Appendix A

An Introduction to Cryptography

Cryptography is a complex and highly mathematical art and science. The basic building blocks are easy enough to understand; we caution you, however, that there are very many subtle interactions when actually using cryptosystems. This appendix is the barest introduction; even elementary cryptography cannot be covered fully here. Readers desiring a more complete treatment should consult any of a number of standard references, such as [Schneier, 1996], [Stinson, 1995], or [Menezes *et al.*, 1997]. See [Kahn, 1996] for the history of cryptography.

Selecting an encryption system is comparatively easy; actually using one is less so. That is the domain of *cryptographic protocols*, a field that is even more strewn with subtle traps than are the basic algorithms. Put bluntly, it is not a field for amateurs; if you are not experienced in the field, you will do far better using reputable published algorithms and protocols than inventing your own.

We should add a note on proprietary cryptography. On occasion, you will encounter an advertisement that brags about a firm's own, proprietary cryptographic algorithm or protocol, generally with the assertion that the system is safer precisely because it does not use well-known standards. They may be right, but don't bet on it. Good cryptosystems are very hard to create, and even systems by the best designers can have subtle (or not so subtle) flaws. You're almost always better off using a published design. Look at t this way: Why would one firm have more cryptographic expertise than the entire field?

A.1 Notation

Modern cryptosystems consist of an operation that maps a *plaintext* (P) and a *key* (K) to a *ciphertext* (C). We write this as

$$C \leftarrow K[P]$$

Usually, there is an inverse operation that maps a ciphertext and key K^{-1} to the original plaintext:

$$P \leftarrow K^{-1}[C]$$

Types of Attacks

Cryptographic systems are subject to a variety of attacks. It is impossible to provide a complete taxonomy—but we discuss a few of the more important ones.

***Cryptanalysis*:** The science—or art—of reading encrypted traffic without prior knowledge of the key.

***"Practical" cryptanalysis*:** In a sense, the converse; it refers to obtaining a key, by any means necessary.

***Rubber hose cryptanalysis*:** It may be easier to obtain a key by physical or monetary means.

***Known-plaintext attack*:** Often, an enemy will have one or more pairs of ciphertext and a known plaintext encrypted with the same key. These pairs, known as *cribs*, can be used to aid in cryptanalysis.

***Chosen-plaintext*:** Attacks in which you trick the enemy into encrypting your messages with the enemy's key. For example, if your opponent encrypts traffic to and from a file server, you can mail that person a message and watch the encrypted copy being delivered.

***Exhaustive search*:** Trying every possible key. Also known as *brute force*.

The attacker's usual goal is to recover the keys K and K^{-1}. For a strong cipher, it should be impossible to recover them by any means short of trying all possible values. This should hold true no matter how much ciphertext and plaintext the enemy has captured. (Actually, the attacker's real goal is to recover the plaintext. While recovering K^{-1} is one way to proceed, there are often many others.)

It is generally accepted that one must assume that attackers are familiar with the encryption function—if nothing else, disassembly and reverse compilation are easy—thus, the security of the cryptosystem must rely entirely on the secrecy of the keys. Protecting them is therefore of the greatest importance. In general, the more a key is used, the more vulnerable it is to compromise. Accordingly, separate keys, called *session keys*, are used for each job. Distributing session keys is a complex matter, about which we will say little; let it suffice to say that session keys are generally transmitted encrypted by a *master key*, and often come from a centralized *Key Distribution Center (KDC)*.

Types of Attacks (continued)

***Replay*:** These take a legitimate message and reinject it into the network later.

***Passive eavesdropping*:** A passive attacker simply listens to traffic flowing by.

***Active attack*:** In an active attack, the enemy can insert messages and—in some variants—delete or modify legitimate messages.

***Man-in-the-middle*:** The enemy sits between you and the party with whom you wish to communicate, and impersonates each of you to the other.

***Cut-and-paste*:** Given two messages encrypted with the same key, it is sometimes possible to combine portions of two or more messages to produce a new message. You may not know what it says, but you can use it to trick your enemy into doing something for you.

***Time-resetting*:** In protocols that use the current time, this attack will try to confuse you about what the correct time is.

***Birthday attack*:** An attack on hash functions in which the goal is to find any two messages that yield the same value. If exhaustive search takes 2^n steps, a birthday attack would take only $2^{n/2}$ tries.

***Oracle attack*:** An attacker may gains some benefit by sending queries to one party or another, using the protocol participants as *oracles*.

A.2 Secret-Key Cryptography

In conventional cryptosystems—sometimes known as *secret-key* or *symmetric* cryptosystems—there is only one key. That is,

$$K = K^{-1}$$

Writing out K^{-1} is simply a notational convenience to indicate decryption. There are many different types of symmetric cryptosystems; here, we concentrate on the *Advanced Encryption Standard (AES)* [Daemen and Rijmen, 2002]. AES is the successor to the *Data Encryption Standard (DES)* [NBS, 1977]. Several standard modes of operation were approved for DES [NBS, 1980], and while DES is no longer strong enough to be considered secure, its modes of operation are still

valid, and we continue to recommend those. A new variant, *counter mode*, has been approved for AES [NIST, 2001]. Note, though, that most things we say are applicable to other modern cipher systems, with the obvious exception of such parameters as encryption block size and key size. AES is a form of encryption system known as a *block cipher*. That is, it operates on fixed-size blocks. It maps blocks of plaintext into blocks of ciphertext and vice versa. The block lengths that are supported are 128, 196, and 256 bits, though only 128-bit blocks are standardized. The keys in AES are also variable and the same bit lengths are supported—namely, 128, 196, or 256. Any combination of block size and key size using these values is possible.

Encryption in AES is performed via substitution and transformation with 10, 12, or 14 rounds, depending on the size of the key; longer keys require more rounds of mixing. Each round of AES consists of four operations. In the first, an 8×8 *substitution box (S-box)* is applied to each byte. The second and third operations involve shifting rows and substituting columns in a data array, and in the fourth operation, bits from the key are mixed in (XORed) with the data. The data is then sent to the next round for scrambling. Decryption in AES is very simple. The same code that is used to encrypt a block is used to decrypt it. The only changes are the tables and polynomials used in each operation. The description of the algorithm is compact relative to other symmetric ciphers, and this elegance makes it simpler to analyze. This is considered one of its strengths.

The predecessor to AES, DES, was developed at IBM in response to a solicitation for a cryptographic standard from the National Bureau of Standards (NBS, now known as the National Institute of Standards and Technology, or NIST). It was originally adopted for nonclassified federal government use, effective January 15, 1978. Every five years, a recertification review was held. Clearly, though, DES is no longer adequately strong for many uses. There has been a fair amount of controversy about DES over the years; see, for example, [Diffie and Hellman, 1977]. Some have charged that the design was deliberately sabotaged by the National Security Agency (NSA), or that the key size is just small enough that a major government or large corporation could afford to build a machine that tried all 2^{56} possible keys for a given ciphertext. That said, the algorithm successfully resisted attack by civilian cryptographers for two decades. Moreover, research [Biham and Shamir, 1991, 1993] indicates that the basic design of DES is actually quite strong, and was almost certainly not sabotaged.

In 1998, a team under the auspices of the Electronic Frontier Foundation built a DES-cracker after investing less than $250,000 [Gilmore, 1998]. Full details of the design, both hardware and software, have been published. Obviously, any group interested in reading DES traffic could build its own version for rather less money. This renders DES unsuitable for keeping out any but the joy hackers.

Cryptographic key length is another arms race. Longer key lengths have been more expensive in terms of time and hardware, though cheap, fast CPUs have largely negated this issue. Brute force attacks tend to require implausibly large computation devices, often only available to the spooks in large governments.

Besides being more secure, AES is considerably faster than DES, both in hardware and in software. Unlike its predecessor, it was developed in an open process led by NIST. Candidates algorithms were solicited from the research community at large, and 15 finalists were chosen from around the world. Conferences were held to discuss all of the candidates and to narrow the list down. Eventually, there were five possibilities left, and Rijndael (a loose combination of letters

in its inventors' names, Vincent Rijmen and Joan Daemen) was selected as the best combination of security, efficiency, key agility (the cost of switching among different keys), and versatility. It is the standard for both low power and low memory devices, such as smart cards, and for high performance computers as well. To date, it is resistant to all known attacks, and no improvement over exhaustive key search is known. Given even the shortest key of 128 bits, an attacker would have to search an average of 2^{127} times to find the key. This is not feasible on today's hardware, and will not be for a long time, if ever. If you used a million processors, and each could try one key per nanosecond, it would still take over 5 quadrillion years to find the answer...

Stream ciphers operate on individual bytes. Typically (though not always), they operate by generating a running key stream that is exclusive-ORed with the data. The best-known stream cipher is RC4, devised by Rivest. It is extremely elegant and extremely fast. However, it is claimed as a trade secret by RSADSI. That notwithstanding, a source code version of RC4 is widely available on the Internet. The legal status is a bit murky; check with your own attorney.

A.3 Modes of Operation

Block ciphers, such as AES and DES, are generally used as primitive operators to implement more complex modes of operation. The five standard modes are described next. All of them can be used with any block cipher, although we have used AES in the examples.

A.3.1 Electronic Code Book Mode

The simplest mode of operation, *Electronic Code Book (ECB)* mode, is also the most obvious: AES is used, as is, on 16-byte blocks of data. Because no context goes into each encryption, every time the same 16 bytes are encrypted with the same key, the same ciphertext results. This allows an enemy to collect a "code book" of sorts, a list of 16-byte ciphertexts and their likely (or known) plaintext equivalents. Because of this danger, ECB mode should be used only for transmission of keys and initialization vectors (see below). It should *never* be used to encrypt session data.

A.3.2 Cipher Block Chaining Mode

Cipher Block Chaining (CBC) is the most important mode of operation. In CBC mode, each block of plaintext is exclusive-ORed with the previous block of ciphertext before encryption:

$$C_n \leftarrow K[P_n \oplus C_{n-1}]$$

To decrypt, we reverse the operation:

$$P_n \leftarrow K^{-1}[C_n] \oplus C_{n-1}$$

Two problems immediately present themselves: how to encrypt the first block when there is no C_0, and how to encrypt the last block if the message is not a multiple of 16 bytes in length.

To solve the first problem, both parties must agree upon an *initialization vector (IV)*. The IV acts as C_0, the first block of cipher; it is exclusive-ORed with the first block of plaintext before

encryption. Some subtle attacks are possible if IVs are not chosen properly; to be safe, IVs should be (a) chosen randomly (and not be something predictable like a counter); (b) not used with more than one other partner; and (c) either transmitted encrypted in ECB mode or chosen anew for each separate message, even to the same partner [Voydock and Kent, 1983]. A good choice is to use the last block of ciphertext from one packet as the IV for the next packet.

Apart from solving the initialization problem, IVs have another important role: They disguise stereotyped beginnings of messages. That is, if the IV is held constant, two encryptions of the same start of a message will yield the same ciphertext. This not only gives clues to cryptanalysts and traffic analysts, in some contexts it is possible to replay an intercepted message. Replays may still be possible if the IV has changed, but the attacker will not know what message to use.

Dealing with the last block is somewhat more complex. In some situations, length fields are used; in others, bytes of padding are acceptable. One useful technique is to add padding such that the last byte indicates how many of the trailing bytes should be ignored. It will thus always contain a value between 1 and 16.

A transmission error in a block of ciphertext will corrupt both that block and the following block of plaintext when using CBC mode.

A.3.3 Output Feedback Mode

For dealing with asynchronous streams of data, such as keyboard input, *output feedback (OFB)* mode is sometimes used. OFB uses AES as a random number generator, by looping its output back to its input, and exclusive-ORing the output with the plaintext:

$$\begin{aligned} AES_n &\leftarrow K[AES_{n-1}] \\ C_n &\leftarrow P_n \oplus AES_n \end{aligned}$$

If the P_n blocks are single bytes, we are, in effect, throwing away 128 bits of output from each AES cycle. In theory, the remaining bits could be kept and used to encrypt the next 16 bytes of plaintext, but that is not standard. As with CBC, an IV must be agreed on. It may be sent in the clear, because it is encrypted before use. Indeed, if it is sent encrypted, that encryption should be done with a key other than the one used for the OFB loop.

With OFB, errors do not propagate. Corruption in any received ciphertext byte will affect only that plaintext byte. However, an enemy who can control the received ciphertext can control the changes that are introduced in the plaintext: A complemented ciphertext bit will cause the same bit in the plaintext to be complemented. If this is a significant threat, OFB should be used only in conjunction with a cryptographically strong *message authentication code (MAC)*.

A serious threat is lurking here. (In fact, the same threat applies to most other stream ciphers, including RC4.) If the same key and IV pair are ever reused, an attacker can exclusive-OR the two ciphertext sequences together, producing an exclusive-OR of the two plaintexts. This is fairly easy to split into its component pieces of plaintext.

A.3.4 Cipher Feedback Mode

Cipher Feedback (CFB) mode is a more complex mechanism for encrypting streams. If we are encrypting 128-bit blocks, we encipher as follows:

$$C_n \leftarrow P_n \oplus K[C_{n-1}]$$

Decryption is essentially the same operation:

$$P_n \leftarrow C_n \oplus K[C_{n-1}]$$

That is, the last ciphertext block sent or received is fed back into the encryptor. As in OFB mode, AES is used in encryption mode only.

If we are sending 8-bit bytes, CFB_8 mode is used. The difference is that the input to the AES function is from a shift register; the 8 bits of the transmitted ciphertext are shifted in from the right, and the leftmost 8 bits are discarded.

Errors in received CFB data affect the decryption process while the garbled bits are in the shift register. Thus, for CFB_8 mode, 9 bytes are affected. The error in the first of these 9 bytes can be controlled by the enemy.

As with OFB mode, the IV for CFB encryption may, and arguably should, be transmitted in the clear.

A.3.5 Counter Mode

Counter mode is a new mode of operation suitable for use with AES. The underlying block cipher is used to encrypt a counter T. If the starting counter for plaintext block m is T_m:

$$\begin{aligned} C_i &\leftarrow P_i \oplus K[T_m] \\ T_m &\leftarrow T_m + 1 \end{aligned}$$

where P_i represents the AES blocks of a single message.

A new T_m is picked for each message. While there is no mandatory mechanism for picking these counters, care is needed: Counter mode is a stream cipher, with all the dangers that implies if a counter is ever reused. The usual scheme is to divide T into two sections. The left-hand section is a per-message value; it can either be a message counter or some pseudorandom value. The right-hand section is the count of blocks within a message. It must be long enough to handle the longest message possible.

The advantage of counter mode is that it's parallelizable. That is, each block within a message can be encrypted or decrypted simultaneously with any other block. This allows a hardware designer to throw lots of chips at the problem of very high speed cryptography. The older modes, such as CBC, are limited to a "mere" 2.5 Gbps with the chips currently available.

Unfortunately, no authentication algorithms run faster than that, and stream ciphers are extremely vulnerable if used without authentication. To our minds, this makes counter mode of questionable utility [Bellovin and Blaze, 2001].

A.3.6 One-Time Passwords

Conventional cryptosystems are often used to implement the authentication schemes described in Chapter 7. In a challenge/response authenticator, the user's token holds the shared secret key K. The challenge *Ch* acts as plaintext; both the token and the host calculate $K[Ch]$. Assuming that a strong cryptosystem is used, there is no way to recover K from the challenge/response dialogue.

A similar scheme is used with time-based authenticators. The clock value T is the plaintext; $K[T]$ is displayed.

PINs can be implemented in either form of token in a number of different ways. One technique is to use the PIN to encrypt the device's copy of K. An incorrect PIN will cause an incorrect copy of K to be retrieved, thereby corrupting the output. Note that the host does not need to know the PIN, and need not be involved in PIN-change operations.

A.3.7 Master Keys

It is worth taking extra precautions with sensitive information, especially when using master keys. An enemy who cracks a session key can read that one session, but someone who cracks a master key can read all traffic—past, present, and future. The most sensitive message of all is a session key encrypted by a master key, as two brute force attacks—first to recover the session key and then to match that against its encrypted form—will reveal the master [Garon and Outerbridge, 1991]. Accordingly, *triple encryption* or use of a longer key length is recommended if you think your enemy is well financed.

A.4 Public Key Cryptography

With conventional cipher systems, both parties must share the same secret key before communication begins. This is problematic. For one thing, it is impossible to communicate with someone with whom you have no prior arrangements. Additionally, the number of keys needed for a complete communications mesh is very large, n^2 keys for an n-party network. While both problems can be solved by recourse to a trusted, centralized key distribution center, KDCs are not panaceas. If nothing else, the KDC must be available in real time to initiate a conversation. This makes KDC access difficult for store-and-forward message systems.

Public key, or asymmetric, cryptosystems [Diffie and Hellman, 1976] offer a different solution. In such systems, $K \neq K^{-1}$. Furthermore, given K, the encryption key, it is not feasible to discover the decryption key K^{-1}. We write encryption as

$$C \leftarrow E_A[P]$$

and decryption as

$$P \leftarrow D_A[C]$$

for the keys belonging to A.

Each party publishes its encryption key in a directory, while keeping its decryption key secret. To send a message to someone, simply look up their public key and encrypt the message with that key.

The best known, and most important, public key cryptosystem is RSA, named for its inventors, Ronald Rivest, Adi Shamir, and Leonard Adleman [Rivest *et al.*, 1978]. Its security relies on the difficulty of factoring very large numbers. For many years, RSA was protected by a U.S. patent that expired in September, 2000; arguably, this held back its deployment.

To use RSA, pick two large prime numbers p and q; each should be at least several hundred bits long. Let $n = pq$. Pick some random integer d relatively prime to $(p-1)(q-1)$, and e such that

$$ed \equiv 1 \pmod{(p-1)(q-1)}$$

That is, when the product ed is divided by $(p-1)(q-1)$, the remainder is 1.

We can now use the pair (e, n) as the public key, and the pair (d, n) as the private key. Encryption of some plaintext P is performed by exponentiation modulo n:

$$C \leftarrow P^e \pmod{n}$$

Decryption is the same operation, with d as the exponent:

$$\begin{aligned} P \leftarrow C^d \pmod{n} &\equiv (P^e)^d \pmod{n} \\ &\equiv P^{ed} \pmod{n} \\ &\equiv P \pmod{n} \end{aligned}$$

No way to recover d from e is known that does not involve factoring n, and that is believed to be a very difficult operation. (Oddly enough, primality testing is very much easier than factoring.)

Securely building a message to use with RSA is remarkably difficult. Published standards such as PKCS 1 [RSA Laboratories, 2002] should generally be used.

Public key systems suffer from two principal disadvantages. First, the keys are very large compared with those of conventional cryptosystems. This might be a problem when it comes to entering or transmitting the keys, especially over low-bandwidth links. Second, encryption and decryption are much slower. Not much can be done about the first problem. The second is dealt with by using such systems primarily for key distribution. Thus, if A wanted to send a secret message M to B, A would transmit something like

$$E_B[K], K[M] \tag{A.1}$$

where K is a randomly generated session key for DES or some other conventional cryptosystem.

A.5 Exponential Key Exchange

A concept related to public key cryptography is *exponential key exchange*, sometimes referred to as the *Diffie-Hellman* algorithm [Diffie and Hellman, 1976]. Indeed, it is an older algorithm; the scheme was first publicly described in the same paper that introduced the notion of public key cryptosystems, but without providing any examples.

Exponential key exchange provides a mechanism for setting up a secret but *unauthenticated* connection between two parties. That is, the two can negotiate a secret session key, without fear

of eavesdroppers. However, neither party has any strong way of knowing who is really at the other end of the circuit.

In its most common form, the protocol uses arithmetic operations in the *field* of integers modulo some large number β. When doing arithmetic $\pmod{\beta}$, you perform the operation as usual, but then divide by β, discarding the quotient and keeping the remainder. In general, you can do the arithmetic operations either before or after taking the remainder. Both parties must also agree on some integer α, $1 < \alpha < \beta$.

Suppose A wishes to talk to B. They each generate secret random numbers, R_A and R_B. Next, A calculates and transmits to B the quantity

$$\alpha^{R_A} \pmod{\beta}$$

Similarly, B calculates and transmits

$$\alpha^{R_B} \pmod{\beta}$$

Now, A knows R_A and $\alpha^{R_B} \pmod{\beta}$, and hence can calculate

$$\begin{aligned} (\alpha^{R_B})^{R_A} \pmod{\beta} &\equiv \alpha^{R_B R_A} \pmod{\beta} \\ &\equiv \alpha^{R_A R_B} \pmod{\beta} \end{aligned}$$

Similarly, B can calculate the same value. An outsider cannot; the task of recovering R_A from $\alpha^{R_A} \pmod{\beta}$ is believed to be very hard. (This problem is known as the *discrete logarithm* problem.) Thus, A and B share a value known only to them; it can be used as a session key for a symmetric cryptosystem.

Again, caution is indicated when using exponential key exchange. As noted, there is no authentication provided; *anyone* could be at the other end of the circuit, or even in the middle, relaying messages to each party. Simply transmitting a password over such a channel is risky, because of "man-in-the-middle" attacks. There are techniques for secure transmission of authenticating information when using exponential key exchange; see, for example, [Rivest and Shamir, 1984; Bellovin and Merritt, 1992, 1993, 1994]. They are rather more complex and still require some prior knowledge of authentication data.

A.6 Digital Signatures

Often, the source of a message is at least as important as its contents. Digital signatures can be used to identify the source of a message. Like public key cryptosystems, digital signature systems employ public and private keys. The sender of a message uses a private key to sign it; this signature can be verified by means of the public key.

Digital signature systems do not necessarily imply secrecy. Indeed, a number of them do not provide it. However, the RSA cryptosystem can be used for both purposes.

To sign a message with RSA, the sender *decrypts* it, using a private key. Anyone can verify—and recover—this message by *encrypting* with the corresponding public key. (The mathematical

operations used in RSA are such that one can decrypt plaintext, and encrypt to recover the original message.) Consider the following message:

$$E_B[D_A[M]]$$

Because it is encrypted with B's public key, only B can strip off the outer layer. Because the inner section $D_A[M]$ is encrypted with A's private key, only A could have generated it. We therefore have a message that is both private and authenticated. We write a message M signed by A as

$$S_A[M]$$

There are a number of other digital signature schemes besides RSA. The most important one is the *Digital Signature Standard (DSS)* adopted by NIST [1994]. Apparently by intent, its keys cannot be used to provide secrecy, only authentication. It has been adopted as a U.S. federal government standard.

How does one know that the published public key is authentic? The cryptosystems themselves may be secure, but that matters little if an enemy can fool a publisher into announcing the wrong public keys for various parties. That is dealt with via *certificates*. A certificate is a combination of a name and a public key, collectively signed by another, and more trusted, party T:

$$S_T[A, E_A]$$

That signature requires its own public key of course. It may require a signature by some party more trusted yet, and so on:

$$S_{T_1}[A, E_A]$$
$$S_{T_2}[T_1, E_{T_1}]$$
$$S_{T_3}[T_2, E_{T_2}]$$

Certificates may also include additional information, such as the key's expiration date. One should not use any one key for too long for fear of compromise, and one does not want to be tricked into accepting old, and possibly broken, keys.

While there are many ways to encode certificates, the most common is described in the X.509 standard. X.509 is far too complex, in both syntax and semantics, to describe here. Interested readers should see [Feghhi *et al.*, 1998]; the truly dedicated can read the formal specification. A profile of X.509 for use in the Internet is described in [Adams *et al.*, 1999].

A concept related to digital signatures is that of the MAC. A MAC is a symmetric function that takes a message and a key as input, and produces a unique value for the message and the key. Of course, because MAC outputs are finite and messages are infinite, the value cannot really be unique, but if the length of the output is high enough, the value can be viewed as unique for all practical purposes. It is essentially a fancy checksum.

When MACs are used with encrypted messages, the same key should not be used for both encryption and message authentication. Typically, some simple transform of the encryption key, such as complementing some of the bits, is used in the MAC computation, though this may be a bad idea if the secrecy algorithm is weak.

A.7 Secure Hash Functions

It is often impractical to apply a signature algorithm to an entire message. A function like RSA can be too expensive for use on large blocks of data. In such cases, a *secure hash function* can be employed. A secure hash function has two interesting properties. First, its output is generally relatively short—on the order of 128 to 512 bits. Second, and more important, it must be unfeasible to create an input that will produce an arbitrary output value. Thus, an attacker cannot create a fraudulent message that is authenticated by means of an intercepted genuine hash value.

Secure hash functions are used in two main ways. First, and most obvious, any sort of digital signature technique can be applied to the hash value instead of to the message itself. In general, this is a much cheaper operation, simply because the input is so much smaller. Thus, if A wished to send to B a signed version of message (A.1), A would transmit

$$E_B[K], K[M], S_A[H(M)]$$

where H is a secure hash function. (As before, K is the secret key used to encrypt the message itself.) If, instead, it sent

$$E_B[K], K[M, S_A[H(M)]]$$

the signature, too, and hence the origin of the message, will be protected from all but B's eyes.

The second major use of secure hash functions is less obvious. In conjunction with a shared secret key, the hash functions themselves can be used to authenticate messages. By prepending the secret key to the desired message, and then calculating the hash value, one produces a signature that cannot be forged by a third party:

$$H(M, K) \tag{A.2}$$

where K is a shared secret string and M is the message to be signed.

This concept extends in an obvious way to challenge/response authentication schemes. Normally, in response to a challenge C_A from A, B would respond with $K[C_A]$, where K is a shared key. The same effect can be achieved by sending something like $H(C_A, K)$ instead. This technique has sometimes been used to avoid export controls on encryption software: Licenses to export authentication technology, as opposed to secrecy technology, are easy to obtain.

It turns out that using just $H(C_A, K)$ is not quite secure enough. A simple modification, known as HMAC [Bellare *et al.*, 1996], is considerably better, and only slightly more expensive. An HMAC is calculated by

$$H(H(C_A, K'), K'')$$

where K' and K'' are padded versions of the secret key.

(It's also possible to build a MAC from a block cipher. The current scheme of choice is RMAC, described in a draft NIST recommendation [NIST, 2002]. But RMAC has been shown to be weak under certain circumstances.)

It is important that secure hash functions have an output length of at least 128 bits. If the output value is too short, it is possible to find two messages that hash to the same value. This is much easier than finding a message with a given hash value. If a brute force attack on the latter takes 2^m operations, a birthday attack takes just $2^{m/2}$ tries. If the hash function yielded as

short an output value as DES, two collisions of this type could be found in only 2^{32} tries. That's far too low. (The term "birthday attack" comes from the famous *birthday paradox*. On average, there must be 183 people in a room for there to be a 50% probability that someone has the same birthday as you, but only 23 people are needed for there to be a 50% probability that *some* two people share the same birthday.)

There are a number of well-known hash functions from which to choose. Some care is needed, because the criteria for evaluating their security are not well established [Nechvatal, 1992]. Among the most important such functions are MD5 [Rivest, 1992b], RIPEMD-160 [Dobbertin *et al.*, 1996], and NIST's Secure Hash Algorithm [NIST, 1993], a companion to its digital signature scheme. Hints of weakness have shown up in MD5 and RIPEMD-160; cautious people will eschew them, though none of the attacks are of use against either function when used with HMAC. As of this writing, the NIST algorithm appears to be the best choice. For many purposes, the newer versions of SHA are better; these have block sizes ranging from 256 to 512 bits.

On occasion, it has been suggested that a MAC calculated with a known key is a suitable hash function. Such usages are not secure [Winternitz, 1984; Mitchell and Walker, 1988]. Secure hash functions can be derived from block ciphers, but a more complex function is required [Merkle, 1990].

A.8 Timestamps

Haber and Stornetta [Haber and Stornetta, 1991a, 1991b] have shown how to use secure hash functions to implement a *digital timestamp* service. Messages to be timestamped are *linked* together. The hash value from the previous timestamp is used in creating the hash for the next one.

Suppose we want to timestamp document D_n at some time T_n. We create a *link value* L_n by calculating

$$L_n \leftarrow H(T_n, H(D_n), n, L_{n-1})$$

This value L_n serves as the timestamp. The time T_n is, of course, unreliable; however, L_n is used as an input when creating L_{n+1}, and uses L_{n-1} as an input value. The document D_n must therefore have been timestamped before D_{n+1} and after D_{n-1}. If these documents belonged to a company other than D_n, the evidence is persuasive. The entire sequence can be further tied to reality by periodically publishing the link values. Surety does just that, in a legal notice in the *New York Times*.[1]

Note, incidentally, that one need not disclose the contents of a document to secure a timestamp; a hash of it will suffice. This preserves the secrecy of the document, but proves its existence at a given point in time.

1. This scheme has been patented.