



ECE 4309

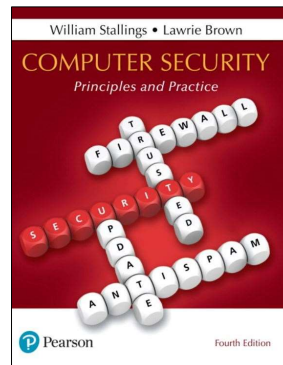
Basics of cryptography – part 2 – MAC and hash functions

Dr. Valerio Formicola



Computer Security: Principles and Practice

Fourth Edition



Chapter 2 and 21

Cryptographic Tools



Copyright © 2018, 2015, 2012 Pearson Education, Inc. All Rights Reserved

If this PowerPoint presentation contains mathematical equations, you may need to check that your computer has the following installed:


- 1) MathType Plugin
- 2) Math Player (free versions available)
- 3) NVDA Reader (free versions available)

An important element in many computer security services and applications is the use of cryptographic algorithms. This chapter provides an overview of the various types of algorithms, together with a discussion of their applicability. For each type of algorithm, we will introduce the most important standardized algorithms in common use. For the technical details of the algorithms themselves, see Part Four.


We begin with symmetric encryption, which is used in the widest variety of contexts, primarily to provide confidentiality. Next, we examine secure hash functions and discuss their use in message authentication. The next section examines public-key encryption, also known as asymmetric encryption. We then discuss the two most important applications of public-key encryption, namely digital signatures and key management. In the

case of digital signatures, asymmetric encryption and secure hash functions are combined to produce an extremely useful tool.

Finally, in this chapter, we provide an example of an application area for cryptographic algorithms by looking at the encryption of stored data.



Message authentication and hash functions



Message encryption alone cannot provide authentication

- Merely encrypting a message content, does not guarantee that the message is generated by an authentic source.
 - It's only for **confidentiality**
- A malicious actor might intercept the sequence of blocks in ECB encrypted by legitimate user and retransmit the blocks in a different order, hence changing the meaning of the message.
 - (aka, *message reordering attack*)
 - In general, we need more protection against *active attacks*

Message Authentication

- It's a mechanism to guarantee a message is integer (i.e., not altered) and authentic:
 - Authenticity: data comes from authentic source
 - Integrity: we guarantee that contents of message/data have not been altered
- As part of the Integrity, we also add information that indicates a message is:
 - Timely and in correct sequence
- How do we transmit this information? In a *Message Tag* preceding or following the original message

Original message

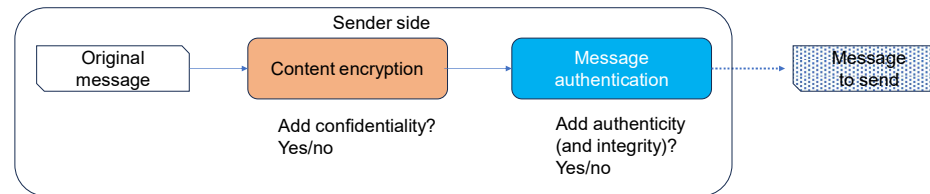
Tag

Encryption protects against passive attack (eavesdropping). A different requirement is to protect against active attack (falsification of data and transactions). Protection against such attacks is known as message or data authentication.

A message, file, document, or other collection of data is said to be authentic when it is genuine and came from its alleged source. Message or data authentication is a procedure that allows communicating parties to verify that received or stored messages are authentic. The two important aspects are to verify that the contents of the message have not been altered and that the source is authentic. We may also wish to verify a message's timeliness (it has not been artificially delayed and replayed) and sequence relative to other messages flowing between two parties. All of these concerns come under the category of data integrity as described in Chapter 1.

Observation: Combining Authentication Vs Confidentiality as a functions

- Content encryption and Message authentication/integrity can work as separate functions
- Multiple combinations are possible and for each, cryptography can be symmetric or asymmetric:
 1. Content Encryption (No) – Authentication (No)
 2. Content Encryption (Yes) – Authentication (No)
 3. Content Encryption (No) – Authentication (Yes)
 4. Content Encryption (Yes) – Authentication (Yes)



Case 1 (not used): Message authentication as effect of content symmetric encryption

- Sender and receiver are the only ones having the key and they know each other.
 - Apparently, we might achieve authenticity this way
 - However, ...
- Pure encryption of messages does not guarantee:
 - Protection from **reordering attacks**: the attacker simply stays in the middle of a communication and changes the sequence of data messages (1-2-3 -> 3-1-2)
 - Protection from **replay/delay attacks**: the attacker replays packets even if encrypted, to generate the same data at later time
- In practice: symmetric encryption alone is not a suitable tool for data authentication



7

It would seem possible to perform authentication simply by the use of symmetric encryption. If we assume that only the sender and receiver share a key (which is as it should be), then only the genuine sender would be able to encrypt a message successfully for the other participant, provided the receiver can recognize a valid message. Furthermore, if the message includes an error-detection code and a sequence number, the receiver is assured that no alterations have been made and that sequencing is proper. If the message also includes a timestamp, the receiver is assured that the message has not been delayed beyond that normally expected for network transit.

In fact, symmetric encryption alone is not a suitable tool for data authentication. To give one simple example, in the ECB mode of encryption, if an attacker reorders the blocks of ciphertext, then each block will still decrypt successfully. However, the reordering may alter the meaning of the overall data sequence. Although sequence numbers may be used at some level (e.g., each IP packet), it is typically not the case that a separate sequence number will be associated with each *b-bit block of plaintext*. Thus, *block reordering is a threat*.

Case 2: Message authentication without content encryption (i.e., no confidentiality)

- We guarantee the message is sent from authentic source and it's not altered, but we don't encrypt the content of the message
 - Only authenticity and integrity, but not confidentiality
- Situations in which message authentication without confidentiality may be preferable include:
 - There are several applications in which the same message is broadcast to several destinations, like alarms or public messages from an authoritative source (e.g., police department)
 - An exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages
 - Only a few messages are encrypted in the content, most likely the ones that contain more critical information
 - Authentication of a computer program in plaintext is an attractive service

Because the approaches discussed in this section do not encrypt the message, message confidentiality is not provided. As was mentioned, message encryption by itself does not provide a secure form of authentication. However, it is possible to combine authentication and confidentiality in a single algorithm by encrypting a message plus its authentication tag. Typically, however, message authentication is provided as a separate function from message encryption. [DAVI89] suggests three situations in which message authentication without confidentiality is preferable:

1. There are a number of applications in which the same message is broadcast to a number of destinations. Two examples are notification to users that the network is now unavailable, and an alarm signal in a control center. It is cheaper and more reliable to have only one destination responsible for monitoring authenticity. Thus, the message must be broadcast in plaintext with an associated message authentication tag. The responsible system performs authentication. If a violation occurs, the other destination systems are alerted by a general alarm.
2. Another possible scenario is an exchange in which one side has a heavy load and cannot afford the time to

decrypt all incoming messages. Authentication is carried out on a selective basis, with messages being chosen at random for checking.

3. Authentication of a computer program in plaintext is an attractive service. The computer program can be executed without having to decrypt it every time, which would be wasteful of processor resources. However, if a message authentication tag were attached to the program, it could be checked whenever assurance is required of the integrity of the program.

Thus, there is a place for both authentication and encryption in meeting security requirements.

Case 2: approaches

1. **Approach 1:** Use of plain MAC (Message Authentication Code):
 - Compute a Tag that depends on the content of original message, plus a sequence number (*anti-reordering attack protection*)
 - Encrypt the Tag using a symmetric key (shared key)
 - Append to the original message (in clear)
 - Receiver will use secret key to decrypt the encrypted MAC tag and compare to the actual MAC of the message received
2. **Approach 2:** One-way hash function:
 - As before, compute a Tag but this time use a *one-way hash function* of the original message + original length information + padding
 - Encrypt the hash:
 - (2A) Using a symmetric key (shared key)
 - (2B) Using an asymmetric key (aka, public key)
 - Append to the original message (in clear)
 - Receiver will use the key (public or shared) to decrypt the encrypted Hash tag and compare to the actual Hash of message received
3. **Approach 3:** Keyed hash MAC:
 - You evaluate the Hash of a message concatenated to a secret key (shared key): $H(K || M || K)$
 - Append the keyed Hash tag to the original message
 - Receiver will concatenate the received message with secret key, recalculate the keyed hash and compare with received Hash tag

Approach 1: Message Authentication Using a Message Authentication Code (MAC)

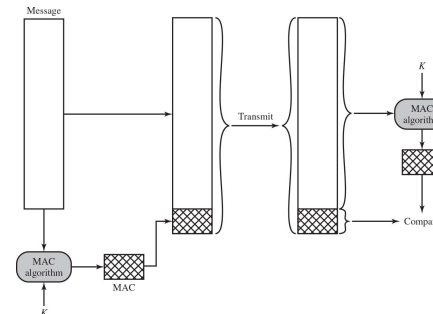
The message authentication code (MAC) is a complex function using as input 1) the message and 2) the secret key K_{AB} exchanged between sender and receiver: $MAC_M = F(K_{AB}, M)$.

MAC guarantees integrity AND authentication. Use of a key as an input to the function F guarantees **authentication**.
Use of a message content M guarantees **integrity**.

Symmetric key is still used but only for the MAC tag

- 1 - secret key is used to create a MAC tag (authenticity)
- 2 - no alteration of message is possible (integrity because nobody has secret key but legitimate ends of the communication)
- 3 - **if the message contains a sequence number**, any alteration of the sequence number will be detected as a integrity violation; moreover, sequence number will protect from *replay attacks* (next slide)

Examples of common algorithms for MAC (key + message, no content encryption):
CMAC, CBC-MAC.



One authentication technique involves the use of a secret key to generate a small block of data, known as a message authentication code, that is appended to the message. This technique assumes that two communicating parties, say A and B, share a common secret key K_{AB} . When A has a message to send to B, it calculates the message authentication code as a complex function of the message and the key: $MAC_M = F(K_{AB}, M)$. The message plus code are transmitted to the intended recipient. The recipient performs the same calculation on the received message, using the same secret key, to generate a new message authentication code. The received code is compared to the calculated code (Figure 2.3). If we assume that only the receiver and the sender know the identity of the secret key, and if the received code matches the calculated code, then

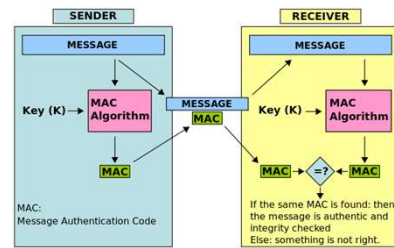
1. The receiver is assured that the message has not been altered. If an attacker alters the message but does not alter the code, then the receiver's calculation of the code will differ from the received code. Because the attacker is assumed not to know the secret key, the attacker cannot alter the code to correspond to the alterations in the message.
2. The receiver is assured that the message is from the alleged sender. Because no one else knows the secret

key, no one else could prepare a message with a proper code.

3. If the message includes a sequence number (such as is used with X.25, HDLC, and TCP), then the receiver can be assured of the proper sequence, because an attacker cannot successfully alter the sequence number.

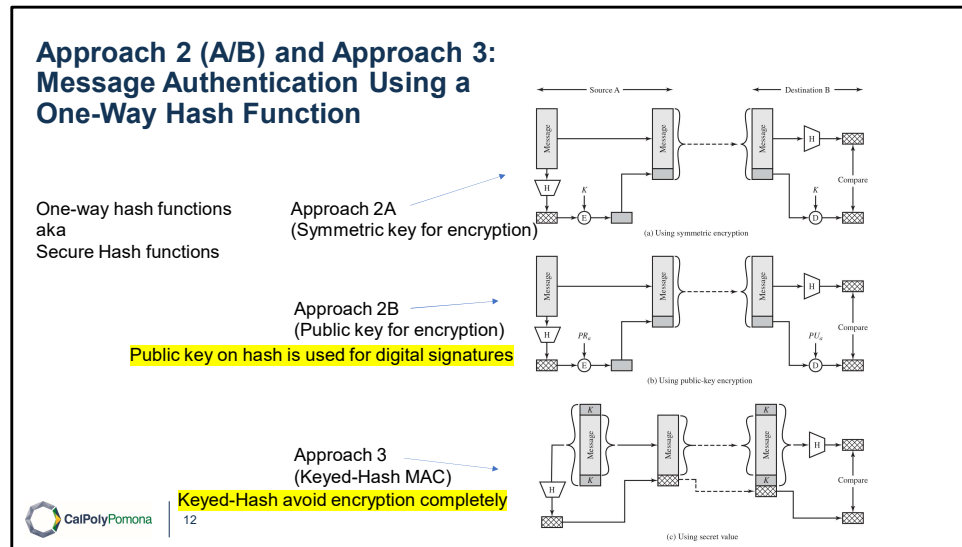
The steps are as follows. The first block labeled message is shared to M A C algorithm and a key K is shared to M A C algorithm. Step 2. The first block transmits the message to the second block with M A C and M A C is shared to the M A C in the second block. Step 3. The second block transmits the message to the third block with M A C. Step 4. The message in the third block is transmitted to M A C algorithm. A key is shared to the algorithm. The output from the algorithm is compared by M A C from the third block.

Approach 1 (another example)



In this example, the sender of a message runs it through a MAC algorithm to produce a MAC data tag. The message and the MAC tag are then sent to the receiver. The receiver in turn runs the message portion of the transmission through the same MAC algorithm using the same key, producing a second MAC data tag. The receiver then compares the first MAC tag received in the transmission to the second generated MAC tag. If they are identical, the receiver can safely assume that the message was not altered or tampered with during transmission (data integrity).

However, to allow the receiver to be able to detect replay attacks, the message itself must contain data that assures that this same message can only be sent once (e.g. time stamp, sequence number or use of a one-time MAC). Otherwise an attacker could – without even understanding its content – record this message and play it back at a later time, producing the same result as the original sender.



Unlike the MAC, a hash function does not also take a secret key as input. To authenticate a message, the message digest is sent with the message in such a way that the message digest is authentic. Figure 2.5 illustrates three ways in which the message can be authenticated using a hash code. The message digest can be encrypted using symmetric encryption (part a); if it is assumed that only the sender and receiver share the encryption key, then authenticity is assured. The message digest can also be encrypted using public-key encryption (part b); this is explained in Section 2.3. The public-key approach has two advantages: It provides a digital signature as well as message authentication; and it does not require the distribution of keys to communicating parties.

These two approaches have an advantage over approaches that encrypt the entire message in that less computation is required. But an even more common approach is the use of a technique that avoids encryption altogether. Several reasons for this interest are pointed out in [TSUD92]:

- Encryption software is quite slow. Even though the amount of data to be encrypted per message is small, there may be a steady stream of messages into and out of a system.

- Encryption hardware costs are non-negligible. Low-cost chip implementations of DES are available, but the cost adds up if all nodes in a network must have this capability.
- Encryption hardware is optimized toward large data sizes. For small blocks of data, a high proportion of the time is spent in initialization/invocation overhead.
- An encryption algorithm may be protected by a patent.

Figure 2.5c shows a technique that uses a hash function but no encryption for message authentication. This technique, known as a keyed hash MAC, assumes that two communicating parties, say A and B, share a common secret key K. This secret key is incorporated into the process of generating a hash code. In the approach illustrated in Figure 2.5c, when A has a message to send to B, it calculates the hash function over the concatenation of the secret key and the message:

$MD_M = H(K \parallel M \parallel K)$. It then sends $[M \parallel MD_M]$ to B. Because B possesses K, it can recompute $H(K \parallel M \parallel K)$ and verify MD_M . Because the secret key itself is not sent, it should not be possible for an attacker to modify an intercepted message. As long as the secret key remains secret, it should not be possible for an attacker to generate a false message.

Note that the secret key is used as both a prefix and a suffix to the message. If the secret key is used as either only a prefix or only a suffix, the scheme is less secure. This topic is discussed in Chapter 21. Chapter 21 also describes a scheme known as HMAC, which is somewhat more complex than the approach of Figure 2.5c and which has become the standard approach for a keyed hash MAC.

Diagram a, using symmetric encryption. The first block labeled message is shared to Hash function H algorithm M A C algorithm. A key K, is shared to the encryption algorithm. The first block transmits the message to the second block and the encrypted message is shared to the encrypted message in the second block. The process is represented as source A. The second block with encrypted message transmits the message to the third block with encrypted message. The message in the third block is transmitted to hash function and M A C. The encrypted message is fed to the decryption algorithm. A key is shared to the decryption algorithm. The output from the decryption algorithm is compared by M A C from the third block. This process is labeled, Destination B.

Diagram b, using public key encryption. The first block labeled message is shared to Hash function H algorithm M A C algorithm. A public key P_R sub a, is shared to the encryption algorithm. The first block transmits the message to the second block and the encrypted message is shared to the encrypted message in the second block. The process is represented as source A. The second block with encrypted message transmits the message to the third block with encrypted message. The message in the third block is transmitted to hash function and M A C. The encrypted message is fed to the decryption algorithm. A public key P_U sub a is shared to the decryption algorithm. The output from the decryption algorithm is compared by M A C from the third block. This process is labeled, Destination B.

Diagram c, using secret value. The first block labeled, message with keys at the top and bottom is transmitted to hash function, H. The hash function is fed to M A C and the message from the first block is transmitted to second block with M A C. The M A C is fed to the M A C in the second block. The message from the second block is transmitted to message in the third block with keys at the top and bottom. The M A C from the second block is fed to M A C in the third block. The message is transmitted to the hash function H. The output from H is fed to M A C. The M A C from the third block and the M A C from the hash function are compared.

Secure Hash Functions: To Be Useful for Message Authentication, a Hash Function H Must Have the Following Properties:

1. Can be applied to a block of data of any size
2. Produces a fixed-length output
3. $H(x)$ is relatively easy to compute for any given x
4. **One-way or pre-image resistant**
 - Computationally infeasible to find x such that $H(x) = h$, i.e., nobody should be able to find the inverse function of H because, otherwise $H^{-1}(y) = x$; x might be $K || M || K$ and the key K would be revealed if you also have M
5. **Second pre-image resistant or weak collision resistant**
 - Computationally infeasible to find $y \neq x$ such that $H(y) = H(x)$
 - Note: here you get the hash of a message, but you cannot create another one that has the same hash of the first (*anti-forgery protection*)
6. **Collision resistant or strong collision resistance**
 - Computationally infeasible to find any pair (x, y) such that $H(y) = H(x)$
 - Note: Here you don't have any starting message; the property states you should not be able to create two messages with the same hash



13

The purpose of a hash function is to produce a “fingerprint” of a file, message, or other block of data. To be useful for message authentication, a hash function H must have the following properties:

1. H can be applied to a block of data of any size.
2. H produces a fixed-length output.
3. $H(x)$ is relatively easy to compute for any given x , making both hardware and software implementations practical.
4. For any given code h , it is computationally infeasible to find x such that $H(x) = h$. A hash function with this property is referred to as one-way or preimage resistant.
5. For any given block x , it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$. A hash function with this property is referred to as second preimage resistant. This is sometimes referred to as weak collision resistant.

6. It is computationally infeasible to find any pair (x, y) such that $H(x) = H(y)$. A hash function with this property is referred to as collision resistant. This is sometimes referred to as strong collision resistant.

The first three properties are requirements for the practical application of a hash function to message authentication.

The fourth property is the one-way property: It is easy to generate a code given a message, but virtually impossible to generate a message given a code. This property is important if the authentication technique involves the use of a secret value (Figure 2.5c). The secret value itself is not sent; however, if the hash function is not one way, an attacker can easily discover the secret value: If the attacker can observe or intercept a transmission, the attacker obtains the message M and the hash code $MD_M = H(K \parallel M \parallel K)$. The attacker then inverts the hash function to obtain $K \parallel M \parallel K = H^{-1}(MD_M)$. Because the attacker now has both M and $K \parallel M \parallel K$, it is a trivial matter to recover K .

The fifth property guarantees that it is impossible to find an alternative message with the same hash value as a given message. This prevents forgery when an encrypted hash code is used (Figures 2.5a and b). If this property were not true, an attacker would be capable of the following sequence: First, observe or intercept a message plus its encrypted hash code; second, generate an unencrypted hash code from the message; third, generate an alternate message with the same hash code.

A hash function that satisfies the first five properties in the preceding list is referred to as a weak hash function. If the sixth property is also satisfied, then it is referred to as a strong hash function. A strong hash function protects against an attack in which one party generates a message for another party to sign. For example, suppose Bob gets to write an IOU message, send it to Alice, and she signs it. Bob finds two messages with the same hash, one of which requires Alice to pay a small amount and one that requires a large payment. Alice signs the first message and Bob is then able to claim that the second message is authentic.

Security of Hash Functions

- There are two approaches to attacking a secure hash function:
 - Cryptanalysis
 - Exploit logical weaknesses in the algorithm
 - Brute-force attack
 - Strength of hash function depends solely on the length of the hash code produced by the algorithm
- MD5 which generates 128 bit hash, has been found to be breakable and it's not secure anymore
- SHA most widely used hash algorithm family. Currently SHA-2 (256, 384, 512 bit hashes) are the most used and SHA-3 will be in the future
- Additional secure hash function applications:
 - Passwords
 - Hash of a password is stored by an operating system
 - Intrusion detection
 - Store $H(\text{File})$ for each file on a system and secure the hash values

As with symmetric encryption, there are two approaches to attacking a secure hash function: cryptanalysis and brute-force attack. As with symmetric encryption algorithms, cryptanalysis of a hash function involves exploiting logical weaknesses in the algorithm.

The strength of a hash function against brute-force attacks depends solely on the length of the hash code produced by the algorithm. For a hash code of length n , the level of effort required is proportional to the following:

Preimage resistant 2^n

Second preimage resistant 2^n

Collision resistant $2^{n/2}$

If collision resistance is required (and this is desirable for a general-purpose secure hash code), then the value $2^{n/2}$ determines the strength of the hash code against brute-force attacks. Van Oorschot and Wiener [VANO94] presented a design for a \$10 million collision search machine for MD5, which has a 128-bit hash length, that could find a collision in 24 days. Thus a 128-bit code may be viewed as inadequate. The next step up, if a hash

code is treated as a sequence of 32 bits, is a 160-bit hash length. With a hash length of 160 bits, the same search machine would require over four thousand years to find a collision. With today's technology, the time would be much shorter, so that 160 bits now appears suspect.

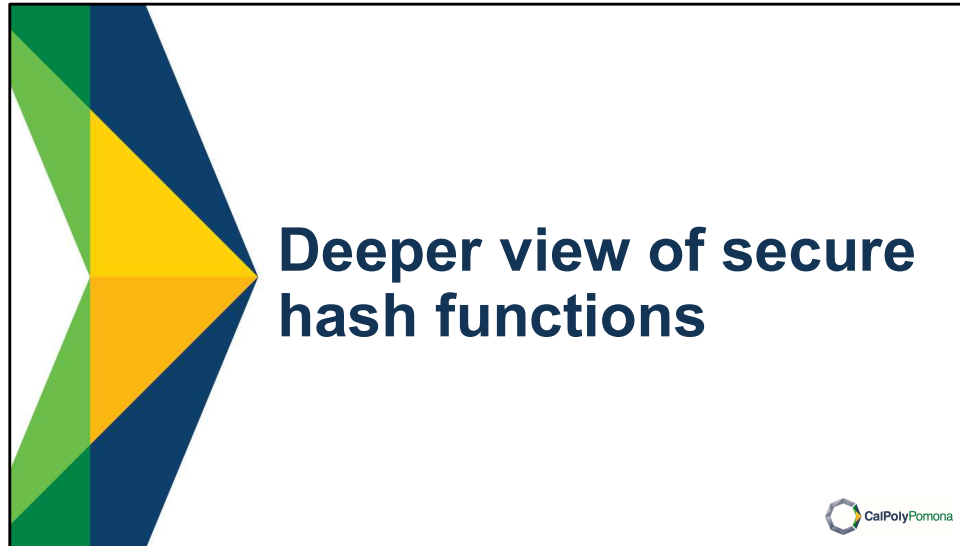
In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). SHA was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993. When weaknesses were discovered in SHA, a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as SHA-1. SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on the other SHA versions by 2010. As discussed in Chapter 21, researchers have demonstrated that SHA-1 is far weaker than its 160-bit hash length suggests, necessitating the move to the newer versions of SHA.

We have discussed the use of hash functions for message authentication and for the creation of digital signatures (the latter is discussed in more detail later in this chapter). Here are two other examples of secure hash function applications:

- **Passwords:** Chapter 3 explains a scheme in which a hash of a password is stored by an operating system rather than the password itself. Thus, the actual password is not retrievable by a hacker who gains access to the password file. In simple terms, when a user enters a password, the hash of that password is compared to the stored hash value for verification. This application requires preimage resistance and perhaps second preimage resistance.
- **Intrusion detection:** Store $H(F)$ for each file on a system and secure the hash values (e.g., on a CD-R that is kept secure). One can later determine if a file has been modified by recomputing $H(F)$. An intruder would need to change F without changing $H(F)$. This application requires weak second preimage resistance

In summary

- MAC function:
 - Calculates an output Tag from the message in combination with a key
 - Hence provides integrity and authenticity
- One-way hash function or Secure-hash function:
 - message Tag is calculated only using the message content, not a key
 - Hence, it provides integrity but not authenticity
 - You need to combine with encryption for authenticity (symmetric or public encryption) to obtain authenticity
 - It has applicable to any blocks size
 - In contrast to stream ciphers that cannot be applied to blocks shorter than the key size
 - Generates the same output, whatever the input size
 - It's not computationally stressful
 - It can have several levels of security based on the resistance level:
 - preimage resistance: for essentially all pre-specified outputs, it is computationally infeasible to find any input that hashes to that output; i.e., given y , it is difficult to find an x such that $h(x) = y$.
 - second-preimage resistance: for a specified input, it is computationally infeasible to find another input which produces the same output; i.e., given x , it is difficult to find a second input $x' \neq x$ such that $h(x) = h(x')$.
 - collision resistance (strong resistance): it is computationally infeasible to find any two distinct inputs x, x' that hash to the same output; i.e., such that $h(x) = h(x')$



rounds; this module is labeled F in Figure 21.2.

- Step 5: Output. After all N 1024-bit blocks have been processed, the output from the N th stage is the 512-bit message digest.

A message of L bits is padded with bits, numbering in range of 1 to 1024, so that its length is congruent to N times 1024 bits. The padding consists of a single 1 bit followed by a number of 0 bits. A 128 bit block is appended to the message. The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length. The expanded message is represented as the sequence of 1024-bit blocks, in sequence from $M_{sub\ 1}$ to $M_{sub\ N}$. An initialization vector H_0 with 512 bits buffers are added to the message. Each round takes as input the 512 bit buffer value and updates the contents of the buffer. The hash code obtained is H_N .

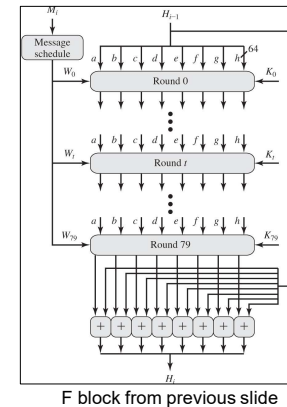
Figure 21.3 SHA-512 Processing of a Single 1024-Bit Block

W_t = 64 bit data derived from the block being processed

K_t = additive constant for each round of the 80

Initialization buffer at the beginning is always the same
 $H_0 = a|b|c|d|e|f|g|h$

```
h[0..7] := 0x6a09e667f3bcc908, 0x2b67ee8584caa73b, 0x3c6ef372fe94f82b,  
0xa54ff53a5f1d36f1, 0x510e527fade682d1, 0x9b05688c2b3e6c1f,  
0x1f83d9abfb41bd6b, 0x5be0cd19137e2179
```



The logic is illustrated in Figure 21.3.

Each round takes as input the 512-bit buffer value $abcdefgh$, and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value, H_{i-1} . Each round t makes use of a 64-bit value W_t , derived from the current 1024-bit block being processed (M_i). Each round also makes use of an additive constant K_t , where $0 \leq t \leq 79$ indicates one of the 80 rounds. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. The constants provide a “randomized” set of 64-bit patterns, which should eliminate any regularities in the input data. The operations performed during a round consist of circular shifts, and primitive Boolean functions based on AND, OR, NOT, and XOR.

The output of the eightie-th round is added to the input to the first round (H_{i-1}) to produce H_i . The addition is done independently for each of the eight words in the buffer with each of the corresponding words in H_{i-1} , using addition modulo 264.

The SHA-512 algorithm has the property that every bit of the hash code is a function of every bit of the input. The complex repetition of the basic function F produces results that are well mixed; that is, it is unlikely that two messages chosen at random, even if they exhibit similar regularities, will have the same hash code. Unless there is some hidden weakness in SHA-512, which has not so far been published, the difficulty of coming up with two messages having the same message digest is on the order of 2^{256} operations, while the difficulty of finding a message with a given digest is on the order of 2^{512} operations.

The message schedule $M_{sub\ i}$ is processed for 80 rounds to produce $H_{sub\ i}$. Each round takes as input the 512 bit buffer value, $a\ b\ c\ d\ e\ f\ g\ h$, and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value, $H_{sub\ start\ expression\ i\ minus\ 1\ end\ expression}$. Each round t makes use of a 64 bit value $W_{sub\ t}$, derived from the current 1024 bit block being processed $M_{sub\ i}$. Each round also makes use of an additive constant $K_{sub\ t}$, where 0 to 79 indicates one of the 80 rounds. The constants provide a randomized set of 64 bit patterns. The output of the eightieth round is added to the input to the first round, $H_{sub\ start\ expression\ i\ minus\ 1\ end\ expression}$ to produce $H_{sub\ i}$. The addition is done independently for each of the eight words in the buffer with each of the corresponding words in $H_{sub\ start\ expression\ i\ minus\ 1\ end\ expression}$, using addition modulo 2 to the 64 power.

SHA-3

- SHA-2 shares same structure and mathematical operations as its predecessors and causes concern
 - SHA-512 is pretty solid so far
- Due to time required to replace SHA-2 should it become vulnerable, NIST announced in 2007 a competition to produce SHA-3
- Requirements:
 - Must support hash value lengths of 224, 256, 384, and 512 bits
 - Algorithm must process small blocks at a time instead of requiring the entire message to be buffered in memory before processing it



19

SHA-2, particularly the 512-bit version, would appear to provide unassailable security. However, SHA-2 shares the same structure and mathematical operations as its predecessors, and this is a cause for concern. Because it will take years to find

a suitable replacement for SHA-2, should it become vulnerable, NIST announced in 2007 a competition to produce the next generation NIST hash function, to be called SHA-3.

The basic requirements that needed to be satisfied by any candidate for SHA-3 are the following:

1. It must be possible to replace SHA-2 with SHA-3 in any application by a simple drop-in substitution. Therefore, SHA-3 must support hash value lengths of 224, 256, 384, and 512 bits.
2. SHA-3 must preserve the online nature of SHA-2. That is, the algorithm must process comparatively small blocks (512 or 1024 bits) at a time instead of requiring that the entire message be buffered in memory before processing it.

After an extensive consultation and vetting process, NIST selected a winning submission and formally published SHA-3

as FIPS 202 (*SHA-3 Standard: Permutation- Based Hash and Extendable-Output Functions*, August 2015).

The structure and functions used for SHA-3 are substantially different from those shared by SHA-2 and SHA-1. Thus, if weaknesses are discovered in either SHA-2 or SHA-3, users have the option to switch to the other standard. SHA-2 has held up well and NIST considers it secure for general use. So for now, SHA-3 is a complement to SHA-2 rather than a replacement. The relatively compact nature of SHA-3 may make it useful for so-called “embedded” or smart devices that connect to electronic networks but are not themselves full-fledged computers. Examples include sensors in a building-wide security system and home appliances that can be controlled remotely. A detailed presentation of SHA-3 is provided in Appendix K.

HMAC

- Interest in developing a MAC derived from a cryptographic hash code
 - Cryptographic hash functions generally execute faster
 - Library code is widely available
 - SHA-1 was not designed for use as a MAC because it does not rely on a secret key
- Issued as RFC2104
- Has been chosen as the mandatory-to-implement MAC for IP security
 - Used in other Internet protocols such as Transport Layer Security (TLS) and Secure Electronic Transaction (SET)

In this section, we look at the hash-code approach to message authentication. Appendix E looks at message authentication based on block ciphers. In recent years, there has been increased interest in developing a MAC derived from a cryptographic hash code, such as SHA-1. The motivations for this interest are as follows:

- Cryptographic hash functions generally execute faster in software than conventional encryption algorithms such as DES.
- Library code for cryptographic hash functions is widely available.

A hash function such as SHA-1 was not designed for use as a MAC and cannot be used directly for that purpose because it does not rely on a secret key. There have been a number of proposals for the incorporation of a secret key into an existing hash algorithm. The approach that has received the most support is HMAC [BELL96]. HMAC has been issued as RFC 2104 (HMAC: Keyed-Hashing for Message Authentication, 1997), has been chosen as the mandatory-to-implement MAC for IP Security, and is used in other Internet protocols, such as Transport Layer Security (TLS, soon to replace Secure Sockets Layer) and Secure Electronic Transaction (SET).

HMAC Design Objectives

- To use, without modifications, available hash functions
- To allow for easy replaceability of the embedded hash function in case faster or more secure hash functions are found or required
- To preserve the original performance of the hash function without incurring a significant degradation
- To use and handle keys in a simple way
- To have a well-understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions on the embedded hash function

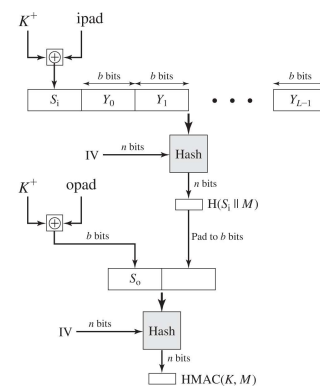
RFC 2104 lists the following design objectives for HMAC:

- To use, without modifications, available hash functions—in particular, hash functions that perform well in software, and for which code is freely and widely available
- To allow for easy replaceability of the embedded hash function in case faster or more secure hash functions are found or required
- To preserve the original performance of the hash function without incurring a significant degradation
- To use and handle keys in a simple way
- To have a well-understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions on the embedded hash function

The first two objectives are important to the acceptability of HMAC. HMAC treats the hash function as a “black box.” This has two benefits. First, an existing implementation of a hash function can be used as a module in implementing HMAC. In this way, the bulk of the HMAC code is prepackaged and ready to use without modification. Second, if it is ever desired to replace a given hash function in an HMAC implementation, all that is required is to remove the existing hash function module and drop in the new module. This could be done if a faster hash function were desired. More important, if the security of the embedded hash function were compromised, the security of HMAC could be retained simply by replacing the embedded hash function with a more secure one.

The last design objective in the preceding list is, in fact, the main advantage of HMAC over other proposed hash-based schemes. HMAC can be proven secure provided that the embedded hash function has some reasonable cryptographic strengths. We return to this point later in this section, but first we examine the structure of HMAC.

Figure 21.4 HMAC Structure



- H = embedded hash function (e.g., SHA)
- M = message input to HMAC (including the padding specified in the embedded hash function)
- Y_i = i -th block of M , $0 \leq i \leq (L-1)$
- L = number of blocks in M
- b = number of bits in a block (e.g., 512)
- n = length of hash code produced by embedded hash function
- K = secret key; if key length is greater than b , the key is input to the hash function to produce an n -bit key; recommended length is $\geq n$
- K^+ = K padded with zeros on the left so that the result is b bits in length
- ipad = 00110110 (36 in hexadecimal) repeated $b/8$ times
- opad = 01011100 (5C in hexadecimal) repeated $b/8$ times
- IV = initialization vector (unused)

$$\text{HMAC}(K, M) = H[(K^+ \text{ XOR opad}) \parallel H[(K^+ \text{ XOR ipad}) \parallel M]]$$

Where XOR is the XOR operation between two sequences

Figure 21.4 illustrates the overall operation of HMAC.

In words:

1. Append zeros to the left end of K to create a b -bit string K^+ (e.g., if K is of length 160 bits and $b = 512$, then K will be appended with 44 zero bytes 0x00).
2. XOR (bitwise exclusive-OR) K^+ with ipad to produce the b -bit block S_i .
3. Append M to S_i .
4. Apply H to the stream generated in step 3.
5. XOR K^+ with opad to produce the b -bit block S_o .
6. Append the hash result from step 4 to S_o .

7. Apply H to the stream generated in step 6 and output the result.

Note that the XOR with $ipad$ results in flipping one-half of the bits of K . Similarly, the XOR with $opad$ results in flipping one-half of the bits of K , but a different set of bits. In effect, by passing S_i and S_o through the hash algorithm, we have pseudorandomly generated two keys from K .

HMAC should execute in approximately the same time as the embedded hash function for long messages. HMAC adds three executions of the basic hash function (for S_i , S_o , and the block produced from the inner hash).

The process is as follows. 1. Append zeros to the left end of K to create a b bit string K^+ . 2. XOR, bitwise exclusive OR, K^+ with $ipad$ to produce the b bit block $S_{sub\ i}$. 3. Append $Y_{sub\ 0}$ and $Y_{sub\ 1}$, to $Y_{sub\ start}$ expression $L - 1$ end expression each of b bits, to $S_{sub\ i}$. 4. Apply H to the stream generated in step 3. 5. XOR K^+ with $opad$ to produce the b bit block $S_{sub\ o}$. 6. Append the hash result from step 4 to $S_{sub\ o}$. 7. Apply H to the stream generated in step 6 and output the result. The XOR with $ipad$ results in flipping one half of the bits of K . Similarly, the XOR with $opad$ results in flipping one half of the bits of K , but a different set of bits. In effect, pass $S_{sub\ i}$ and $S_{sub\ o}$ through the hash algorithm to get the pseudo randomly generated two keys from K . HMAC adds three executions of the basic hash function, for $S_{sub\ i}$, $S_{sub\ o}$, and the block produced from the inner hash.

Security of HMAC

- Security depends on the cryptographic strength of the underlying hash function
- The appeal of HMAC is that its designers have been able to prove an exact relationship between the strength of the embedded hash function and the strength of HMAC
- For a given level of effort on messages generated by a legitimate user and seen by the attacker, the probability of successful attack on HMAC is equivalent to one of the following attacks on the embedded hash function:
 - The attacker is able to compute an output of the compression function even with an IV that is random, secret, and unknown to the attacker: same difficulty of a brute-force attack to guess a secret key with $O(2^n)$
 - The attacker finds collisions in the hash function even when the IV is random and secret. $H(M) = H(M')$. This attack is also known as *birthday attack*

The security of any MAC function based on an embedded hash function depends in some way on the cryptographic strength of the underlying hash function. The appeal of HMAC is that its designers have been able to prove an exact relationship between the strength of the embedded hash function and the strength of HMAC.

The security of a MAC function is generally expressed in terms of the probability of successful forgery with a given amount of time spent by the forger and a given number of message-MAC pairs created with the same key. In essence, it is proved in [BELL96] that for a given level of effort (time, message-MAC pairs) on messages generated by a legitimate user and seen by the attacker, the probability of successful attack on HMAC is equivalent to one of the following attacks on the embedded hash function:

1. The attacker is able to compute an output of the compression function even with an IV that is random, secret, and unknown to the attacker.
2. The attacker finds collisions in the hash function even when the IV is random and secret.

In the first attack, we can view the compression function as equivalent to the hash function applied to a message consisting of a single b -bit block. For this attack, the IV of the hash function is replaced by a secret, random value of n bits. An attack on this hash function requires either a brute-force attack on the key, which is a level of effort on the order of 2^n , or a birthday attack, which is a special case of the second attack, discussed next.

In the second attack, the attacker is looking for two messages M and M' that produce the same hash: $H(M) = H(M')$. This is the birthday attack mentioned previously. We have stated that this requires a level of effort of $2^{n/2}$ for a hash length of n . On this basis, the security of the earlier MD5 hash function was called into question, because a level of effort of 2^{64} looks feasible with today's technology. Does this mean that a 128-bit hash function such as MD5 is unsuitable for HMAC? The answer is no, because of the following argument. To attack MD5, the attacker can choose any set of messages and work on these offline on a dedicated computing facility to find a collision. Because the attacker knows the hash algorithm and the default IV, the attacker can generate the hash code for each of the messages that the attacker generates. However, when attacking HMAC, the attacker cannot generate message/code pairs offline because the attacker does not know K . Therefore, the attacker must observe a sequence of messages generated by HMAC under the same key and perform the attack on these known messages. For a hash code length of 128 bits, this requires 2^{64} observed blocks (2^{72} bits) generated using the same key. On a 1-Gbps link, one would need to observe a continuous stream of messages with no change in key for about 150,000 years in order to succeed. Thus, if speed is a concern, it is acceptable to use MD5 rather than SHA as the embedded hash function for HMAC, although use of MD5 is now uncommon.