Hash Functions

- A hash function H accepts a variable-length block of data *M* as input and produces a fixed-size hash value
  - h = H(M)
  - Principal object is data integrity
- Cryptographic hash function
  - An algorithm for which it is computationally infeasible to find either:

    (a) a data object that maps to a pre-specified hash result (the one-way property)

    (b) two data objects that map to the same hash result (the collision-free property)

- A hash function H accepts a variable-length block of data M as input and produces a fixed-size hash value h = H(M). A “good” hash function has the property that the results of applying the function to a large set of inputs will produce outputs that are evenly distributed and apparently random. In general terms, the principal object of a hash function is data integrity. A change to any bit or bits in M results, with high probability, in a change to the hash code.

Contd…

- Auxiliary function in cryptography.
- The kind of hash function needed for security applications is referred to as a **cryptographic hash function.**

- A cryptographic hash function is an algorithm for which it is computationally infeasible:
- (a) a data object that maps to a pre-specified hash result (the one-way property) or
- (b) two data objects that map to the same hash result

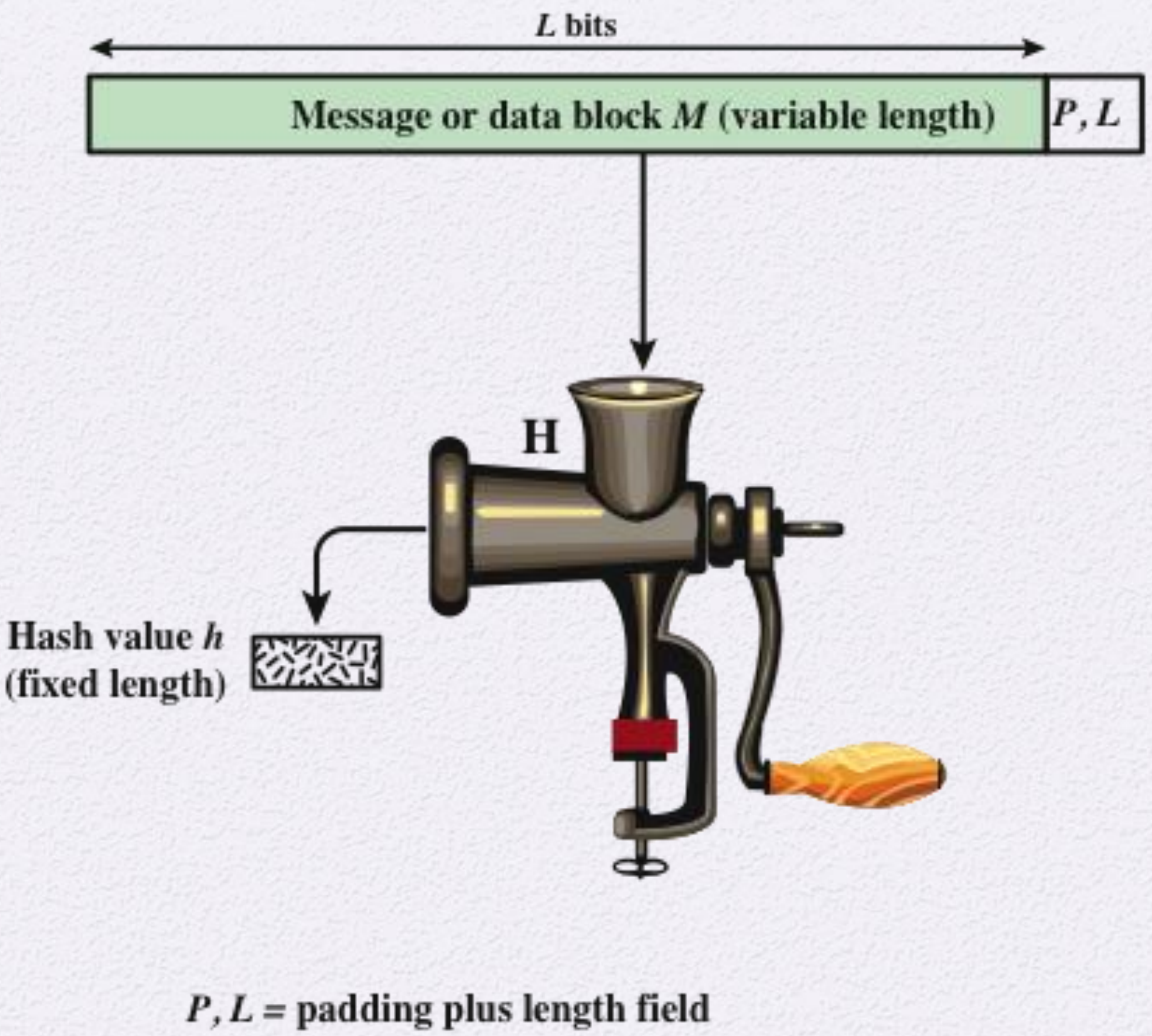
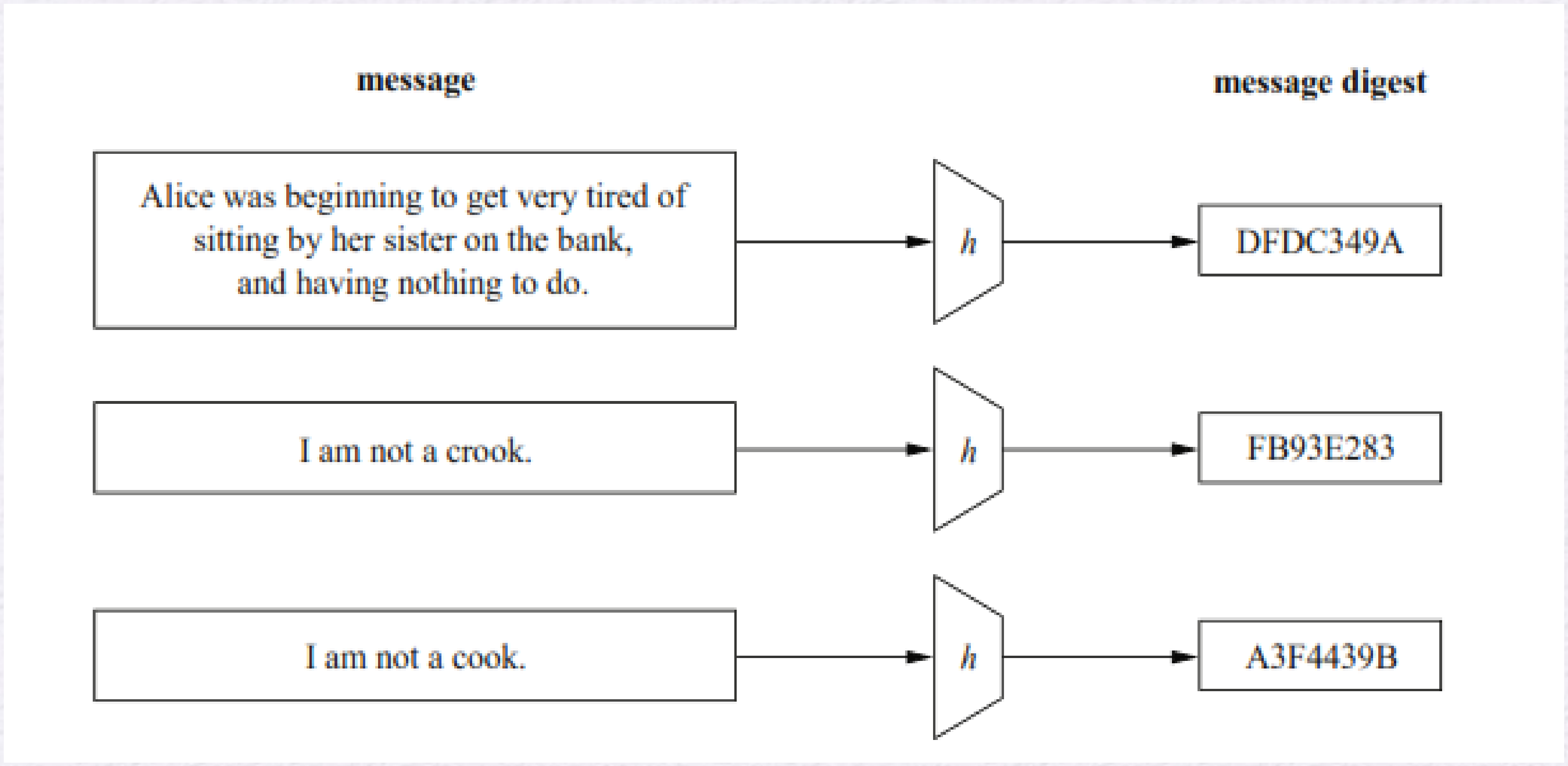

L bits
Message or data block M (variable length)
P, L
H
Hash value h
(fixed length)
P, L = padding plus length field

Figure 11.1 Cryptographic Hash Function; $h = \mathrm{H}(M)$


message
message digest
Alice was beginning to get very tired of sitting by her sister on the bank, and having nothing to do.
h
DFDC349A
I am not a crook.
h
FB93E283
I am not a cook.
h
A3F4439B

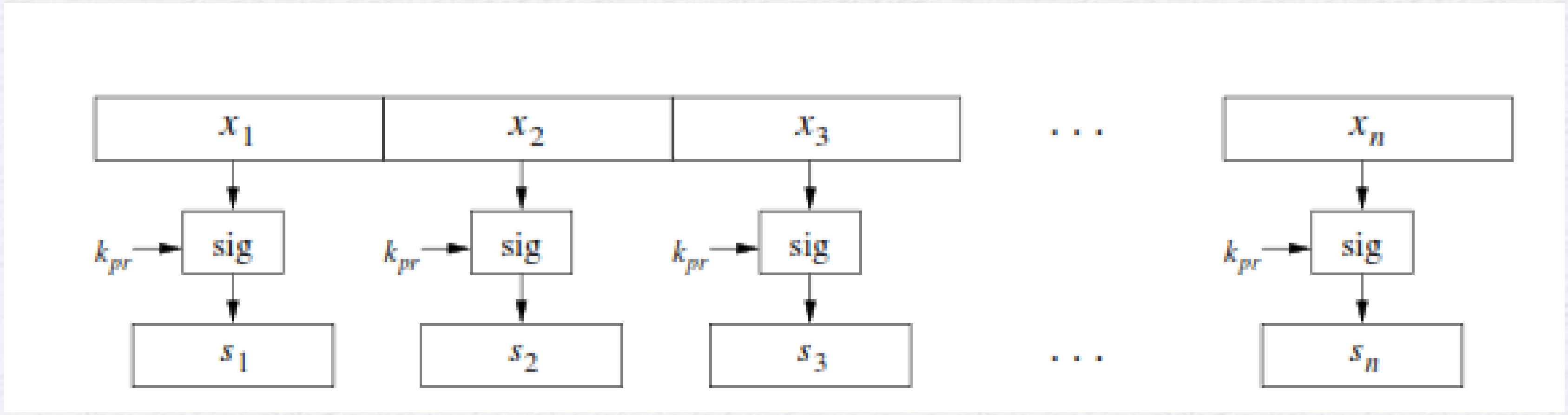
Contd…

- Hash functions are an important cryptographic primitive and are widely used in protocols. They compute a *digest of a message which is a short, fixed-length bitstring.*
- For a particular message, the message digest, or *hash value, can be seen as* the fingerprint of a message, i.e., a unique representation of a message.
- hash functions do not have a key.

Application

- Hash functions are also widely used for other cryptographic applications, e.g., for storing of password hashes or key derivation.
- Hash functions can be used for **intrusion detection and virus detection**
- **PRNG – Pseudo random number generator**

Signing Long Message



Contd…

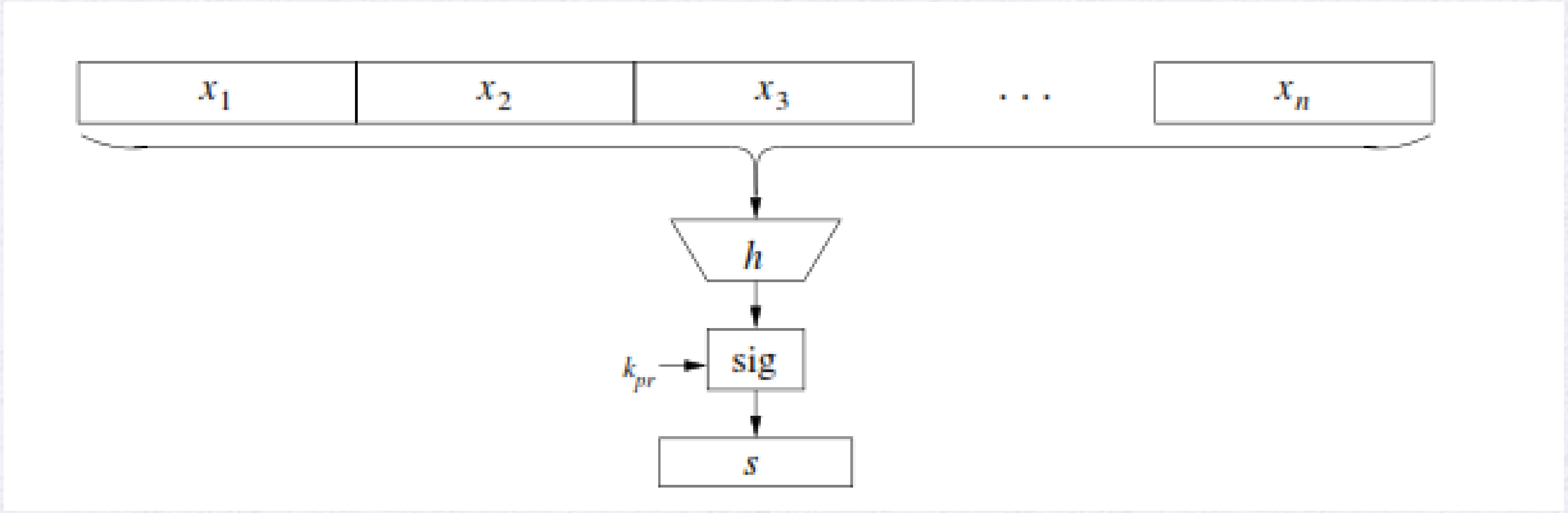
- For instance, in the case of RSA, the message cannot be larger than the modulus, which is in practice often between 1024 and 3072-bits long. Remember this translates into only 128–384 bytes; most emails are longer than that.

Problems

- **High Computational Load:**
  - **Digital signatures are based on computa**tionally intensive asymmetric operations such as modular exponentiations of large integers.
- **Message Overhead:**
  - **Obviously, this na**ive approach doubles the message overhead because not only must the message be sent but also the signature, which is of the same length in this case.

Contd…

- **Security Limitations:**
  - **This is the most serious problem if we attempt** to sign a long message by signing a sequence of message blocks *individually.*



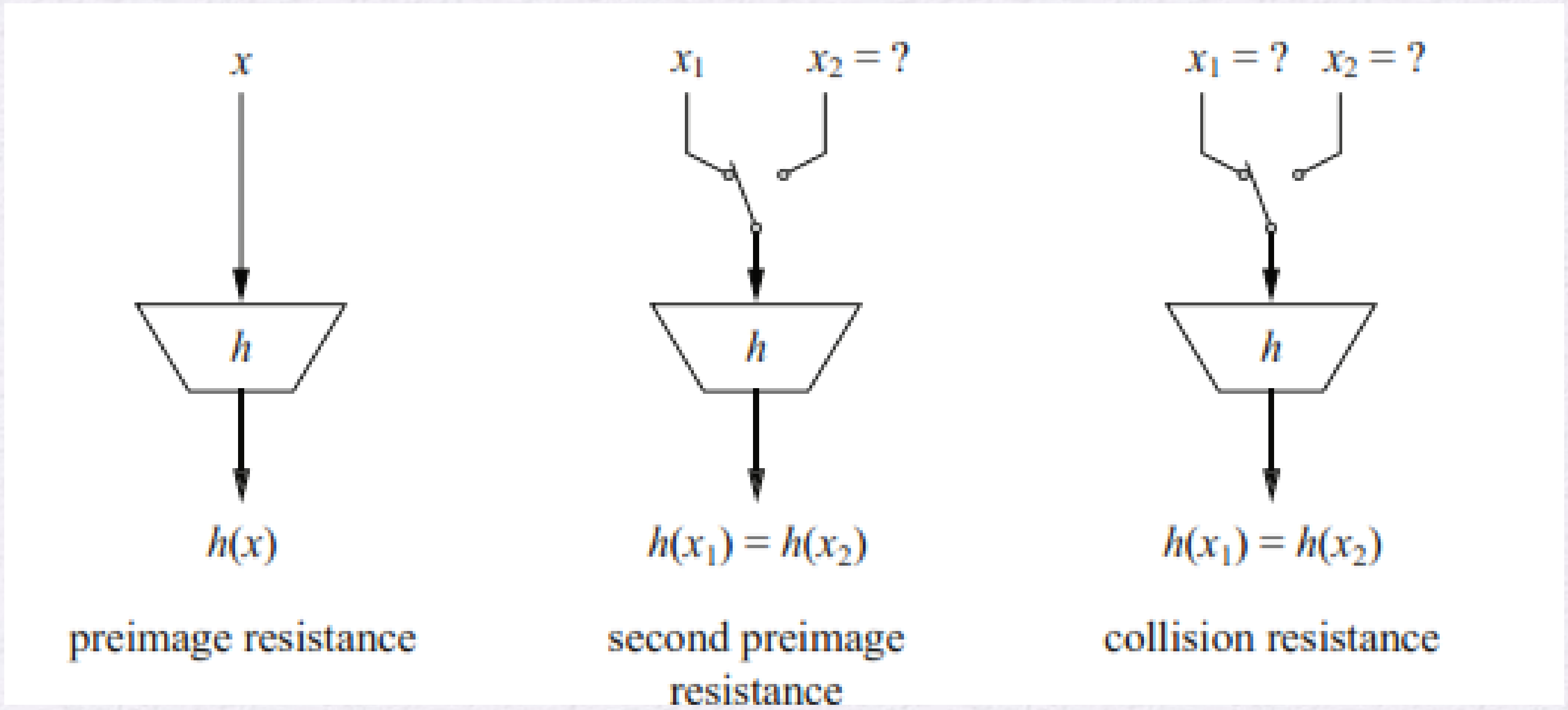
If we had a hash function that somehow computes a fingerprint of the message x, we could perform the signature operation

Basic Protocol for Digital Signatures with a Hash Function:

$$\begin{array}{ccc}
\textbf{Alice} & & \textbf{Bob} \\
 & \xleftarrow{\quad k_{pub,B} \quad} & \\
 & & z = h(x) \\
 & & s = \mathrm{sig}_{k_{pr,B}}(z) \\
 & \xleftarrow{\quad (x,s) \quad} & \\
z' = h(x) & & \\
\mathrm{ver}_{k_{pub,B}}(s, z') = \text{true/false} & &
\end{array}$$

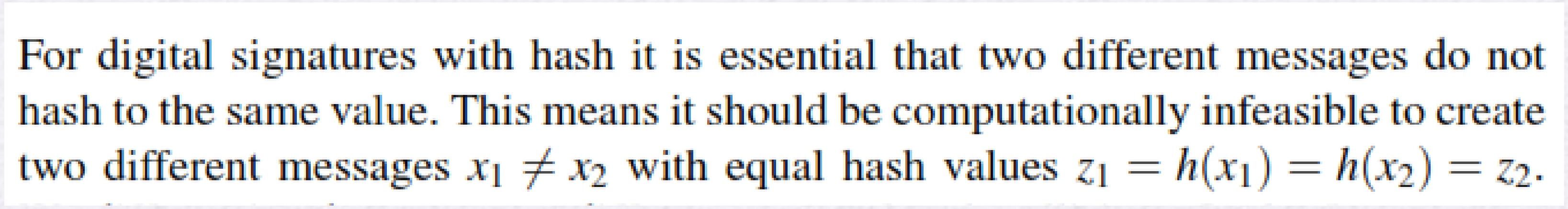
Properties

- Three central properties of hash functions:
- 1. preimage resistance (or one-wayness)
- 2. second preimage resistance (or weak collision resistance)
- 3. collision resistance (or strong collision resistance)

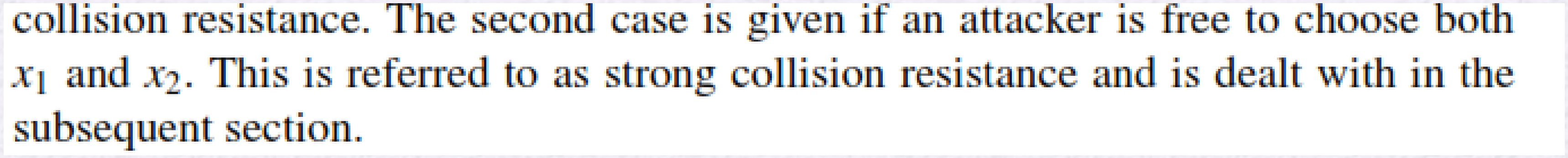


(a) a data object that maps to a pre-specified hash result
(the one-way property) or (b) two data objects that map to the same hash result

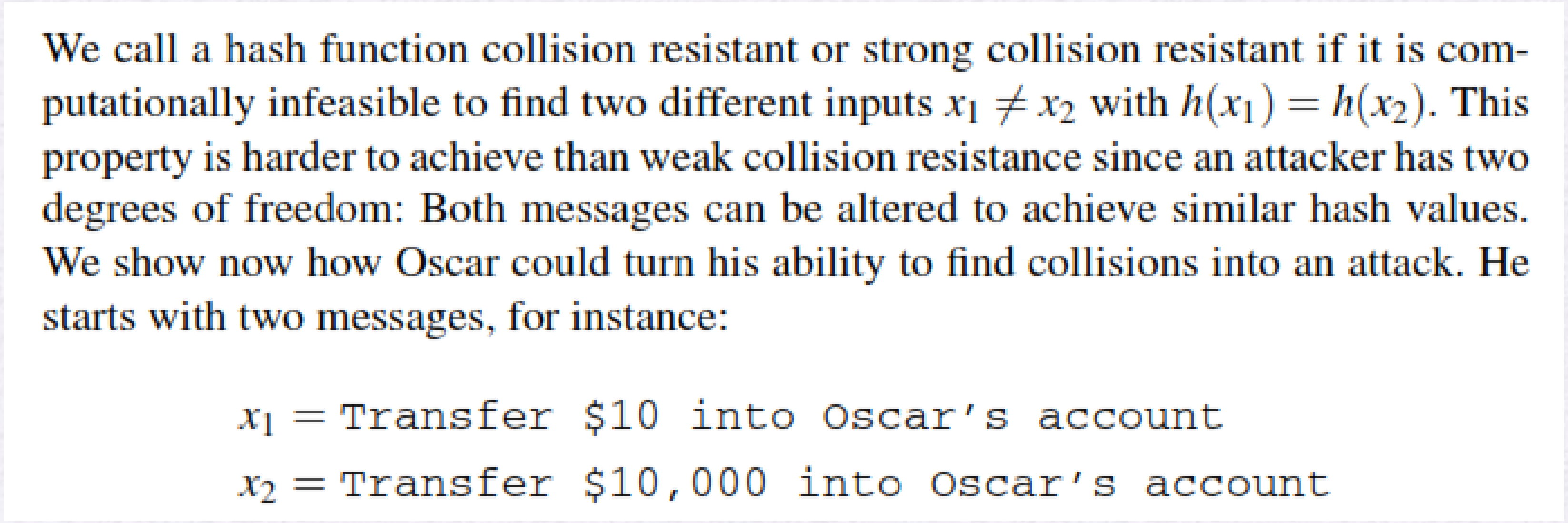
- If the hash function is *not one-way, Oscar can now compute the message x from $h^{-1}(z)=x$. Thus, the symmetric encryption of x is circumvented by the signature,* which leaks the plaintext. For this reason, *h(x) should be a one-way function.*


For digital signatures with hash it is essential that two different messages do not hash to the same value. This means it should be computationally infeasible to create two different messages $x_1 \neq x_2$ with equal hash values $z_1 = h(x_1) = h(x_2) = z_2$.

In the first case x1 is given, and we can try to find x2 – Weak Collision


collision resistance. The second case is given if an attacker is free to choose both x_1 and x_2. This is referred to as strong collision resistance and is dealt with in the subsequent section.

Birthday attack



We call a hash function collision resistant or strong collision resistant if it is computationally infeasible to find two different inputs $x_1 \neq x_2$ with $h(x_1) = h(x_2)$. This property is harder to achieve than weak collision resistance since an attacker has two degrees of freedom: Both messages can be altered to achieve similar hash values. We show now how Oscar could turn his ability to find collisions into an attack. He starts with two messages, for instance:

$x_1 =$ `Transfer $10 into Oscar's account`

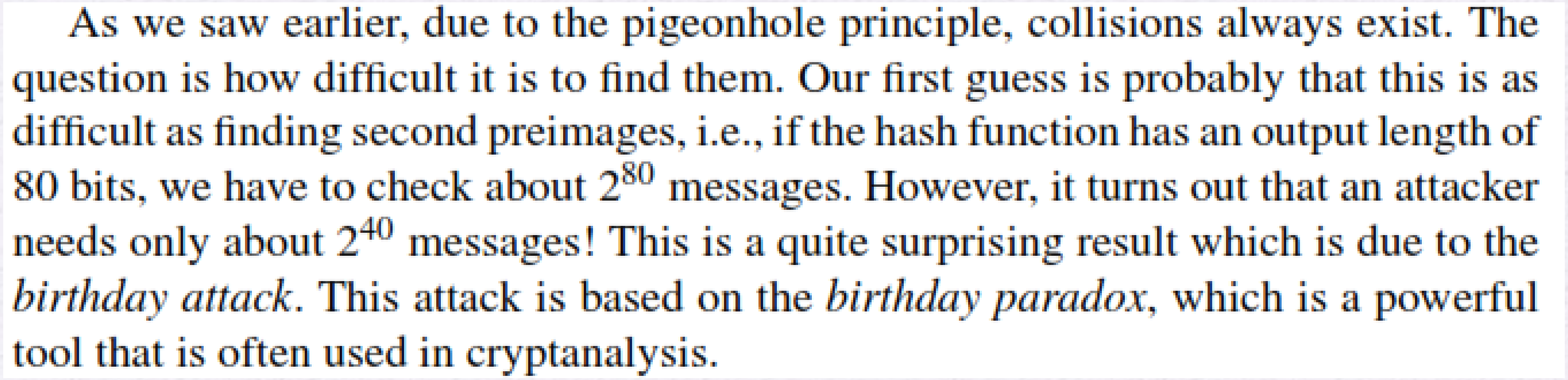
$x_2 =$ `Transfer $10,000 into Oscar's account`

$$\begin{array}{lcccl}
\textbf{Alice} & & \textbf{Oscar} & & \textbf{Bob} \\
 & & & \xleftarrow{k_{pub,B}} & \\
 & & & \xrightarrow{x_1} & \\
 & & & & z = h(x_1) \\
 & & & & s = \mathrm{sig}_{k_{pr,B}}(z) \\
 & \xleftarrow{(x_2,s)} & \lightning \text{ substitute} & \xleftarrow{(x_1,s)} & \\
z = h(x_2) & & & & \\
\mathrm{ver}_{k_{pub,B}}(s,z) = \mathrm{true} & & & &
\end{array}$$

Example

- This attack assumes that Oscar can trick Bob into signing the message x_1. This is, of course, not possible in every situation, but one can imagine scenarios where Oscar can pose as an innocent party, e.g., an e-commerce vendor on the Internet, and x is the purchase order that is generated by Oscar.

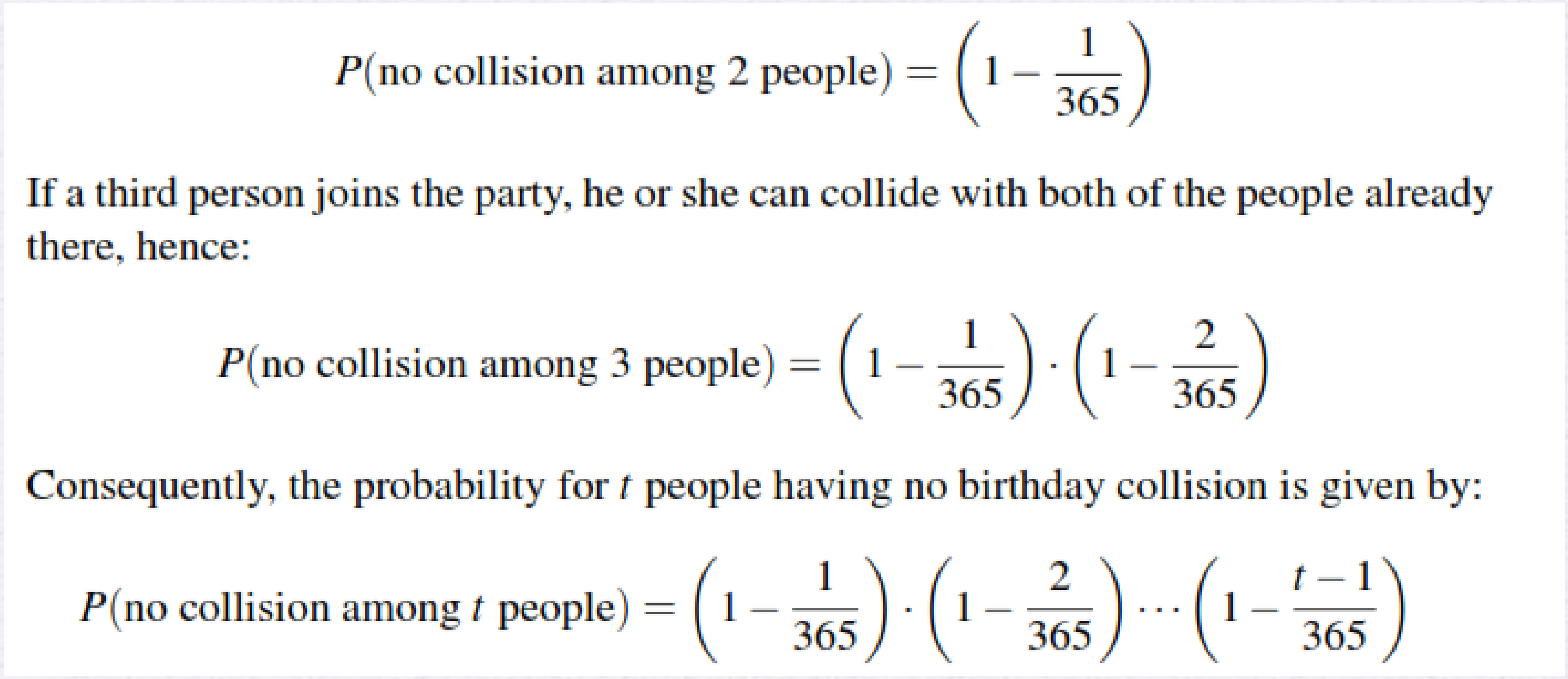
Collision attacks



As we saw earlier, due to the pigeonhole principle, collisions always exist. The question is how difficult it is to find them. Our first guess is probably that this is as difficult as finding second preimages, i.e., if the hash function has an output length of 80 bits, we have to check about 2^{80} messages. However, it turns out that an attacker needs only about 2^{40} messages! This is a quite surprising result which is due to the *birthday attack*. This attack is based on the *birthday paradox*, which is a powerful tool that is often used in cryptanalysis.

- How many people are needed at a party such that there is a reasonable chance that at least two people have the same birthday?
- We have 365 days in a year. Our intuition might lead us to assume that we need around 183 people for a collision to occur.

- For one person, the probability of no collision is 1, which is trivial since a single birthday cannot collide with anyone else's.
- For the second person, the probability of no collision is 364 over 365, since there is only one day, the birthday of the first person, to collide with:



$$P(\text{no collision among 2 people}) = \left(1 - \frac{1}{365}\right)$$

If a third person joins the party, he or she can collide with both of the people already there, hence:

$$P(\text{no collision among 3 people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right)$$

Consequently, the probability for t people having no birthday collision is given by:

$$P(\text{no collision among } t \text{ people}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{t-1}{365}\right)$$

- For *t = 366 people we will have a collision with probability 1 since a year has only* 365 days.

- how many people are needed to have a 50% chance of two colliding birthdays? Surprisingly—following from the equations above—it only requires 23 people to obtain a probability of about 0.5for a birthday collision since:

$$\begin{aligned} P(\text{at least one collision}) &= 1 - P(\text{no collision}) \\ &= 1 - \left(1 - \frac{1}{365}\right) \cdots \left(1 - \frac{23-1}{365}\right) \\ &= 0.507 \approx 50\%. \end{aligned}$$

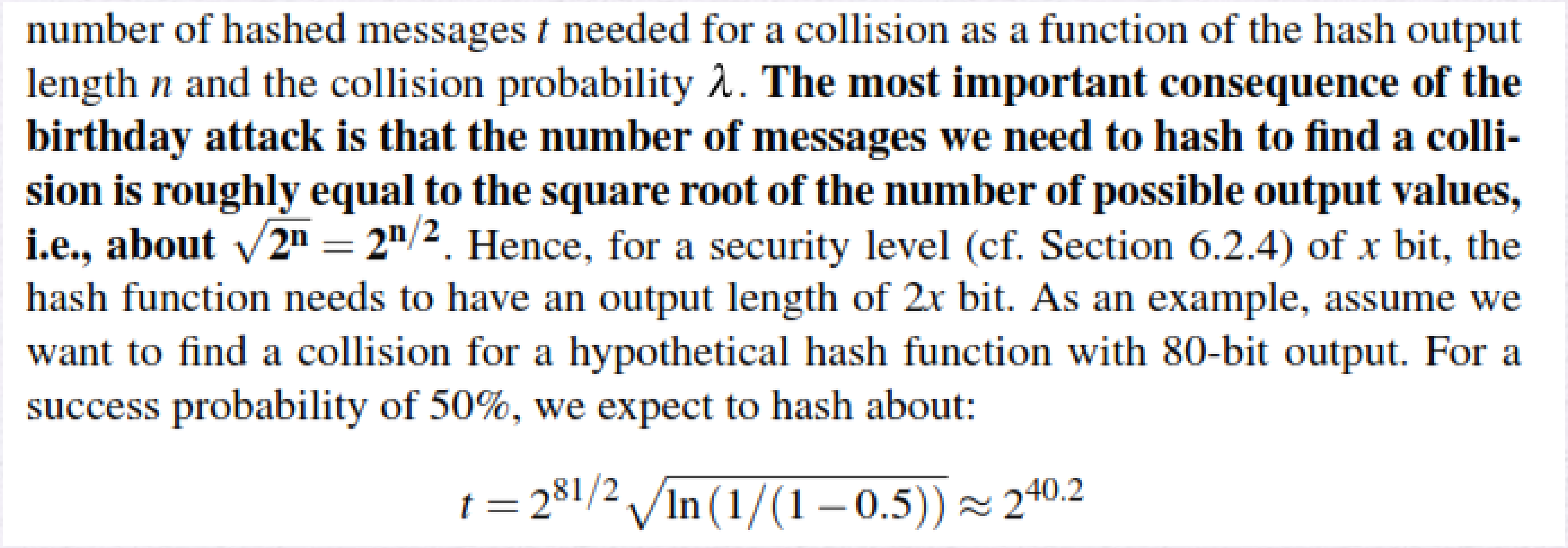
40 people the probability is about 90%

- Collision search for a hash function *h() is exactly the same problem as finding* birthday collisions among party attendees. For a hash function there are not 365 values each element can take but 2^n , where *n is the output width of h()*

how many messages $(x_1, x_2, \ldots, x_t)$ does Oscar need to hash until he has a reasonable chance that $h(x_i) = h(x_j)$ for some x_i and x_j that he picked. The probability for no collisions among t hash values is:

$$\begin{aligned} P(\text{no collision}) &= \left(1 - \frac{1}{2^n}\right)\left(1 - \frac{2}{2^n}\right)\cdots\left(1 - \frac{t-1}{2^n}\right) \\ &= \prod_{i=1}^{t-1}\left(1 - \frac{i}{2^n}\right) \end{aligned}$$

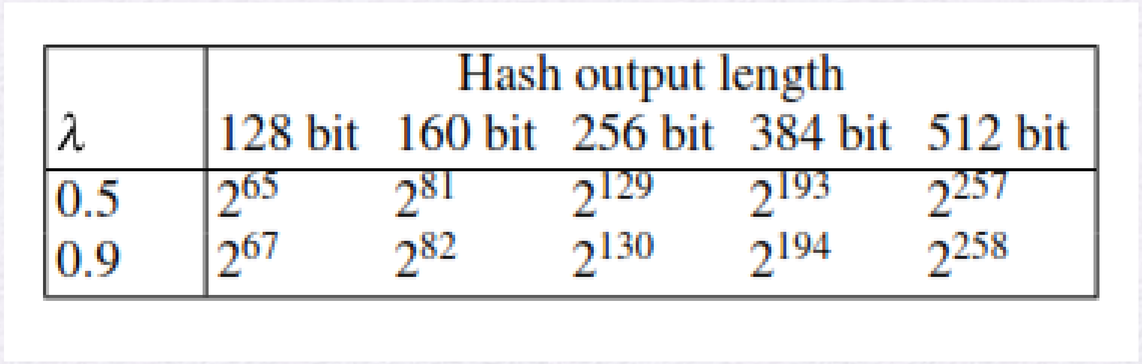
$$t \approx 2^{(n+1)/2}\sqrt{\ln\left(\frac{1}{1-\lambda}\right)}.$$


number of hashed messages t needed for a collision as a function of the hash output length n and the collision probability λ. **The most important consequence of the birthday attack is that the number of messages we need to hash to find a collision is roughly equal to the square root of the number of possible output values, i.e., about $\sqrt{2^n} = 2^{n/2}$.** Hence, for a security level (cf. Section 6.2.4) of x bit, the hash function needs to have an output length of $2x$ bit. As an example, assume we want to find a collision for a hypothetical hash function with 80-bit output. For a success probability of 50%, we expect to hash about:

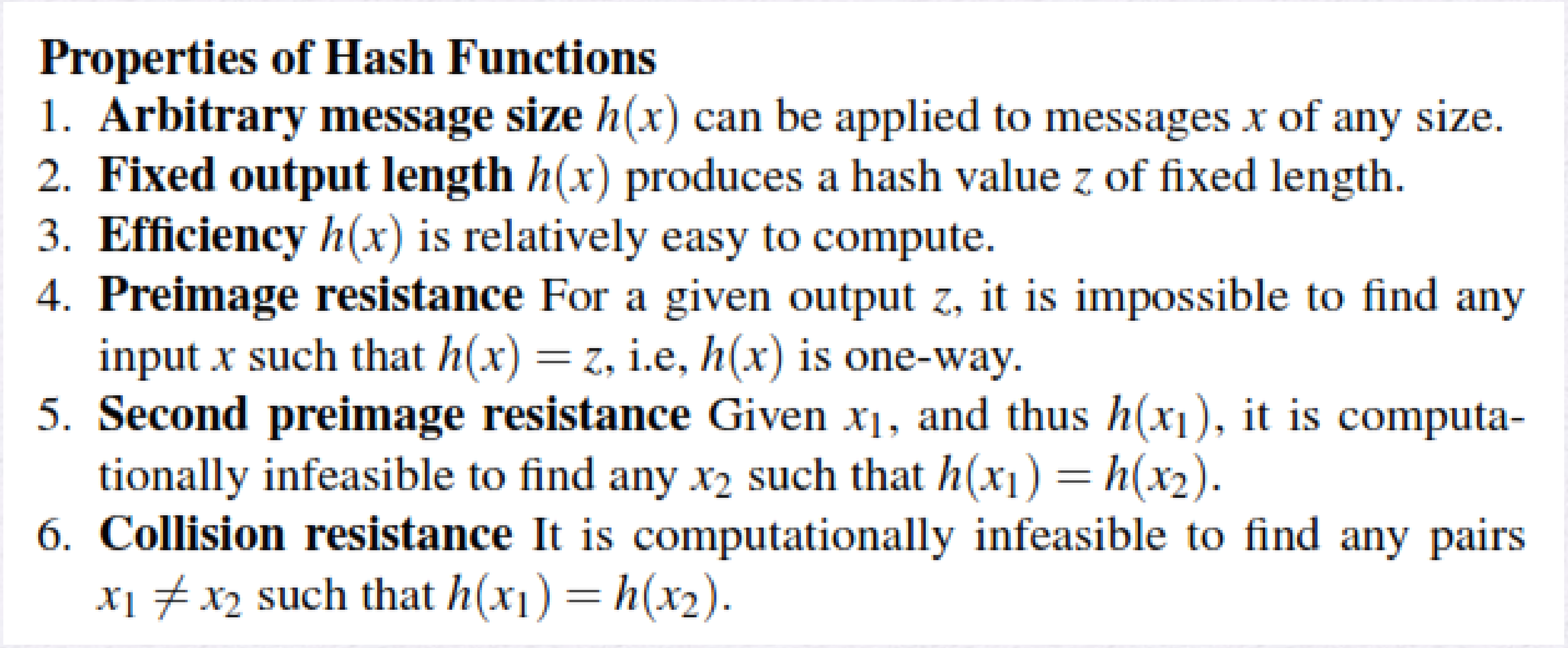
$$t = 2^{81/2}\sqrt{\ln(1/(1-0.5))} \approx 2^{40.2}$$

Hash output

- Computing around 2^{40} hashes and checking for collisions can be done with current laptops!
- For this reason, all hash functions have an output length of at least 128 bit, where most modern ones are much longer.



	Hash output length				
λ	128 bit	160 bit	256 bit	384 bit	512 bit
0.5	2^{65}	2^{81}	2^{129}	2^{193}	2^{257}
0.9	2^{67}	2^{82}	2^{130}	2^{194}	2^{258}



Properties of Hash Functions

1. **Arbitrary message size** $h(x)$ can be applied to messages x of any size.
2. **Fixed output length** $h(x)$ produces a hash value z of fixed length.
3. **Efficiency** $h(x)$ is relatively easy to compute.
4. **Preimage resistance** For a given output z, it is impossible to find any input x such that $h(x) = z$, i.e, $h(x)$ is one-way.
5. **Second preimage resistance** Given x_1, and thus $h(x_1)$, it is computationally infeasible to find any x_2 such that $h(x_1) = h(x_2)$.
6. **Collision resistance** It is computationally infeasible to find any pairs $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$.

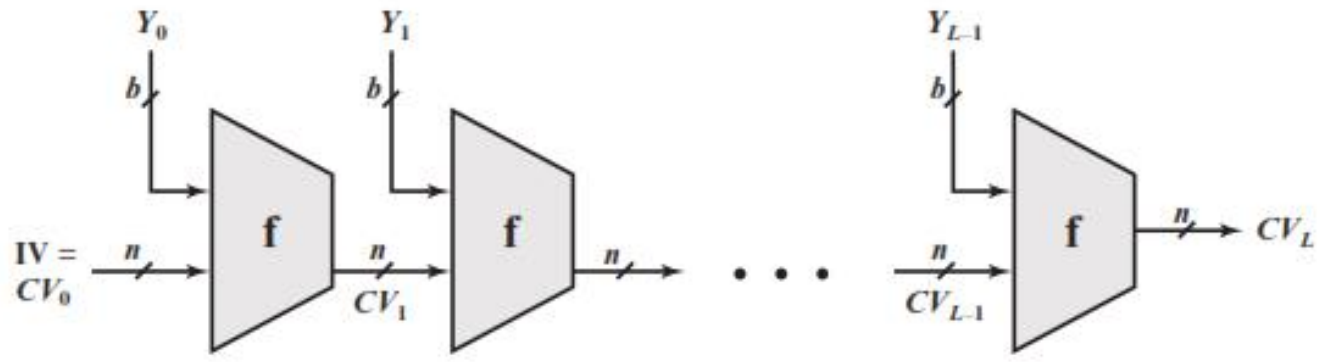
Types of Hash Functions

- There are two general types of hash functions:
- 1. **Dedicated hash functions These are algorithms that are specifically designed to** serve as hash functions.
- 2. **Block cipher-based hash functions It is also possible to use block ciphers such** as AES to construct hash functions.

Dedicated Hash Functions

- MD5 –brokern , SHA 1 (Weak), 2, 3 Secure Hash Algorithm
- Both algorithms have an output length of 160 bit

Contd…



IV = Initial value
CV_i = Chaining variable
Y_i = ith input block
f = Compression algorithm

L = Number of input blocks
n = Length of hash code
b = Length of input block

Figure 11.8 General Structure of Secure Hash Code

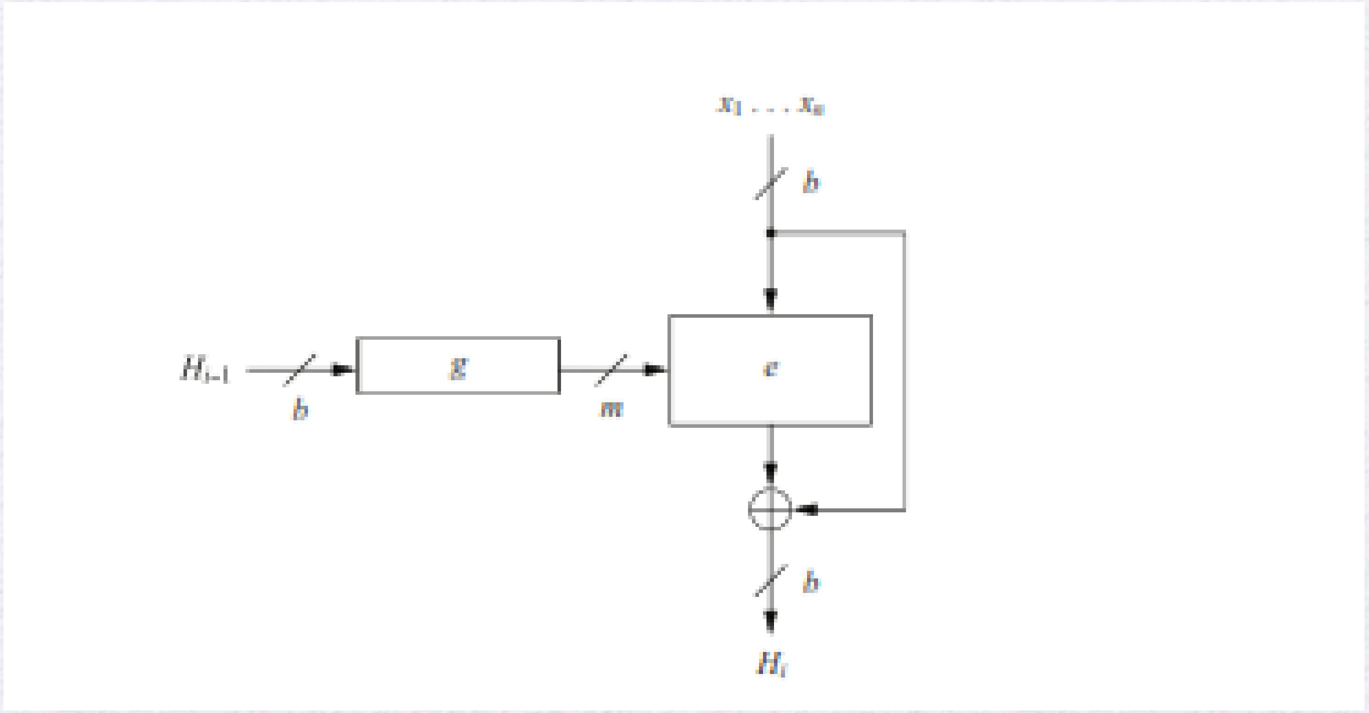
The hash function can be summarized as

$$\begin{aligned} CV_0 &= IV = \text{initial } n\text{-bit value} \\ CV_i &= \mathrm{f}(CV_{i-1}, Y_{i-1}) \quad 1 \le i \le L \\ \mathrm{H}(M) &= CV_L \end{aligned}$$

where the input to the hash function is a message M consisting of the blocks $Y_0, Y_1, \ldots, Y_{L-1}$.

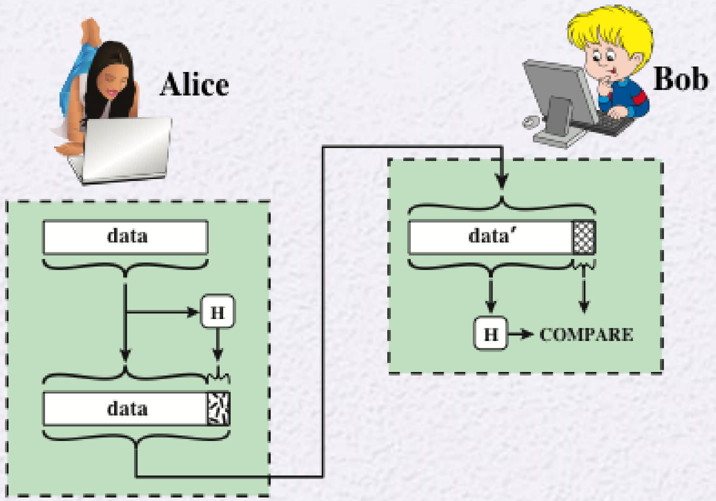
Hash function from Block ciphers

is a b-to-m-bit mapping. In the case of $b = m$, which is, for instance, given if AES with a 128-bit key is being used, the function g can be the identity mapping. After the encryption of the message block x_i, we XOR the result to the original message block. The last output value computed is the hash of the whole message $x_1, x_2, \ldots, x_n$, i.e., $H_n = h(x)$.

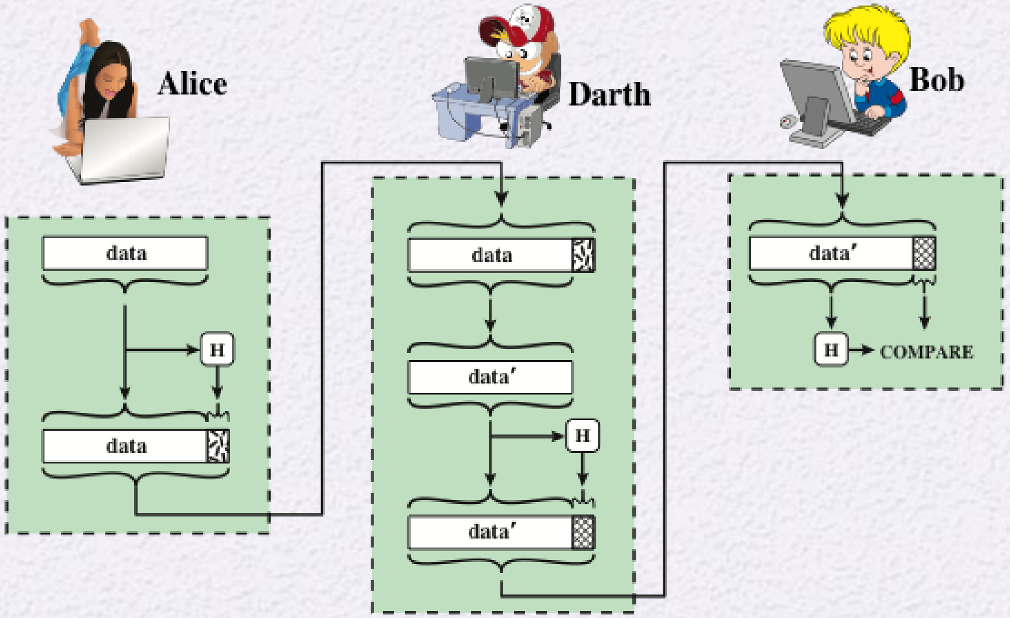


proposals was that of Rabin [RABI78]. Divide a message M into fixed-size blocks $M_1, M_2, \dots, M_N$ and use a symmetric encryption system such as DES to compute the hash code G as

$$\begin{aligned} H_0 &= \text{initial value} \\ H_i &= \mathrm{E}(M_i, H_{i-1}) \\ G &= H_N \end{aligned}$$



(a) Use of hash function to check data integrity



(b) Man-in-the-middle attack

Figure 11.2 Attack Against Hash Function

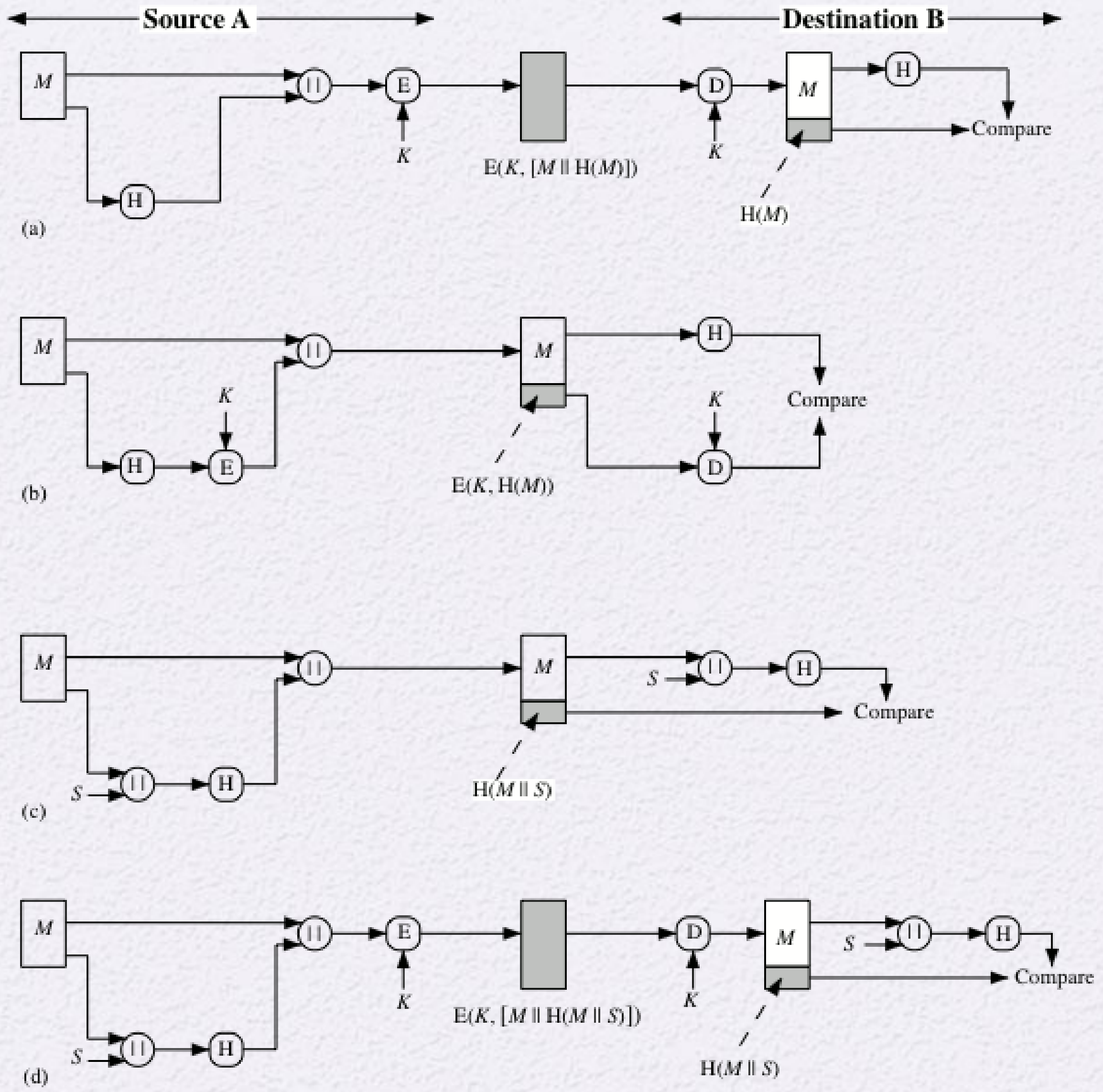


Figure 11.3 Simplified Examples of the Use of a Hash Function for Message Authentication

Message Authentication Code (MAC)

- Also known as a *keyed hash function*
- Typically used between two parties that share a secret key to authenticate information exchanged between those parties

Takes as input a secret key and a data block and produces a hash value (MAC) which is associated with the protected message

- If the integrity of the message needs to be checked, the MAC function can be applied to the message and the result compared with the associated MAC value
- An attacker who alters the message will be unable to alter the associated MAC value without knowledge of the secret key

Digital Signature

- Operation is similar to that of the MAC
- The hash value of a message is encrypted with a user’s private key
- Anyone who knows the user’s public key can verify the integrity of the message
- An attacker who wishes to alter the message would need to know the user’s private key
- Implications of digital signatures go beyond just message authentication

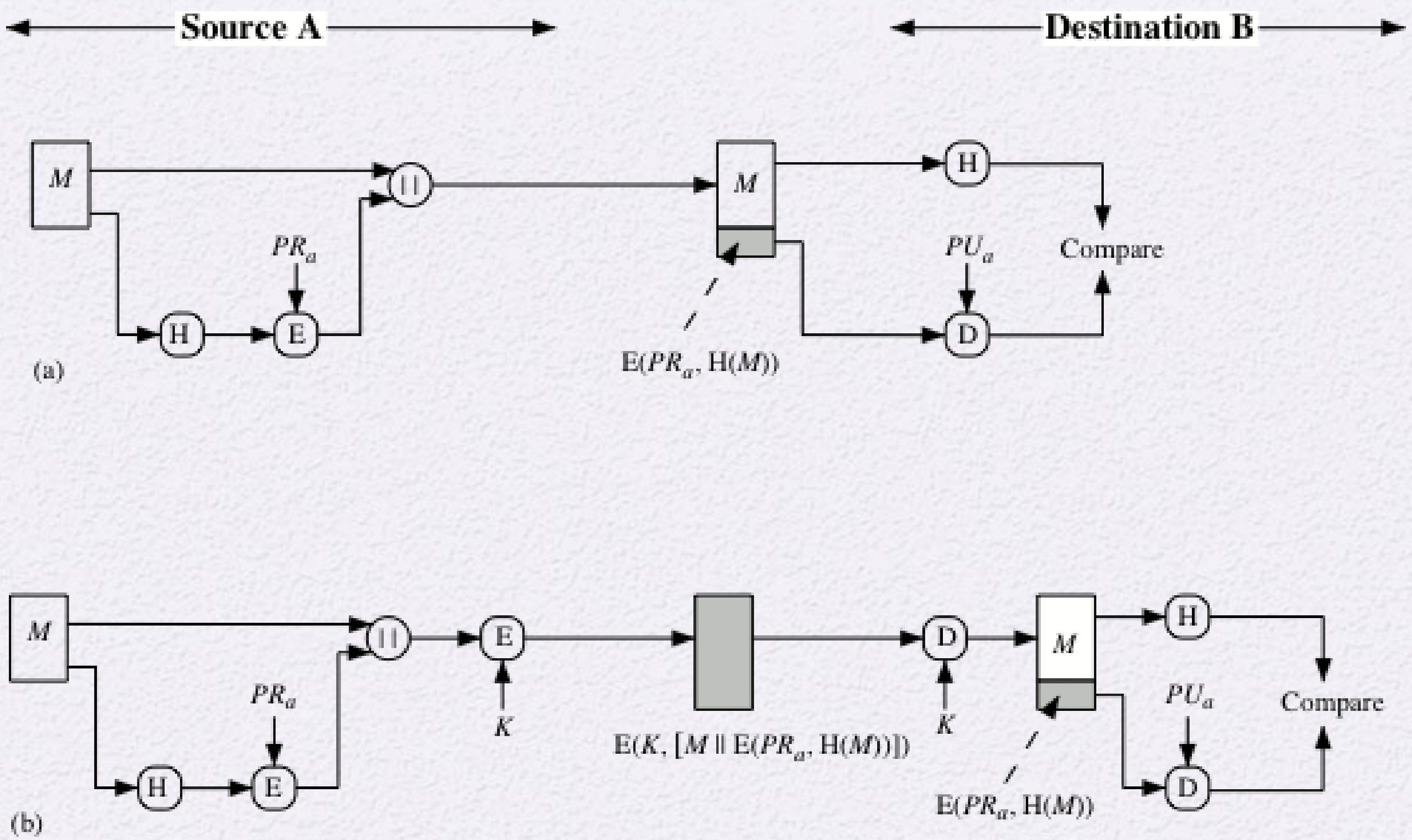


Figure 11.4 Simplified Examples of Digital Signatures

Other Hash Function Uses

Commonly used to create a one-way password file

When a user enters a password, the hash of that password is compared to the stored hash value for verification

This approach to password protection is used by most operating systems

Can be used for intrusion and virus detection

Store H(F) for each file on a system and secure the hash values

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)

Can be used to construct a pseudorandom function (PRF) or a pseudorandom number generator (PRNG)

A common application for a hash-based PRF is for the generation of symmetric keys

Two Simple Hash Functions

- Consider two simple insecure hash functions that operate using the following general principles:
  - The input is viewed as a sequence of *n*-bit blocks
  - The input is processed one block at a time in an iterative fashion to produce an *n*-bit hash function
- Bit-by-bit exclusive-OR (XOR) of every block
  - $C_i = b_{i1}$ *xor* b_{i2} *xor . . . xor* b_{im}
  - Produces a simple parity for each bit position and is known as a longitudinal redundancy check
  - Reasonably effective for random data as a data integrity check
- Perform a one-bit circular shift on the hash value after each block is processed
  - Has the effect of randomizing the input more completely and overcoming any regularities that appear in the input

Two Simple Hash Functions

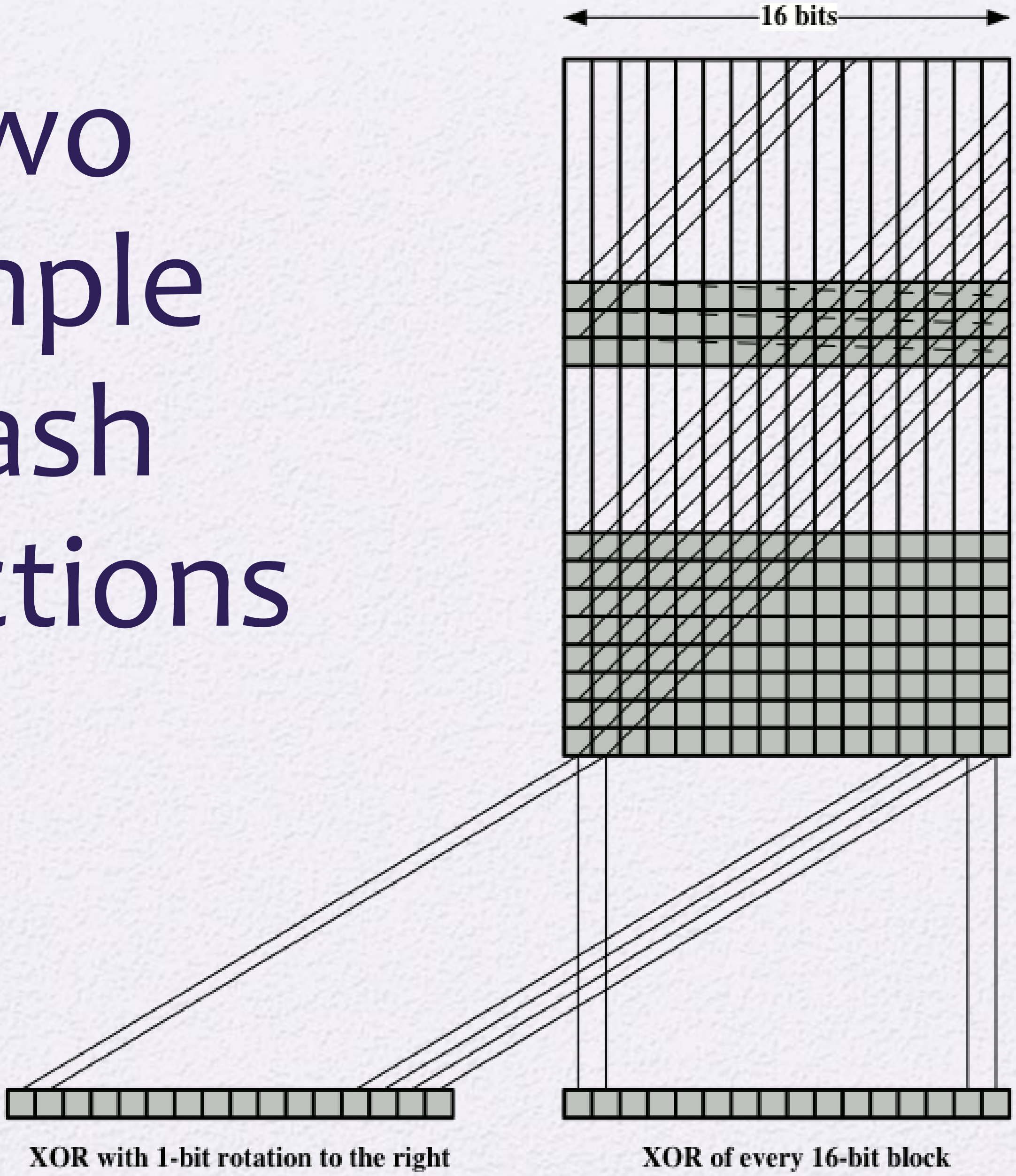

16 bits
XOR with 1-bit rotation to the right
XOR of every 16-bit block

Figure 11.5 Two Simple Hash Functions

Requirements and Security

Preimage

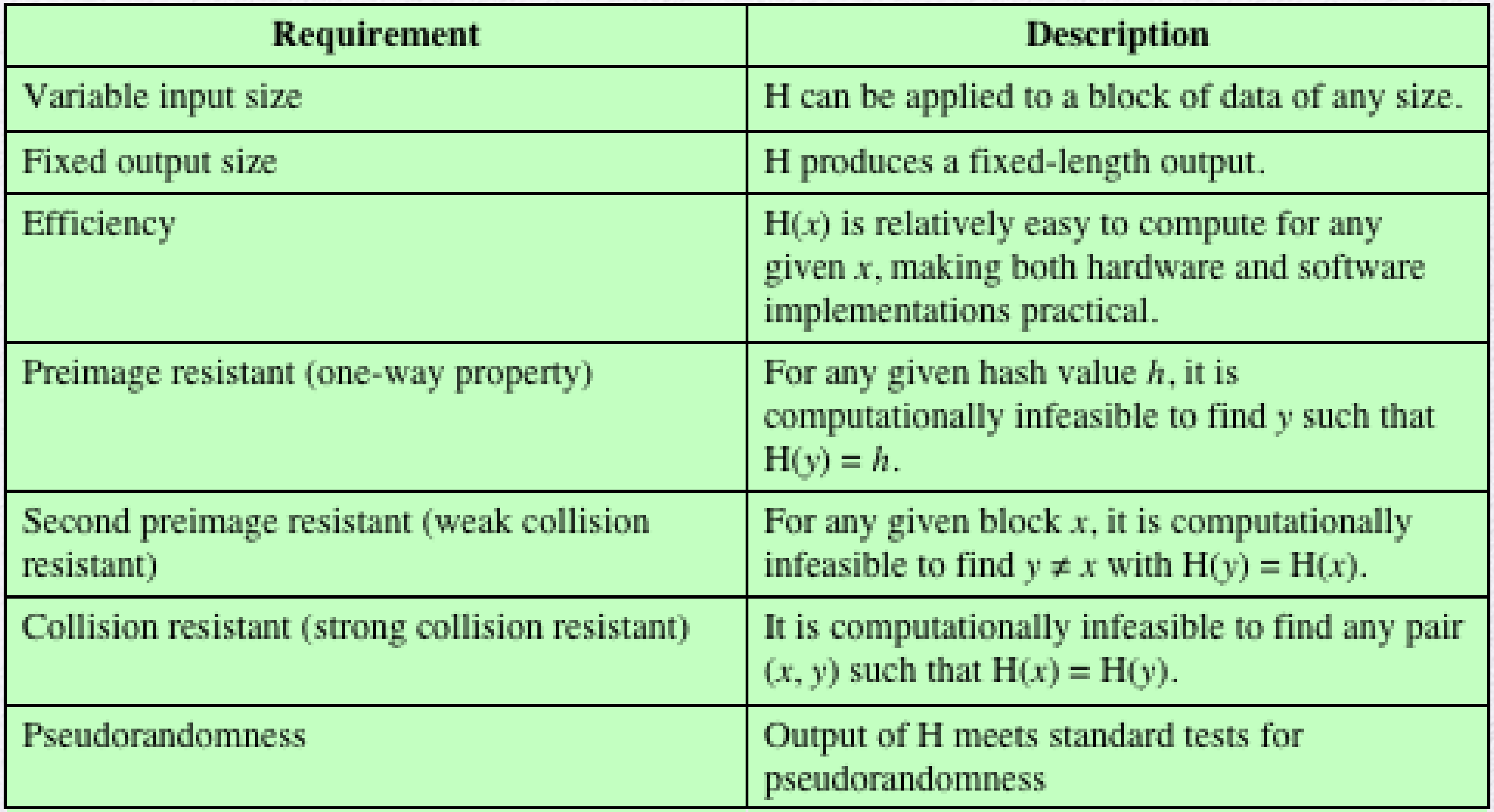
- x is the preimage of h for a hash value $h = H(x)$
- Is a data block whose hash function, using the function H, is h
- Because H is a many-to-one mapping, for any given hash value h, there will in general be multiple preimages

Collision

- Occurs if we have $x \neq y$ and $H(x) = H(y)$
- Because we are using hash functions for data integrity, collisions are clearly undesirable

Table 11.1

Requirements for a Cryptographic Hash Function H



Requirement	Description
Variable input size	H can be applied to a block of data of any size.
Fixed output size	H produces a fixed-length output.
Efficiency	H(x) is relatively easy to compute for any given x, making both hardware and software implementations practical.
Preimage resistant (one-way property)	For any given hash value h, it is computationally infeasible to find y such that H(y) = h.
Second preimage resistant (weak collision resistant)	For any given block x, it is computationally infeasible to find $y \neq x$ with H(y) = H(x).
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (x, y) such that H(x) = H(y).
Pseudorandomness	Output of H meets standard tests for pseudorandomness

(Table can be found on page 323 in textbook.)

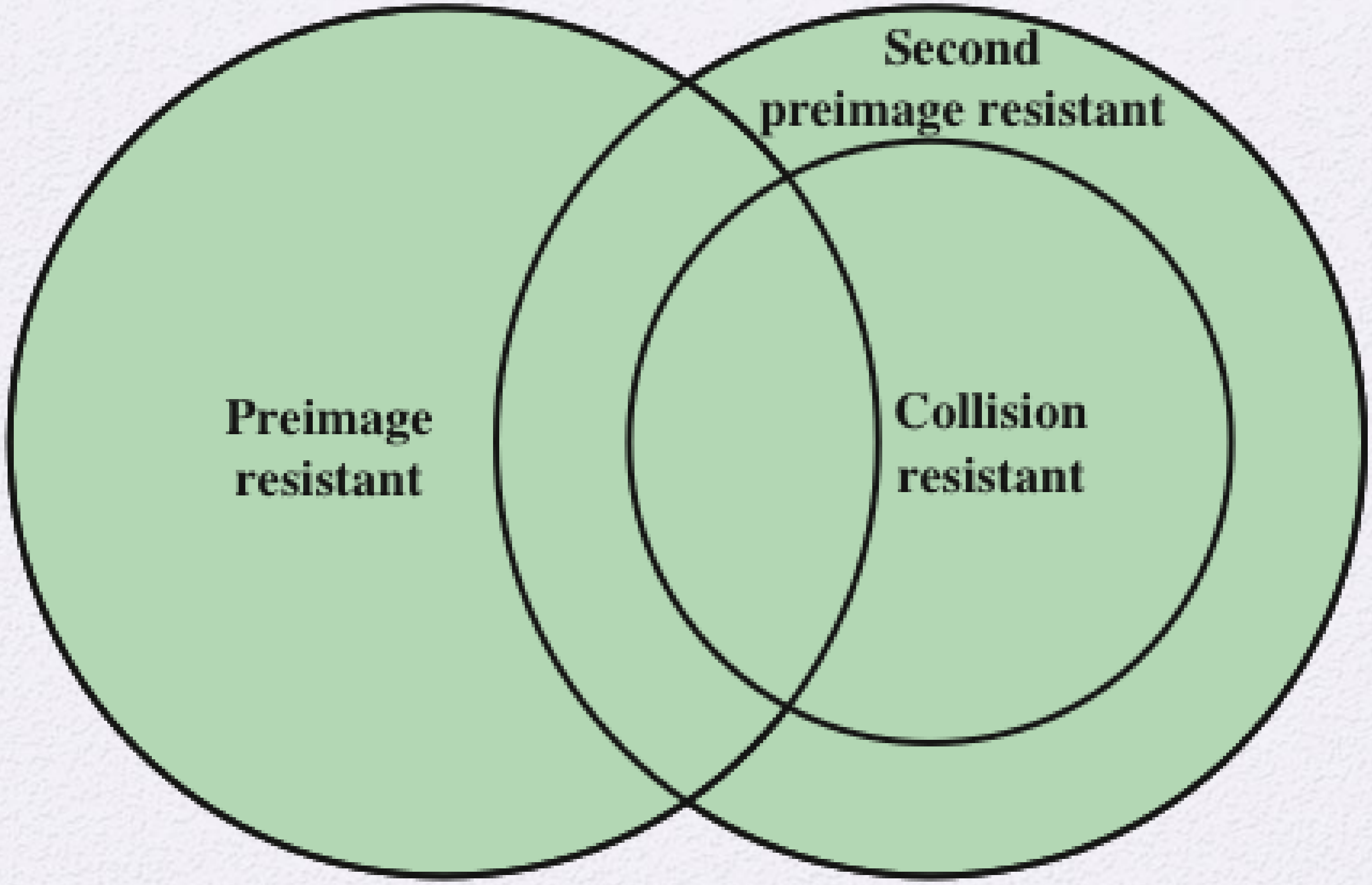
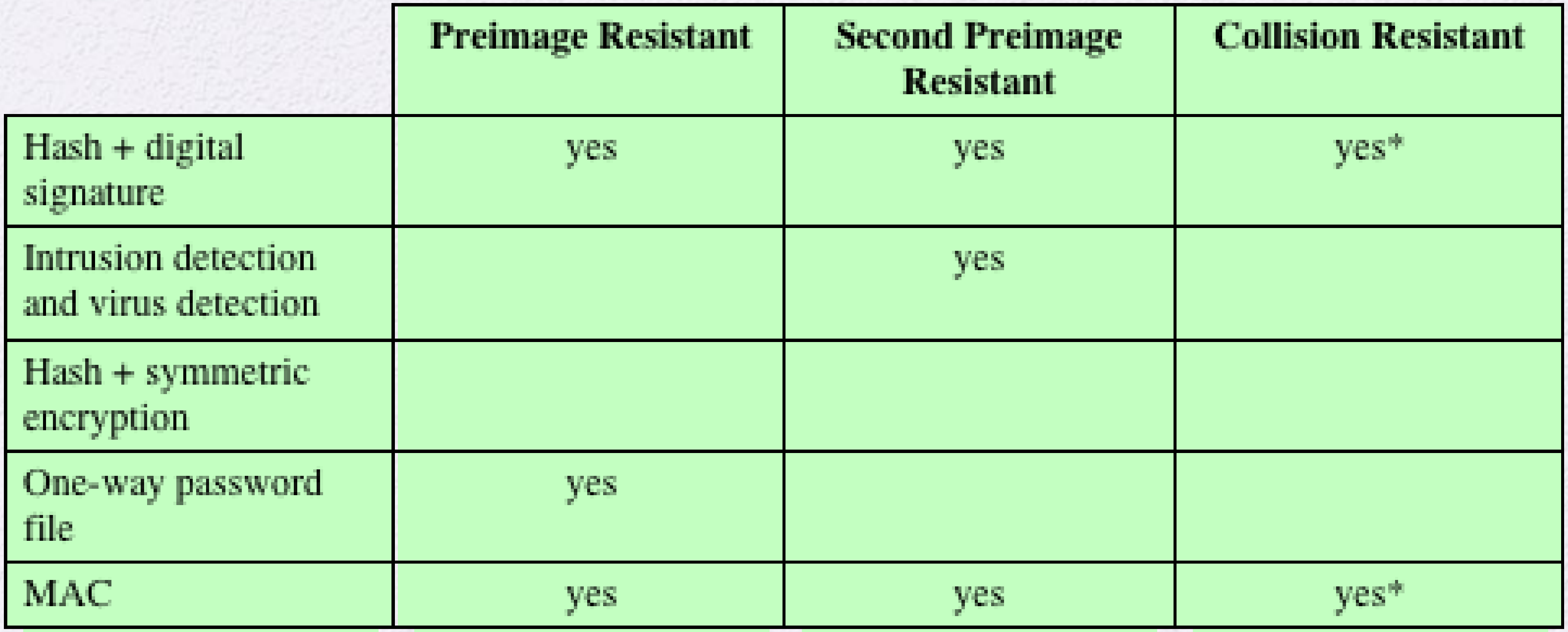


Figure 11.6 Relationship Among Hash Function Properties

Table 11.2

Hash Function Resistance Properties Required for Various Data Integrity Applications



	Preimage Resistant	Second Preimage Resistant	Collision Resistant
Hash + digital signature	yes	yes	yes*
Intrusion detection and virus detection		yes	
Hash + symmetric encryption			
One-way password file	yes		
MAC	yes	yes	yes*

* Resistance required if attacker is able to mount a chosen message attack

Attacks on Hash Functions

Brute-Force Attacks

- Does not depend on the specific algorithm, only depends on bit length
- In the case of a hash function, attack depends only on the bit length of the hash value
- Method is to pick values at random and try each one until a collision occurs

Cryptanalysis

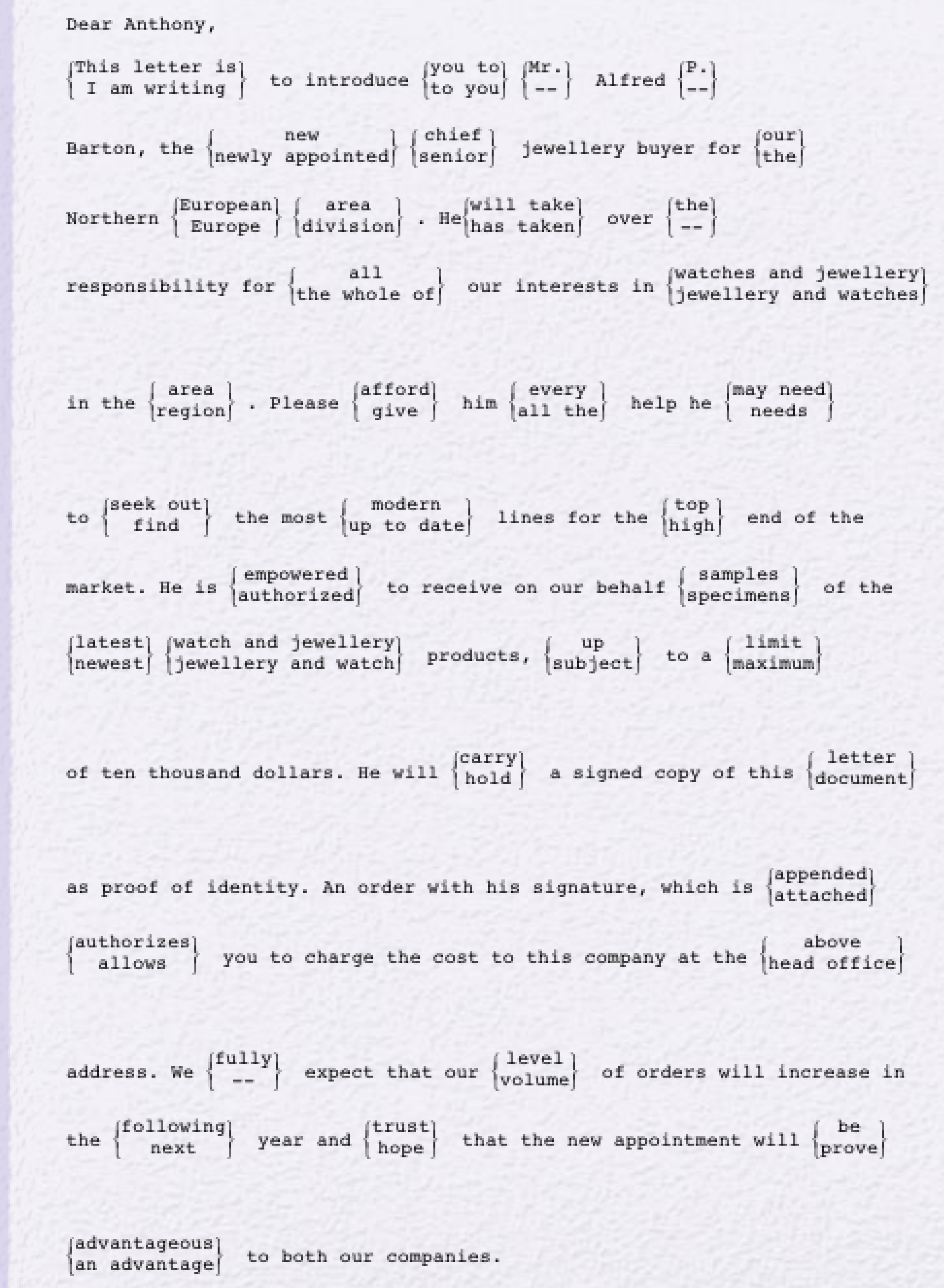
- An attack based on weaknesses in a particular cryptographic algorithm
- Seek to exploit some property of the algorithm to perform some attack other than an exhaustive search

Birthday Attacks

- For a collision resistant attack, an adversary wishes to find two messages or data blocks that yield the same hash function
  - The effort required is explained by a mathematical result referred to as the *birthday paradox*

- How the birthday attack works:
  - The source (A) is prepared to sign a legitimate message *x* by appending the appropriate *m*-bit hash code and encrypting that hash code with A's private key
  - Opponent generates $2^{m/2}$ variations *x'* of *x*, all with essentially the same meaning, and stores the messages and their hash values
  - Opponent generates a fraudulent message *y* for which A's signature is desired
  - Two sets of messages are compared to find a pair with the same hash
  - The opponent offers the valid variation to A for signature which can then be attached to the fraudulent variation for transmission to the intended recipient
    - Because the two variations have the same hash code, they will produce the same signature and the opponent is assured of success even though the encryption key is not known

A Letter in 2^{37} Variation

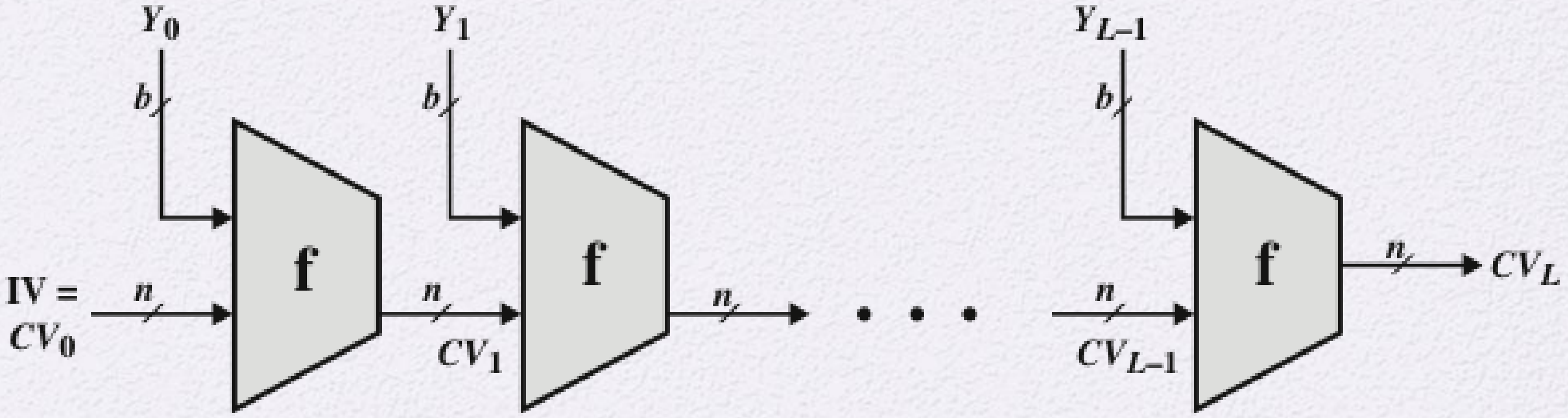


Dear Anthony,

{This letter is / I am writing} to introduce {you to / to you} {Mr. / --} Alfred {P. / --} Barton, the {new / newly appointed} {chief / senior} jewellery buyer for {our / the} Northern {European / Europe} {area / division}. He {will take / has taken} over {the / --} responsibility for {all / the whole of} our interests in {watches and jewellery / jewellery and watches} in the {area / region}. Please {afford / give} him {every / all the} help he {may need / needs} to {seek out / find} the most {modern / up to date} lines for the {top / high} end of the market. He is {empowered / authorized} to receive on our behalf {samples / specimens} of the {latest / newest} {watch and jewellery / jewellery and watch} products, {up / subject} to a {limit / maximum} of ten thousand dollars. He will {carry / hold} a signed copy of this {letter / document} as proof of identity. An order with his signature, which is {appended / attached} {authorizes / allows} you to charge the cost to this company at the {above / head office} address. We {fully / --} expect that our {level / volume} of orders will increase in the {following / next} year and {trust / hope} that the new appointment will {be / prove} {advantageous / an advantage} to both our companies.

Figure 11.7 A Letter in 2^{37} Variations

(Letter is located on page 326 in textbook)



IV	=	Initial value	L	=	number of input blocks
CV_i	=	chaining variable	n	=	length of hash code
Y_i	=	ith input block	b	=	length of input block
f	=	compression algorithm			

Figure 11.8 General Structure of Secure Hash Code

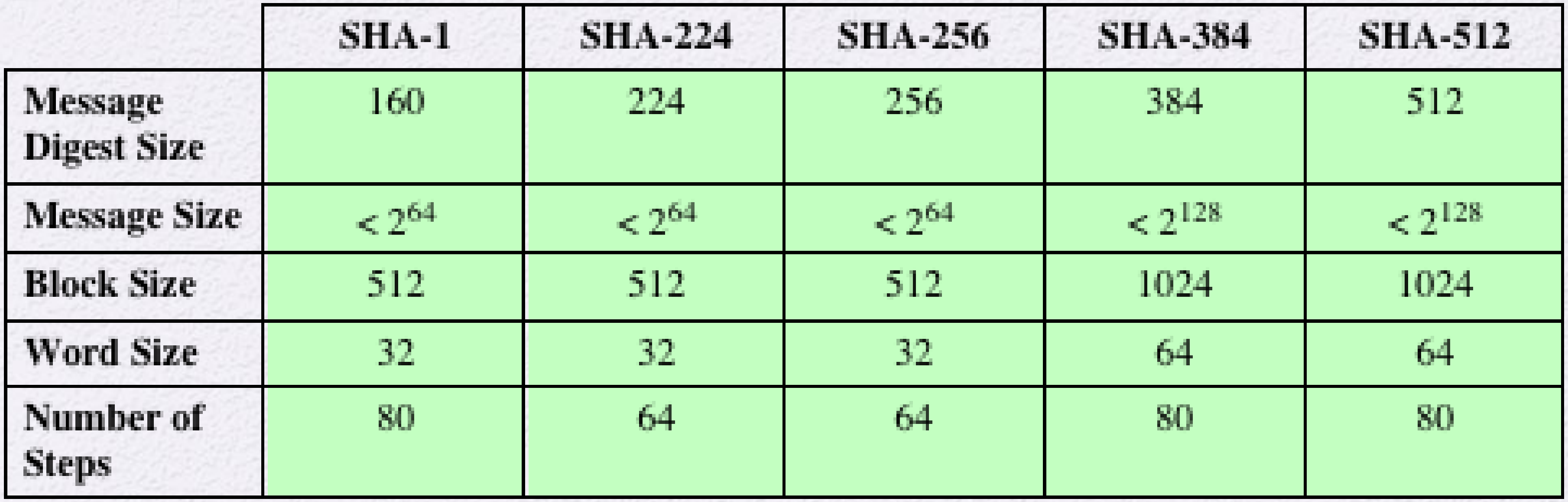
Hash Functions Based on Cipher Block Chaining

- Can use block ciphers as hash functions
  - Using H_0=0 and zero-pad of final block
  - Compute: $H_i = E(M_i\ H_{i-1})$
  - Use final block as the hash value
  - Similar to CBC but without a key
- Resulting hash is too small (64-bit)
  - Both due to direct birthday attack
  - And "meet-in-the-middle" attack
- Other variants also susceptible to attack

Secure Hash Algorithm (SHA)

- SHA was originally designed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993
- Was revised in 1995 as SHA-1
- Based on the hash function MD4 and its design closely models MD4
- Produces 160-bit hash values
- In 2002 NIST produced a revised version of the standard that defined three new versions of SHA with hash value lengths of 256, 384, and 512
  - Collectively known as SHA-2

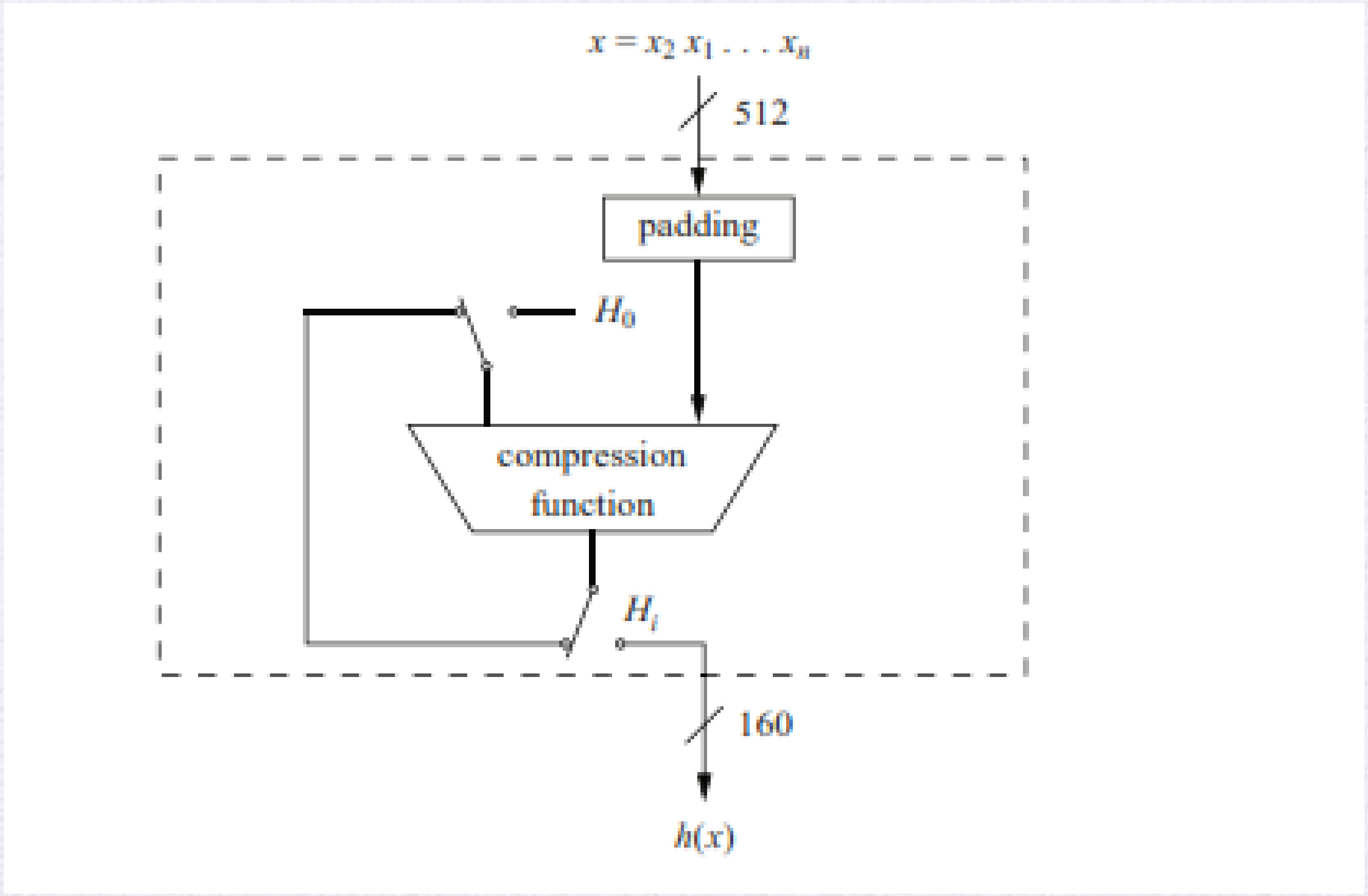
Table 11.3
Comparison of SHA Parameters



	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Message Digest Size	160	224	256	384	512
Message Size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block Size	512	512	512	1024	1024
Word Size	32	32	32	64	64
Number of Steps	80	64	64	80	80

Note: All sizes are measured in bits.

SHA 1



$x = x_2\, x_1 \ldots x_n$
512
padding
H_0
compression function
H_i
160
$h(x)$

Contd…

- During the actual computation, the compression function processes the message in 512-bit chunks. The compression function consists of 80 rounds which are divided into four stages of 20 rounds each.

Pre-processing

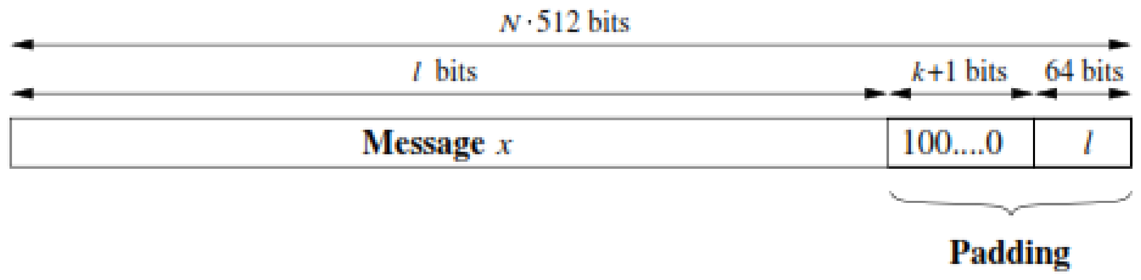
- Before the actual hash computation, the message *x has to be padded to fit a size of* a multiple of 512 bit.
- For the internal processing, the padded message must then be divided into blocks. Also, the initial value *H* is set to a predefined constant.

padding

Padding Assume that we have a message x with a length of l bit. To obtain an overall message size of a multiple of 512 bits, we append a single “1” followed by k zero bits and the binary 64-bit representation of l. Consequently, the number of required zeros k is given by

$$\begin{aligned} k &\equiv 512 - 64 - 1 - l \\ &= 448 - (l + 1) \bmod 512. \end{aligned}$$

Figure 11.10 illustrates the padding of a message x.



Example 11.1. Given is the message "`abc`" consisting of three 8-bit ASCII characters with a total length of $l = 24$ bits:

$$\underbrace{\texttt{01100001}}_{\texttt{a}}\quad\underbrace{\texttt{01100010}}_{\texttt{b}}\quad\underbrace{\texttt{01100011}}_{\texttt{c}}.$$

We append a "1" followed by $k = 423$ zero bits, where k is determined by

$$k \equiv 448 - (l + 1) = 448 - 25 = 423 \bmod 512.$$

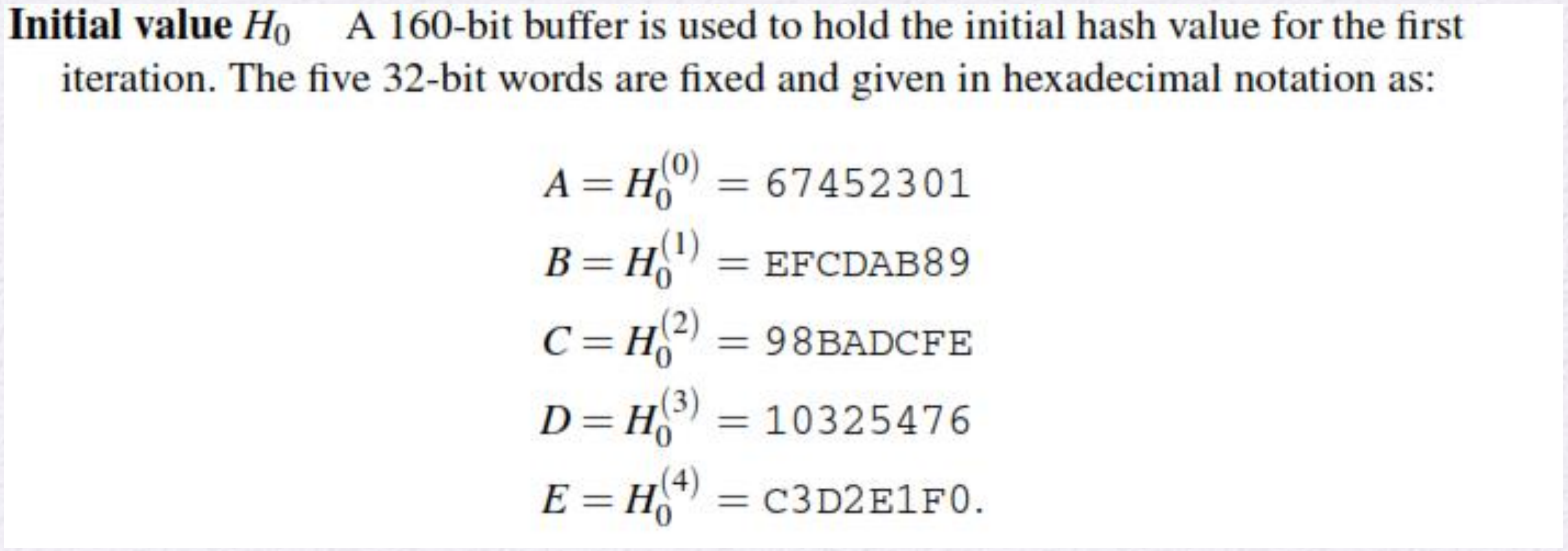
Finally, we append the 64-bit value which contains the binary representation of the length $l = 24_{10} = 11000_2$. The padded message is then given by

$$\underbrace{\texttt{01100001}}_{\texttt{a}}\quad\underbrace{\texttt{01100010}}_{\texttt{b}}\quad\underbrace{\texttt{01100011}}_{\texttt{c}}\quad\texttt{1}\quad\underbrace{\texttt{00...0}}_{\text{423 zeros}}\quad\underbrace{\texttt{00...011000}}_{l=24}.$$

Dividing the padded message Prior to applying the compression function, we need to divide the message into 512-bit blocks $x_1, x_2, \ldots, x_n$. Each 512-bit block can be subdivided into 16 words of size of 32 bits. For instance, the ith block of the message x is split into:

$$x_i = (x_i^{(0)}\ x_i^{(1)}\ \ldots\ x_i^{(15)})$$

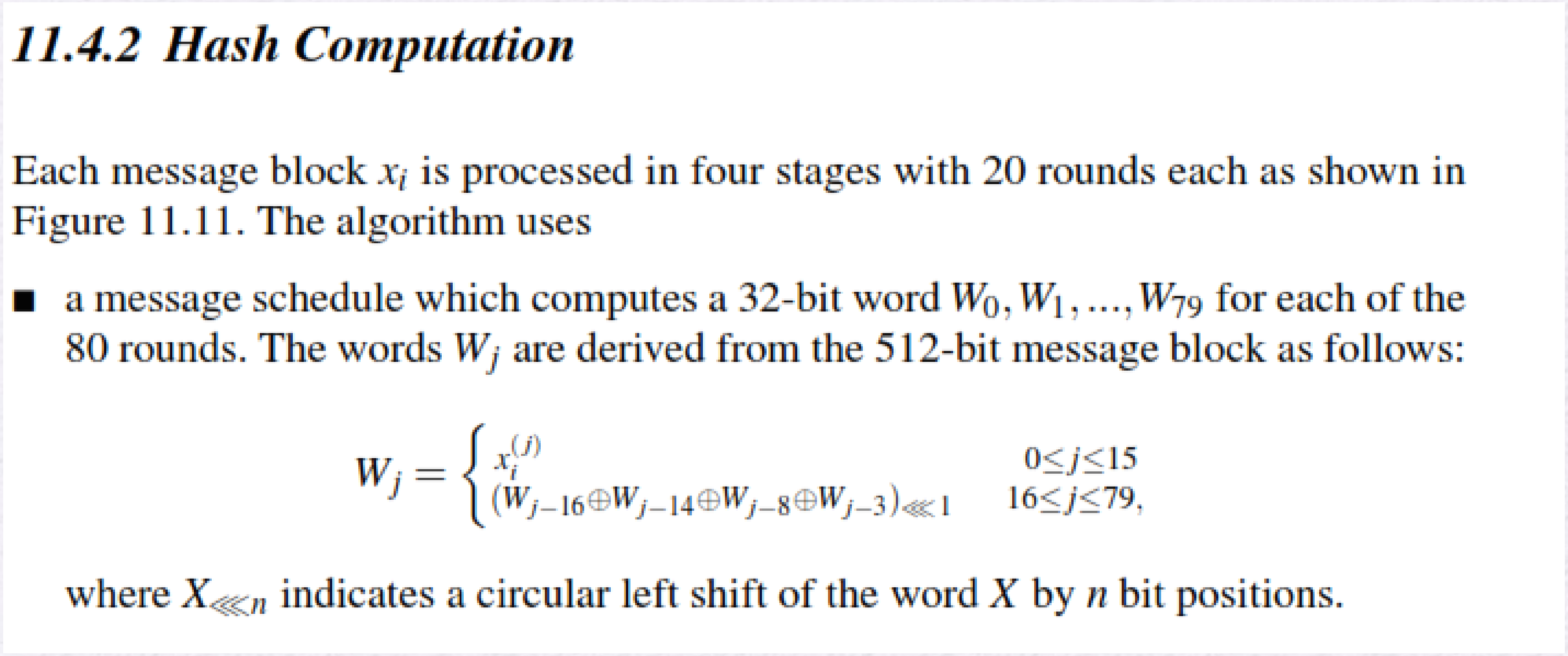
where $x_i^{(k)}$ are words of size of 32 bits.



Initial value H_0 A 160-bit buffer is used to hold the initial hash value for the first iteration. The five 32-bit words are fixed and given in hexadecimal notation as:

$$A = H_0^{(0)} = \texttt{67452301}$$
$$B = H_0^{(1)} = \texttt{EFCDAB89}$$
$$C = H_0^{(2)} = \texttt{98BADCFE}$$
$$D = H_0^{(3)} = \texttt{10325476}$$
$$E = H_0^{(4)} = \texttt{C3D2E1F0}.$$

Hash Computation



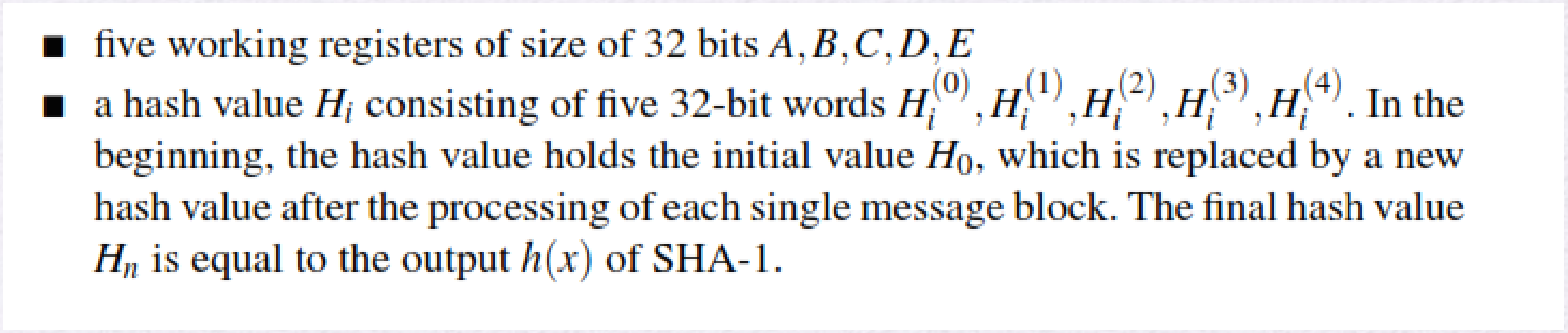
11.4.2 *Hash Computation*

Each message block x_i is processed in four stages with 20 rounds each as shown in Figure 11.11. The algorithm uses

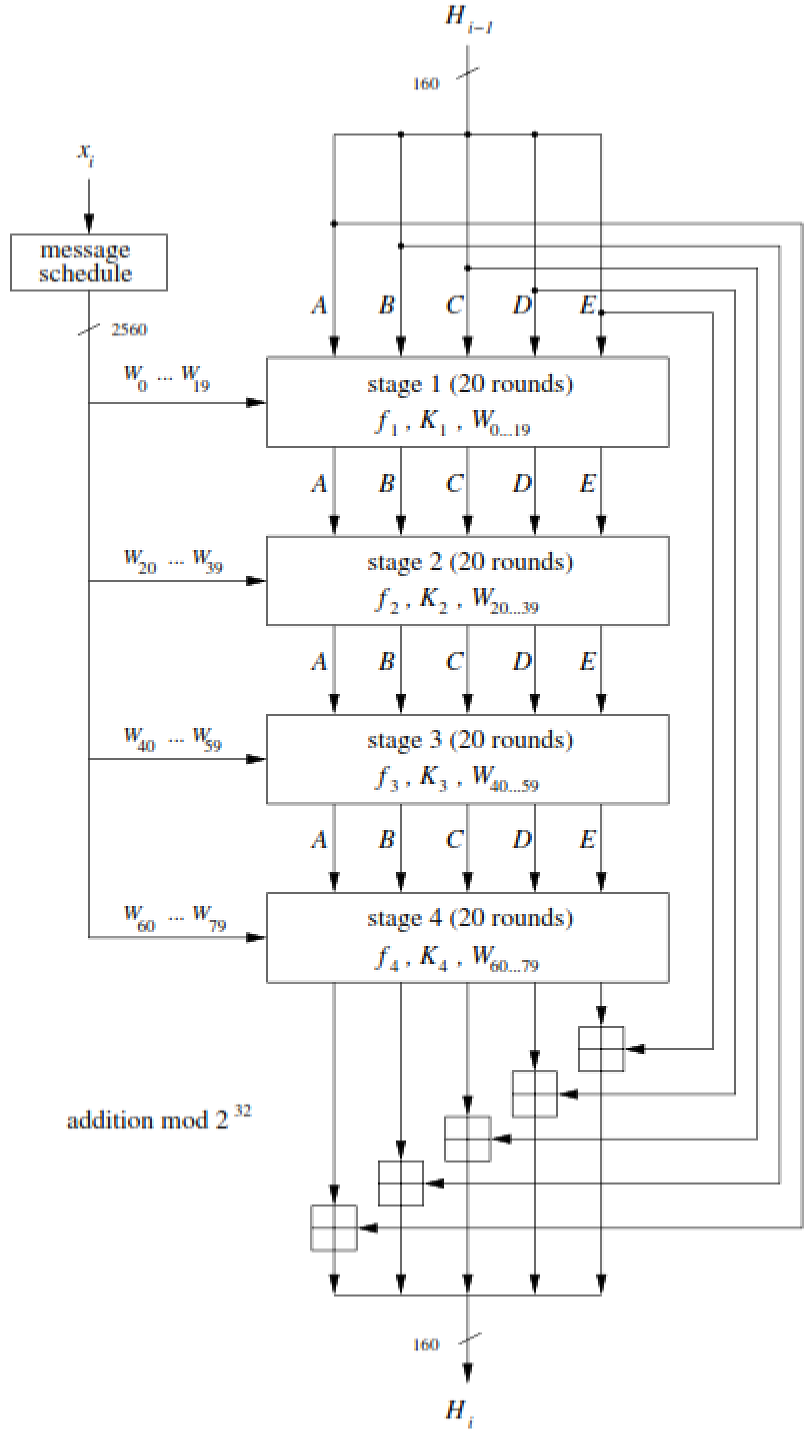
- a message schedule which computes a 32-bit word $W_0, W_1, \ldots, W_{79}$ for each of the 80 rounds. The words W_j are derived from the 512-bit message block as follows:

$$W_j = \begin{cases} x_i^{(j)} & 0 \leq j \leq 15 \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3})_{\lll 1} & 16 \leq j \leq 79, \end{cases}$$

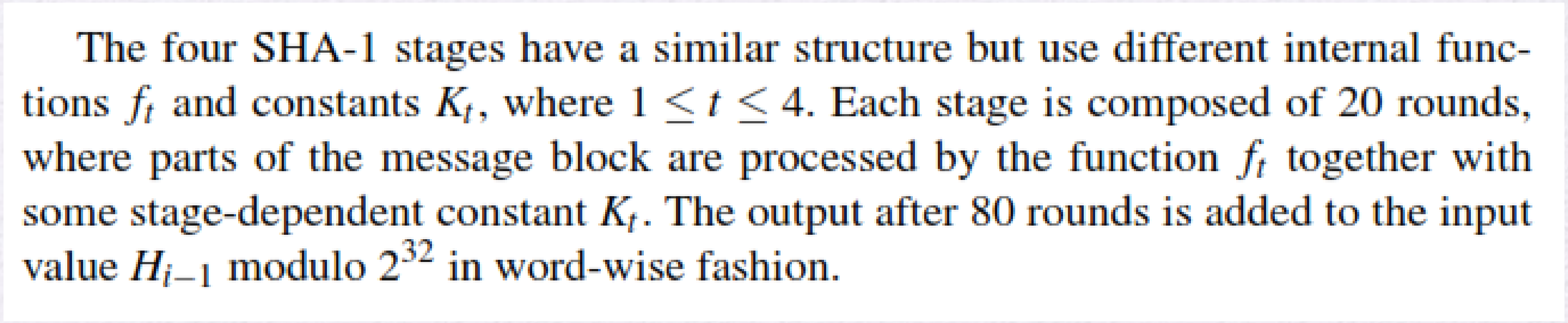
  where $X_{\lll n}$ indicates a circular left shift of the word X by n bit positions.



- five working registers of size of 32 bits A, B, C, D, E
- a hash value H_i consisting of five 32-bit words $H_i^{(0)}, H_i^{(1)}, H_i^{(2)}, H_i^{(3)}, H_i^{(4)}$. In the beginning, the hash value holds the initial value H_0, which is replaced by a new hash value after the processing of each single message block. The final hash value H_n is equal to the output $h(x)$ of SHA-1.



H_{i-1}
160
x_i
message schedule
2560
A B C D E
$W_0 \dots W_{19}$
stage 1 (20 rounds)
$f_1, K_1, W_{0...19}$
$W_{20} \dots W_{39}$
stage 2 (20 rounds)
$f_2, K_2, W_{20...39}$
$W_{40} \dots W_{59}$
stage 3 (20 rounds)
$f_3, K_3, W_{40...59}$
$W_{60} \dots W_{79}$
stage 4 (20 rounds)
$f_4, K_4, W_{60...79}$
addition mod 2^{32}
160
H_i

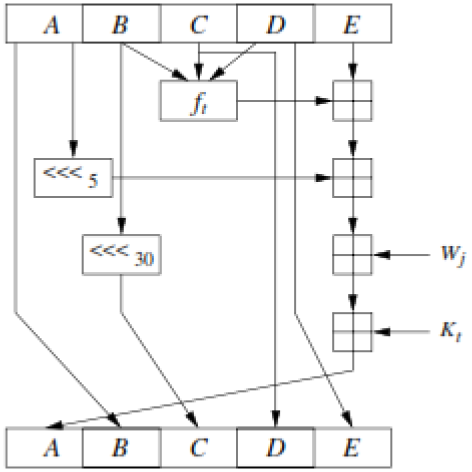


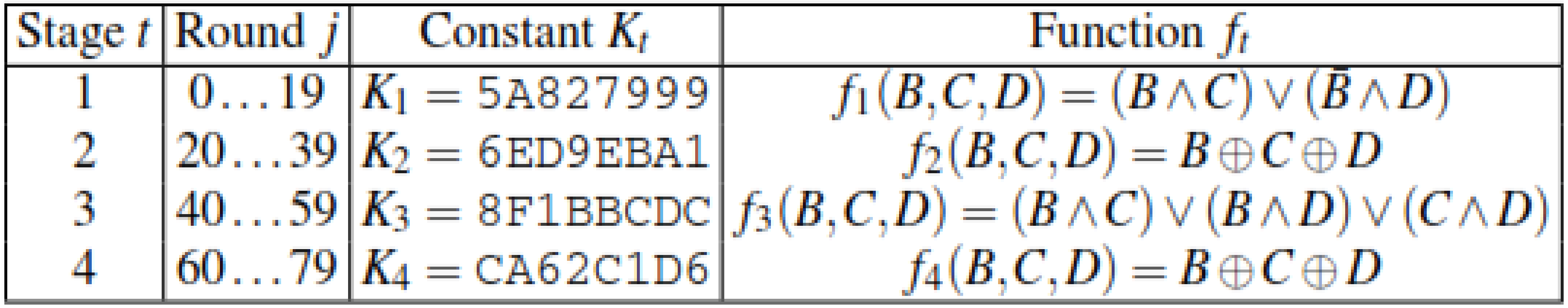
The four SHA-1 stages have a similar structure but use different internal functions f_t and constants K_t, where $1 \leq t \leq 4$. Each stage is composed of 20 rounds, where parts of the message block are processed by the function f_t together with some stage-dependent constant K_t. The output after 80 rounds is added to the input value H_{i-1} modulo 2^{32} in word-wise fashion.

The operation within round j in stage t is given by

$$A,B,C,D,E = (E + f_t(B,C,D) + (A)_{\lll 5} + W_j + K_t), A, (B)_{\lll 30}, C, D$$

and is depicted in Figure 11.12. The internal functions f_t and constants K_t change





Stage t	Round j	Constant K_t	Function f_t
1	$0 \ldots 19$	$K_1 =$ `5A827999`	$f_1(B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)$
2	$20 \ldots 39$	$K_2 =$ `6ED9EBA1`	$f_2(B,C,D) = B \oplus C \oplus D$
3	$40 \ldots 59$	$K_3 =$ `8F1BBCDC`	$f_3(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	$60 \ldots 79$	$K_4 =$ `CA62C1D6`	$f_4(B,C,D) = B \oplus C \oplus D$

Digital Signature

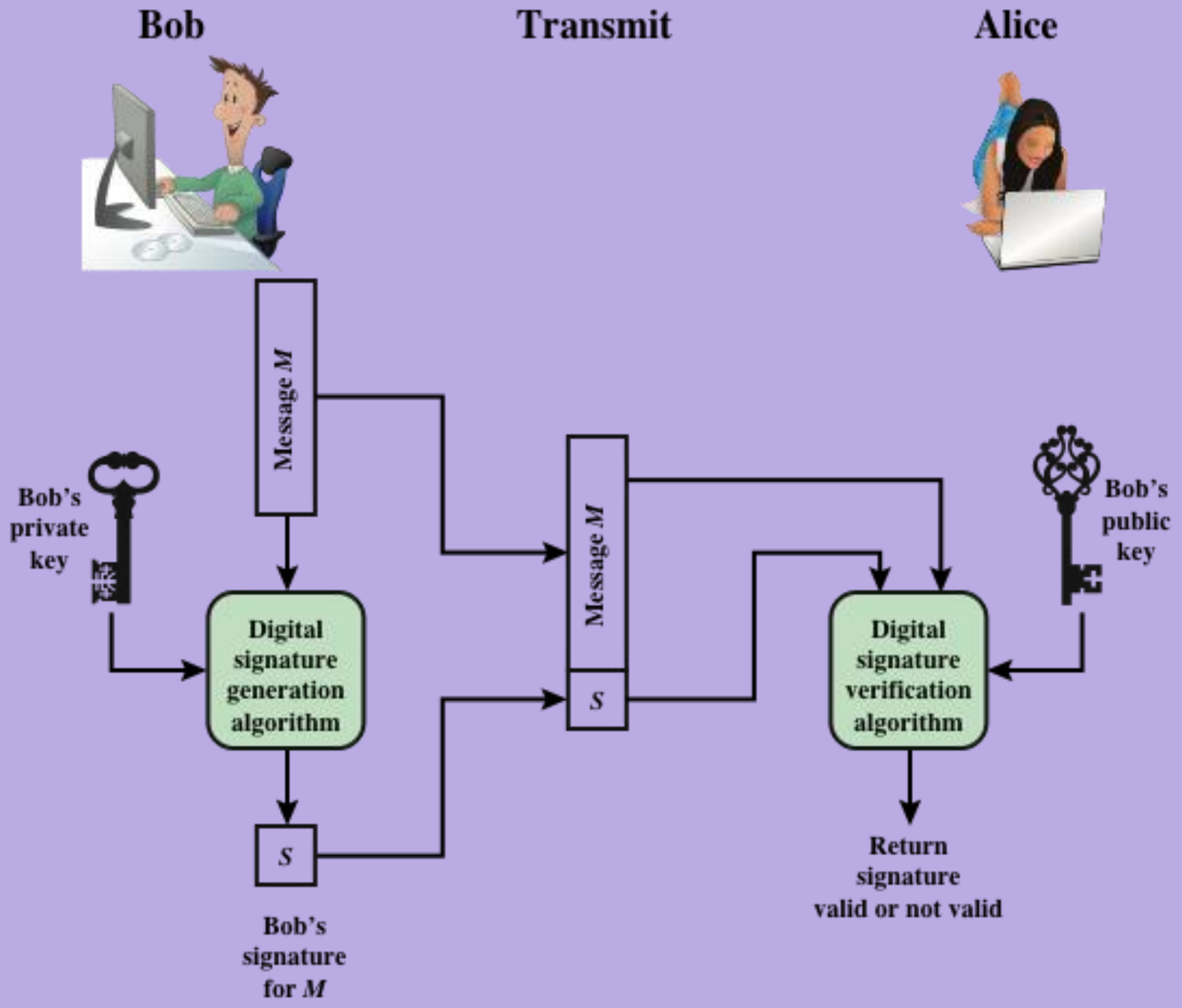

Bob
Transmit
Alice
Message M
Bob's private key
Digital signature generation algorithm
S
Bob's signature for M
Message M
S
Bob's public key
Digital signature verification algorithm
Return signature valid or not valid

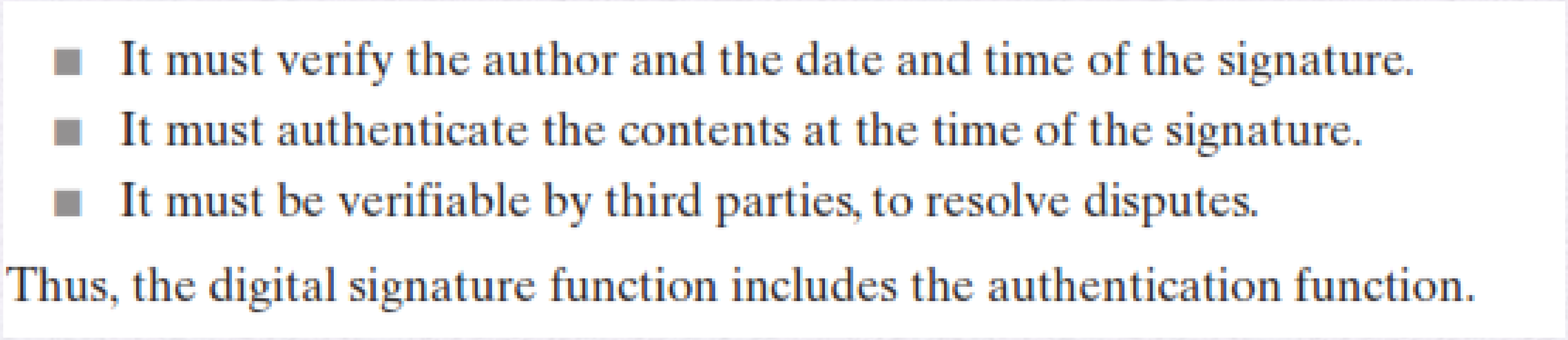
Figure 13.1 Generic Model of Digital Signature Process

- Bob can sign a message using a digital signature generation algorithm. The inputs to the algorithm are the message and Bob’s private key. Any other user, say Alice, can verify the signature using a verification algorithm, whose inputs are the message, the signature, and Bob’s public key.

Contd…

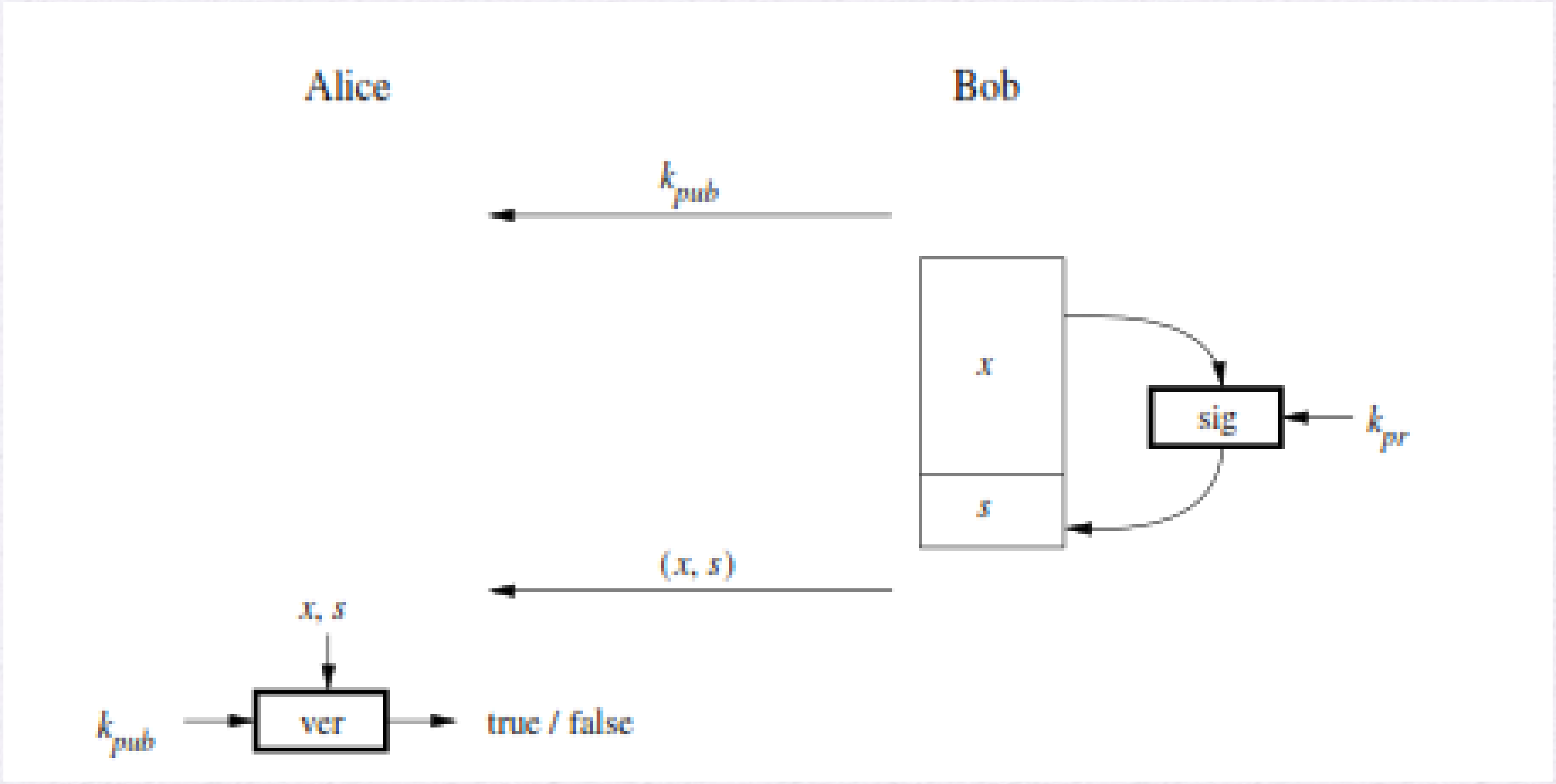
- Digital signatures are one of the most important cryptographic tools they and are widely used today. Applications for digital signatures range from digital certificates for secure e-commerce to legal signing of contracts to secure software updates. Together with key establishment over insecure channels, they form the most important instance for public-key cryptography. Digital signatures share some functionality with handwritten signatures.

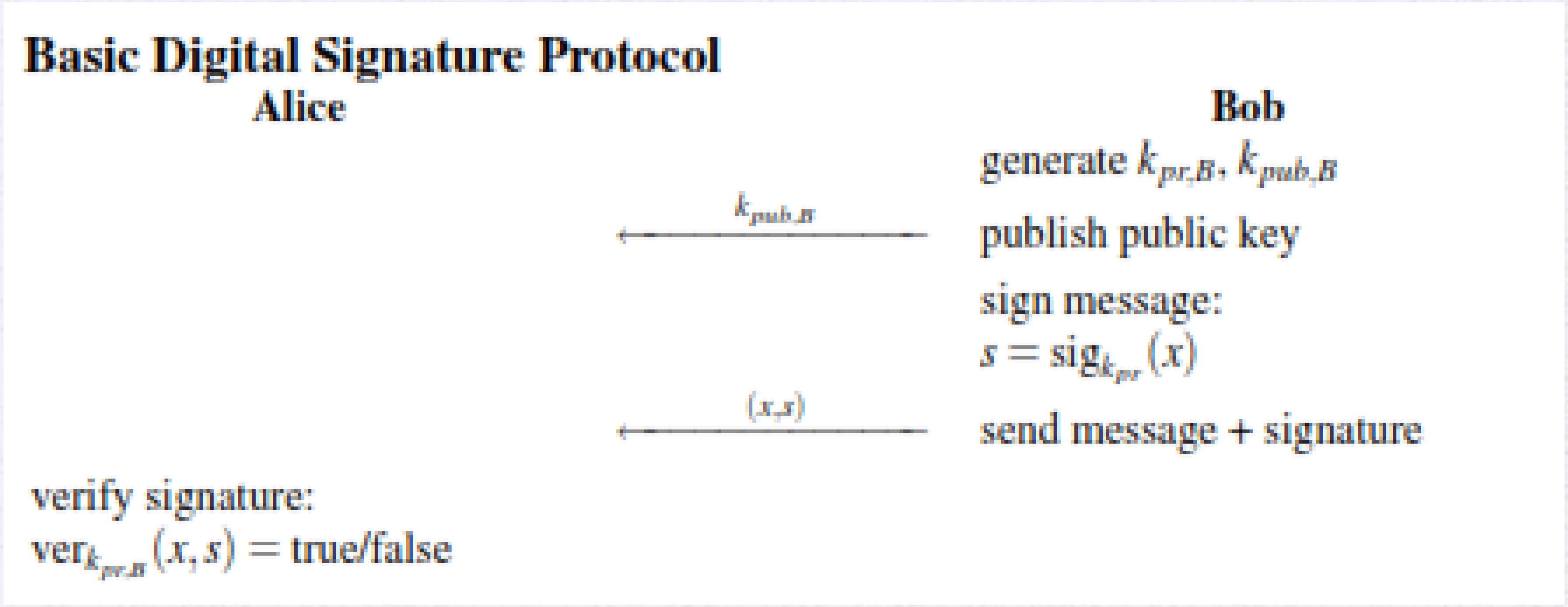
Digital Sign Properties



- It must verify the author and the date and time of the signature.
- It must authenticate the contents at the time of the signature.
- It must be verifiable by third parties, to resolve disputes.

Thus, the digital signature function includes the authentication function.


Alice
Bob
k_{pub}
x
s
sig
k_{pr}
(x, s)
x, s
k_{pub}
ver
true / false



Basic Digital Signature Protocol

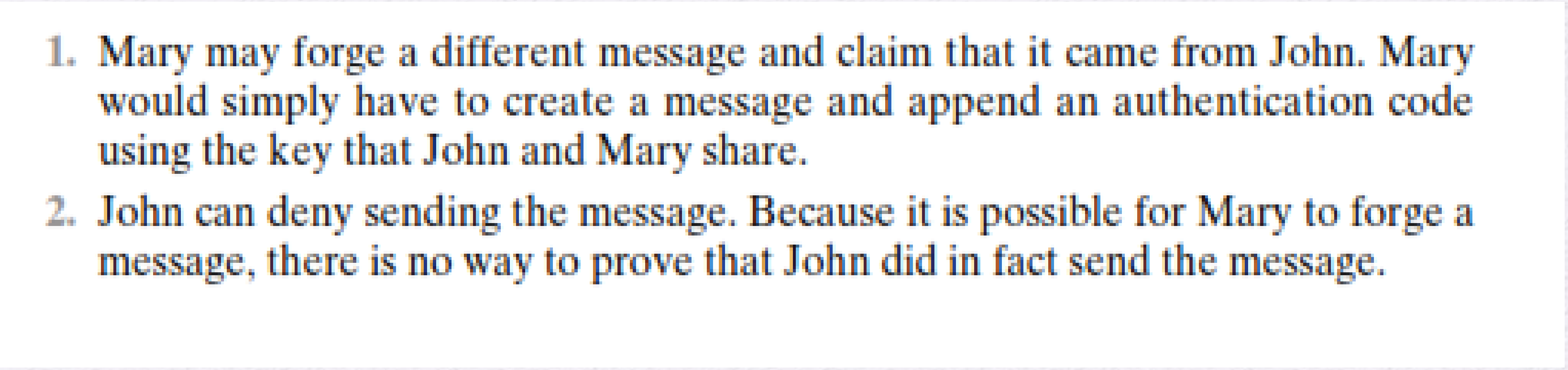
Alice		**Bob**
		generate $k_{pr,B}$, $k_{pub,B}$
	$\xleftarrow{k_{pub,B}}$	publish public key
		sign message: $s = \mathrm{sig}_{k_{pr}}(x)$
	$\xleftarrow{(x,s)}$	send message + signature
verify signature: $\mathrm{ver}_{k_{pr,B}}(x,s) = \text{true/false}$		

Security services

- **Confidentiality: Information is kept secret from all but authorized parties.**
- **Integrity: Messages have not been modified in transit.**
- **Message Authentication: The sender of a message is authentic. An alternative** term is *data origin authentication.*
- **Nonrepudiation: The sender of a message can not deny the creation of the mess**age.

- Identification
- Access control
- Availability
- Auditing

Dispute



1. Mary may forge a different message and claim that it came from John. Mary would simply have to create a message and append an authentication code using the key that John and Mary share.
2. John can deny sending the message. Because it is possible for Mary to forge a message, there is no way to prove that John did in fact send the message.

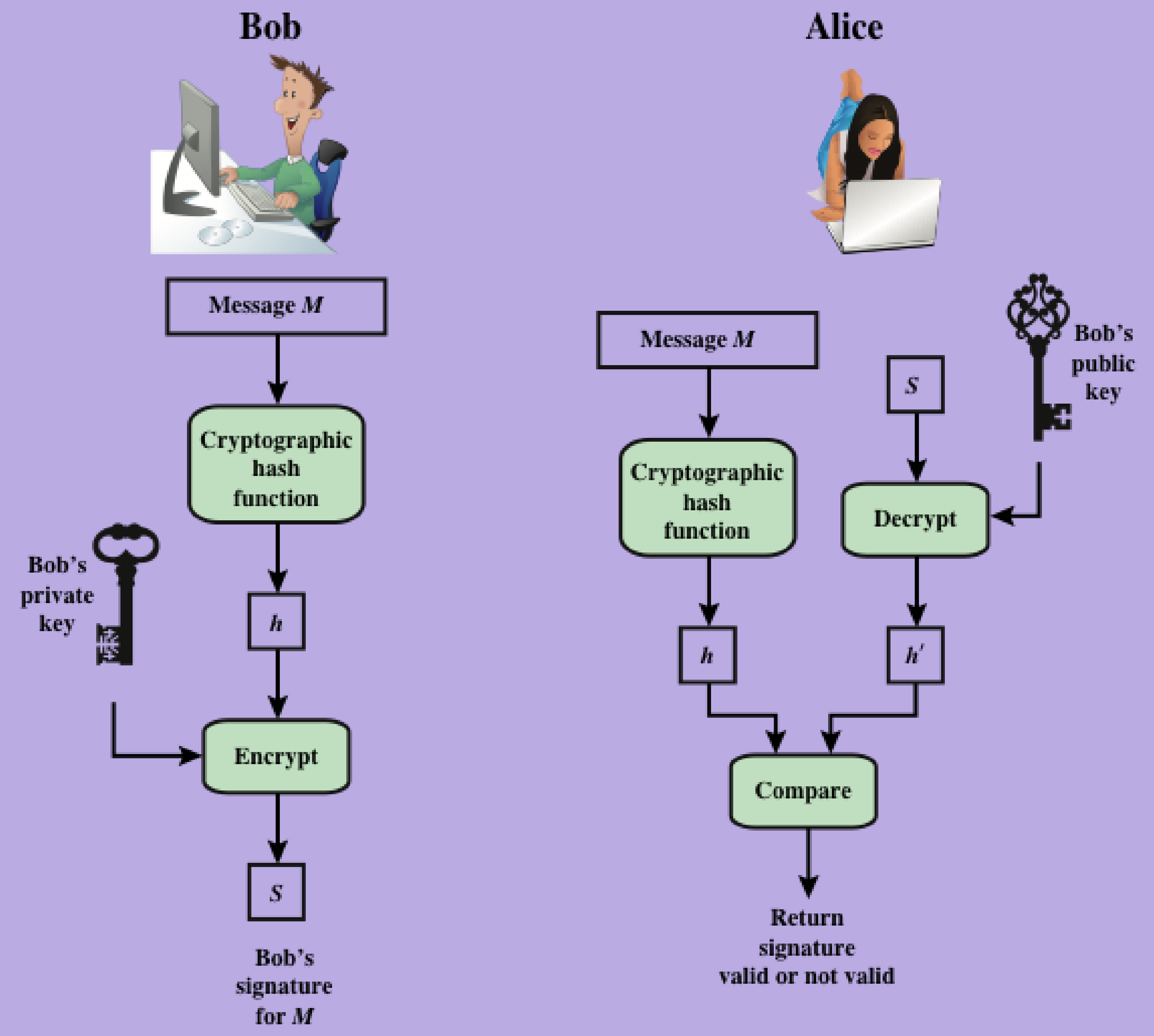


Figure 13.2 Simplified Depiction of Essential Elements of Digital Signature Process

Digital Signature Properties

It must verify the author and the date and time of the signature

It must authenticate the contents at the time of the signature

It must be verifiable by third parties, to resolve disputes

Attacks

- A denotes the user whose signature method is being attacked, and C denotes the attacker.
- Key-only attack: C only knows A’s public key.
- Known message attack: C is given access to a set of messages and their signatures.

- Generic chosen message attack: C chooses a list of messages before attempting to breaks A’s signature scheme, independent of A’s public key. C then obtains from A valid signatures for the chosen messages. The attack is generic, because it does not depend on A’s public key; the same attack is used against everyone.

- Directed chosen message attack: Similar to the generic attack, except that the list of messages to be signed is chosen after C knows A’s public key but before any signatures are seen.

- Adaptive chosen message attack: C is allowed to use A as an “oracle.” This means that C may request from A signatures of messages that depend on previously obtained message-signature pairs.

Attacks

Key-only attack

- C only knows A's public key

Known message attack

- C is given access to a set of messages and their signatures

Generic chosen message attack

- C chooses a list of messages before attempting to break A's signature scheme, independent of A's public key; C then obtains from A valid signatures for the chosen messages

Directed chosen message attack

- Similar to the generic attack, except that the list of messages to be signed is chosen after C knows A's public key but before any signatures are seen

Adaptive chosen message attack

- C may request from A signatures of messages that depend on previously obtained message-signature pairs

Forgeries

Total break

- C determines A's private key

Universal forgery

- C finds an efficient signing algorithm that provides an equivalent way of constructing signatures on arbitrary messages

Selective forgery

- C forges a signature for a particular message chosen by C

Existential forgery

- C forges a signature for at least one message; C has no control over the message

Example

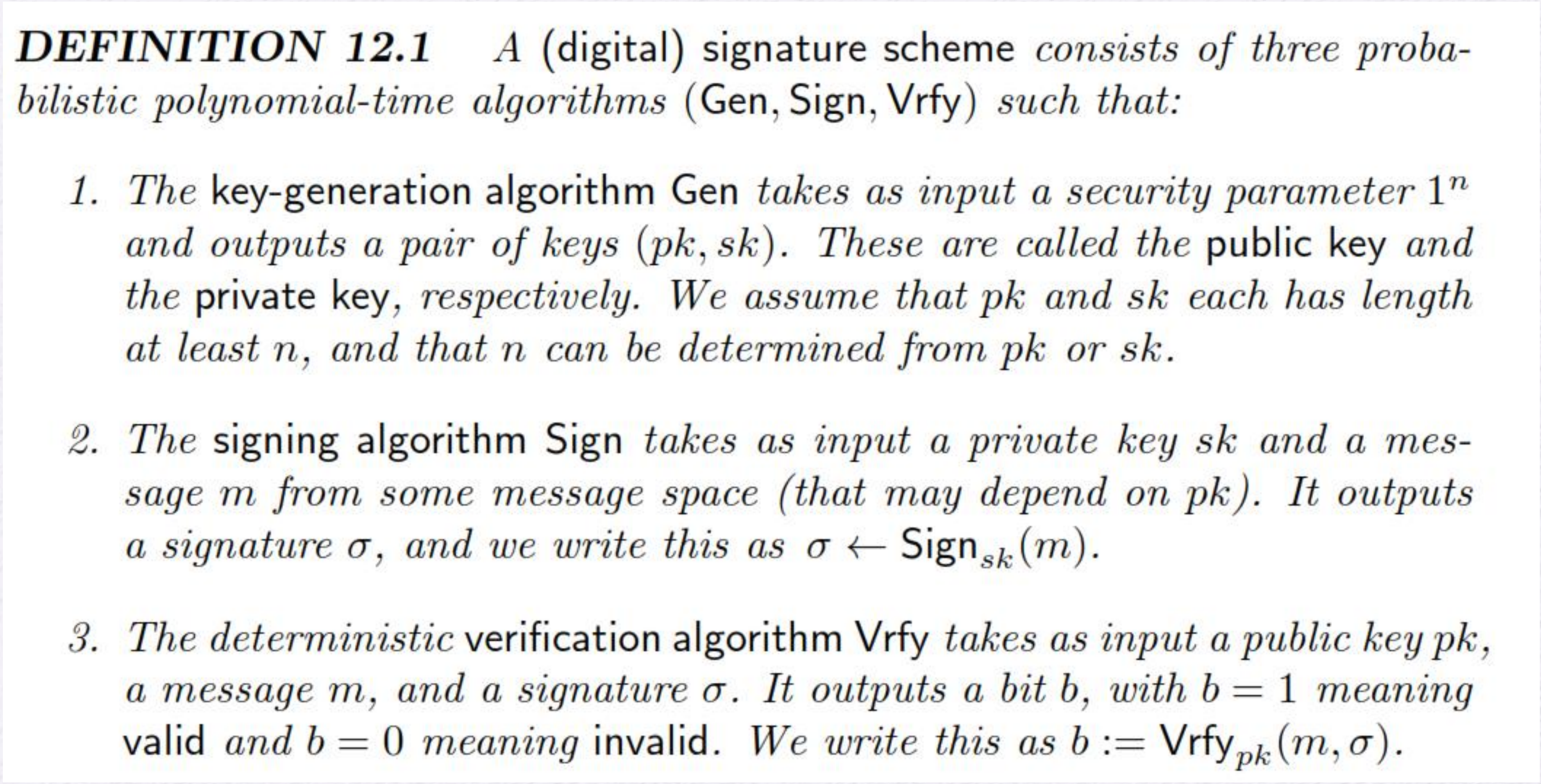
- As a prototypical application, consider a software company that wants to disseminate software updates in an authenticated manner; that is, when the company releases an update it should be possible for any of its clients to verify that the update is authentic, and a malicious third party should never be able to fool a client into accepting an update that was not actually released by the company.

- In the current example, pk could be bundled with the original software purchased by a client.) When releasing a software update m, the company computes a digital signature σ on m using its private key sk, and sends (m, σ) to every client. Each client can verify the authenticity of m by checking that σ is a correct signature on m with respect to the public key pk.

- Both message authentication codes and digital signature schemes are used to ensure the **integrity** of transmitted messages.
- Using digital signatures rather than message authentication codes simplifies key distribution and management, especially when a sender needs to communicate with multiple receivers.

- A qualitative advantage that digital signatures have as compared to message authentication codes is that signatures are **publicly verifiable**.
- Digital signature schemes also provide the very important property of **nonrepudiation**.

- Digital signatures are the public-key counterpart of message authentication codes, and their syntax and security guarantees are analogous. The algorithm that the sender applies to a message is here denoted Sign (rather than Mac), and the output of this algorithm is now called a signature (not tag)



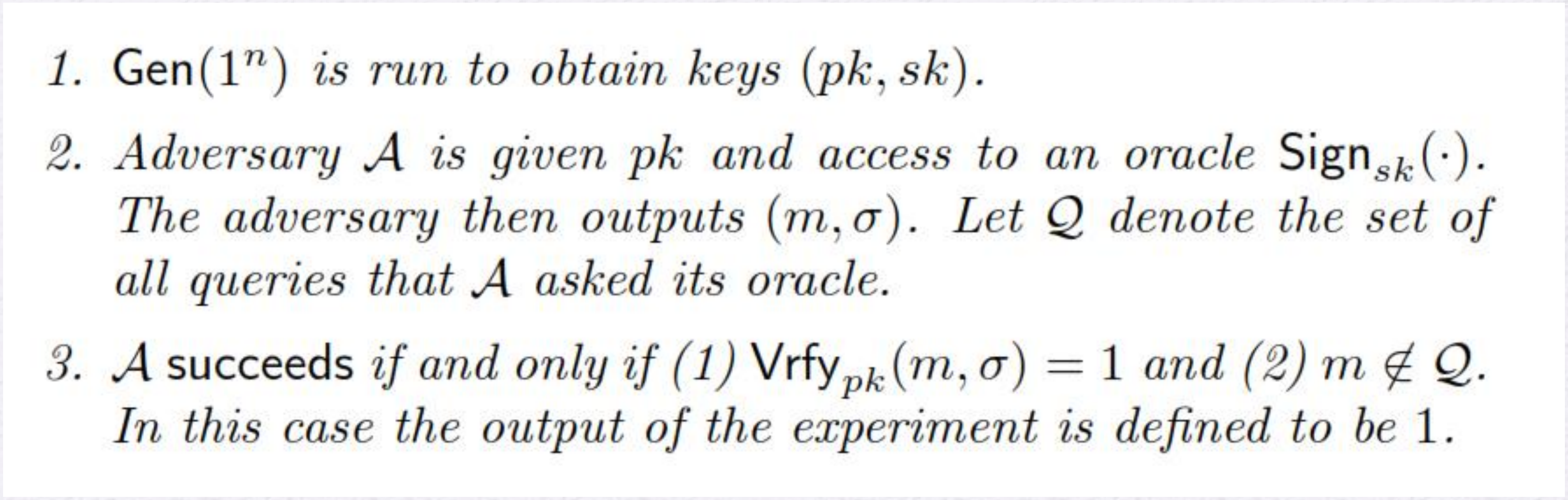
DEFINITION 12.1 *A* (digital) signature scheme *consists of three probabilistic polynomial-time algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *such that:*

1. *The* key-generation algorithm Gen *takes as input a security parameter* 1^n *and outputs a pair of keys* (pk, sk). *These are called the* public key *and the* private key, *respectively. We assume that* pk *and* sk *each has length at least* n, *and that* n *can be determined from* pk *or* sk.

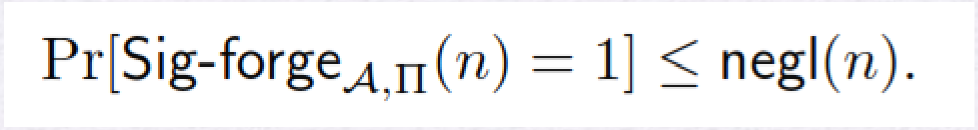
2. *The* signing algorithm Sign *takes as input a private key* sk *and a message* m *from some message space (that may depend on* pk*). It outputs a signature* σ, *and we write this as* $\sigma \leftarrow \mathsf{Sign}_{sk}(m)$.

3. *The deterministic* verification algorithm Vrfy *takes as input a public key* pk, *a message* m, *and a signature* σ. *It outputs a bit* b, *with* $b = 1$ *meaning* valid *and* $b = 0$ *meaning* invalid. *We write this as* $b := \mathsf{Vrfy}_{pk}(m, \sigma)$.

Adversary Model

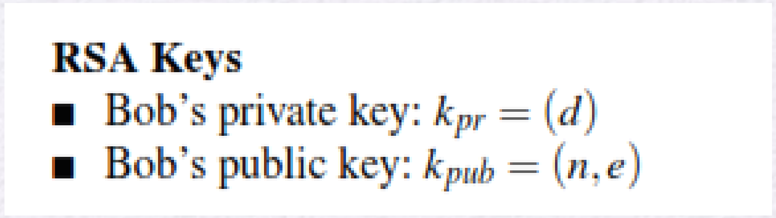


1. $\mathsf{Gen}(1^n)$ *is run to obtain keys* (pk, sk).
2. *Adversary* $\mathcal{A}$ *is given* pk *and access to an oracle* $\mathsf{Sign}_{sk}(\cdot)$. *The adversary then outputs* (m, σ). *Let* $\mathcal{Q}$ *denote the set of all queries that* $\mathcal{A}$ *asked its oracle.*
3. $\mathcal{A}$ succeeds *if and only if (1)* $\mathsf{Vrfy}_{pk}(m, \sigma) = 1$ *and (2)* $m \notin \mathcal{Q}$. *In this case the output of the experiment is defined to be* 1.



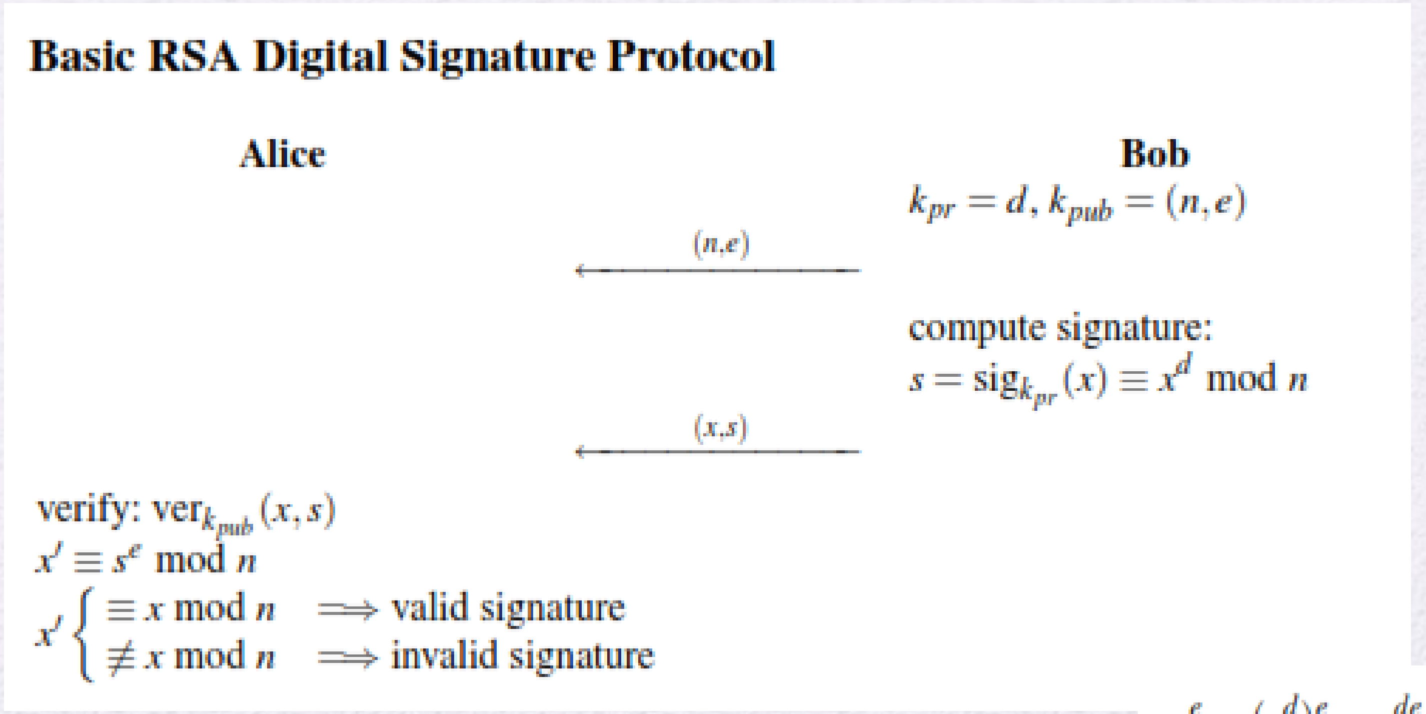
$$\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] \leq \mathsf{negl}(n).$$

RSA Digital Signature



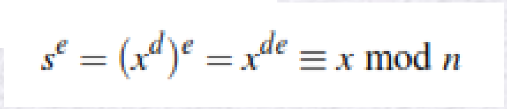
RSA Keys

- Bob's private key: $k_{pr} = (d)$
- Bob's public key: $k_{pub} = (n, e)$



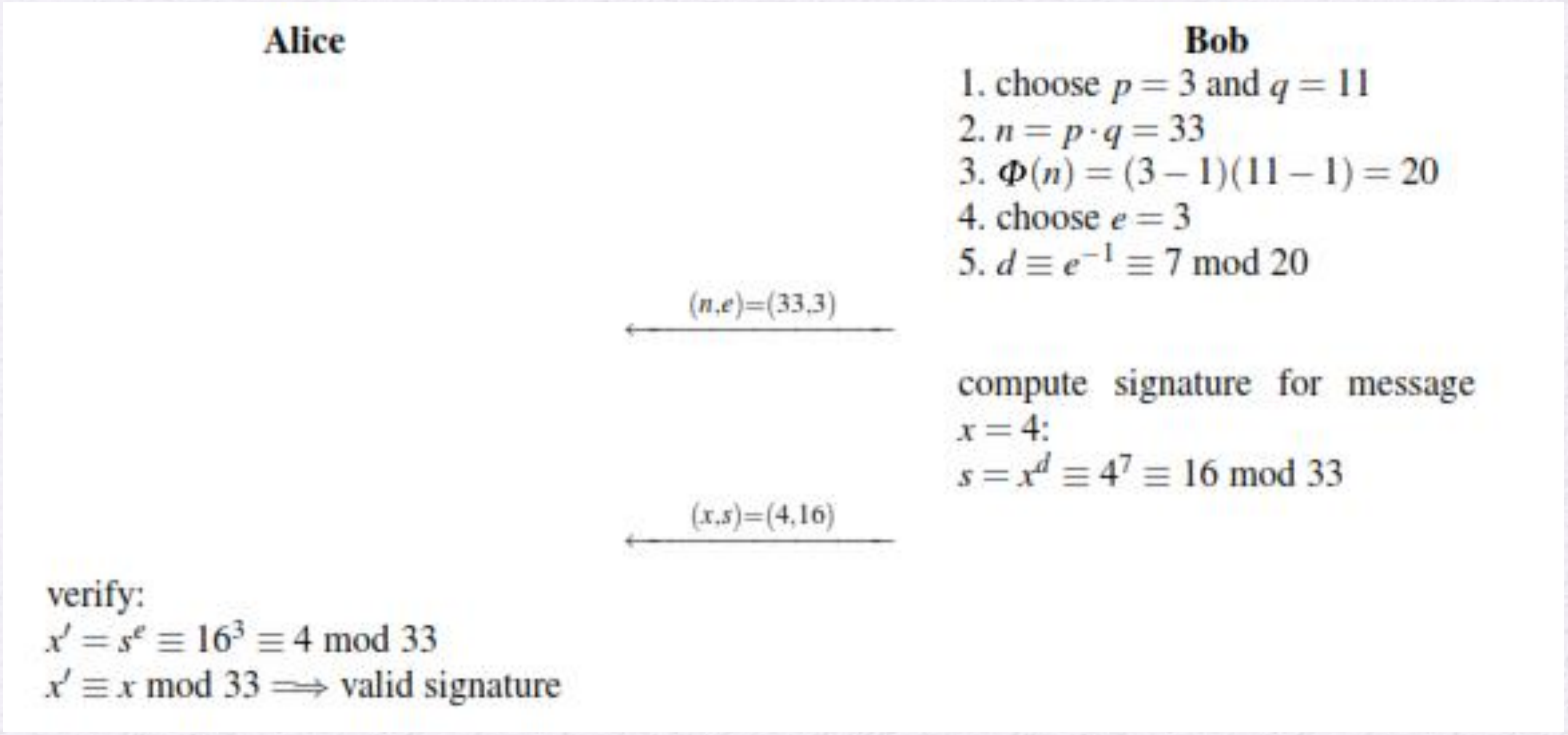
Basic RSA Digital Signature Protocol

Alice		**Bob**
		$k_{pr} = d,\ k_{pub} = (n,e)$
	$\xleftarrow{(n,e)}$	
		compute signature: $s = \mathrm{sig}_{k_{pr}}(x) \equiv x^d \bmod n$
	$\xleftarrow{(x,s)}$	
verify: $\mathrm{ver}_{k_{pub}}(x,s)$		
$x' \equiv s^e \bmod n$		
$x' \begin{cases} \equiv x \bmod n & \Longrightarrow \text{valid signature} \\ \not\equiv x \bmod n & \Longrightarrow \text{invalid signature} \end{cases}$		



$$s^e = (x^d)^e = x^{de} \equiv x \bmod n$$

Example



Alice | **Bob**

Bob

1. choose $p = 3$ and $q = 11$
2. $n = p \cdot q = 33$
3. $\Phi(n) = (3-1)(11-1) = 20$
4. choose $e = 3$
5. $d \equiv e^{-1} \equiv 7 \bmod 20$

$$\xleftarrow{(n,e)=(33,3)}$$

compute signature for message $x = 4$:

$s = x^d \equiv 4^7 \equiv 16 \bmod 33$

$$\xleftarrow{(x,s)=(4,16)}$$

Alice

verify:

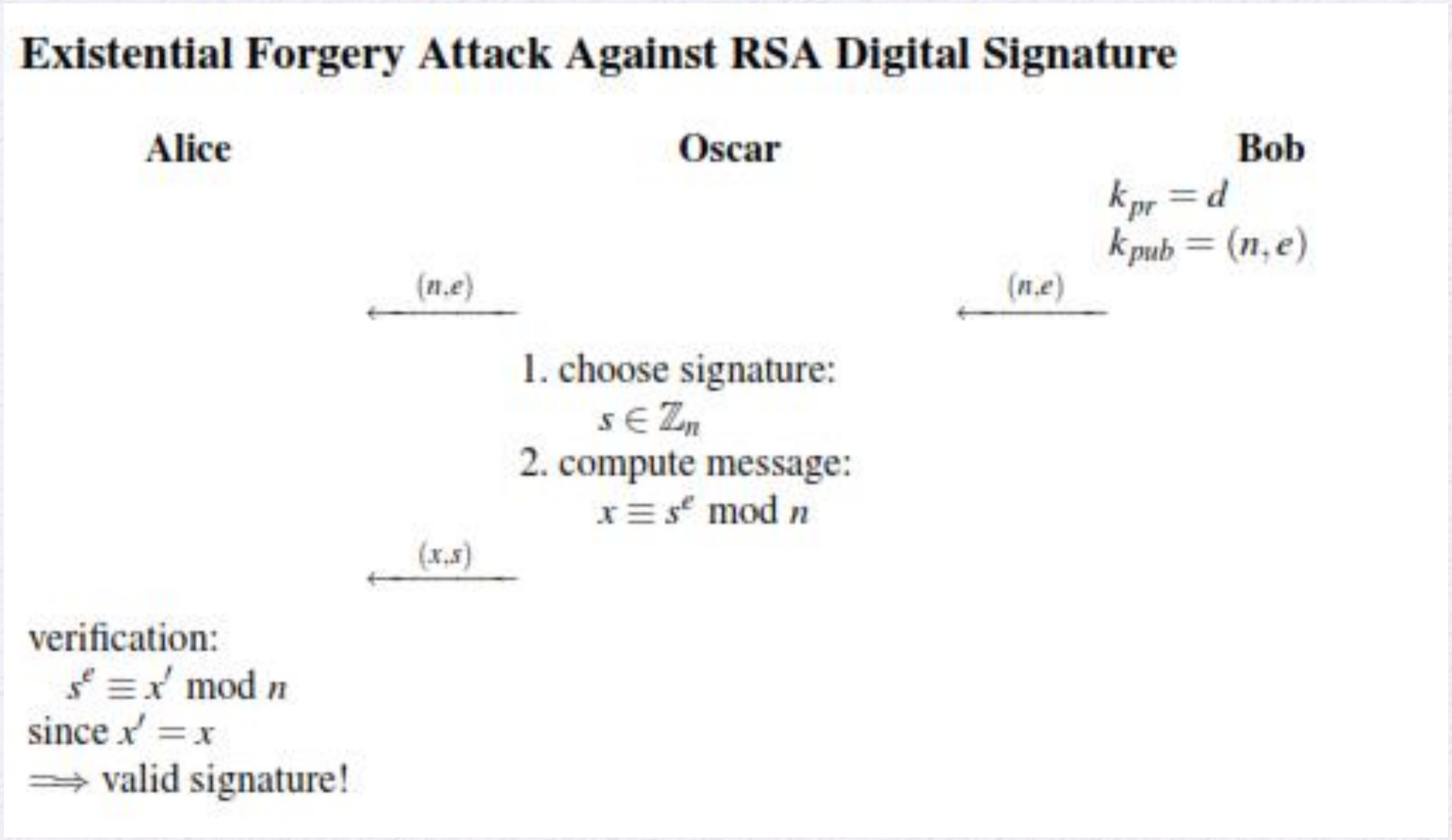
$x' = s^e \equiv 16^3 \equiv 4 \bmod 33$

$x' \equiv x \bmod 33 \Longrightarrow$ valid signature

Contd…

- It should be noted that we introduced a digital signature scheme only. In particular, the message itself is not encrypted and, thus, there is not confidentiality. If this security service is required, the message together with the signature should be encrypted, e.g., using a symmetric algorithm like AES.

Forgery by Oscar



Existential Forgery Attack Against RSA Digital Signature

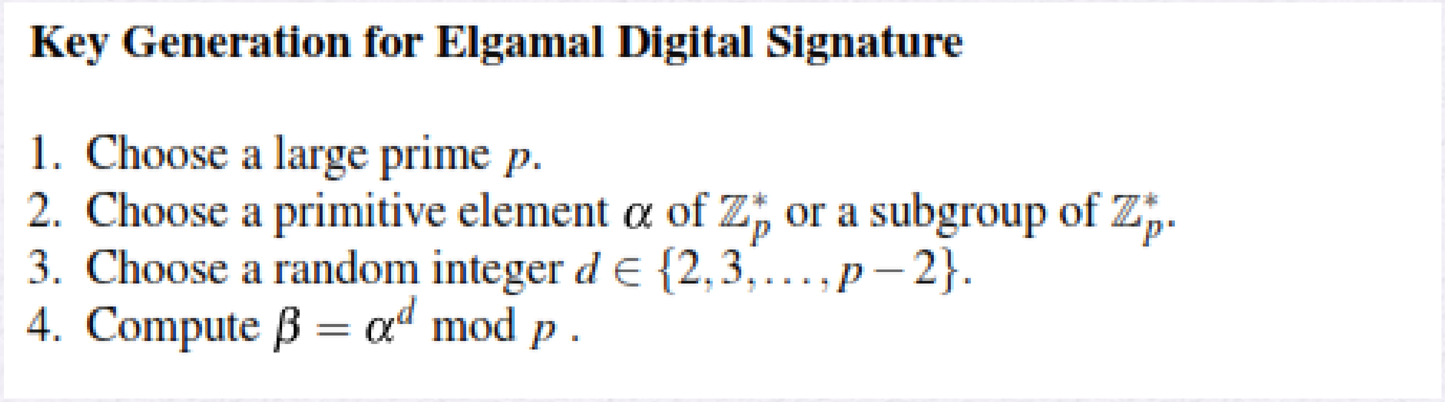
Alice	Oscar	Bob
		$k_{pr} = d$ $k_{pub} = (n, e)$
$\xleftarrow{(n,e)}$	$\xleftarrow{(n,e)}$	
	1. choose signature: $s \in \mathbb{Z}_n$ 2. compute message: $x \equiv s^e \bmod n$	
$\xleftarrow{(x,s)}$		
verification: $s^e \equiv x' \bmod n$ since $x' = x$ $\Longrightarrow$ valid signature!		

Elgamal DSS

- The Elgamal signature scheme, which was published in 1985, is based on the difficulty of computing discrete logarithms. Unlike RSA, where encryption and digital signature are almost identical operations,
- the Elgamal digital signature is quite different from the encryption scheme with the same name.

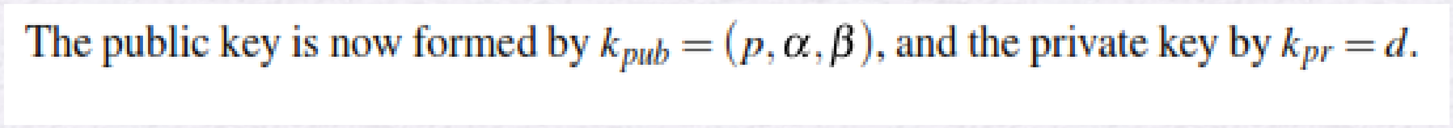
- Setup – KeyGeneration
- **Signature and**
- **Verification**

Key Generation



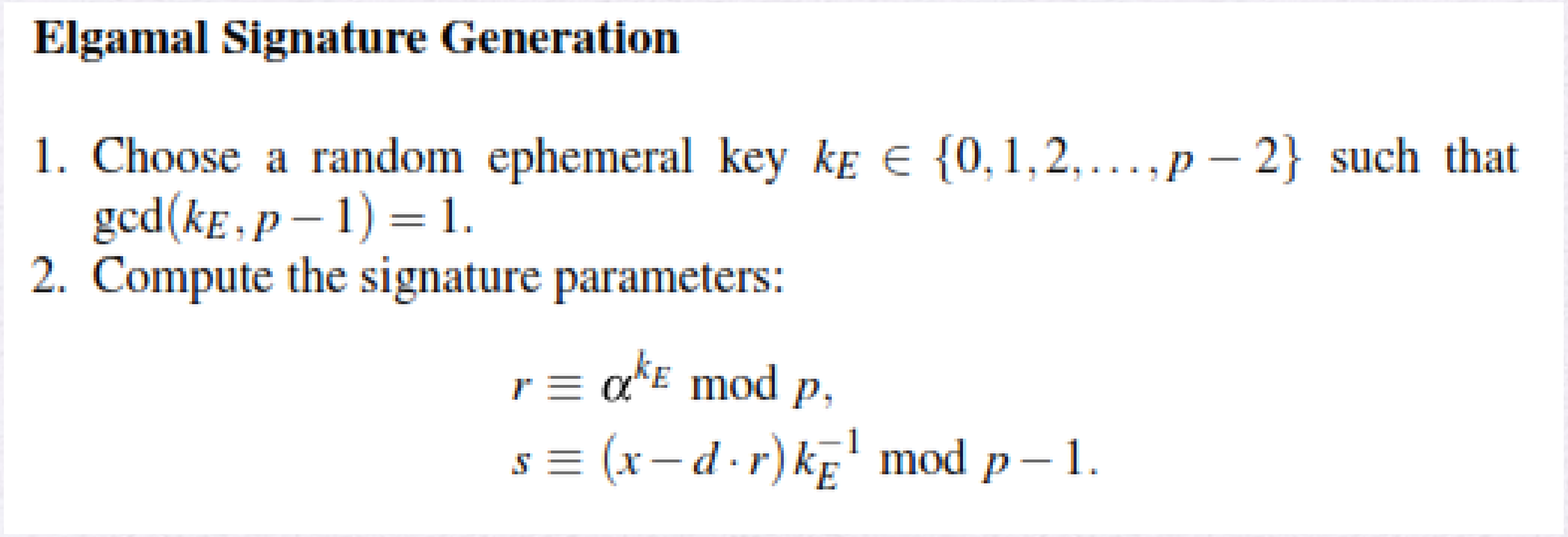
Key Generation for Elgamal Digital Signature

1. Choose a large prime p.
2. Choose a primitive element α of $\mathbb{Z}_p^*$ or a subgroup of $\mathbb{Z}_p^*$.
3. Choose a random integer $d \in \{2, 3, \ldots, p-2\}$.
4. Compute $\beta = \alpha^d \bmod p$.



The public key is now formed by $k_{pub} = (p, \alpha, \beta)$, and the private key by $k_{pr} = d$.

Signature Generation (r,s)

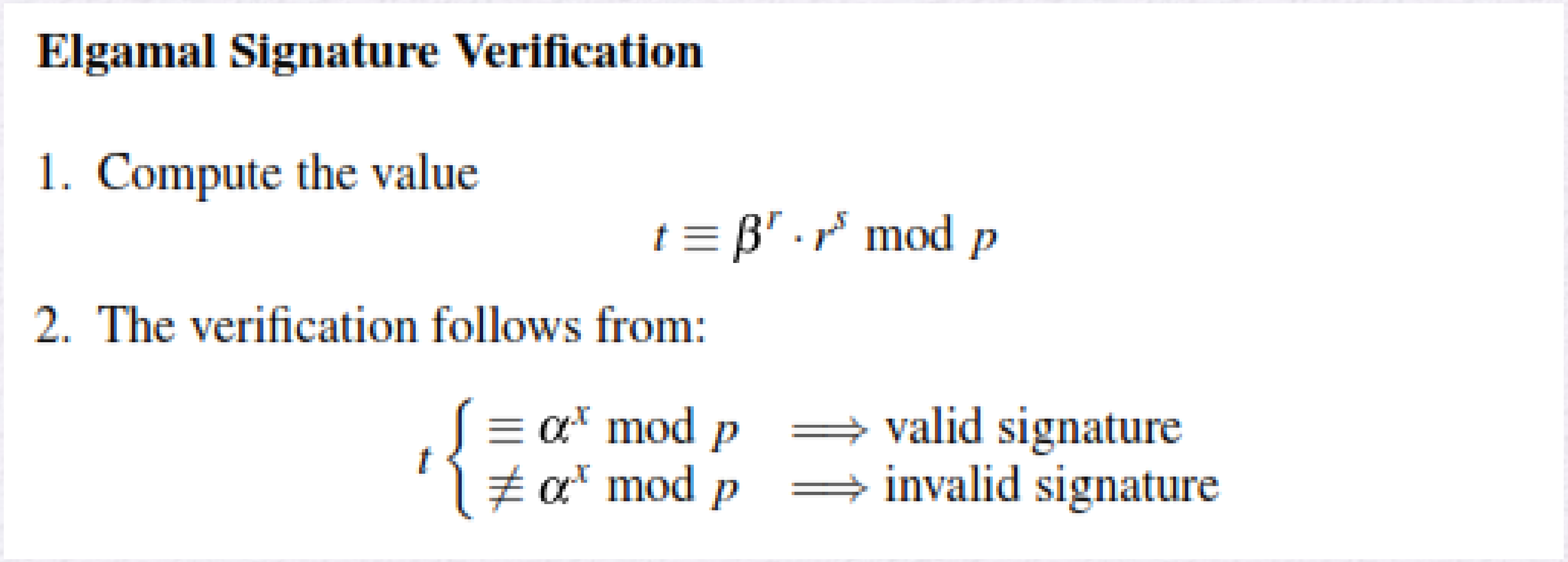


Elgamal Signature Generation

1. Choose a random ephemeral key $k_E \in \{0,1,2,\ldots,p-2\}$ such that $\gcd(k_E, p-1) = 1$.
2. Compute the signature parameters:

$$r \equiv \alpha^{k_E} \bmod p,$$
$$s \equiv (x - d \cdot r) k_E^{-1} \bmod p-1.$$

Verification



Elgamal Signature Verification

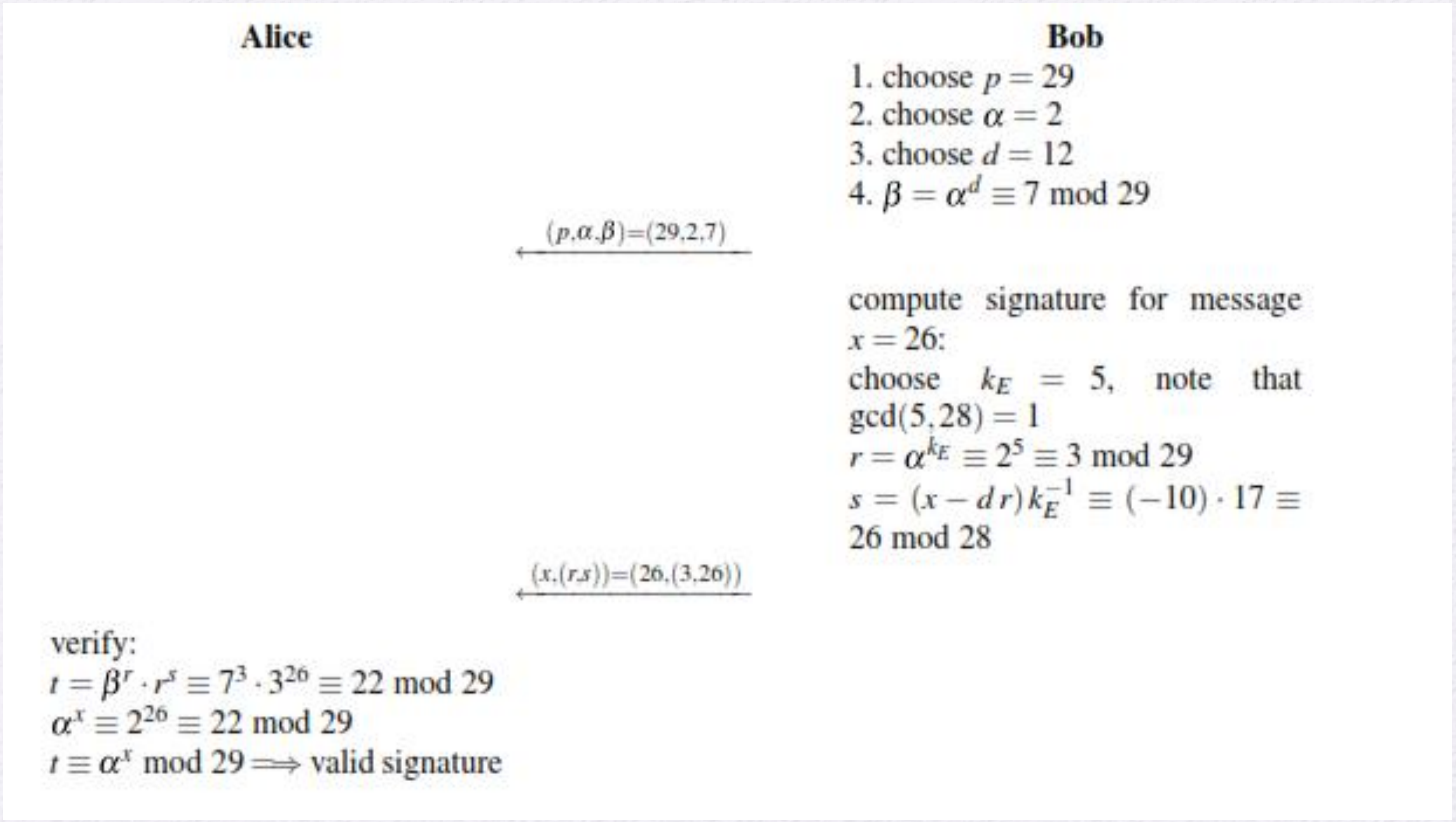
1. Compute the value
$$t \equiv \beta^r \cdot r^s \bmod p$$
2. The verification follows from:
$$t \begin{cases} \equiv \alpha^x \bmod p & \Longrightarrow \text{valid signature} \\ \not\equiv \alpha^x \bmod p & \Longrightarrow \text{invalid signature} \end{cases}$$

Proof

$$\begin{aligned}\beta^r \cdot r^s &\equiv (\alpha^d)^r(\alpha^{k_E})^s \bmod p \\ &\equiv \alpha^{d\,r+k_E\,s} \bmod p.\end{aligned}$$

$$\alpha^x \equiv \alpha^{d\,r+k_E\,s} \bmod p.$$

Example



Alice

Bob

1. choose $p = 29$
2. choose $\alpha = 2$
3. choose $d = 12$
4. $\beta = \alpha^d \equiv 7 \bmod 29$

$$\xleftarrow{(p,\alpha,\beta)=(29,2,7)}$$

compute signature for message $x = 26$:

choose $k_E = 5$, note that $\gcd(5,28) = 1$

$r = \alpha^{k_E} \equiv 2^5 \equiv 3 \bmod 29$

$s = (x - d\,r)\,k_E^{-1} \equiv (-10) \cdot 17 \equiv 26 \bmod 28$

$$\xleftarrow{(x,(r,s))=(26,(3,26))}$$

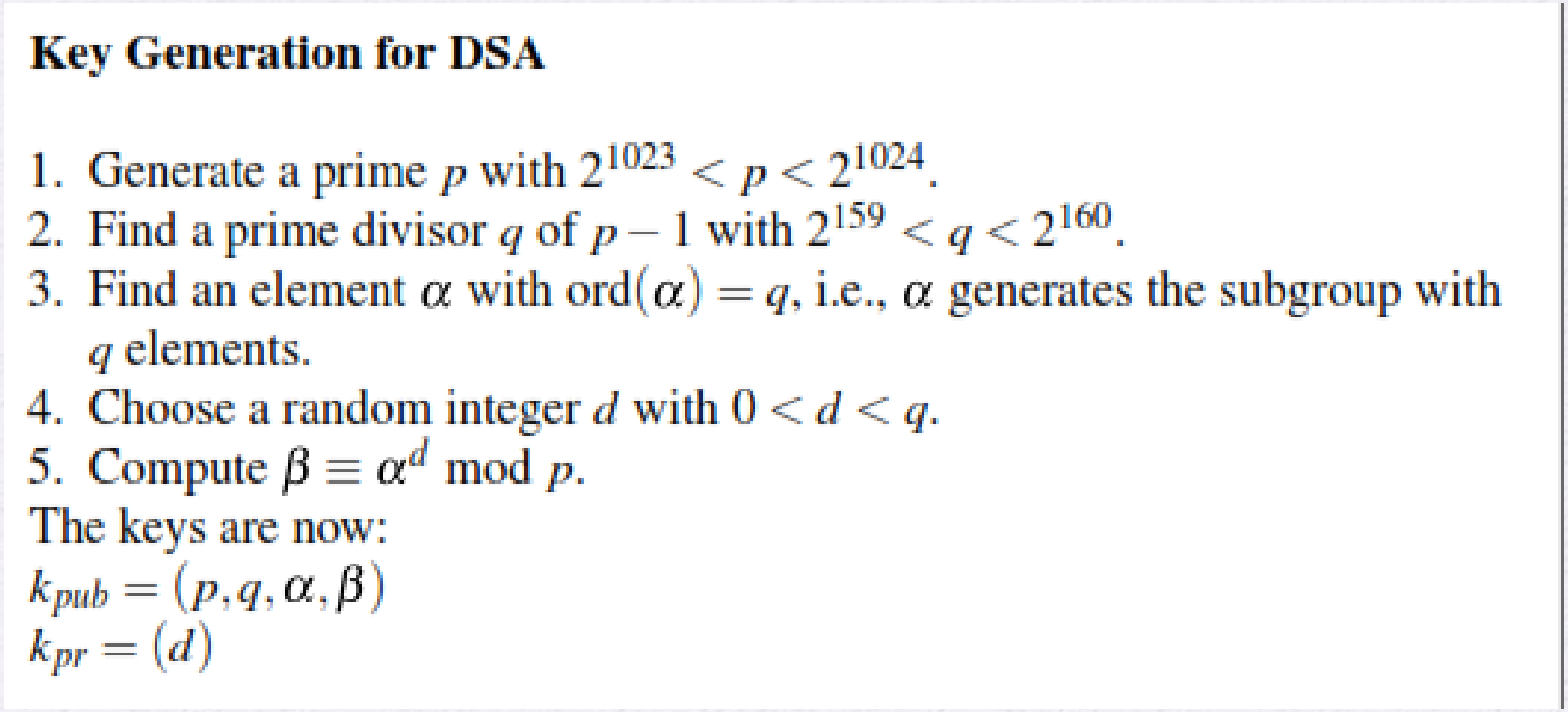
verify:

$t = \beta^r \cdot r^s \equiv 7^3 \cdot 3^{26} \equiv 22 \bmod 29$

$\alpha^x \equiv 2^{26} \equiv 22 \bmod 29$

$t \equiv \alpha^x \bmod 29 \Longrightarrow$ valid signature

DSA 1024 bits



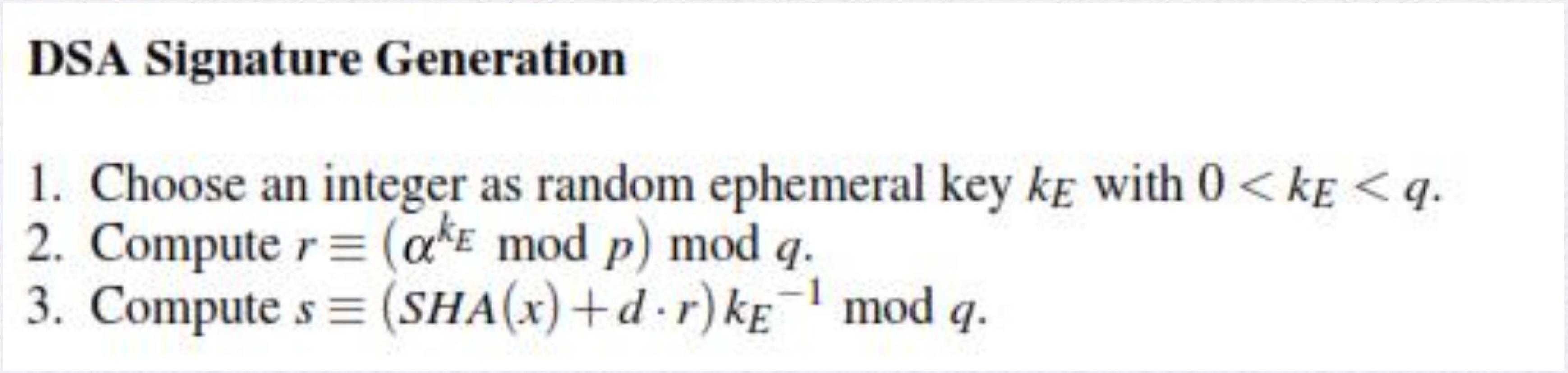
Key Generation for DSA

1. Generate a prime p with $2^{1023} < p < 2^{1024}$.
2. Find a prime divisor q of $p-1$ with $2^{159} < q < 2^{160}$.
3. Find an element α with $\text{ord}(\alpha) = q$, i.e., α generates the subgroup with q elements.
4. Choose a random integer d with $0 < d < q$.
5. Compute $\beta \equiv \alpha^d \bmod p$.

The keys are now:
$k_{pub} = (p, q, \alpha, \beta)$
$k_{pr} = (d)$

- In addition to the 1024-bit prime *p and a 160-bit prime q, there are two other bit* length combinations possible for the primes *p and q.*
- This set-up yields shorter signatures

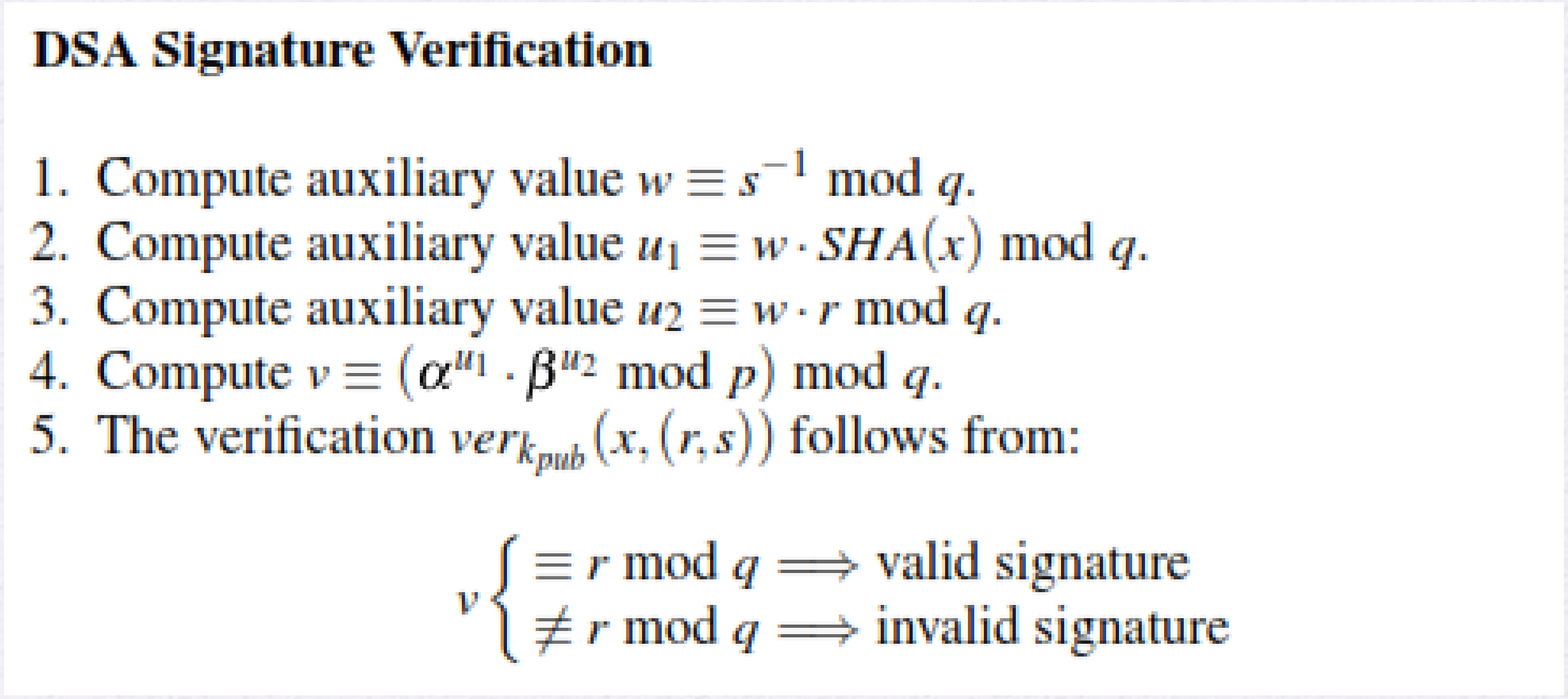
Sign generation



DSA Signature Generation

1. Choose an integer as random ephemeral key k_E with $0 < k_E < q$.
2. Compute $r \equiv (\alpha^{k_E} \bmod p) \bmod q$.
3. Compute $s \equiv (SHA(x) + d \cdot r)\, k_E^{-1} \bmod q$.

Verification

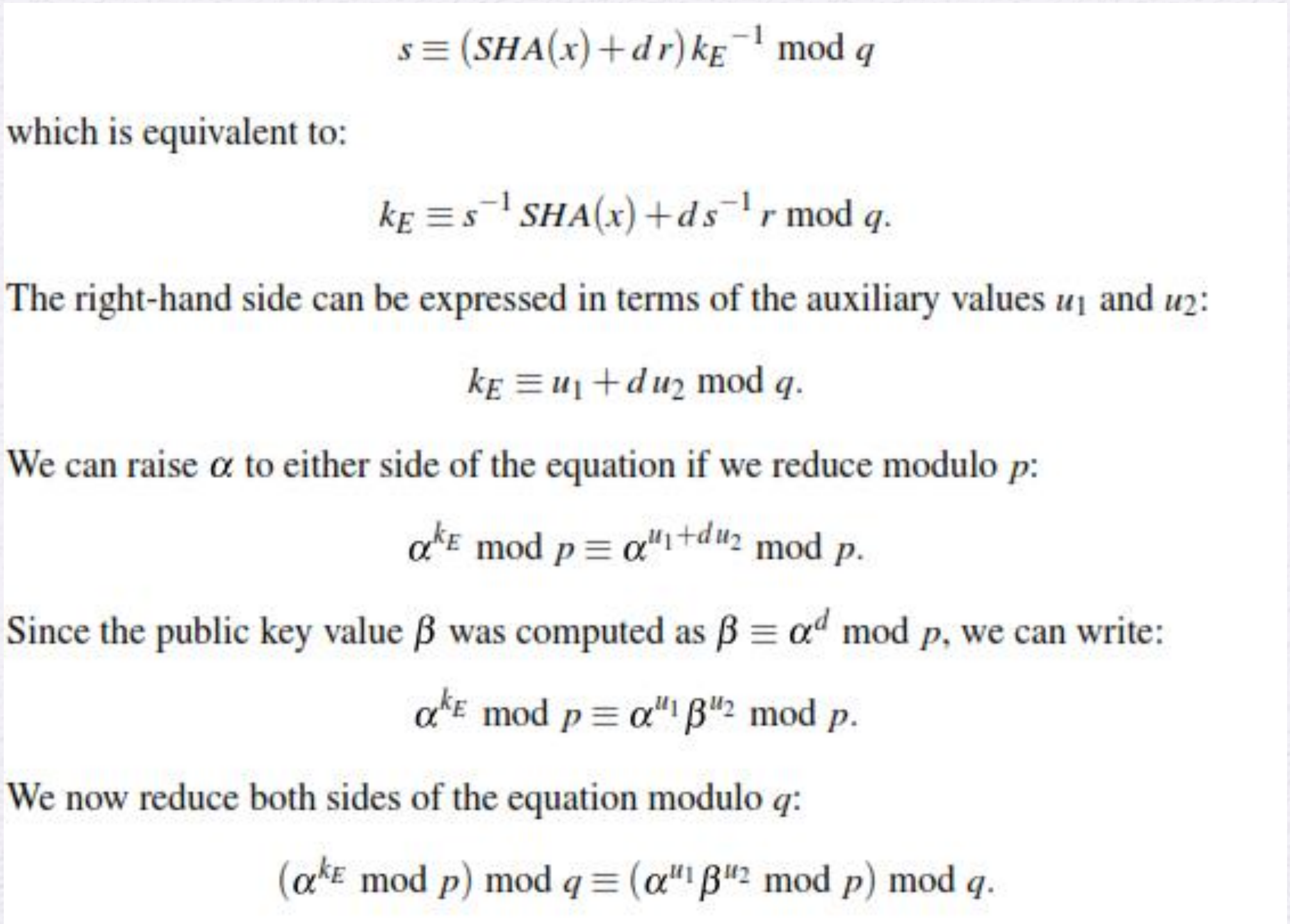


DSA Signature Verification

1. Compute auxiliary value $w \equiv s^{-1} \bmod q$.
2. Compute auxiliary value $u_1 \equiv w \cdot SHA(x) \bmod q$.
3. Compute auxiliary value $u_2 \equiv w \cdot r \bmod q$.
4. Compute $v \equiv (\alpha^{u_1} \cdot \beta^{u_2} \bmod p) \bmod q$.
5. The verification $ver_{k_{pub}}(x,(r,s))$ follows from:

$$v \begin{cases} \equiv r \bmod q \Longrightarrow \text{valid signature} \\ \not\equiv r \bmod q \Longrightarrow \text{invalid signature} \end{cases}$$

Proof



$$s \equiv (SHA(x) + d\,r)\,k_E^{-1} \bmod q$$

which is equivalent to:

$$k_E \equiv s^{-1}\,SHA(x) + d\,s^{-1}\,r \bmod q.$$

The right-hand side can be expressed in terms of the auxiliary values u_1 and u_2:

$$k_E \equiv u_1 + d\,u_2 \bmod q.$$

We can raise α to either side of the equation if we reduce modulo p:

$$\alpha^{k_E} \bmod p \equiv \alpha^{u_1 + d\,u_2} \bmod p.$$

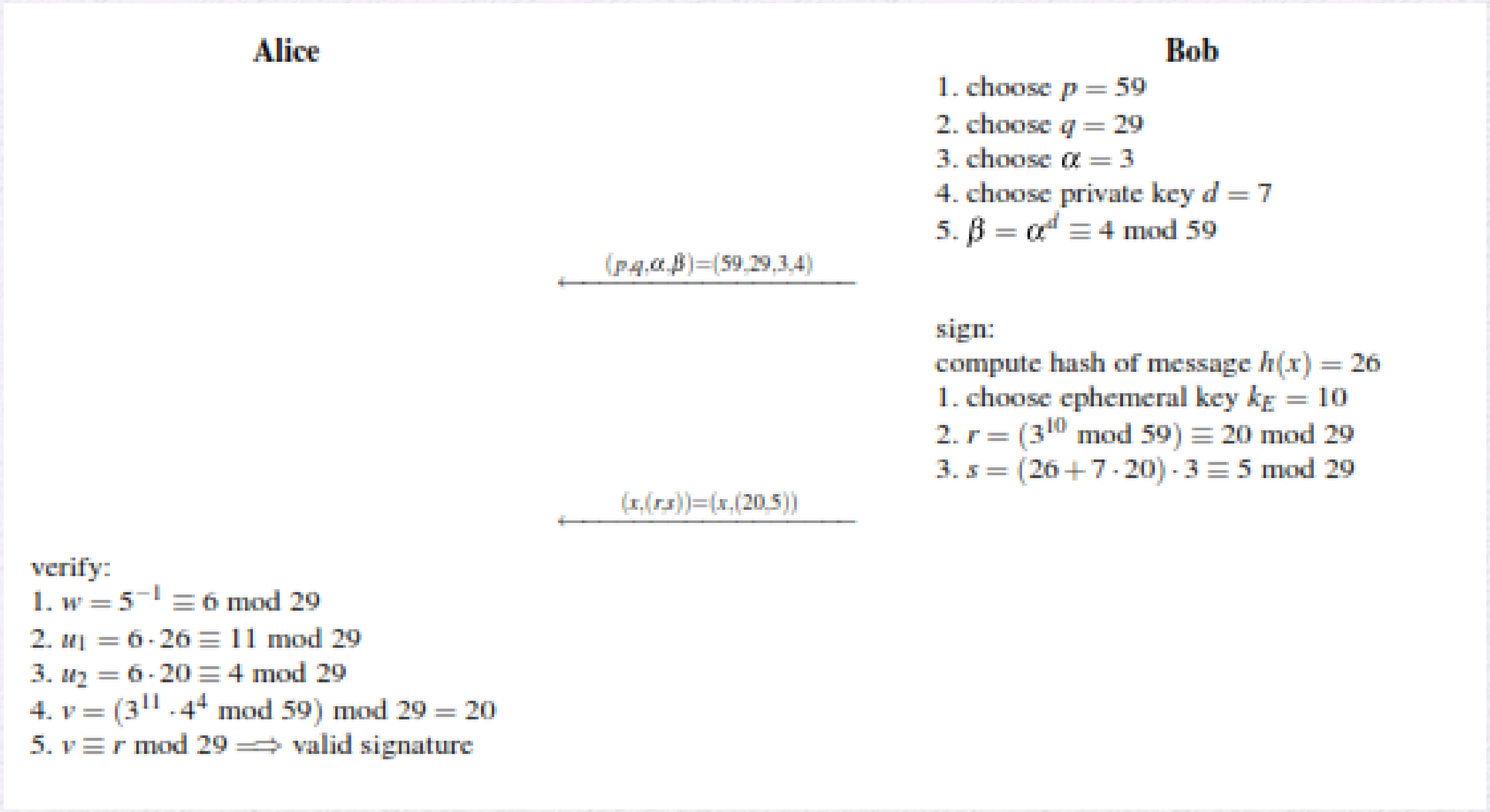
Since the public key value β was computed as $\beta \equiv \alpha^d \bmod p$, we can write:

$$\alpha^{k_E} \bmod p \equiv \alpha^{u_1}\beta^{u_2} \bmod p.$$

We now reduce both sides of the equation modulo q:

$$(\alpha^{k_E} \bmod p) \bmod q \equiv (\alpha^{u_1}\beta^{u_2} \bmod p) \bmod q.$$

Example



Alice | **Bob**

Bob:

1. choose $p = 59$
2. choose $q = 29$
3. choose $\alpha = 3$
4. choose private key $d = 7$
5. $\beta = \alpha^d \equiv 4 \bmod 59$

$$\xleftarrow{(p,q,\alpha,\beta)=(59,29,3,4)}$$

sign:
compute hash of message $h(x) = 26$
1. choose ephemeral key $k_E = 10$
2. $r = (3^{10} \bmod 59) \equiv 20 \bmod 29$
3. $s = (26 + 7 \cdot 20) \cdot 3 \equiv 5 \bmod 29$

$$\xleftarrow{(x,(r,s))=(x,(20,5))}$$

Alice:

verify:
1. $w = 5^{-1} \equiv 6 \bmod 29$
2. $u_1 = 6 \cdot 26 \equiv 11 \bmod 29$
3. $u_2 = 6 \cdot 20 \equiv 4 \bmod 29$
4. $v = (3^{11} \cdot 4^4 \bmod 59) \bmod 29 = 20$
5. $v \equiv r \bmod 29 \Longrightarrow$ valid signature

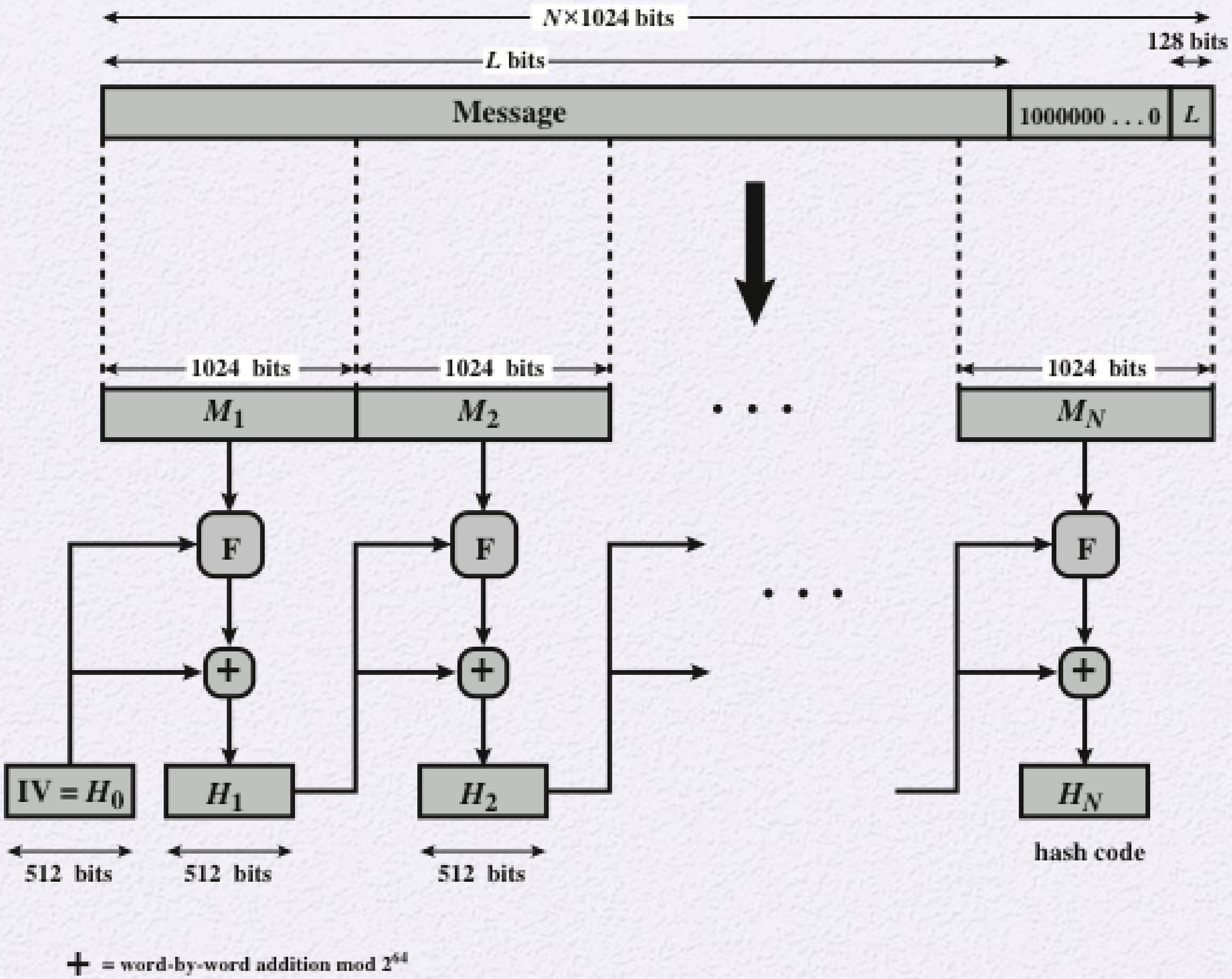


Figure 11.9 Message Digest Generation Using SHA-512

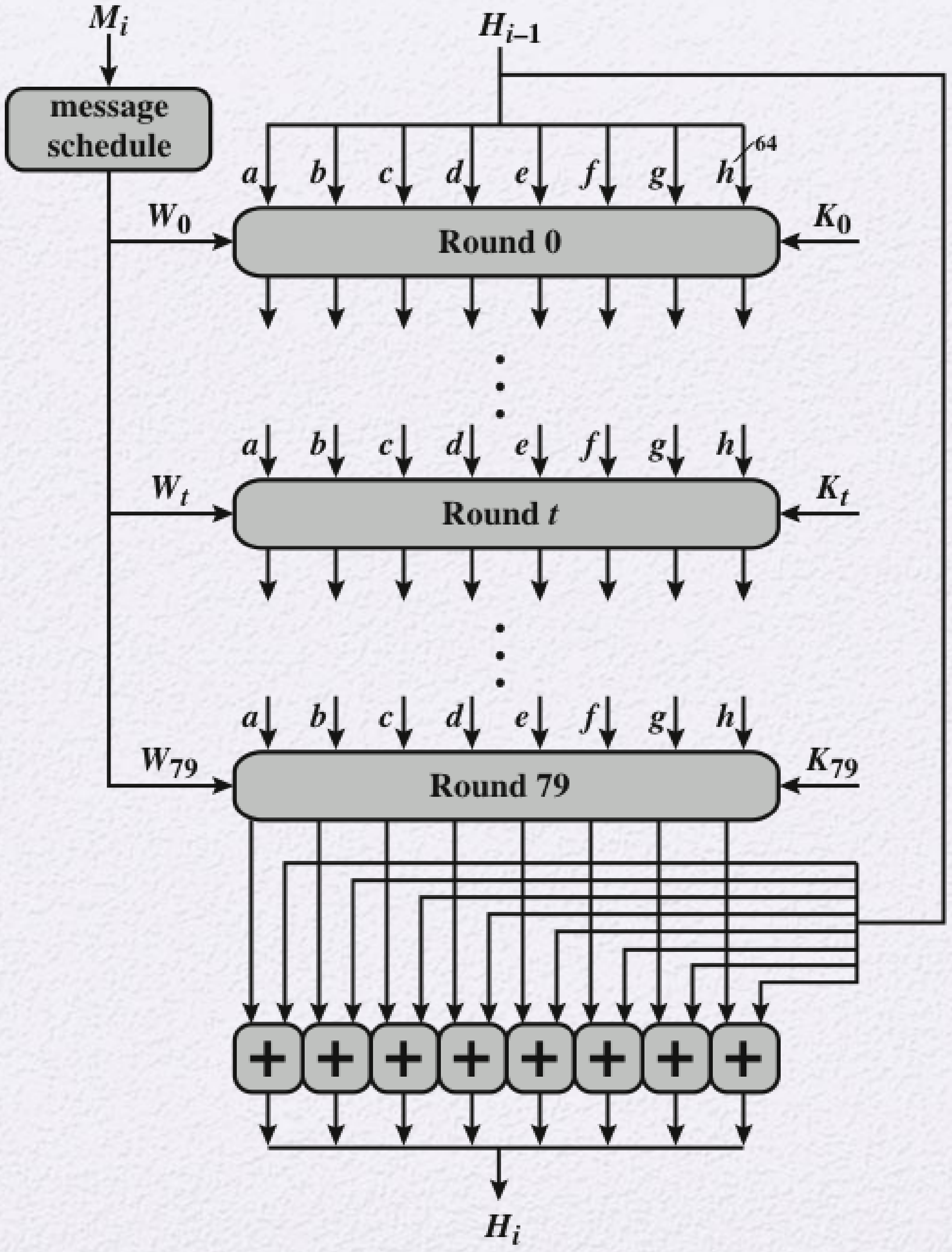
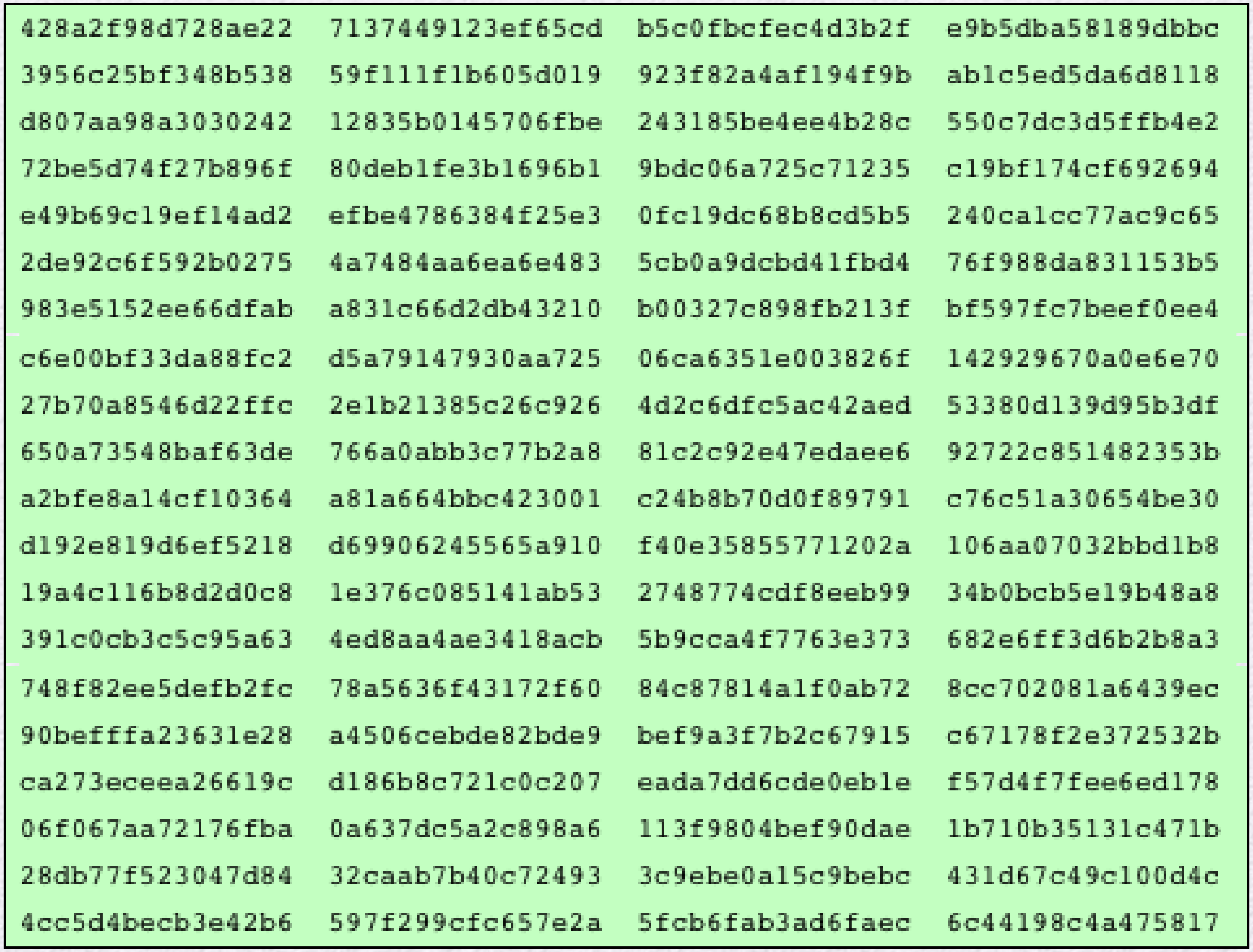


Figure 11.10 SHA-512 Processing of a Single 1024-Bit Block

Table 11.4
SHA-512 Constants



428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

(Table can be found on page 333 in textbook)

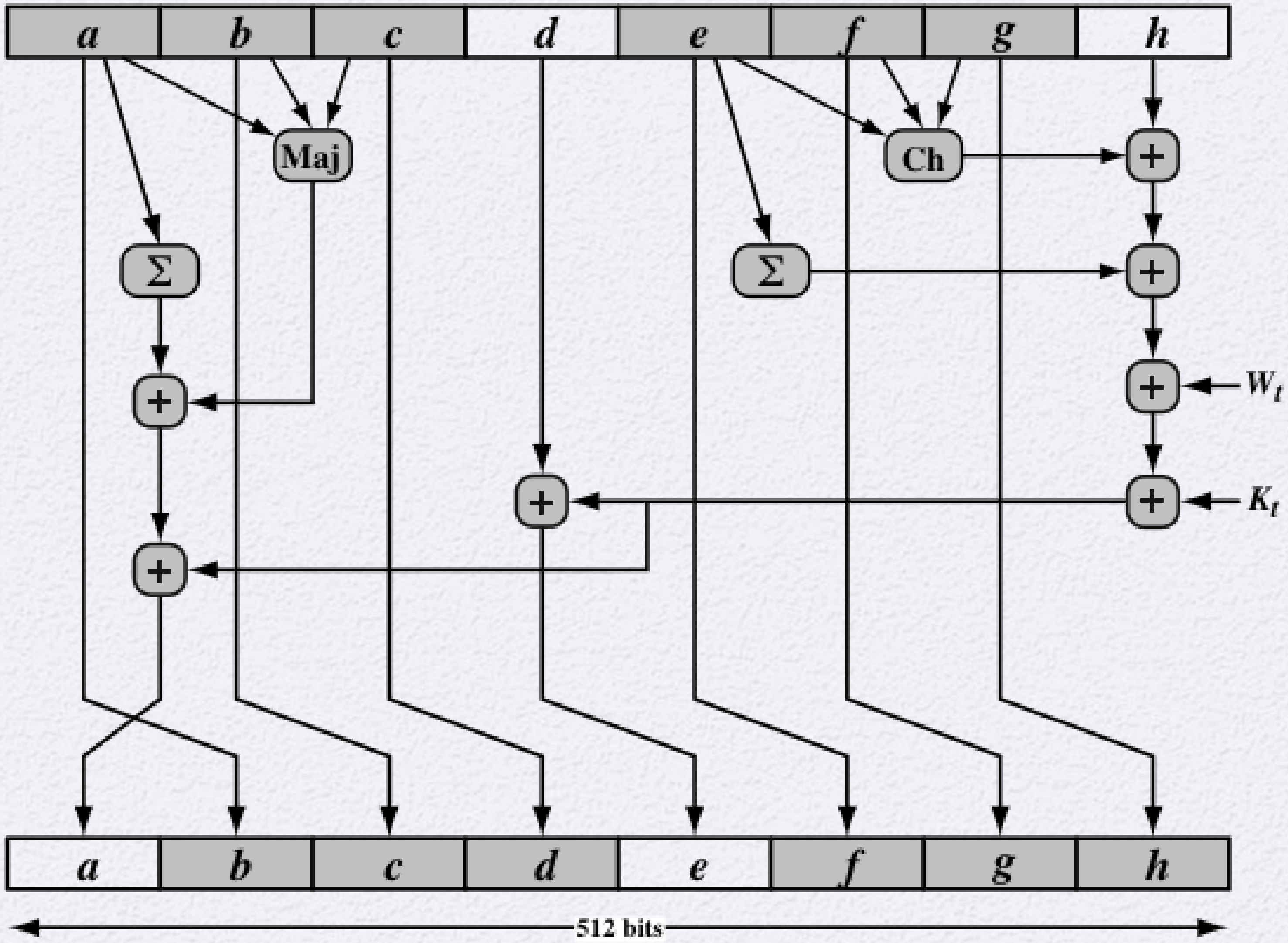
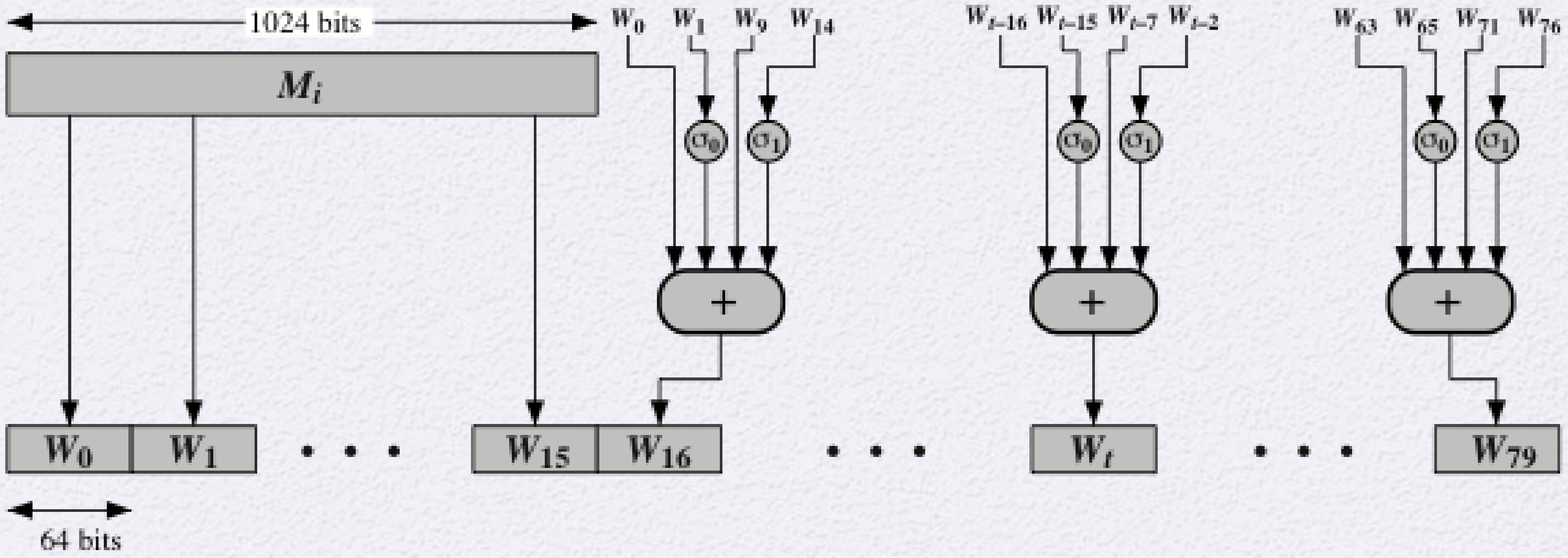


Figure 11.11 Elementary SHA-512 Operation (single round)

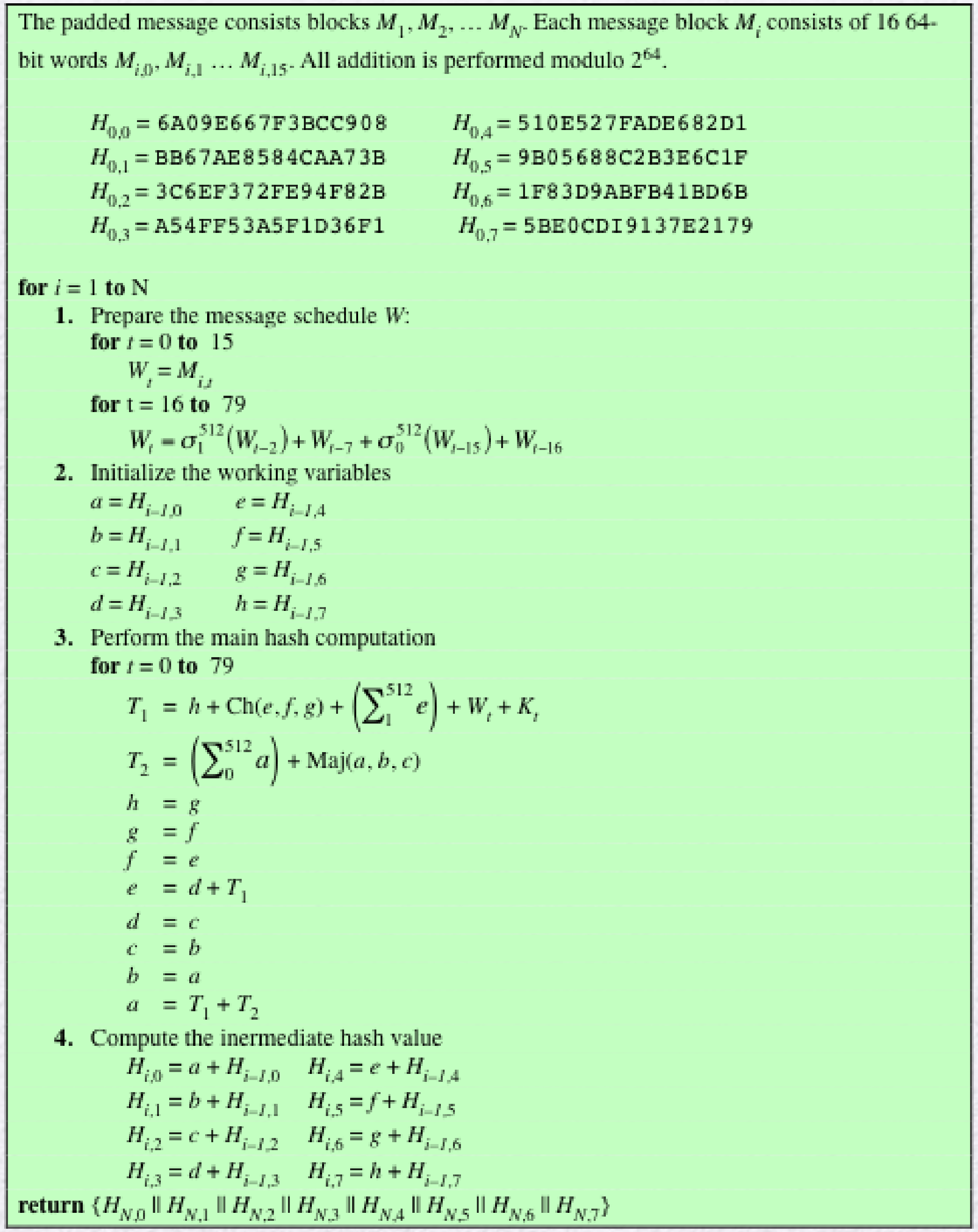


1024 bits
M_i
W_0 W_1 W_9 W_{14}
W_{t-16} W_{t-15} W_{t-7} W_{t-2}
W_{63} W_{65} W_{71} W_{76}
σ_0 σ_1
+
W_0 W_1 • • • W_{15} W_{16} • • • W_t • • • W_{79}
64 bits

Figure 11.12 Creation of 80-word Input Sequence for SHA-512 Processing of Single Block

SHA-512 Logic

(Figure can be found on page 337 in textbook)



The padded message consists blocks $M_1, M_2, \ldots M_N$. Each message block M_i consists of 16 64-bit words $M_{i,0}, M_{i,1} \ldots M_{i,15}$. All addition is performed modulo 2^{64}.

$H_{0,0}$ = 6A09E667F3BCC908	$H_{0,4}$ = 510E527FADE682D1
$H_{0,1}$ = BB67AE8584CAA73B	$H_{0,5}$ = 9B05688C2B3E6C1F
$H_{0,2}$ = 3C6EF372FE94F82B	$H_{0,6}$ = 1F83D9ABFB41BD6B
$H_{0,3}$ = A54FF53A5F1D36F1	$H_{0,7}$ = 5BE0CDI9137E2179

```
for $i$ = 1 to N
    1. Prepare the message schedule $W$:
       for $t = 0$ to 15
           $W_t = M_{i,t}$
       for t = 16 to 79
           $W_t = \sigma_1^{512}(W_{t-2}) + W_{t-7} + \sigma_0^{512}(W_{t-15}) + W_{t-16}$
    2. Initialize the working variables
       $a = H_{i-1,0}$    $e = H_{i-1,4}$
       $b = H_{i-1,1}$    $f = H_{i-1,5}$
       $c = H_{i-1,2}$    $g = H_{i-1,6}$
       $d = H_{i-1,3}$    $h = H_{i-1,7}$
    3. Perform the main hash computation
       for $t = 0$ to 79
           $T_1 = h + \mathrm{Ch}(e, f, g) + \left(\sum_1^{512} e\right) + W_t + K_t$
           $T_2 = \left(\sum_0^{512} a\right) + \mathrm{Maj}(a, b, c)$
           $h = g$
           $g = f$
           $f = e$
           $e = d + T_1$
           $d = c$
           $c = b$
           $b = a$
           $a = T_1 + T_2$
    4. Compute the inermediate hash value
           $H_{i,0} = a + H_{i-1,0}$    $H_{i,4} = e + H_{i-1,4}$
           $H_{i,1} = b + H_{i-1,1}$    $H_{i,5} = f + H_{i-1,5}$
           $H_{i,2} = c + H_{i-1,2}$    $H_{i,6} = g + H_{i-1,6}$
           $H_{i,3} = d + H_{i-1,3}$    $H_{i,7} = h + H_{i-1,7}$
return $\{H_{N,0} \| H_{N,1} \| H_{N,2} \| H_{N,3} \| H_{N,4} \| H_{N,5} \| H_{N,6} \| H_{N,7}\}$
```

Figure 11.13 SHA-512 Logic

SHA-3

SHA-1 has not yet been "broken"

- No one has demonstrated a technique for producing collisions in a practical amount of time
- Considered to be insecure and has been phased out for SHA-2

SHA-2 shares the same structure and mathematical operations as its predecessors so this is a cause for concern

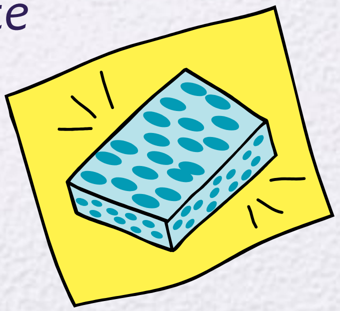
- Because it will take years to find a suitable replacement for SHA-2 should it become vulnerable, NIST decided to begin the process of developing a new hash standard

NIST announced in 2007 a competition for the SHA-3 next generation NIST hash function

- Winning design was announced by NIST in October 2012
- SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications

The Sponge Construction

- Underlying structure of SHA-3 is a scheme referred to by its designers as a *sponge construction*
- Takes an input message and partitions it into fixed-size blocks
- Each block is processed in turn with the output of each iteration fed into the next iteration, finally producing an output block
- The sponge function is defined by three parameters:
  - f = the internal function used to process each input block
  - r = the size in bits of the input blocks, called the *bitrate*
  - pad = the padding algorithm



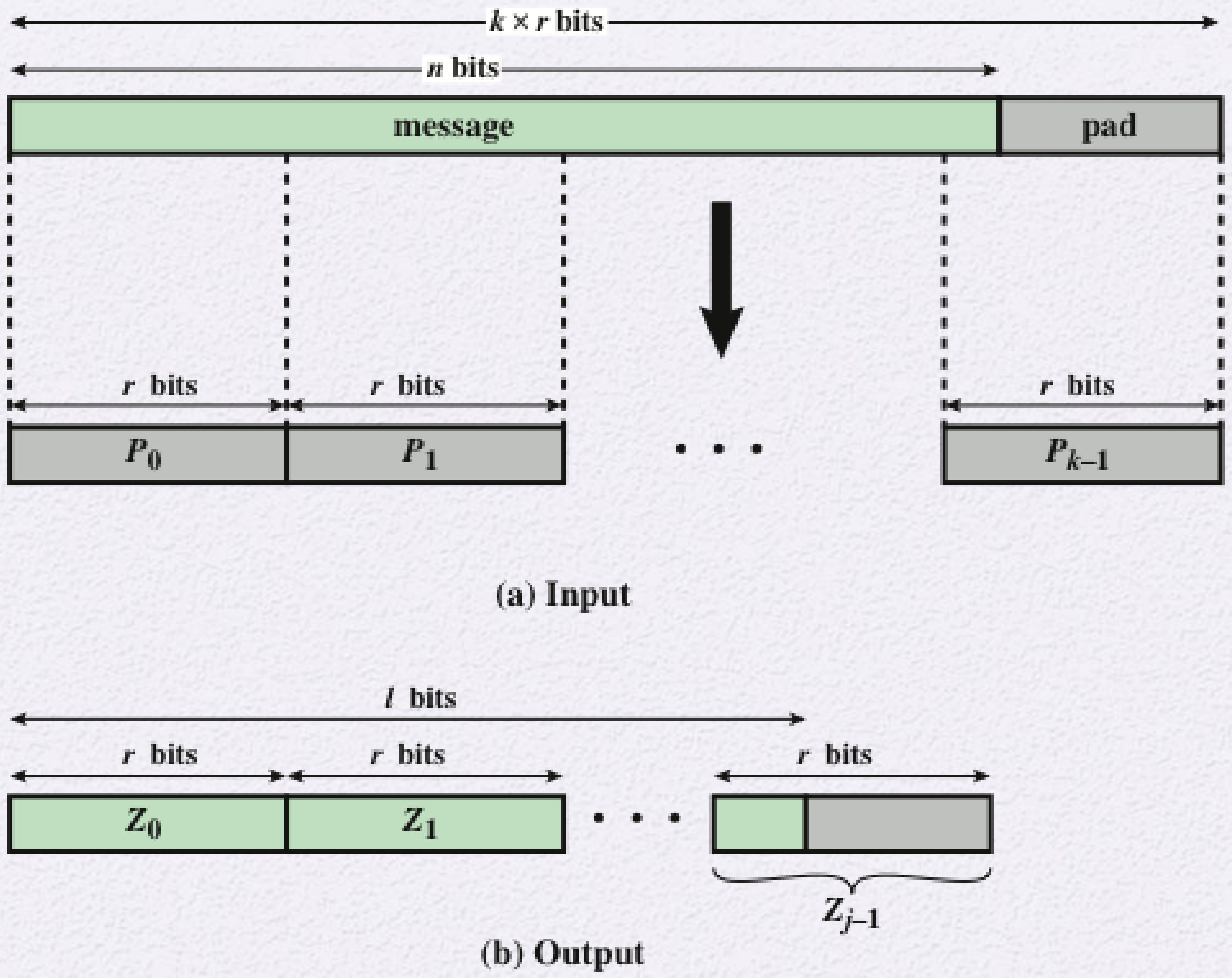


Figure 11.14 Sponge Function Input and Output

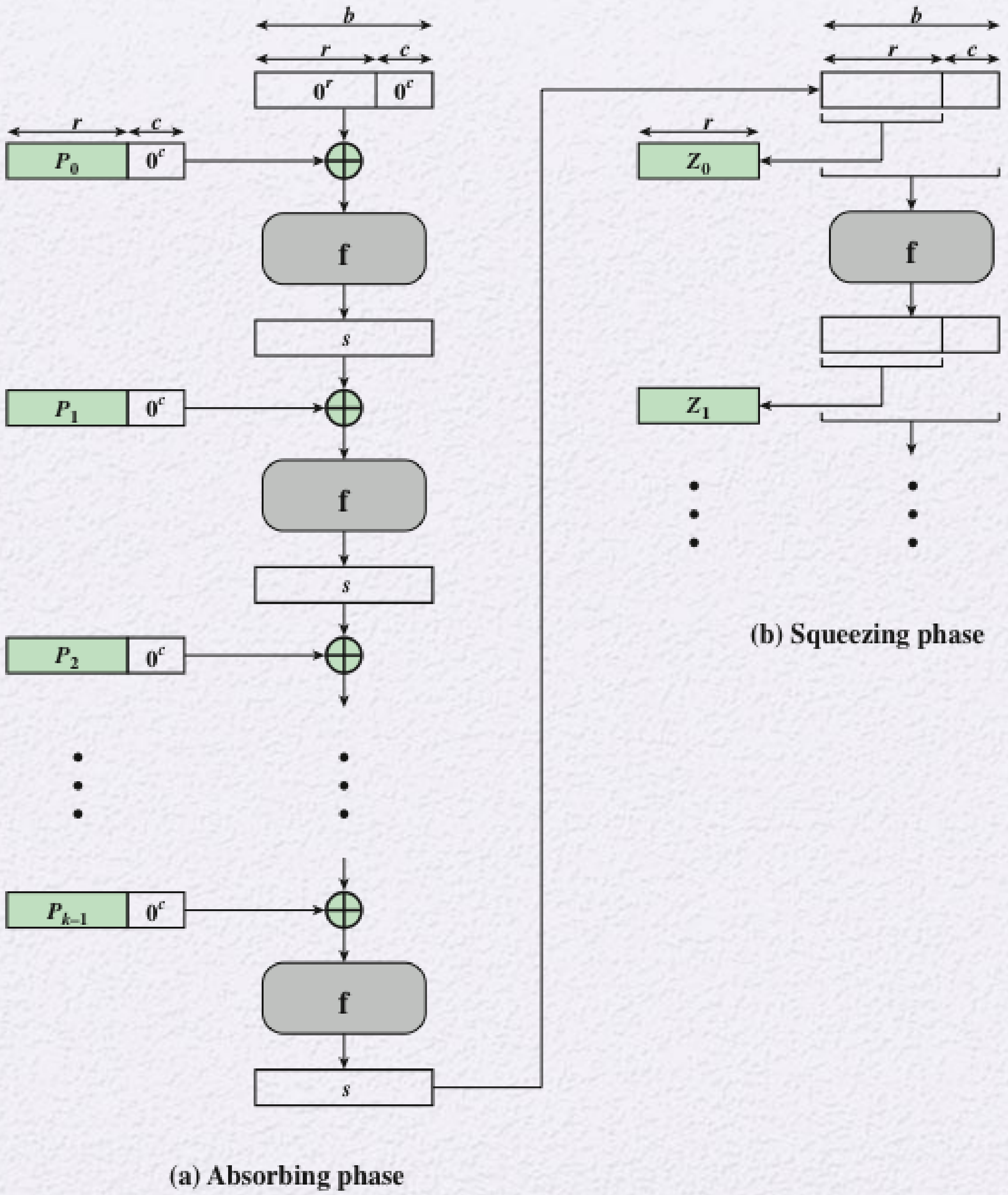
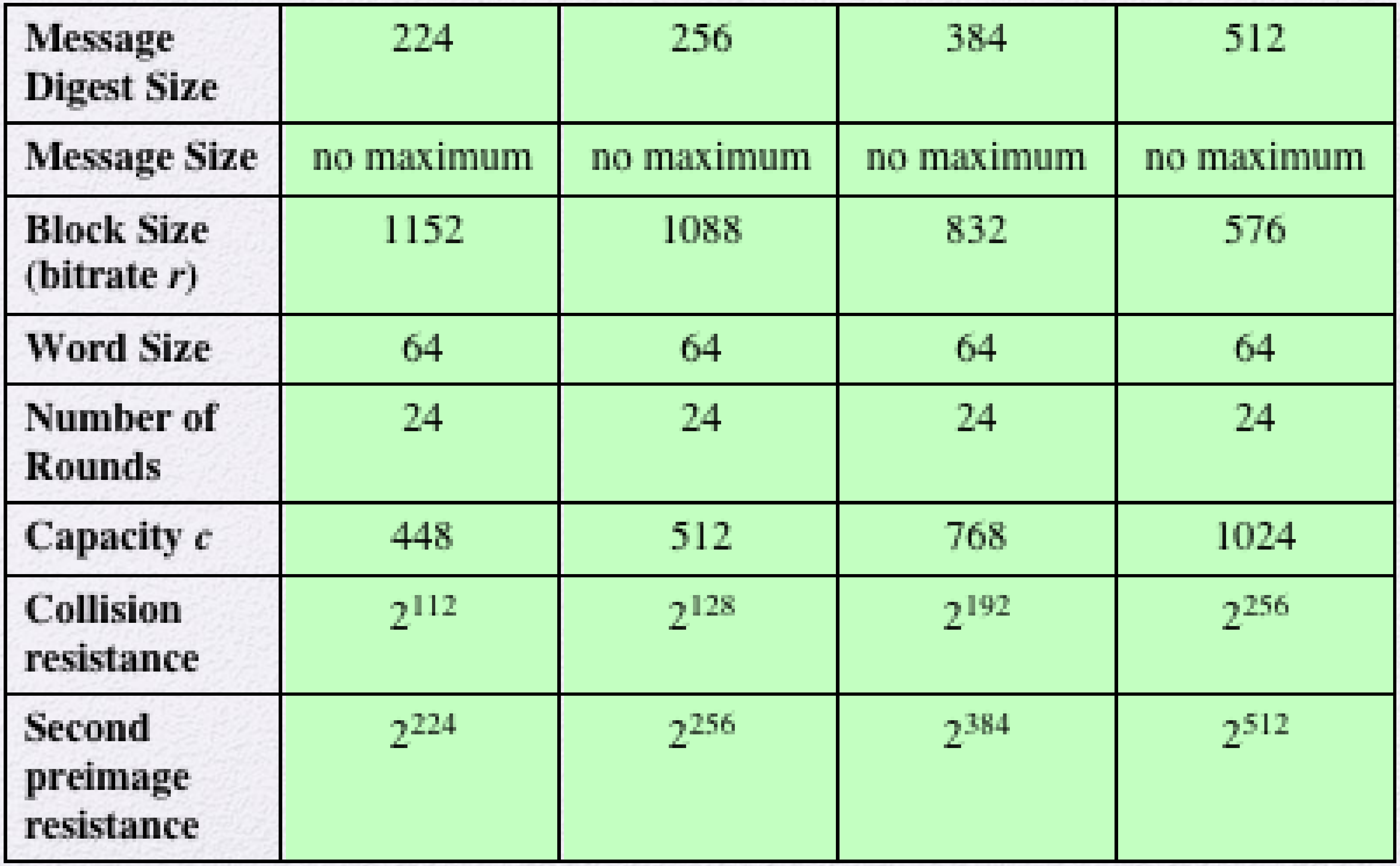


Figure 11.15 Sponge Construction

Table 11.5

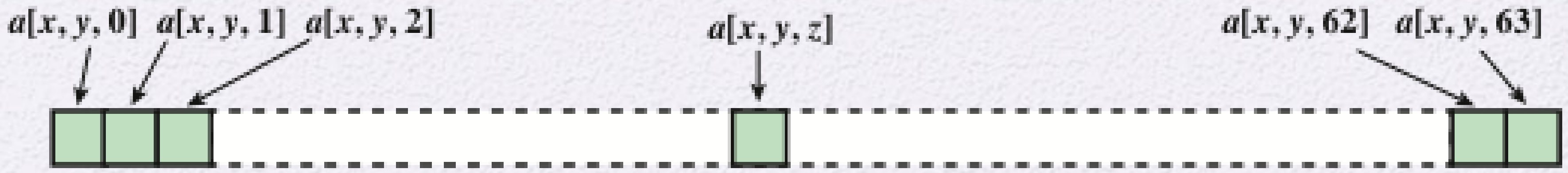
SHA-3 Parameters



Message Digest Size	224	256	384	512
Message Size	no maximum	no maximum	no maximum	no maximum
Block Size (bitrate *r*)	1152	1088	832	576
Word Size	64	64	64	64
Number of Rounds	24	24	24	24
Capacity *c*	448	512	768	1024
Collision resistance	2^{112}	2^{128}	2^{192}	2^{256}
Second preimage resistance	2^{224}	2^{256}	2^{384}	2^{512}

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 4$	$L[0, 4]$	$L[1, 4]$	$L[2, 4]$	$L[3, 4]$	$L[4, 4]$
$y = 3$	$L[0, 3]$	$L[1, 3]$	$L[2, 3]$	$L[3, 3]$	$L[4, 3]$
$y = 2$	$L[0, 2]$	$L[1, 2]$	$L[2, 2]$	$L[3, 2]$	$L[4, 2]$
$y = 1$	$L[0, 1]$	$L[1, 1]$	$L[2, 1]$	$L[4, 1]$	$L[4, 1]$
$y = 0$	$L[0, 0]$	$L[1, 0]$	$L[2, 0]$	$L[3, 0]$	$L[4, 0]$

(a) State variable as 5 × 5 matrix A of 64-bit words



(b) Bit labeling of 64-bit words

Figure 11.16 SHA-3 State Matrix

SHA-3 Iteration Function *f*

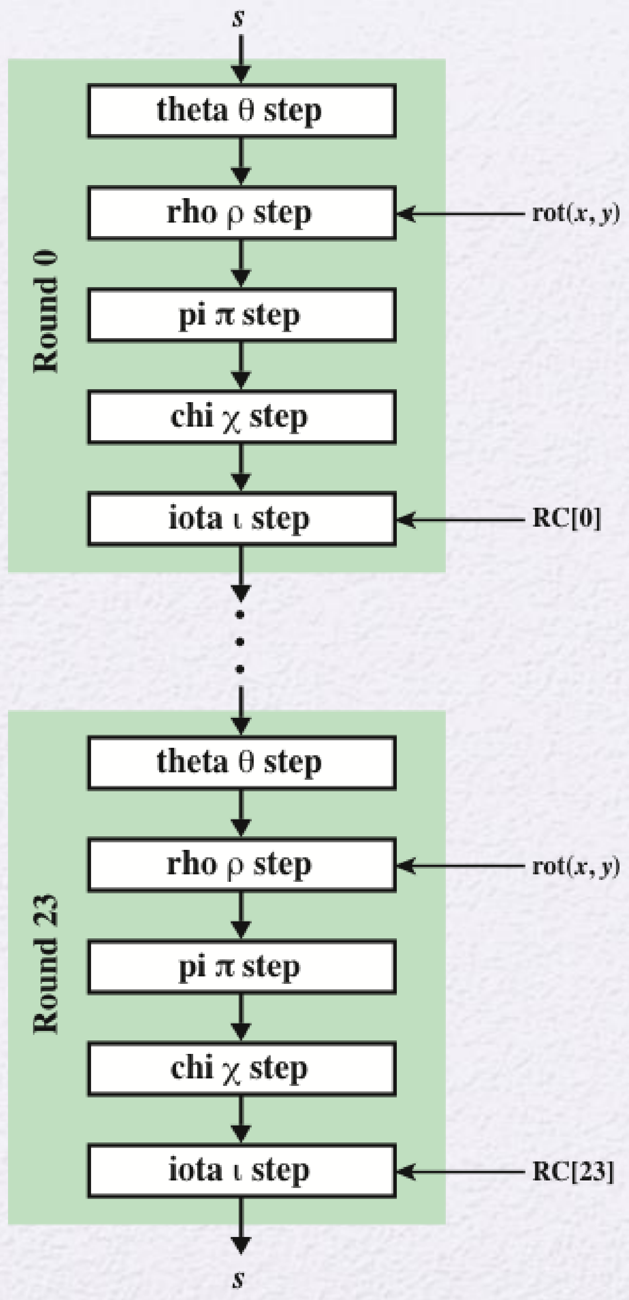
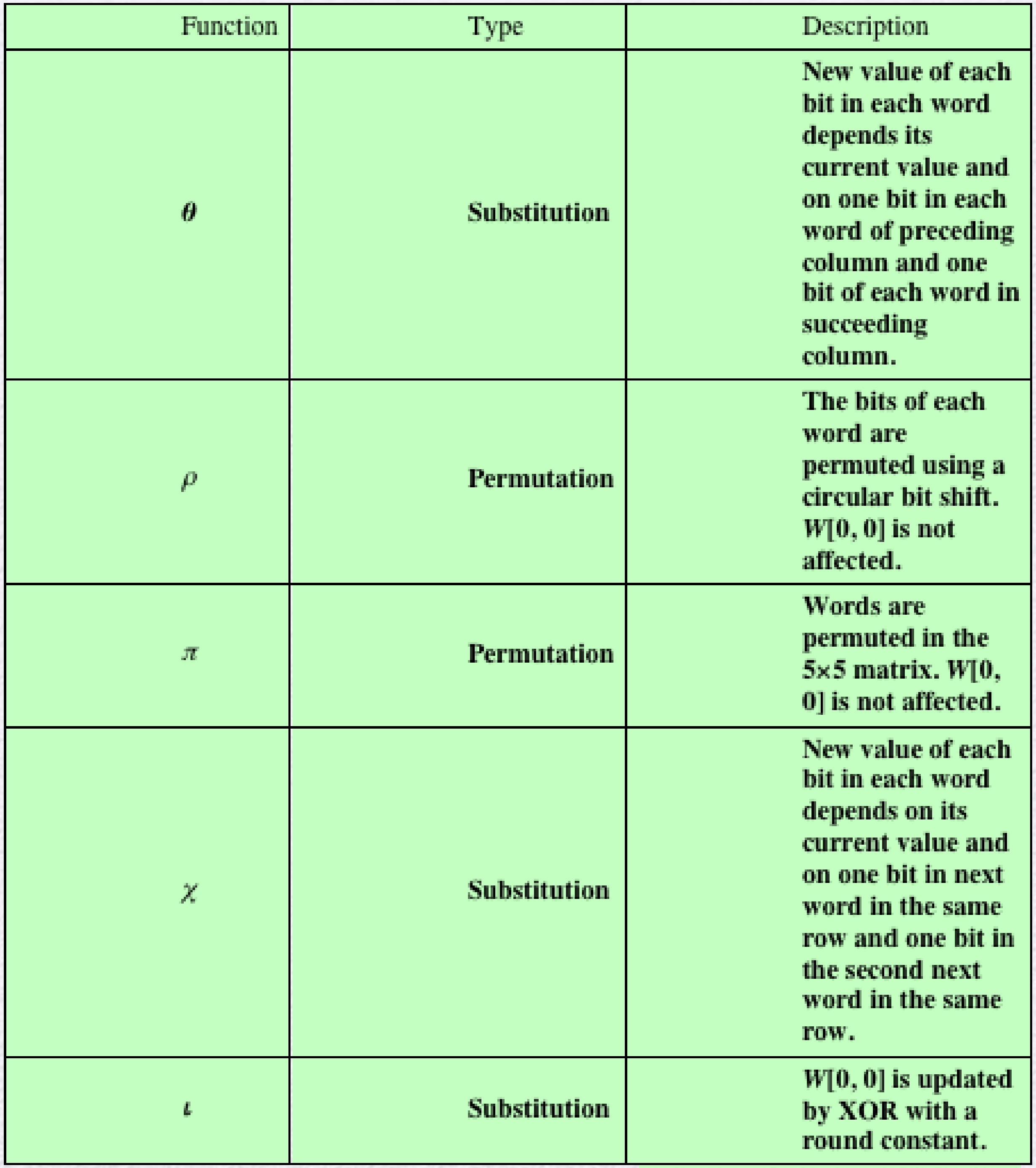


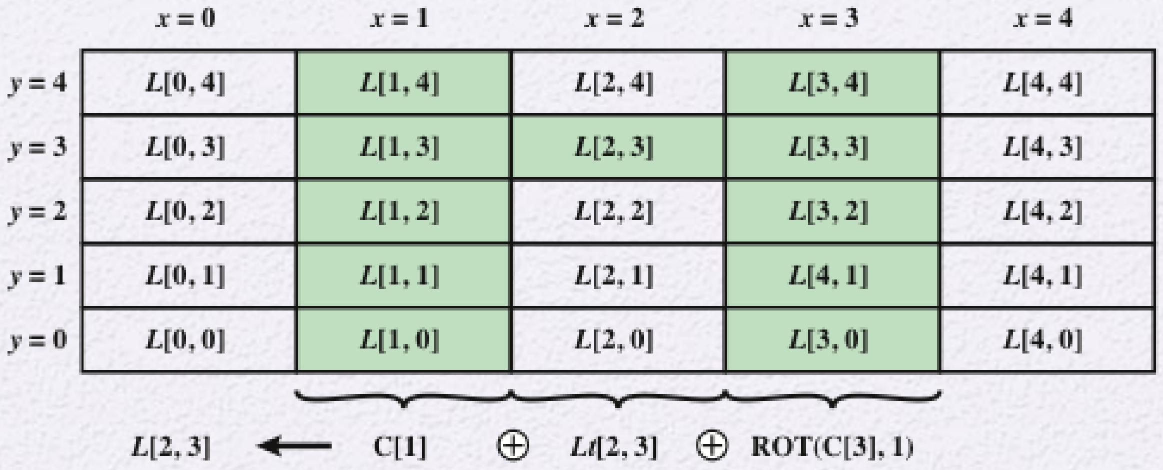
Figure 11.17 SHA-3 Iteration Function *f*

Table 11.6

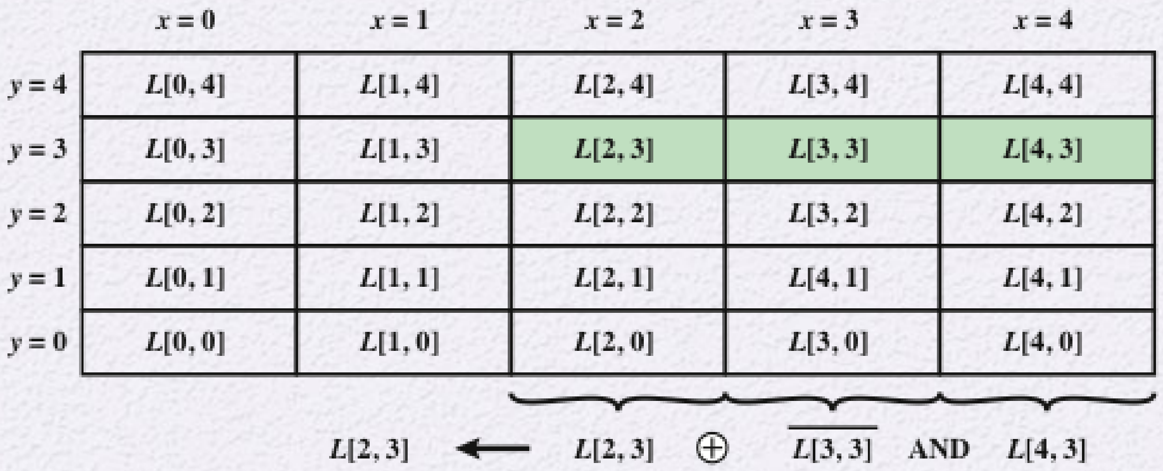
Step Functions in SHA-3



Function	Type	Description
θ	**Substitution**	**New value of each bit in each word depends its current value and on one bit in each word of preceding column and one bit of each word in succeeding column.**
ρ	**Permutation**	**The bits of each word are permuted using a circular bit shift. *W*[0, 0] is not affected.**
π	**Permutation**	**Words are permuted in the 5×5 matrix. *W*[0, 0] is not affected.**
χ	**Substitution**	**New value of each bit in each word depends on its current value and on one bit in next word in the same row and one bit in the second next word in the same row.**
ι	**Substitution**	***W*[0, 0] is updated by XOR with a round constant.**

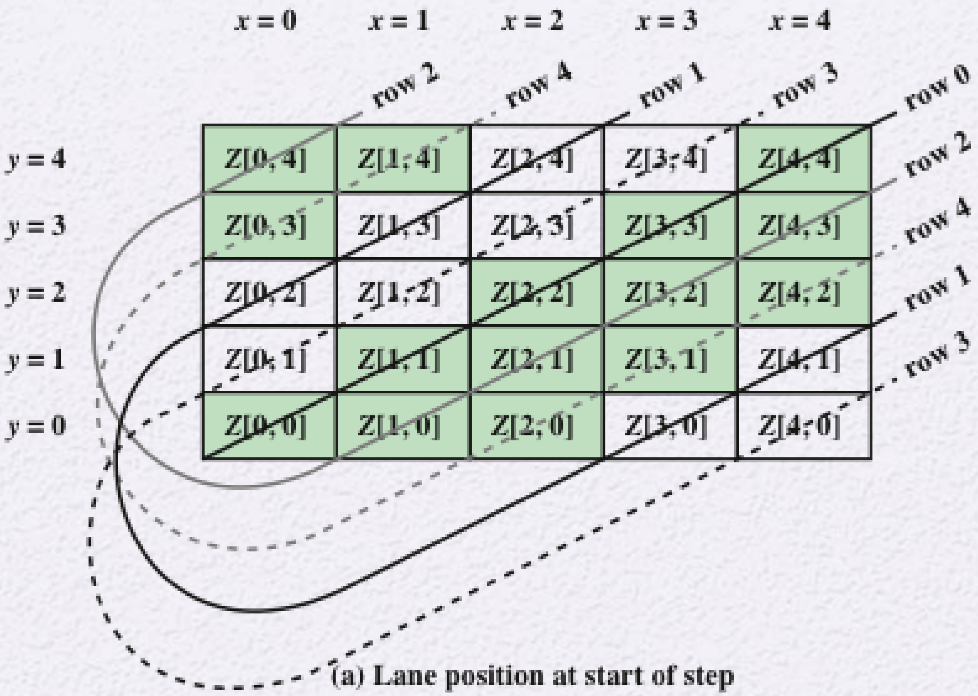

x = 0 x = 1 x = 2 x = 3 x = 4
y = 4 L[0, 4] L[1, 4] L[2, 4] L[3, 4] L[4, 4]
y = 3 L[0, 3] L[1, 3] L[2, 3] L[3, 3] L[4, 3]
y = 2 L[0, 2] L[1, 2] L[2, 2] L[3, 2] L[4, 2]
y = 1 L[0, 1] L[1, 1] L[2, 1] L[4, 1] L[4, 1]
y = 0 L[0, 0] L[1, 0] L[2, 0] L[3, 0] L[4, 0]
L[2, 3] ← C[1] ⊕ L[2, 3] ⊕ ROT(C[3], 1)

(a) θ step function


x = 0 x = 1 x = 2 x = 3 x = 4
y = 4 L[0, 4] L[1, 4] L[2, 4] L[3, 4] L[4, 4]
y = 3 L[0, 3] L[1, 3] L[2, 3] L[3, 3] L[4, 3]
y = 2 L[0, 2] L[1, 2] L[2, 2] L[3, 2] L[4, 2]
y = 1 L[0, 1] L[1, 1] L[2, 1] L[4, 1] L[4, 1]
y = 0 L[0, 0] L[1, 0] L[2, 0] L[3, 0] L[4, 0]
L[2, 3] ← L[2, 3] ⊕ $\overline{L[3, 3]}$ AND L[4, 3]

(b) χ step function

Figure 11.18 Theta and Chi Step Functions



(a) Lane position at start of step

	x = 0	x = 1	x = 2	x = 3	x = 4
y = 4	Z[2, 0]	Z[3, 1]	Z[4, 2]	Z[0, 3]	Z[1, 4]
y = 3	Z[4, 0]	Z[0, 1]	Z[1, 2]	Z[2, 3]	Z[3, 4]
y = 2	Z[1, 0]	Z[2, 1]	Z[3, 2]	Z[4, 3]	Z[0, 4]
y = 1	Z[3, 0]	Z[4, 1]	Z[0, 2]	Z[1, 3]	Z[2, 4]
y = 0	Z[0, 0]	Z[1, 1]	Z[2, 2]	Z[3, 3]	Z[4, 4]

(b) Lane position after permutation

Figure 11.19 Pi Step Function

Summary

- Applications of cryptographic hash functions
  - Message authentication
  - Digital signatures
  - Other applications
- Requirements and security
  - Security requirements for cryptographic hash functions
  - Brute-force attacks
  - Cryptanalysis

- Hash functions based on cipher block chaining
- Secure hash algorithm (SHA)
  - SHA-512 logic
  - SHA-512 round function
- SHA-3
  - The sponge construction
  - The SHA-3 Iteration Function *f*