numbers that are too large. This is an important consideration when using digital computers.

A.3. Information Theory and Entropy

Information theory is the formal study of reliable transmission of information in the least amount of space or, in the vernacular of information theory, the fewest *symbols*. For purposes of digital communication, a symbol can be a byte (i.e., an eight-bit octet) or an even smaller unit of transmission.

The father of information theory is Bell Labs scientist and MIT professor Claude E. Shannon. His seminal paper, "A Mathematical Theory of Communication" (*The Bell System Technical Journal*, Vol. 27, pp. 379-423, 623-656, July, October, 1948), defined a field that has laid the mathematical foundation for so many things that we take for granted today, from data compression, data storage and communication, and quantum computing to language processing, plagiarism detection and other linguistic analysis, and statistical modeling. And, of course, cryptography — although crypto pre-dates information theory by nearly 2000 years.

There are many everyday computer and communications applications that have been enabled by the formalization of information theory, such as:

- *Lossless data compression*, where the compressed data is an exact replication of the uncompressed source (e.g., PKZip, GIF, PNG, and WAV).
- *Lossy data compression*, where the compressed data can be used to reproduce the original uncompressed source within a certain threshold of accuracy (e.g., JPG and MP3).
- *Coding theory*, which describes the impact of bandwidth and noise on the capacity of data communication channels from modems to Digital Subscriber Line (DSL) services, why a CD or DVD with scratches on the surface can still be read, and codes used in error-correcting memory chips and forward error-correcting satellite communication systems.

One of the key concepts of information theory is that of *entropy*. In physics, entropy is a quantification of the disorder in a system; in information theory, entropy describes the uncertainty of a random variable or the randomness of an information symbol. As an example, consider a file that has been compressed using PKZip. The original file and the compressed file have the same information content but the smaller (i.e., compressed) file has more entropy because the content is stored in a smaller space (i.e., with fewer symbols) and each data unit has more randomness than in the uncompressed version. In fact, a perfect compression algorithm would result in compressed files with the maximum possible entropy; i.e., the files would contain the same number of 0s and 1s, and they would be distributed within the file in a totally unpredictable, random fashion.

As another example, consider the entropy of passwords (this text is taken from my paper, "Passwords — Strengths And Weaknesses," citing an example from *Firewalls and Internet Security: Repelling the Wily Hacker* by Cheswick & Bellovin [1994]):

> Most Unix systems limit passwords to eight characters in length, or 64 bits. But Unix only uses the seven significant bits of each character as the encryption key, reducing the key size to 56 bits. But even this is not as good as it might appear because the 128 possible combinations of seven bits per character are not equally likely; users usually do not use control characters or non-alphanumeric characters in their passwords. In fact, most users only use lowercase letters in their passwords (and some password systems are case-insensitive, in any case). The bottom line is that ordinary English text of 8 letters has an information content of about 2.3 bits per letter, yielding an 18.4-bit key length for an 8-letter passwords composed of English words. Many people choose names as a password and this yields an even lower information content of about 7.8 bits for the entire 8-letter name. As phrases get longer, each letter only adds about 1.2 to 1.5 bits of information, meaning that a 16-letter password using words from an English phrase only yields a 19- to 24-bit key, not nearly what we might otherwise expect.

Encrypted files tend to have a great deal of randomness. This is why a compressed file can be encrypted but an encrypted file cannot be compressed; compression algorithms rely on redundancy and repetitive patterns in the source file and such syndromes do not appear in encrypted files.

Randomness is such an integral characteristic of encrypted files that an entropy test is often the basis for searching for encrypted files. Not all highly randomized files are encrypted, but the more random the contents of a file, the more likely that the file is encrypted. As an example, AccessData's Forensic Toolkit (FTK), software widely used in the computer forensics field, uses the following tests to detect encrypted files:

- *Arithmetic Mean:* Calculated by summing all of the bytes in a file and dividing by the file length; if random, the value should be ~1.75.
- *X^2 Error Percent:* This distribution is calculated for a byte stream in a file; the value indicates how frequently a truly random number would exceed the calculated value.
- *Entropy:* Describes the information density (per Shannon) of a file in bits/character; as entropy approaches 8, there is more randomness.
- *MCPI Error Percent:* The Monte Carlo algorithm uses statistical techniques to approximate the value of π; a

high error rate implies more randomness.

- *Serial Correlation Coefficient:* Indicates the amount to which each byte is an e-mail relies on the previous byte. A value close to 0 indicates randomness.

SIDEBAR: An 8-bit byte has 256 possible values. The entropy of a binary file can be calculated using the following formula:

$$E(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

where n=256 and $P(x_i)$ is the probability of a byte in this file having the value *i*. A small Perl program to compute the entropy of a file can be found at entropy.zip.

Given this need for randomness, how do we ensure that crypto algorithms produce random numbers for high levels of entropy? Computers use random number generators (RNGs) for myriad purposes but computers cannot actually generate truly random sequences but, rather, sequences that have mostly random characteristics. To this end, computers use *pseudorandom number generator (PRNG)*, aka *deterministic random number generator*, algorithms. NIST has a series of documents (SP 800-90: Random Bit Generators) that address this very issue:

- *SP 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*
- *Draft SP 800-90 B: Recommendation for the Entropy Sources Used for Random Bit Generation*
- *Draft SP 800-90 C: Recommendation for Random Bit Generator (RBG) Constructions*

SIDEBAR: While the purpose of this document is to be tutorial in nature, I cannot totally ignore the disclosures of Edward Snowden in 2013 about NSA activities related to cryptography. One interesting set of disclosures is around deliberate weaknesses in the NIST PRNG standards at the behest of the NSA. NIST denies any such purposeful flaws but this will be evolving news over time. Interested readers might want to review "NSA encryption backdoor proof of concept published" (M. Lee) or "Dual_EC_DRBG backdoor: a proof of concept" (A. Adamantiadis).

Along these lines, another perspective of the Snowden disclosures relates to the impact on the world's most confidential data and critical infrastructures if governments are able to access encrypted communications. In July 2015, 14 esteemed cryptographers and computer scientists released a paper continuing the debate around cryptography and privacy. The paper, titled "Keys Under Doormats: Mandating insecurity by requiring government access to all data and communications," argues that government access to individual users' encrypted information will ultimately yield significant flaws in larger systems and infrastructures. Also check out the *N.Y. Times* article, "Security Experts Oppose Government Access to Encrypted Communication" (N. Perlroth).

For readers interested in learning more about information theory, see the following sites:

- Wikipedia entry for *Information Theory*
- A Short Course in Information Theory (Eight lectures by David J.C. MacKay)
- *Entropy and Information Theory* by Gray (Revised 1st ed., 1991). In 2011, the second edition was published.

Finally, it is important to note that information theory is an continually evolving field. There is an area of research essentially questioning the "power" of entropy in determining the strength of a cryptosystem. An interesting paper about this is "Brute force searching, the typical set and Guesswork" by Christiansen, Duffy, du Pin Calmon, & Médard (2013 IEEE International Symposium on Information Theory); a relatively non-technical overview of that paper can be found at "Encryption Not Backed by Math Anymore" by Hardesty (*DFI News*, 8/15/2013).

ABOUT THE AUTHOR

Gary C. Kessler, Ph.D., CCE, CCFP, CISSP, is the president and janitor of Gary Kessler Associates, an independent consulting and training firm specializing in computer and network security, computer forensics, Internet access issues, and TCP/IP networking. He has written over 60 papers for industry publications, is co-author of *ISDN, 4th. edition* (McGraw-Hill, 1998), and is a past editor-in-chief of the *Journal of Digital Forensics, Security and Law*. Gary is also a Professor of Cybersecurity and chair of the Security Studies & International Affairs Dept. at Embry-Riddle Aeronautical University in Daytona Beach, Florida. He is a former-member of the Vermont Internet Crimes Against Children (ICAC) Task Force and works with the North Florida ICAC, and is an Adjunct Professor at Edith Cowan University in Perth, Western Australia. Gary was formerly an Associate Professor and Program Director of the M.S. in Information Assurance program at Norwich University in Northfield, Vermont, and he started the M.S. in Digital Investigation Management and undergraduate Computer & Digital Forensics programs at Champlain College in Burlington, Vermont. Gary's e-mail address is *gck@garykessler.net* and his PGP public key can be found at *http://www.garykessler.net/pubkey.html* or on MIT's PGP keyserver (import the latest key!). Some of Gary's other crypto pointers of interest on the Web can be found at his Security-related URLs list.

ACKNOWLEDGEMENTS

An *acknowledgements* section is probably well overdue and so I apologize to all of you who have made helpful comments that remain unacknowledged. If you did make comments that I adopted that improved this paper and I have failed to recognize you, please remind me!

To get the ball rolling, thanks are offered to Sitaram Chamarty, William R. Godwin, Hugh Macdonald, and Douglas P. McNutt.