1.3 Signature Schemes

A signature scheme is a method of signing a message stored in electronic form. As such, a signed message can be transmitted over a computer network. However, there are some fundamental differences between conventional and digital signatures.

First is the question of signing a document. With a conventional signature, a signature is part of the physical document being signed. However, a digital signature is not attached physically to the message that is signed, so the algorithm that is used must somehow "bind" the signature to the message.

Second is the question of verification. A conventional signature is verified by comparing it to other, authentic signatures. For example, when someone signs a credit card purchase, the salesperson is supposed to compare the signature on the sales slip to the signature on the back of the credit card in order to verify the signature. Of course, this is not a very secure method as it is relatively easy to forge someone else's signature. Digital signatures, on the other hand, can be verified using a publicly known verification algorithm. Thus, "anyone" can verify a digital signature. A signature scheme will be considered to be secure if it prevents an adversary from constructing valid forgeries.

Another fundamental difference between conventional and digital signatures is that a "copy" of a signed digital message is identical to the original. On the other hand, a copy of a signed paper document can usually be distinguished from an original. This feature means that care must be taken to prevent a signed digital message from being reused. For example, if Alice signs a digital message authorizing Bob to withdraw $100 from her bank account (i.e., a check), she only wants Bob to be able to do so once. So the message itself should contain information, such as a date, that prevents it from being reused.

A signature scheme consists of two components: a signing algorithm and a verification algorithm. Alice can sign a message x using a (private) signing algorithm sig_K, which depends on a key, K. The resulting signature, $\mathsf{sig}_K(x)$, can subsequently be verified using a public verification algorithm ver_K. Given a pair (x, y), the verification algorithm returns an answer "true" or "false" depending on whether the signature is valid.

Here is a formal definition of a signature scheme.

Definition 1.2: A *signature scheme* is a five-tuple $(\mathcal{P}, \mathcal{A}, \mathcal{K}, \mathcal{S}, \mathcal{V})$, where the following conditions are satisfied:

1. $\mathcal{P}$ is a finite set of possible *messages*
2. $\mathcal{A}$ is a finite set of possible *signatures*
3. $\mathcal{K}$, the *keyspace*, is a finite set of possible *keys*
4. For each $K \in \mathcal{K}$, there is a *signing algorithm* $\mathsf{sig}_K \in \mathcal{S}$ and a corresponding *verification algorithm* $\mathsf{ver}_K \in \mathcal{V}$. Each $\mathsf{sig}_K : \mathcal{P} \to \mathcal{A}$ and $\mathsf{ver}_K : \mathcal{P} \times \mathcal{A} \to \{true, false\}$ are functions such that the following equation is satisfied for every message $x \in \mathcal{P}$ and for every signature $y \in \mathcal{A}$:
$$\mathsf{ver}_K(x, y) = \begin{cases} true & \text{if } y = \mathsf{sig}_K(x) \\ false & \text{if } y \neq \mathsf{sig}_K(x). \end{cases}$$
A pair (x, y) with $x \in \mathcal{P}$ and $y \in \mathcal{A}$ is called a *signed message*.

For various reasons having to do both with efficiency and security, signature schemes are almost always used with in conjunction with a fast public cryptographic hash function. (We will study hash functions in more detail a bit later.) A signature scheme is typically designed to sign a message of fixed length, which usually is fairly small. Therefore some means must be specified to sign messages of arbitrary length. This can be done, using a hash function, as follows.

The hash function $h : \{0, 1\}^* \to \mathcal{Z}$ will take a binary message of arbitrary length and produce a binary message digest of a specified size (160 bits is a popular choice, in which case we would have $\mathcal{Z} = \{0, 1\}^{160}$). The message digest will then be signed using a signature scheme $(\mathcal{P}, \mathcal{A}, \mathcal{K}, \mathcal{S}, \mathcal{V})$, where $\mathcal{Z} \subseteq \mathcal{P}$.

Suppose Alice wants to sign a message x, which is a bitstring of arbitrary length. She first constructs the message digest $z = h(x)$, and then computes the signature on z, namely, $y = sig_K(z)$. Then she transmits the ordered pair (x, y) over the channel. Now the verification can be performed (by anyone) by first reconstructing the message digest $z = h(x)$ using the public hash function h, and then checking that $ver_K(z, y) = true$.

The use of a signature scheme, both with and without a hash function, is depicted diagramatically, as follows:

Alice		Bob
K	without hash function	ver_K
$y = sig_K(x)$	$\xrightarrow{\quad x, y \quad}$	$ver_K(x, y) = true?$
K, h	with hash function	ver_K, h
$z = h(x)$ $y = sig_K(z)$	$\xrightarrow{\quad x, y \quad}$	$z = h(x)$ $ver_K(z, y) = true?$

In practice, the specific hash function *SHA-1* is often used in conjuction with signature schemes. *SHA-1* produces a 160-bit message digest.

1.3.1 Two Signature Schemes: RSA and DSS

We will now give two examples of signature schemes, the *RSA signature scheme* and the *Digital signature scheme* (DSS). The *RSA signature scheme* is defined as follows:

Cryptosystem 1.4: *RSA Signature Scheme*

Let $n = pq$, where p and q are primes. Let $\mathcal{P} = \mathcal{A} = \mathbb{Z}_n$, and define

$$\mathcal{K} = \{(n, p, q, a, b) : n = pq, p, q \text{ prime}, ab \equiv 1 \pmod{\phi(n)}\}.$$

The values n and b are the public key, and the values p, q, a are the private key.

For $K = (n, p, q, a, b)$, define

$$\mathsf{sig}_K(x) = x^a \bmod n$$

and

$$\mathsf{ver}_K(x, y) = true \Leftrightarrow x \equiv y^b \pmod{n}$$

$(x, y \in \mathbb{Z}_n)$.

The set-up of the *RSA signature scheme* is identical to that of the *RSA cryptosystem*. The encryption operation of the *RSA cryptosystem* is identical to the verification operation of the *RSA signature scheme* (i.e., it is a modular exponentiation using the public exponent, b). Similarly, the decryption operation of the *RSA cryptosystem* is identical to the signing operation of the *RSA signature scheme* (it uses the private exponent, a). The proof that the the signature scheme works correctly (i.e., that correctly signed messages will be verified) is the same as the proof that the corresponding cryptosystem works corrrectly (i.e., that correctly encrypted plaintexts can be decrypted).

As is the case with the *RSA cryptosystem*, secure practical implementations of the signature scheme will use two 512-bit primes to form the 1024-bit modulus $n = pq$. Signatures, being elements of $\mathbb{Z}_n$, are binary strings of length 1024.

Another signature scheme widely used in practice, which provides for shorter signatures than *RSA*, is the *Digital signature standard* (or DSS). The *DSS* does not correspond to a cryptosystem as *RSA* does; it is a “special-purpose” signature scheme.

We need to mention a couple of relevant facts about modular arithmetic in $\mathbb{Z}_p$ before describing the algorithm. Suppose p is prime. Then it is known that there exists a *primitive element* in $\mathbb{Z}_p$. This is an element $\alpha \in \mathbb{Z}_p$ such that

$$\{\alpha^i \bmod p : 0 \leq i \leq p-2\} = \mathbb{Z}_p \backslash \{0\}.$$

In other words, the powers of α (modulo p) yield all the non-zero elements in $\mathbb{Z}_p$. It will be the case that $\alpha^{p-1} \equiv 1 \pmod p$, and α is said to have order $p-1$. (In general, the *order* of an element $\beta \in \mathbb{Z}_p \backslash \{0\}$ is the least positive integer j such taht $\beta^j \equiv 1 \pmod p$.)

Example 1.3 Suppose that $p = 13$ and let $\alpha = 2$. It can be verified, by checking all the powers of 2 modulo 13, that 2 is a primitive element modulo 13:

$$\begin{aligned}
2^0 \bmod 13 &= 1\\
2^1 \bmod 13 &= 2\\
2^2 \bmod 13 &= 4\\
2^3 \bmod 13 &= 8\\
2^4 \bmod 13 &= 3\\
2^5 \bmod 13 &= 6\\
2^6 \bmod 13 &= 12\\
2^7 \bmod 13 &= 11\\
2^8 \bmod 13 &= 9\\
2^9 \bmod 13 &= 5\\
2^{10} \bmod 13 &= 10\\
2^{11} \bmod 13 &= 7.
\end{aligned}$$

The reader can also confirm that the primitive elements modulo 13 are 2, 6, 7 and 11. □

Suppose that p is prime and $\alpha \in \mathbb{Z}_p$ is a primitive element. Suppose that $q \mid (p-1)$, and let $\beta = \alpha^{(p-1)/q} \bmod p$. Then it is not hard to see that β has order q.

Here is a description of *Digital signature standard*:

Cryptosystem 1.5: *Digital Signature Standard*

Let p be a 1024-bit prime and let q be a 160-bit prime that divides $p-1$. Let $\alpha \in \mathbb{Z}_p{}^*$ be an element of order q. Let $\mathcal{P} = \{0,1\}^*$, $\mathcal{A} = \mathbb{Z}_q{}^* \times \mathbb{Z}_q{}^*$, and define

$$\mathcal{K} = \{(p, q, \alpha, a, \beta) : \beta \equiv \alpha^a \pmod{p}\},$$

where $0 \leq a \leq q-1$. The values p, q, α and β are the public key, and a is the private key.

For $K = (p, q, \alpha, a, \beta)$, and for a (secret) random number k, $1 \leq k \leq q-1$, define

$$\mathsf{sig}_K(x, k) = (\gamma, \delta),$$

where

$$\gamma = (\alpha^k \bmod p) \bmod q \quad \text{and}$$
$$\delta = (\text{SHA-1}(x) + a\gamma)k^{-1} \bmod q.$$

(If $\gamma = 0$ or $\delta = 0$, a new random value of k should be chosen.)

For $x \in \{0,1\}^*$ and $\gamma, \delta \in \mathbb{Z}_q{}^*$, verification is done by performing the following computations:

$$e_1 = \text{SHA-1}(x)\,\delta^{-1} \bmod q$$
$$e_2 = \gamma\,\delta^{-1} \bmod q$$
$$\mathsf{ver}_K(x, (\gamma, \delta)) = \mathit{true} \Leftrightarrow (\alpha^{e_1}\beta^{e_2} \bmod p) \bmod q = \gamma.$$

A signature in the *DSS* is 320 bits in length. Notice that if Alice computes a value $\delta \equiv 0 \pmod{q}$ in the DSS signing algorithm, she will reject it and construct a new signature with a new random k. This is not likely to cause a problem in practice: the probability that $\delta \equiv 0 \pmod{q}$ is likely to be on the order of 2^{-160}; so for all intents and purposes it will almost never happen.

It is not at all obvious at first glance that correctly formed signatures will be verified. This requires some algebraic manipulation and a bit of number theory to prove it. First, we provide a small toy example as an illustration. In this example, p and q are much smaller than they would be in the "real" *DSS*).

Example 1.4 Suppose we take $q = 101$ and $p = 78q + 1 = 7879$. 3 is a primitive element in $\mathbb{Z}_{7879}$, so we can take

$$\alpha = 3^{78} \bmod 7879 = 170.$$

Suppose $a = 75$; then

$$\beta = \alpha^a \bmod 7879 = 4567.$$

Suppose Alice wants to sign the message digest $\text{SHA-1}(x) = 22$, and she chooses the random value $k = 50$. Then she computes

$$k^{-1} \bmod 101 = 50^{-1} \bmod 101 = 99,$$

$$\begin{aligned}\gamma &= (170^{50} \bmod 7879) \bmod 101 \\ &= 2518 \bmod 101 \\ &= 94,\end{aligned}$$

and

$$\begin{aligned}\delta &= (22 + 75 \times 94)99 \bmod 101 \\ &= 97.\end{aligned}$$

The signature $(94, 97)$ on the message digest 22 is verified by the following computations:

$$\begin{aligned}&\delta^{-1} = 97^{-1} \bmod 101 = 25 \\ &e_1 = 22 \times 25 \bmod 101 = 45 \\ &e_2 = 94 \times 25 \bmod 101 = 27 \\ &(170^{45} 4567^{27} \bmod 7879) \bmod 101 = 2518 \bmod 101 = 94.\end{aligned}$$

Hence, the signature is valid. $\square$

We now give a mathematical proof that a *DSS* signature constructed using sig_K will successfully be verified using ver_K. We use the facts that $\beta = \alpha^a \bmod q$, and all exponents of α can be reduced modulo q (which follows because α has order q). First, observe that

$$\alpha^{e_1}\beta^{e_2} \equiv \alpha^{e_1 + a\, e_2} \pmod p.$$

The exponent, $e_1 + a\, e_2$ can be simplified (modulo q) as follows:

$$\begin{aligned}e_1 + a\, e_2 &\equiv \delta^{-1}(\text{SHA-1}(x) + a\,\gamma) \pmod q \\ &\equiv \delta^{-1} k\, \delta \pmod q \\ &\equiv k \pmod q.\end{aligned}$$

Hence,

$$\alpha^{e_1}\beta^{e_2} \equiv \alpha^k \pmod p,$$

and therefore

$$(\alpha^{e_1}\beta^{e_2} \bmod p) \bmod q = (\alpha^k \bmod p) \bmod q = \gamma,$$

as required.

1.4 Hash Functions and Message Authentication Codes

A cryptographic hash function is used to construct a short "fingerprint" of some data; if the data is altered, then the fingerprint will no longer be valid. Even if the data is stored in an insecure place, its integrity can be checked from time to time by recomputing the fingerprint and verifying that the fingerprint has not changed.

Let h be a hash function and let x be some data. As an illustrative example, x could be a binary string of arbitrary length. The corresponding fingerprint is defined to be $y = h(x)$. This fingerprint is often referred to as a *message digest*. A message digest would typically be a fairly short binary string; as mentioned previously, 160 bits is a common choice.

We suppose that y is stored in a secure place, but x is not. If x is changed, to x', say, then we hope that the "old" message digest, y, is not also a message digest for x'. If this is indeed the case, then the fact that x has been altered can be detected simply by computing the message digest $y' = h(x')$ and verifying that $y' \neq y$.

The motivating example discussed above assumes the existence of a single, fixed hash function. It is also useful to study families of keyed hash functions. A keyed hash function is often used as a *message authentication code*, or MAC. Suppose that Alice and Bob share a secret key, K, which determines a hash function, say h_K. For a message, say x, the corresponding authentication tag, $y = h_K(x)$, can be computed by Alice or Bob. The pair (x, y) can be transmitted over an insecure channel from Alice to Bob. When Bob receives the pair (x, y), he can verify if $y = h_K(x)$. If this condition holds, then he is confident that neither x nor y was altered by an adversary, provided that the hash family is "secure."

Notice the distinction between the type of assurance of data integrity provided by an unkeyed, as opposed to a keyed, hash function. In the case of an unkeyed hash function, the message digest must be securely stored so it cannot be altered. On the other hand, if Alice and Bob use a secret key K to specify the hash function they are using, then they can transmit both the data and the authentication tag over an insecure channel.

The fact that keyed and unkeyed hash functions are used for different purposes means that they will require different properties in order to be considered "secure". Briefly and informally, security for keyed hash functions will mean that it is difficult for an adversary to forge a valid authentication tag (assuming the key is not known). In the case of an unkeyed hash function, h, the following properties turn out to be desirable, especially when hash functions are used in context of signature schemes:

collision resistance
: It should be difficult to find $x \neq x'$ such that $h(x) = h(x')$.

preimage resistance
: Given a randomly chosen message digest, y, it should be difficult to find x such that $h(x) = y$.

We will have more to say about these security conditions later on.

We now give definitions for a keyed hash family.

Definition 1.3: A *hash family* is a four-tuple $(\mathcal{X}, \mathcal{Y}, \mathcal{K}, \mathcal{H})$, where the following conditions are satisfied:

1. $\mathcal{X}$ is a set of possible *messages*
2. $\mathcal{Y}$ is a finite set of possible *message digests* or *authentication tags*
3. $\mathcal{K}$, the *keyspace*, is a finite set of possible *keys*
4. For each $K \in \mathcal{K}$, there is a *hash function* $h_K \in \mathcal{H}$. Each $h_K : \mathcal{X} \to \mathcal{Y}$.

In the above definition, $\mathcal{X}$ could be a finite or infinite set; $\mathcal{Y}$ is always a finite set. In this situation, we will always assume that $|\mathcal{X}| \geq |\mathcal{Y}|$, and we will often assume the stronger condition that $|\mathcal{X}| \geq 2|\mathcal{Y}|$.

An *unkeyed hash function* is a function $h : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are the same as in Definition 1.3. We could think of an unkeyed hash function simply as a hash family in which there is only one possible key, i.e., one in which $|\mathcal{K}| = 1$.

The following diagram compares signature schemes to message authentication codes.

Alice		Bob
K, h	signature scheme	ver_K, h
$z = h(x)$ $y = sig_K(z)$	$\xrightarrow{x,\,y}$	$z = h(x)$ $ver_K(z, y) = true?$
K	MAC	K
$y = h_K(x)$	$\xrightarrow{x,\,y}$	$y = h_K(x)?$

1.4.1 Iterated Hash Functions

If $\mathcal{X}$ is a finite set, an unkeyed hash function is sometimes called a *compression function*. We now study a particular technique by which a compression function, say compress, can be extended to a hash function with an infinite domain, h. A hash function h constructed by this method is called an *iterated hash function*.

In this section, we restrict our attention to hash functions whose inputs and outputs are bitstrings (i.e., strings formed of zeroes and ones). We denote the length of a bitstring x by $|x|$, and the concatenation of bitstrings x and y is denoted $x \parallel y$.

Suppose that $\mathsf{compress} : \{0,1\}^{m+t} \to \{0,1\}^m$ is a compression function (where $t \geq 1$). The evaluation of the iterated hash function h based on compress will consist of three main steps:

preprocessing

Given an input string x, where $|x| \geq m + t + 1$, construct a string y, using a public algorithm, such that $|y| \equiv 0 \pmod{t}$. Denote

$$y = y_1 \parallel y_2 \parallel \cdots \parallel y_r,$$

where $|y_i| = t$ for $1 \leq i \leq r$.

processing

Let IV be a public initial value which is a bitstring of length m. Then compute the following:

$$\begin{aligned}
z_0 &\leftarrow \mathrm{IV} \\
z_1 &\leftarrow \mathsf{compress}(z_0 \parallel y_1) \\
z_2 &\leftarrow \mathsf{compress}(z_1 \parallel y_2) \\
\vdots & \quad \vdots \quad \vdots \\
z_r &\leftarrow \mathsf{compress}(z_{r-1} \parallel y_r).
\end{aligned}$$

optional output transformation

Let $g : \{0,1\}^m \to \{0,1\}^\ell$ be a public function. Define $h(x) = g(z_r)$.

The iterated hash function constructed above is $h : \bigcup_{i=m+t+1}^{\infty} \{0,1\}^i \to \{0,1\}^\ell$.

Figure 1.4 depicts the processing step in the construction of an iterated hash function.

A commonly used preprocessing step is to construct the string y in the following way:

$$y = x \parallel \mathsf{pad}(x),$$

where $\mathsf{pad}(x)$ is constructed from x using a *padding function*. A padding function typically incorporates the value of $|x|$, and pads the result with additional bits

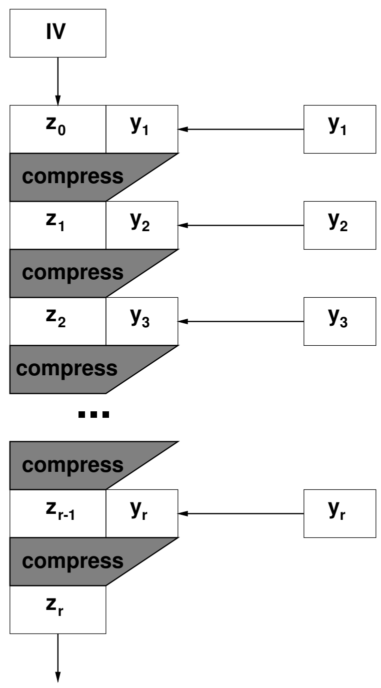

FIGURE 1.4
An iterated hash function

(zeros, for example) so that the resulting string y has a length that is a multiple of t.

The preprocessing step must ensure that the mapping $x \mapsto y$ is an injection. (If the mapping $x \mapsto y$ is not one-to-one, then it may be possible to find $x \neq x'$ so that $y = y'$. Then $h(x) = h(x')$, and h would not be collision-resistant.) Note also that $|y| = rt \geq |x|$ because of the required injective property.

1.4.2 The Secure Hash Algorithm

In this section, we describe *SHA-1* (the *Secure Hash Algorithm*), which is an iterated hash function with a 160-bit message digest. *SHA-1* is built from word-oriented operations on bitstrings, where a word consists of 32 bits (or eight hexadecimal digits). The operations used in *SHA-1* are as follows:

$X \wedge Y$	bitwise "and" of X and Y
$X \vee Y$	bitwise "or" of X and Y
$X \oplus Y$	bitwise "xor" of X and Y
$\neg X$	bitwise complement of X
$X + Y$	integer addition modulo 2^{32}
$\mathsf{ROTL}^s(X)$	circular left shift of X by s positions ($0 \leq s \leq 31$)

We first describe the padding scheme used in *SHA-1*; it is presented as Algorithm 1.4. Note that *SHA-1* requires that $|x| \leq 2^{64} - 1$. Therefore the binary representation of $|x|$, which is denoted by ℓ in Algorithm 1.4, has length at most 64 bits. If $|\ell| < 64$, then it is padded on the left with zeroes so that its length is exactly 64 bits.

Algorithm 1.4: SHA-1-PAD(x)

comment: $|x| \leq 2^{64} - 1$

$d \leftarrow (447 - |x|) \bmod 512$
$\ell \leftarrow$ the binary representation of $|x|$, where $|\ell| = 64$
$y \leftarrow x \parallel 1 \parallel 0^d \parallel \ell$

In the construction of y, we append a single 1 to x, then we concatenate enough 0's so that the length becomes congruent to 448 modulo 512, and finally we concatenate 64 bits that contain the binary representation of the (original) length of x. The resulting string y has length divisible by 512. Then we write y as a concatenation of n blocks, each having 512 bits:

$$y = M_1 \parallel M_2 \parallel \cdots \parallel M_n.$$

Define the functions $\mathsf{f}_0, \ldots, \mathsf{f}_{79}$ as follows:

$$\mathsf{f}_i(B, C, D) = \begin{cases} (B \wedge C) \vee ((\neg B) \wedge D) & \text{if } 0 \leq t \leq 19 \\ B \oplus C \oplus D & \text{if } 20 \leq t \leq 39 \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & \text{if } 40 \leq t \leq 59 \\ B \oplus C \oplus D & \text{if } 60 \leq t \leq 79. \end{cases}$$

Each function f_i takes three words B, C and D as input, and produces one word as output.

Define the word constants $K_0, \ldots, K_{79}$, which are used in the computation of *SHA-1*(x), as follows:

$$K_i = \begin{cases} \texttt{5A827999} & \text{if } 0 \leq t \leq 19 \\ \texttt{6ED9EBA1} & \text{if } 20 \leq t \leq 39 \\ \texttt{8F1BBCDC} & \text{if } 40 \leq t \leq 59 \\ \texttt{CA62C1D6} & \text{if } 60 \leq t \leq 79. \end{cases}$$

Now *SHA-1* is presented as Cryptosystem 1.6.

Cryptosystem 1.6: *SHA-1*(x)

external SHA-1-PAD
global $K_0, \ldots, K_{79}$
$y \leftarrow$ SHA-1-PAD(x)
denote $y = M_1 \parallel M_2 \parallel \cdots \parallel M_n$, where each M_i is a 512-bit block
$H_0 \leftarrow$ `67452301`
$H_1 \leftarrow$ `EFCDAB89`
$H_2 \leftarrow$ `98BADCFE`
$H_3 \leftarrow$ `10325476`
$H_4 \leftarrow$ `C3D2E1F0`
for $i \leftarrow 1$ **to** n

$$
\textbf{do} \begin{cases}
\text{denote } M_i = W_0 \parallel W_1 \parallel \cdots \parallel W_{15}, \text{ where each } W_i \text{ is a word} \\
\textbf{for } t \leftarrow 16 \textbf{ to } 79 \\
\quad \textbf{do } W_t \leftarrow \mathsf{ROTL}^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \\
A \leftarrow H_0 \\
B \leftarrow H_1 \\
C \leftarrow H_2 \\
D \leftarrow H_3 \\
E \leftarrow H_4 \\
\textbf{for } t \leftarrow 0 \textbf{ to } 79 \\
\quad \textbf{do} \begin{cases}
temp \leftarrow \mathsf{ROTL}^5(A) + \mathsf{f}_t(B, C, D) + E + W_t + K_t \\
E \leftarrow D \\
D \leftarrow C \\
C \leftarrow \mathsf{ROTL}^{30}(B) \\
B \leftarrow A \\
A \leftarrow temp
\end{cases} \\
H_0 \leftarrow H_0 + A \\
H_1 \leftarrow H_1 + B \\
H_2 \leftarrow H_2 + C \\
H_3 \leftarrow H_3 + D \\
H_4 \leftarrow H_4 + E
\end{cases}
$$

return $(H_0 \parallel H_1 \parallel H_2 \parallel H_3 \parallel H_4)$

Observe that *SHA-1* closely follows the general model of an iterated hash function. The padding scheme extends the input x by at most one extra 512-bit block. The compression function maps $160 + 512$ bits to 160 bits, where the 512 bits comprise a block of the message.

1.4.3 CBC MACs

One of the most popular ways to construct a message authentication code is to use a block cipher in CBC mode with a fixed (public) initialization vector. In CBC mode, recall that each ciphertext block y_i is x-ored with the next plaintext block, x_{i+1}, before being encrypted with the secret key K. More formally, we start with an initialization vector, denoted by IV, and define $y_0 = \mathrm{IV}$. Then we construct $y_1, y_2, \ldots$ using the rule

$$y_i = e_K(y_{i-1} \oplus x_i),$$

$i \geq 1$.

Suppose that $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ is a cryptosystem, where $\mathcal{P} = \mathcal{C} = \{0,1\}^t$. Let IV be the bitstring consisting of t zeroes, and let $K \in \mathcal{K}$ be a secret key. Finally, let $x = x_1 \parallel \cdots \parallel x_n$ be a bitstring of length tn (for some positive integer n), where each x_i is a bitstring of length t. We compute CBC-MAC(x, K) as follows..

Cryptosystem 1.7: *CBC-MAC*(x, K)

denote $x = x_1 \parallel \cdots \parallel x_n$
$\mathrm{IV} \leftarrow 00\cdots0$
$y_0 \leftarrow \mathrm{IV}$
for $i \leftarrow 1$ **to** n
  do $y_i \leftarrow e_K(y_{i-1} \oplus x_i)$
return (y_n)