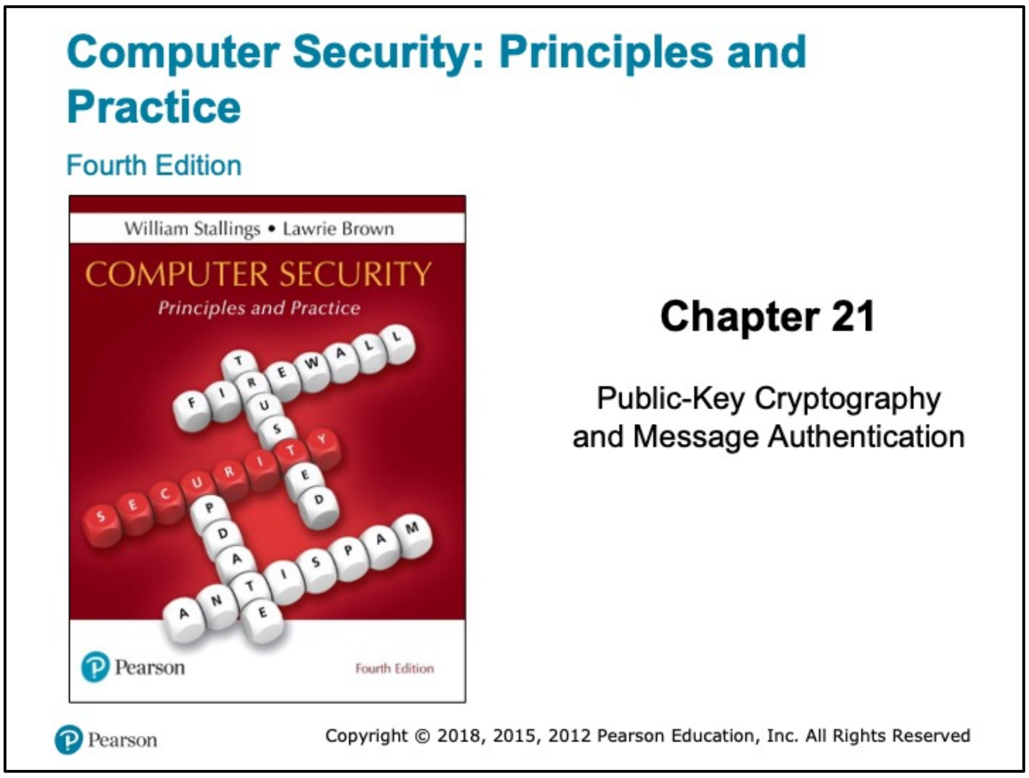
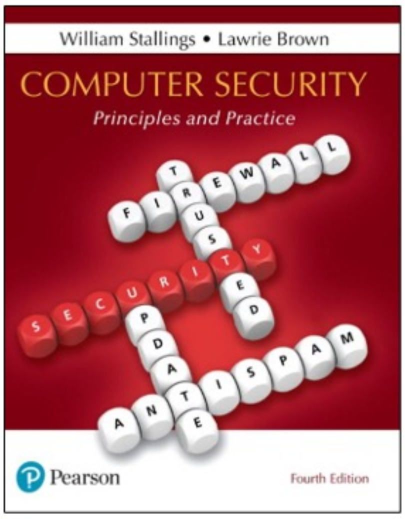


Computer Security: Principles and Practice

Fourth Edition



Chapter 21

Public-Key Cryptography and Message Authentication

This chapter provides technical detail on the topics introduced in Sections 2.2 through 2.4 .

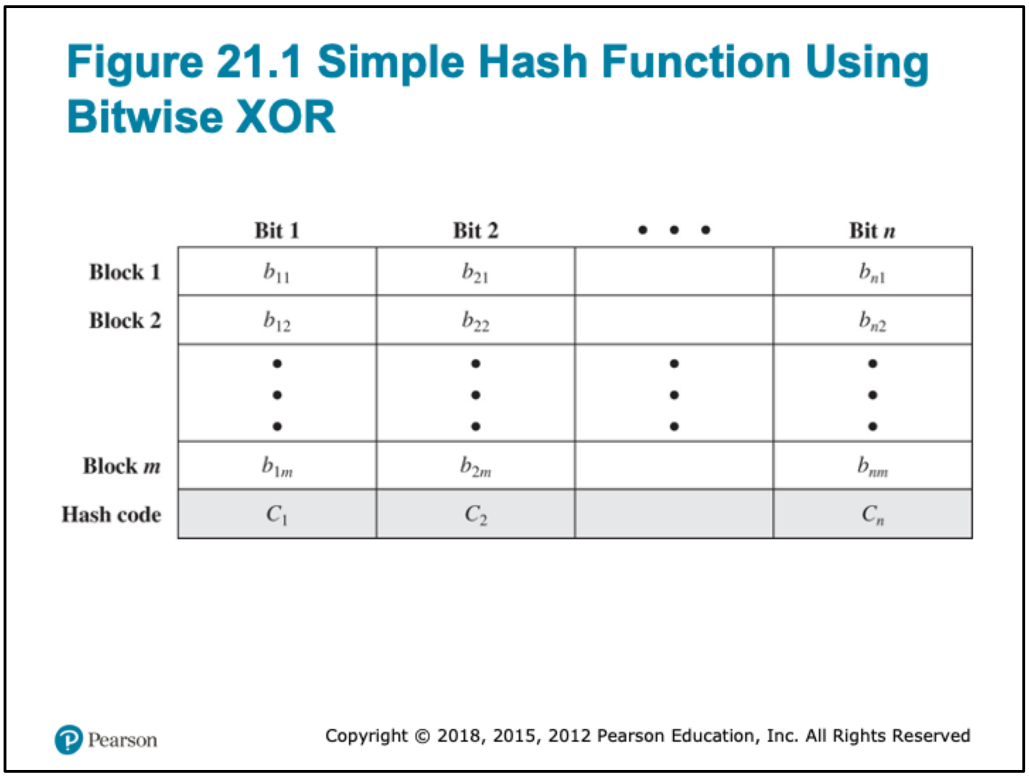


Figure 21.1 Simple Hash Function Using Bitwise XOR

	Bit 1	Bit 2	• • •	Bit n
Block 1	b_{11}	b_{21}		b_{n1}
Block 2	b_{12}	b_{22}		b_{n2}
	⋮	⋮	⋮	⋮
Block m	b_{1m}	b_{2m}		b_{nm}
Hash code	C_1	C_2		C_n

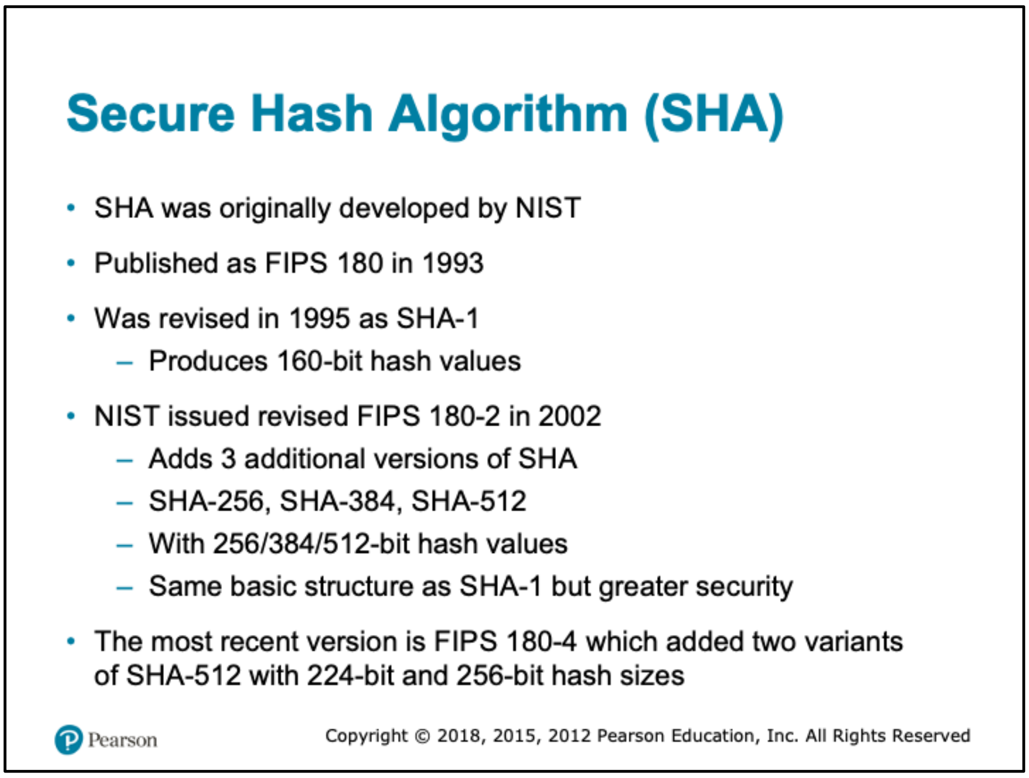
The one-way hash function, or secure hash function, is important not only in message authentication but also in digital signatures. The requirements for and security of secure hash functions are discussed in Section 2.2 . Here, we look at several hash functions, concentrating on perhaps the most widely used family of hash functions: SHA.

All hash functions operate using the following general principles. The input (message, file, etc.) 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. One of the simplest hash functions is the bit-by-bit exclusive-OR (XOR) of every block. This can be expressed as follows:

$$C_i = b_{i1} \oplus b_{i2} \oplus . . . \oplus b_{im}$$

Figure 21.1 illustrates this operation; it produces a simple parity for each bit position and is known as a longitudinal redundancy check. It is reasonably effective for random data as a data integrity check. Each n -bit hash value is equally likely. Thus, the probability that a data error will result in an unchanged hash value is 2-n. With more predictably formatted data, the function is less effective. For example, in most normal text files, the high-order bit of each octet is always zero. So if a 128-bit hash value is used, instead of an effectiveness of 2-128, the hash function on this type of data has an effectiveness of 2-112.

The columns have the following headings from left to right. Block, Bit 1, Bit 2, ellipsis, Bit n, . The row entries are as follows. Row 1. 1, b sub 11, b sub 21, Blank, b sub start expression n 1 end expression. Row 2. 2, b sub 12, b sub 22, Blank, b sub start expression n 2 end expression. Row 3. ellipsis, ellipsis, ellipsis, ellipsis, Ellipsis. Row 4. m, b sub start expression 1 m end expression, b sub start expression 2 m end expression, Blank, b sub start expression n m end expression. Row 5. Hash code, C sub 1, C sub 2, Blank, C sub n.



Secure Hash Algorithm (SHA)

- SHA was originally developed by NIST
- Published as FIPS 180 in 1993
- Was revised in 1995 as SHA-1
  - Produces 160-bit hash values
- NIST issued revised FIPS 180-2 in 2002
  - Adds 3 additional versions of SHA
  - SHA-256, SHA-384, SHA-512
  - With 256/384/512-bit hash values
  - Same basic structure as SHA-1 but greater security
- The most recent version is FIPS 180-4 which added two variants of SHA-512 with 224-bit and 256-bit hash sizes

In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). Indeed, because virtually every other widely used hash function had been found to have substantial cryptanalytic weaknesses, SHA was more or less the last remaining standardized hash algorithm by 2005. SHA was developed by the National Institute of Standards and Technology (NIST) and published as FIPS 180 in 1993. When weaknesses were discovered in SHA (now known as SHA-0), a revised version was issued as FIPS 180-1 in 1995 and is referred to as SHA-1 . The actual standards document is entitled "Secure Hash Standard. SHA-1 is also specified in RFC 3174 (*US Secure Hash Algorithm 1 (SHA1) ,* 2001), which essentially duplicates the material in FIPS 180-1 but adds a C code implementation.

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, respectively (see Table 21.1). Collectively, these hash algorithms are known as **SHA-2** . These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. A revised document was issued as FIPS 180-3 in 2008, which added a 224-bit version of SHA-256, whose hash value is obtained by truncating the 256-bit hash value of SHA-256. SHA-1 and SHA-2 are also specified in RFC 6234 (*US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF),* 2011), which essentially duplicates the material in FIPS 180-3 but adds a C code implementation. The most recent version is FIPS 180-4 [*Secure Hash Standard (SHS)* , August 2015] which added two variants of SHA-512 with 224-bit and 256-bit hash sizes, as SHA-512 is more efficient than SHA-256 on many 64-bit systems.

In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on SHA-2 by 2010. Shortly thereafter, a research team described an attack in which two separate messages could be found that deliver the same SHA-1 hash using 2^{69} operations, far fewer than the 2^{80} operations previously thought needed to find a collision with an SHA-1 hash [WANG05]. This result has hastened the transition to SHA-2.

In this section, we provide a description of SHA-512. The other versions are quite similar. The algorithm takes as input a message with a maximum length of less than 2^{128} bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks.

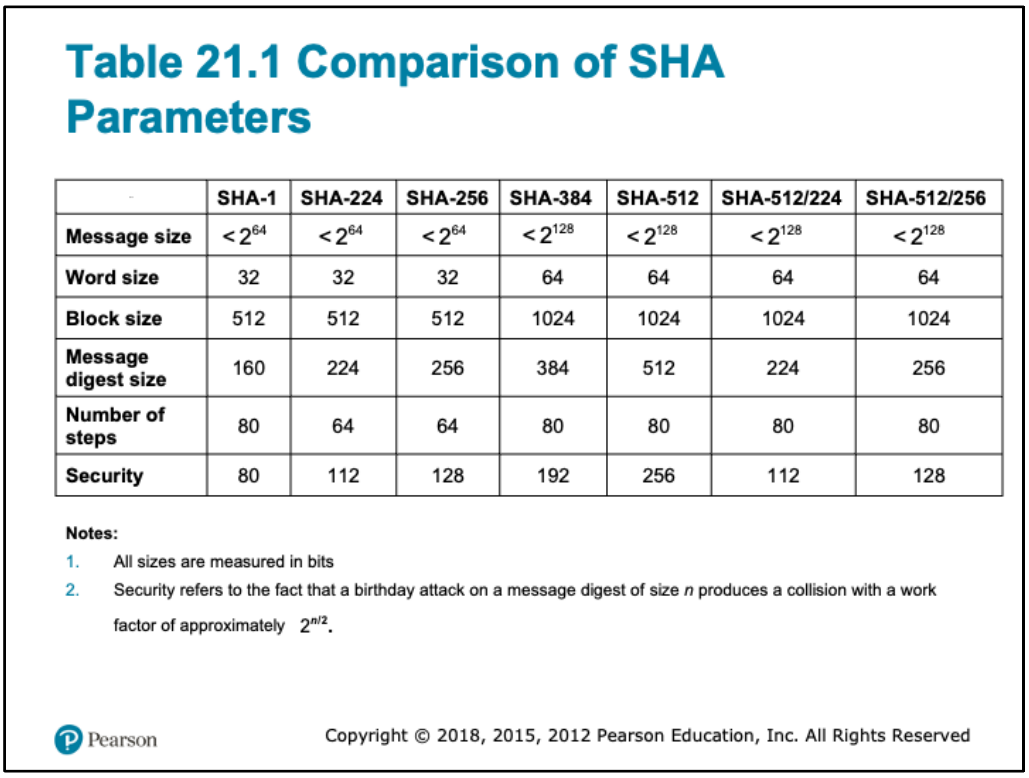


Table 21.1 Comparison of SHA Parameters

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512	SHA-512/224	SHA-512/256
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$	$< 2^{128}$	$< 2^{128}$
Word size	32	32	32	64	64	64	64
Block size	512	512	512	1024	1024	1024	1024
Message digest size	160	224	256	384	512	224	256
Number of steps	80	64	64	80	80	80	80
Security	80	112	128	192	256	112	128

Notes:

1. All sizes are measured in bits
2. Security refers to the fact that a birthday attack on a message digest of size *n* produces a collision with a work factor of approximately $2^{n/2}$.

Pearson

Table 21.1 Comparison of SHA parameters.

Figure 21.2 Message Digest Generation Using SHA-512

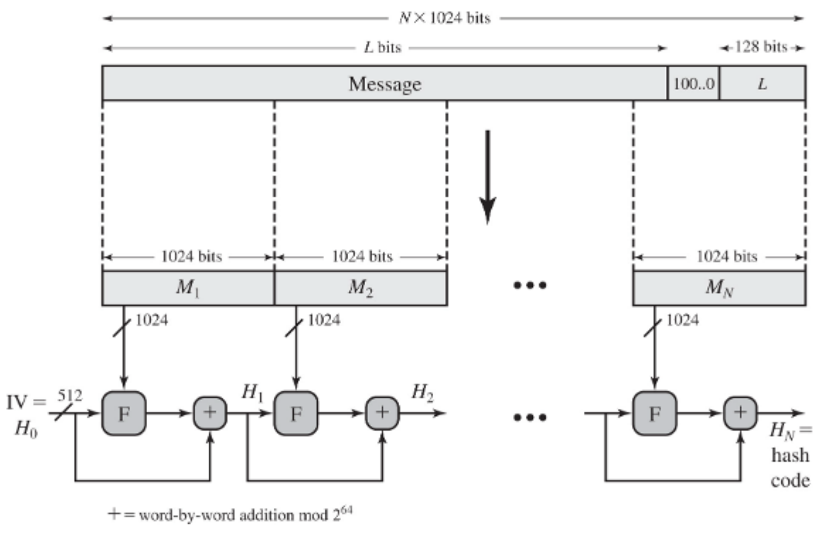


Figure 21.2 depicts the overall processing of a message to produce a digest. The processing consists of the following steps:

• Step 1: Append padding bits: so that message length is congruent to 896 modulo 1024 [length ≡ 896 (mod 1024)]. The padding consists of a single 1-bit followed by the necessary number of 0-bits.

• Step 2: Append length: as a block of 128 bits being an unsigned 128-bit integer length of the original message (before padding).

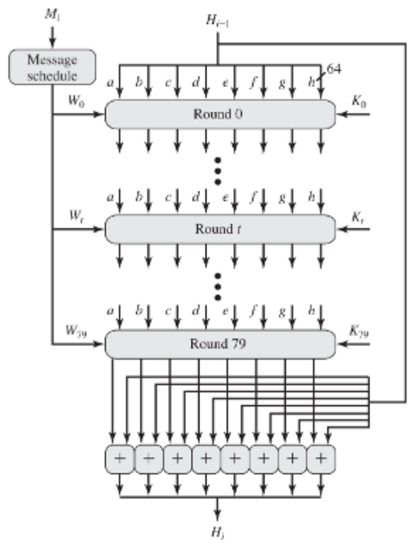
• Step 3: Initialize hash buffer: A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h).

• Step 4: Process the message in 1024-bit (128-word) blocks, The heart of the algorithm is a module that consists of 80 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 is N times 1024 bits. The bits consist of a single 1 bit followed by 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 sub 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 sub N.

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



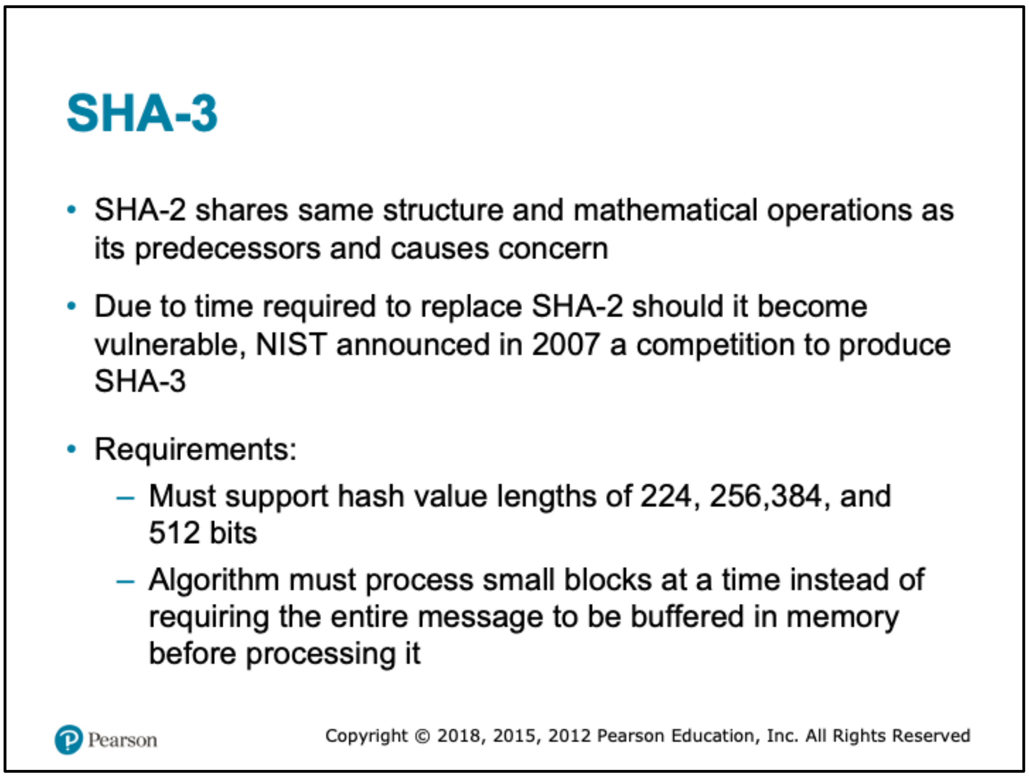
The logic is illustrated in Figure 21.3.

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, Hi–1. Each round t makes use of a 64-bit value Wt, derived from the current 1024-bit block being processed (Mi). Each round also makes use of an additive constant Kt, 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 eightieth round is added to the input to the first round (Hi–1) to produce Hi. The addition is done independently for each of the eight words in the buffer with each of the corresponding words in Hi–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 2256 operations, while the difficulty of finding a message with a given digest is on the order of 2512 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 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
- 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

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.

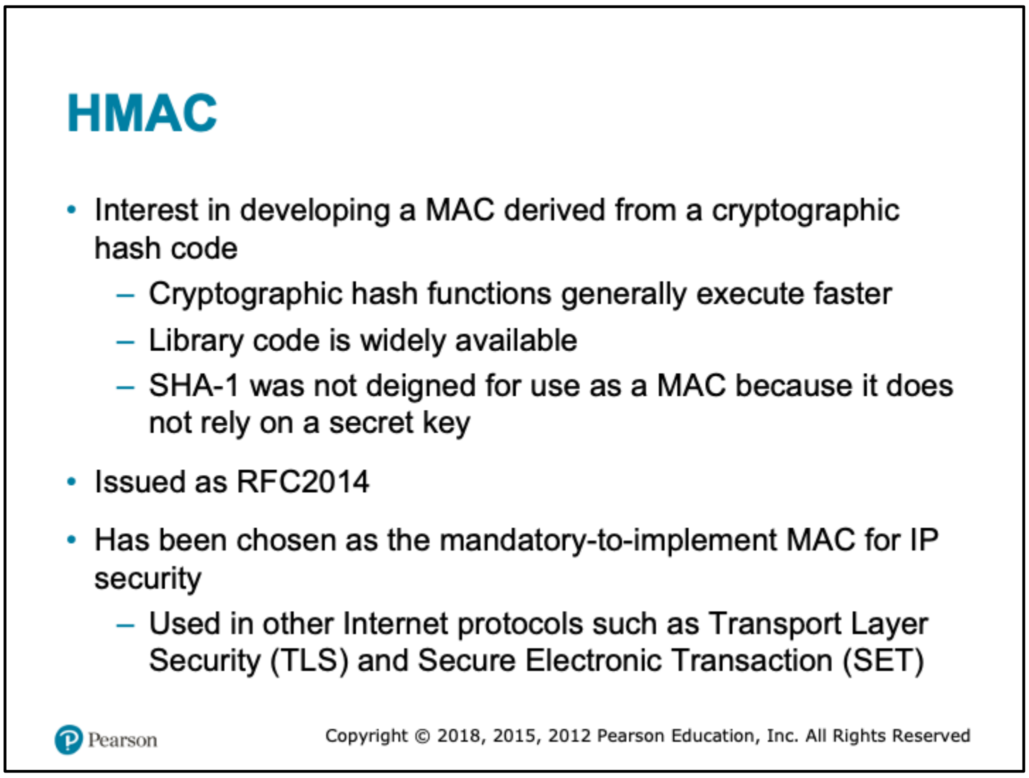
The basic requirements that needed to be satisfied by any candidate for SHA-3 are the following:

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

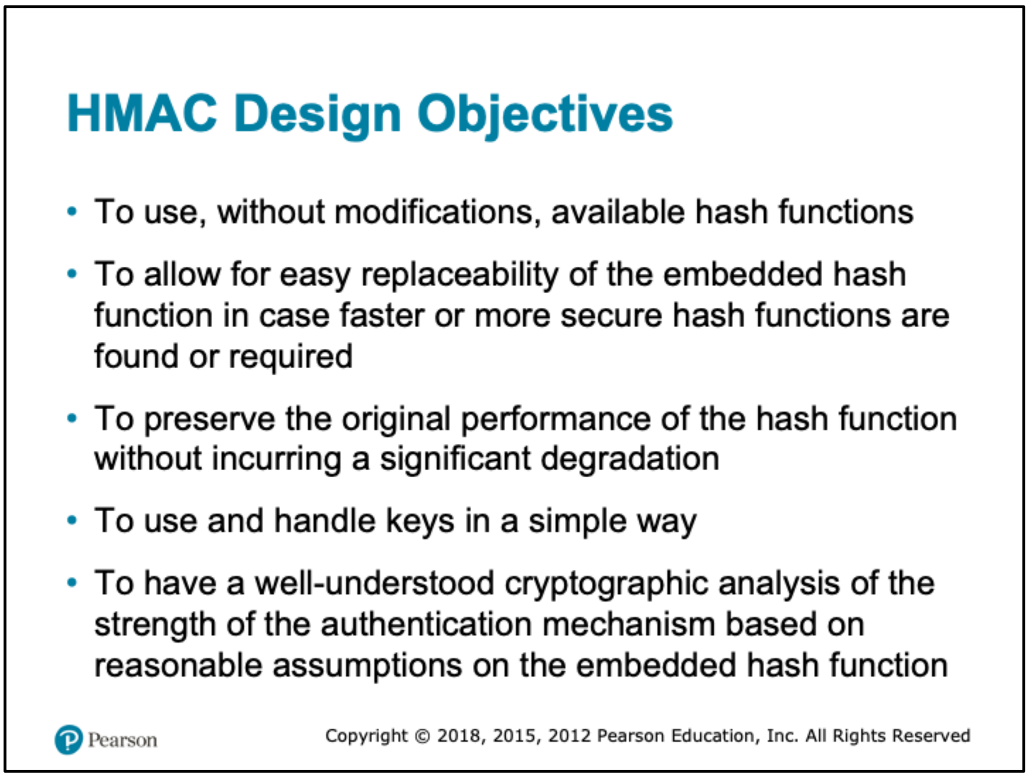
- Interest in developing a MAC derived from a cryptographic hash code
  - Cryptographic hash functions generally execute faster
  - Library code is widely available
  - SHA-1 was not deigned for use as a MAC because it does not rely on a secret key
- Issued as RFC2014
- Has been chosen as the mandatory-to-implement 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

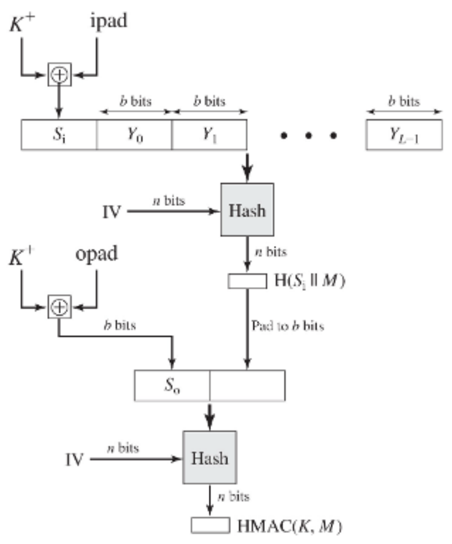


Figure 21.4 illustrates the overall operation of HMAC.
In words:

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 Si.

3. Append M to Si.

4. Apply H to the stream generated in step 3.

5. XOR K+ with opad to produce the b-bit block So.

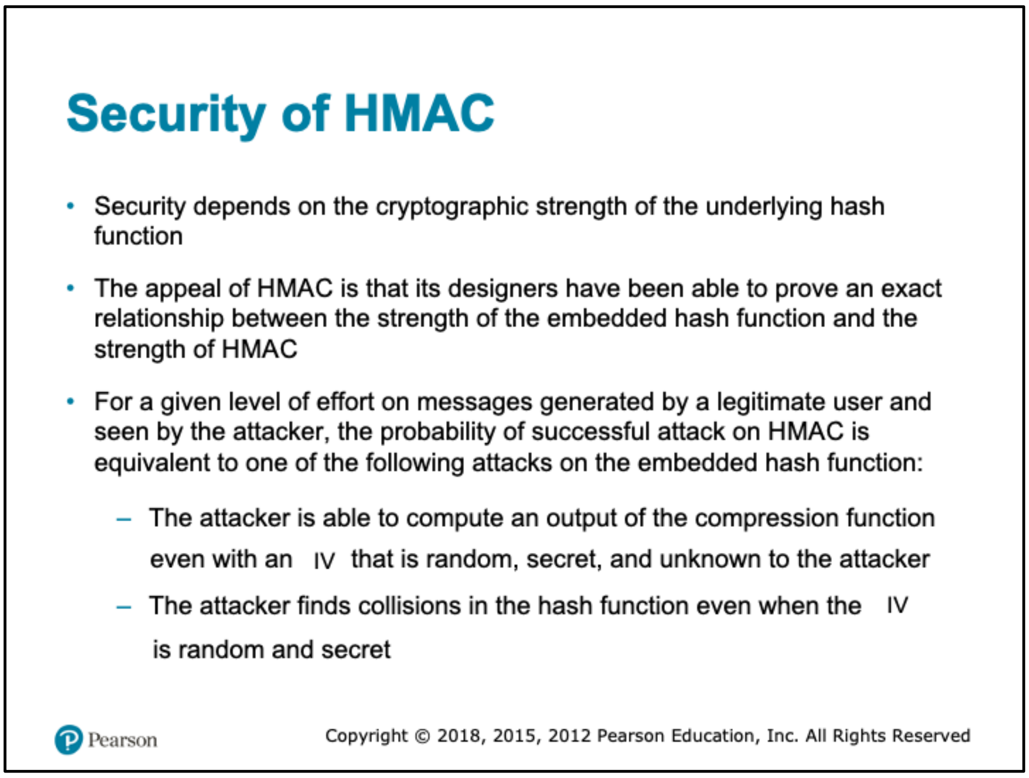
6. Append the hash result from step 4 to So.

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. X O R, bitwise exclusive O R, K + with i pad to produce the b bit block S sub i. 3. Append Y sub 0 and Y sub 1, to Y sub start expression L minus 1 end expression each of b bits, to S sub i. 4. Apply H to the stream generated in step 3. 5. X O R K + with o pad 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 X O R with i pad results in flipping one half of the bits of K. Similarly, the X O R with o pad 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. H M A C 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
  - The attacker finds collisions in the hash function even when the IV is random and secret

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:

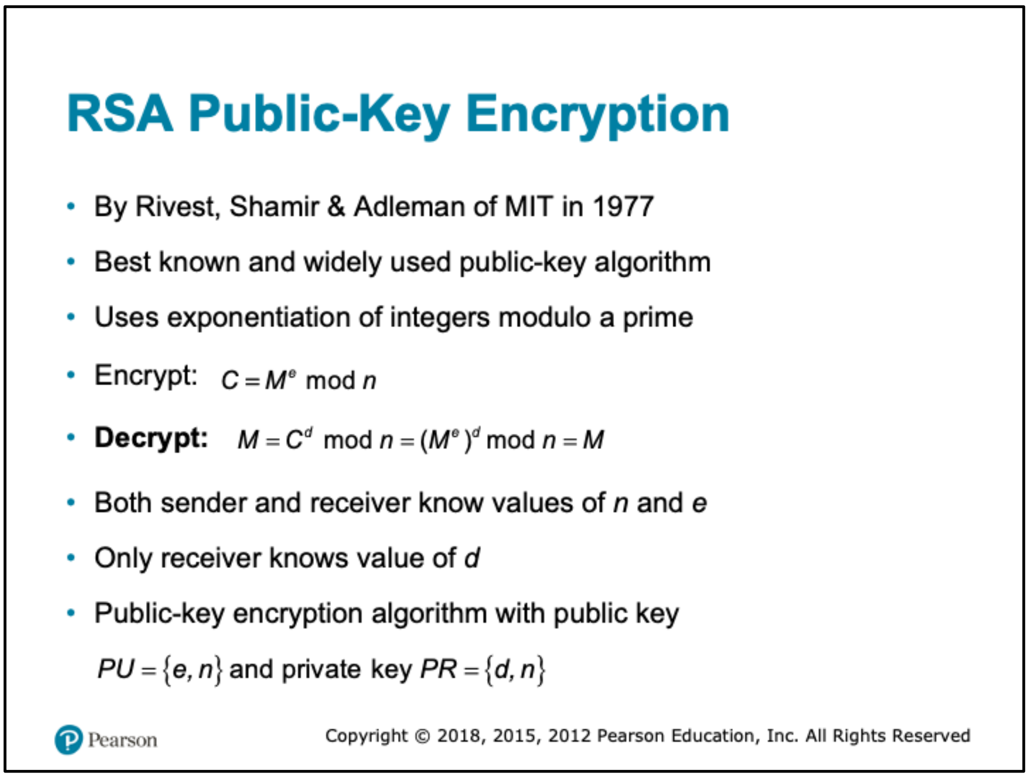
1. The attacker is able to compute an output of the compression function even with an IV that is random, secret, and unknown to the attacker.

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.



RSA Public-Key Encryption

- By Rivest, Shamir & Adleman of MIT in 1977
- Best known and widely used public-key algorithm
- Uses exponentiation of integers modulo a prime
- Encrypt: $C = M^e \bmod n$
- **Decrypt:** $M = C^d \bmod n = (M^e)^d \bmod n = M$
- Both sender and receiver know values of *n* and *e*
- Only receiver knows value of *d*
- Public-key encryption algorithm with public key $PU = \{e, n\}$ and private key $PR = \{d, n\}$

Pearson

Perhaps the most widely used public-key algorithms are RSA and Diffie-Hellman.

One of the first public-key schemes was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978 [RIVE78]. The RSA scheme has since that time reigned supreme as the most widely accepted and implemented approach to public-key encryption. RSA is a block cipher in which the plaintext and ciphertext are integers between 0 and $n - 1$ for some n.

Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

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

$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

Both sender and receiver must know the values of n and e, and only the receiver knows the value of d. This is a public-key encryption algorithm with a public key of $PU = \{e, n\}$ and a private key of $PR = \{d, n\}$. See text for details of how these values are derived, and their requirements.

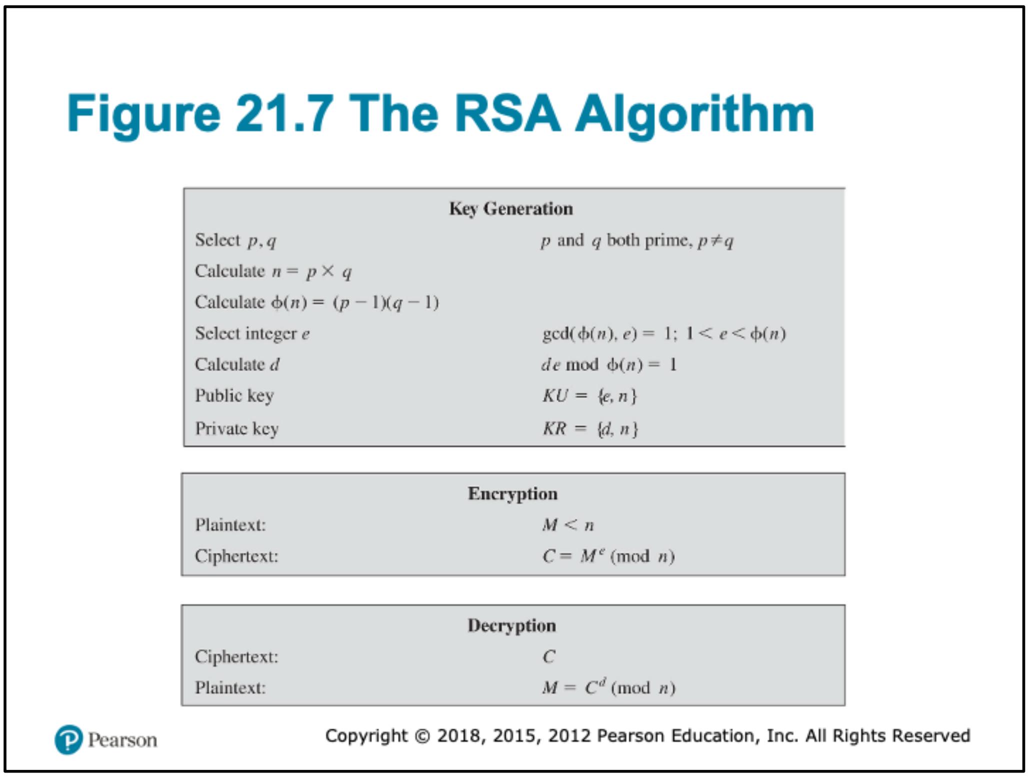


Figure 21.7 The RSA Algorithm

Key Generation	
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1;\ 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

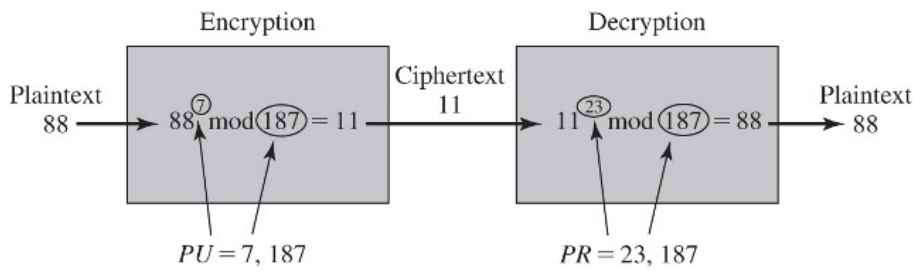
Encryption	
Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod{n}$

Decryption	
Ciphertext:	C
Plaintext:	$M = C^d \pmod{n}$

Figure 21.7 summarizes the RSA algorithm.

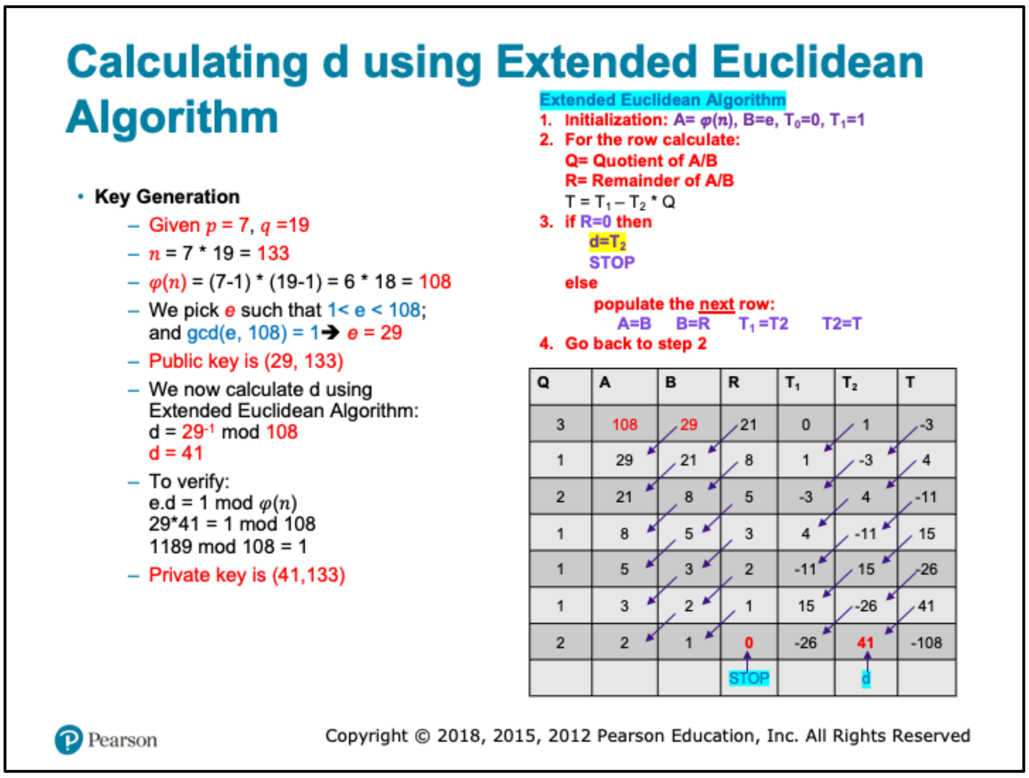
Key generation. Select p, q p and q both prime, p does not equal q. Calculate n = p times q. Calculate phi left parenthesis n right parenthesis = left parenthesis p minus 1 right parenthesis left parenthesis q minus 1 right parenthesis. Select integer e, g c d left parenthesis phi left parenthesis n right parenthesis, e right parenthesis = 1, 1 is less than e and e is less than phi left parenthesis n right parenthesis. Calculate d, d e, m o d phi left parenthesis n right parenthesis = 1. Public key K U = left brace e, n right brace. Private key K R = left brace d, n right brace. Encryption. Plaintext, M is less than n. Cipher text, C = M to the e power left parenthesis m o d, n right parenthesis. Decryption. Ciphertext, C. Plaintext, M = C to the d power left parenthesis m o d, n right parenthesis.

Figure 21.8 Example of RSA Algorithm



An example, from [SING99], is shown in Figure 21.8.

Encryption. Plaintext 88 is given as input. 88 to the 7 power m o d 187, where P U = 7, 87, = 11. Ciphertext 11 is the output. Decryption. Cipher text 88 is decrypted. 11 to the 23 power m o d 187, where P R = 23, 187. Plaintext 88 is the output.

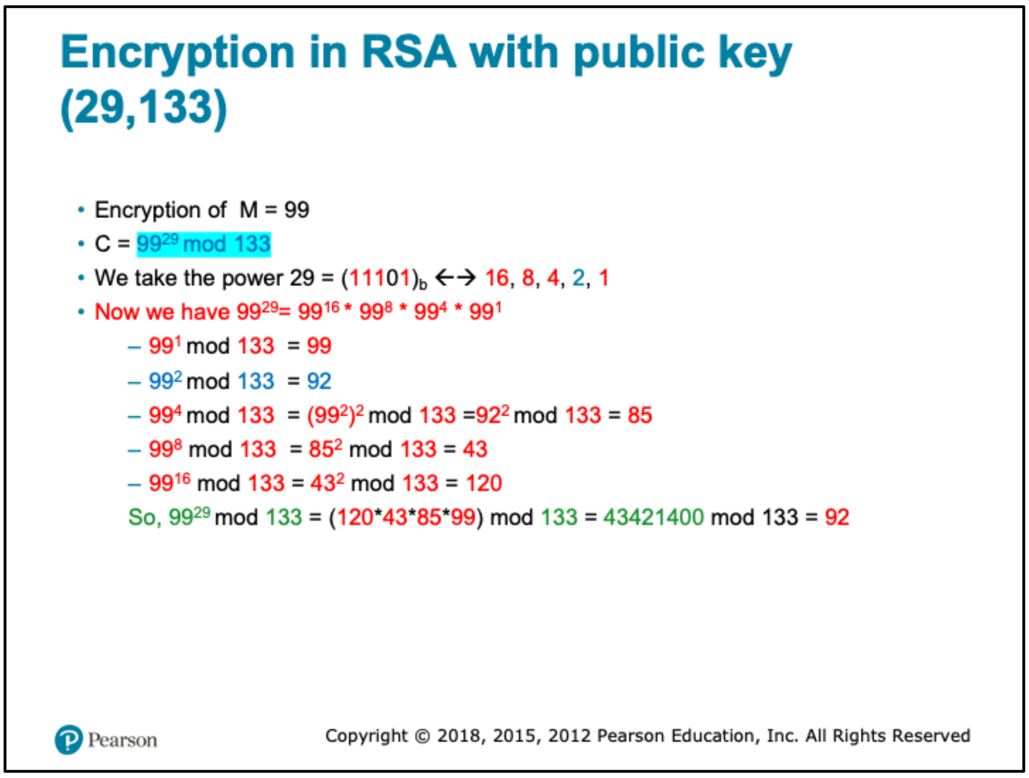


Calculating d using Extended Euclidean Algorithm

- **Key Generation**
  - Given $p = 7$, $q = 19$
  - $n = 7 * 19 = 133$
  - $\varphi(n) = (7-1) * (19-1) = 6 * 18 = 108$
  - We pick e such that $1 < e < 108$; and $\gcd(e, 108) = 1 \rightarrow e = 29$
  - Public key is (29, 133)
  - We now calculate d using Extended Euclidean Algorithm:
    $d = 29^{-1} \bmod 108$
    $d = 41$
  - To verify:
    $e.d = 1 \bmod \varphi(n)$
    $29*41 = 1 \bmod 108$
    $1189 \bmod 108 = 1$
  - Private key is (41,133)

```
Extended Euclidean Algorithm
1. Initialization: A= $\varphi(n)$, B=e, $T_0$=0, $T_1$=1
2. For the row calculate:
   Q= Quotient of A/B
   R= Remainder of A/B
   $T = T_1 - T_2 * Q$
3. if R=0 then
      d=$T_2$
      STOP
   else
      populate the next row:
         A=B   B=R   $T_1$ =T2   T2=T
4. Go back to step 2
```

Q	A	B	R	T_1	T_2	T
3	108	29	21	0	1	-3
1	29	21	8	1	-3	4
2	21	8	5	-3	4	-11
1	8	5	3	4	-11	15
1	5	3	2	-11	15	-26
1	3	2	1	15	-26	41
2	2	1	0	-26	41	-108
			STOP		d	

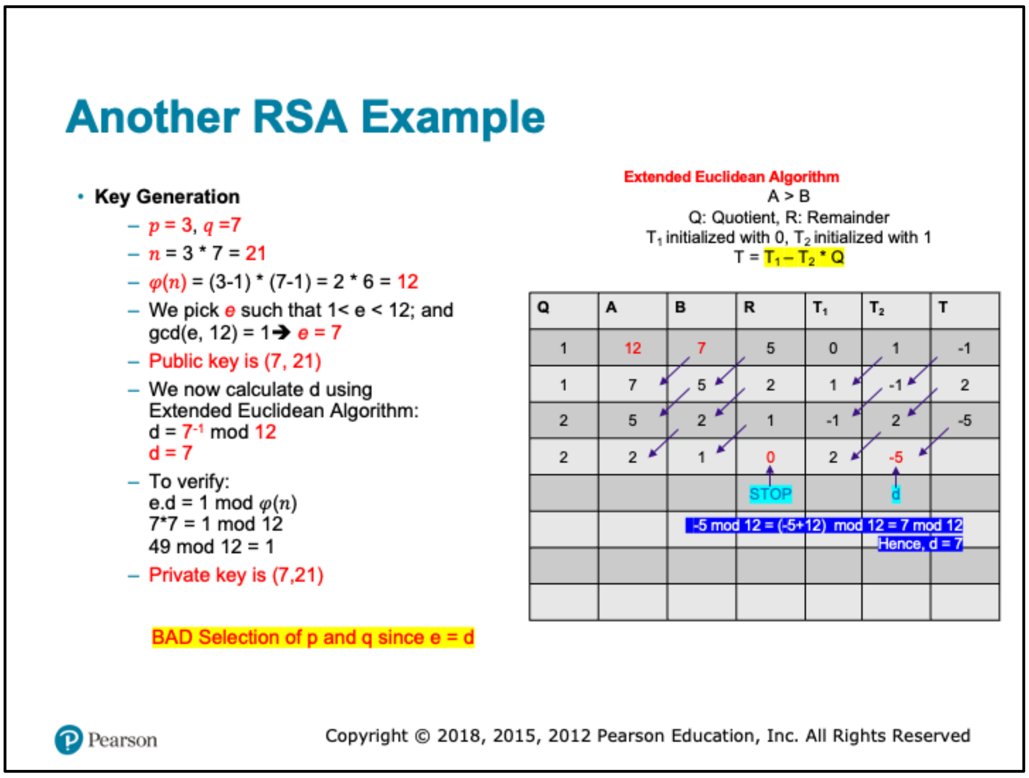


Encryption in RSA with public key (29,133)

- Encryption of $M = 99$
- $C = 99^{29} \bmod 133$
- We take the power $29 = (11101)_b \leftrightarrow 16, 8, 4, 2, 1$
- Now we have $99^{29} = 99^{16} * 99^{8} * 99^{4} * 99^{1}$
  - $99^{1} \bmod 133 = 99$
  - $99^{2} \bmod 133 = 92$
  - $99^{4} \bmod 133 = (99^{2})^{2} \bmod 133 = 92^{2} \bmod 133 = 85$
  - $99^{8} \bmod 133 = 85^{2} \bmod 133 = 43$
  - $99^{16} \bmod 133 = 43^{2} \bmod 133 = 120$

  So, $99^{29} \bmod 133 = (120*43*85*99) \bmod 133 = 43421400 \bmod 133 = 92$

Pearson



Another RSA Example

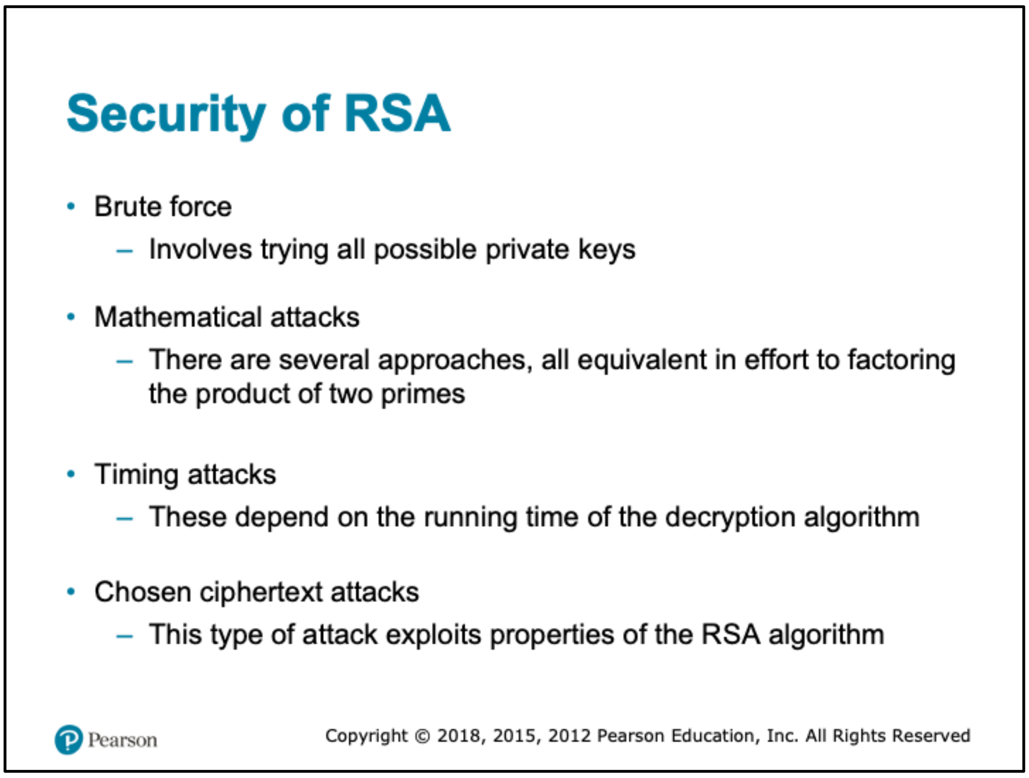
- **Key Generation**
  - p = 3, q =7
  - n = 3 * 7 = 21
  - $\varphi(n)$ = (3-1) * (7-1) = 2 * 6 = 12
  - We pick e such that 1< e < 12; and gcd(e, 12) = 1 → e = 7
  - Public key is (7, 21)
  - We now calculate d using Extended Euclidean Algorithm:
    d = 7^{-1} mod 12
    d = 7
  - To verify:
    e.d = 1 mod $\varphi(n)$
    7*7 = 1 mod 12
    49 mod 12 = 1
  - Private key is (7,21)

BAD Selection of p and q since e = d

Extended Euclidean Algorithm
A > B
Q: Quotient, R: Remainder
T_1 initialized with 0, T_2 initialized with 1
T = $T_1 - T_2$ * Q

Q	A	B	R	T_1	T_2	T
1	12	7	5	0	1	-1
1	7	5	2	1	-1	2
2	5	2	1	-1	2	-5
2	2	1	0	2	-5	
			STOP		d	
		-5 mod 12 = (-5+12) mod 12 = 7 mod 12 Hence, d = 7				

Pearson


Security of RSA

- Brute force
  - Involves trying all possible private keys
- Mathematical attacks
  - There are several approaches, all equivalent in effort to factoring the product of two primes
- Timing attacks
  - These depend on the running time of the decryption algorithm
- Chosen ciphertext attacks
  - This type of attack exploits properties of the RSA algorithm

Four possible approaches to attacking the RSA algorithm are:

• **Brute force:** This involves trying all possible private keys.

• **Mathematical attacks:** There are several approaches, all equivalent in effort to factoring the product of two primes.

• **Timing attacks:** These depend on the running time of the decryption algorithm.

• **Chosen ciphertext attacks:** This type of attack exploits properties of the RSA algorithm. A discussion of this attack is beyond the scope of this book.

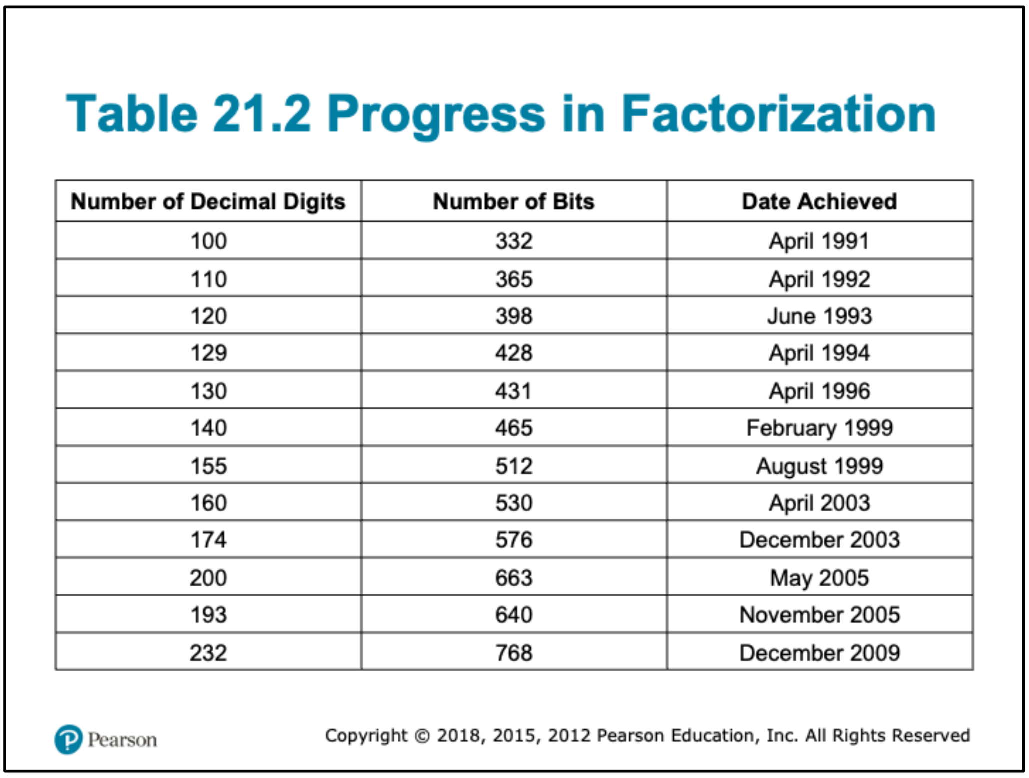


Table 21.2 Progress in Factorization

Number of Decimal Digits	Number of Bits	Date Achieved
100	332	April 1991
110	365	April 1992
120	398	June 1993
129	428	April 1994
130	431	April 1996
140	465	February 1999
155	512	August 1999
160	530	April 2003
174	576	December 2003
200	663	May 2005
193	640	November 2005
232	768	December 2009

For a large *n with large prime factors, factoring is a hard problem, but not* as hard as it used to be. Just as it had done for DES, RSA Laboratories issued challenges for the RSA cipher with key sizes of 100, 110, 120, and so on, digits. The latest challenge to be met is the RSA-200 challenge with a key length of 200 decimal digits, or about 663 bits. Table 21.2 shows the results to date. The level of effort is measured in MIPS-years: a million-instructions-per-second processor running for one year, which is about $3 * 10^{13}$ instructions executed (MIPS-year numbers not available for last 3 entries).

A striking fact about Table 21.2 concerns the method used. Until the mid-1990s, factoring attacks were made using an approach known as the quadratic sieve. The attack on RSA-130 used a newer algorithm, the generalized number field sieve (GNFS), and was able to factor a larger number than RSA-129 at only 20% of the computing effort.

The threat to larger key sizes is twofold: the continuing increase in computing power, and the continuing refinement of factoring algorithms. We have seen that the move to a different algorithm resulted in a tremendous speedup. We can expect further refinements in the GNFS, and the use of an even better algorithm is also a possibility. In fact, a related algorithm, the special number field sieve (SNFS), can factor numbers with a specialized form considerably faster than the generalized number field sieve. It is reasonable to expect a breakthrough that would enable a general factoring performance in about the same time as SNFS, or even better. Thus, we need to be careful in choosing a key size for RSA. For the near future, a key size in the range of 1024 to 2048 bits seems secure.

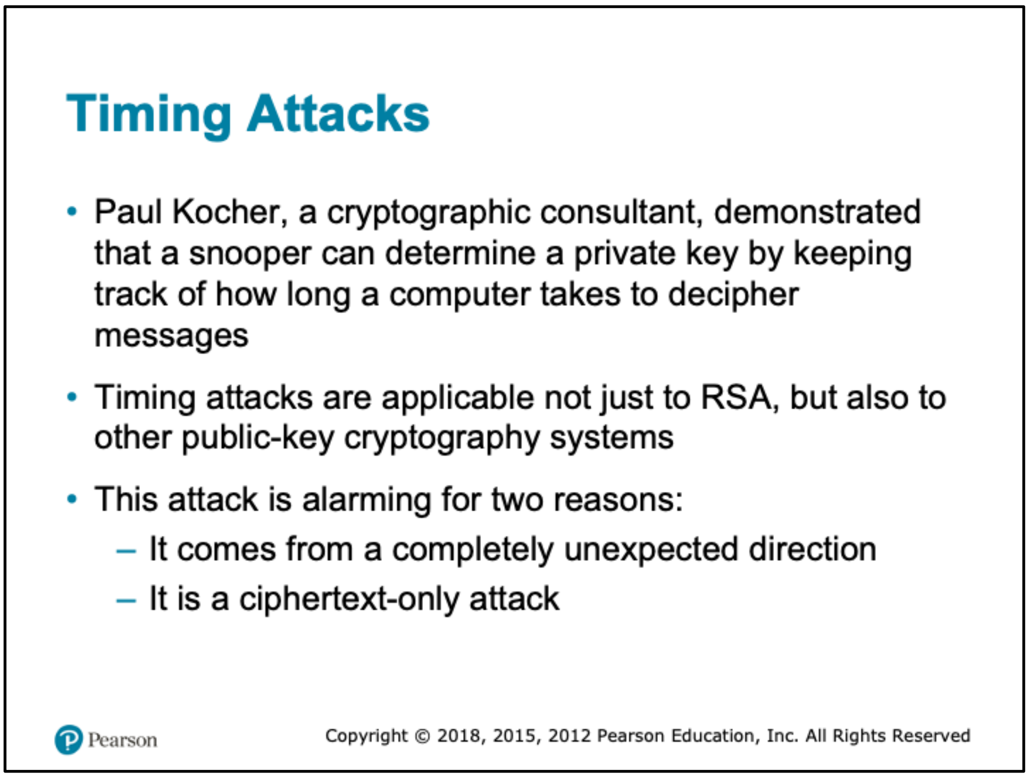
In addition to specifying the size of *n , a number of other constraints have been* suggested by researchers. To avoid values of *n that may be factored more easily, the* algorithm's inventors suggest the following constraints on *p and q :*

1. *p and q should differ in length by only a few digits. Thus, for a 1024-bit key* (309 decimal digits), both *p and q should be on the order of magnitude of* 10^{75} to 10^{100} .

2. Both (*p - 1) and (q - 1) should contain a large prime factor.*

3. gcd (*p - 1, q - 1) should be small.*

In addition, it has been demonstrated that if $e < n$ *and* $d < n^{1/4}$*, then d can be easily* determined [WIEN90].


Timing Attacks

- Paul Kocher, a cryptographic consultant, demonstrated that a snooper can determine a private key by keeping track of how long a computer takes to decipher messages
- Timing attacks are applicable not just to RSA, but also to other public-key cryptography systems
- This attack is alarming for two reasons:
  - It comes from a completely unexpected direction
  - It is a ciphertext-only attack

If one needed yet another lesson about how difficult it is to assess the security of a cryptographic algorithm, the appearance of timing attacks provides a stunning one. Paul Kocher, a cryptographic consultant, demonstrated that a snooper can determine a private key by keeping track of how long a computer takes to decipher messages [KOCH96]. Timing attacks are applicable not just to RSA, but also to other public-key cryptography systems. This attack is alarming for two reasons: It comes from a completely unexpected direction, and it is a ciphertext-only attack.

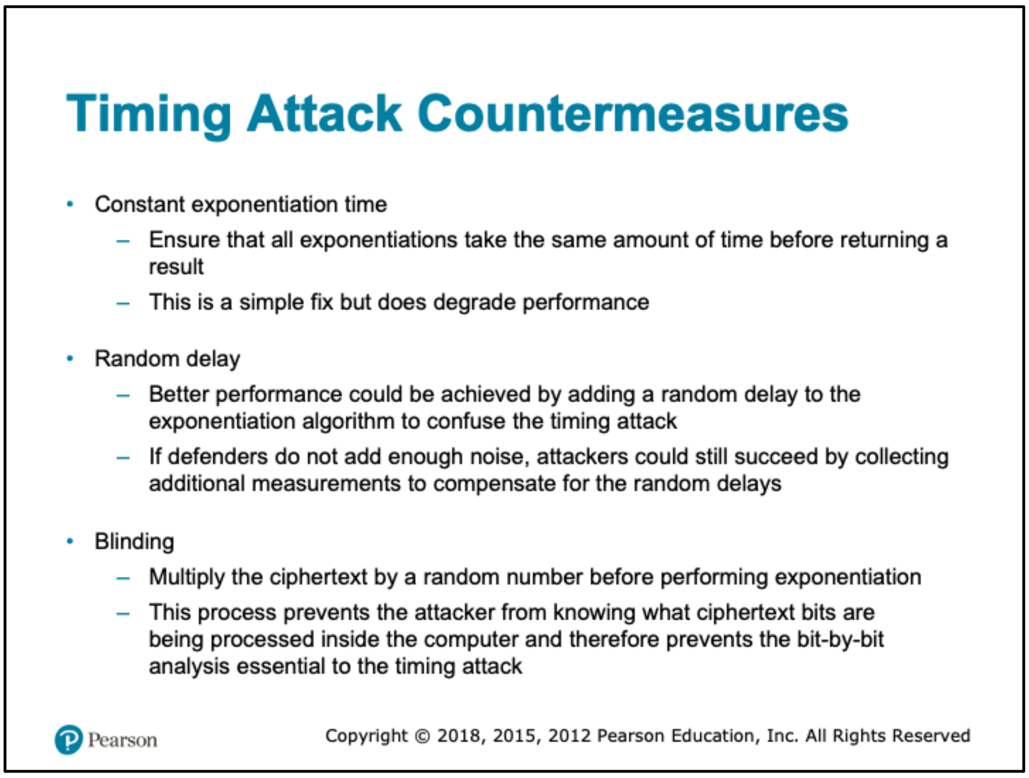
A timing attack is somewhat analogous to a burglar guessing the combination of a safe by observing how long it takes for someone to turn the dial from number to number. The attack exploits the common use of a modular exponentiation algorithm in RSA encryption and decryption, but the attack can be adapted to work with any implementation that does not run in fixed time. In the modular exponentiation algorithm, exponentiation is accomplished bit by bit, with one modular multiplication performed at each iteration and an additional modular multiplication performed for each 1 bit.

As Kocher points out in his paper, the attack is simplest to understand in an extreme case. Suppose the target system uses a modular multiplication function that is very fast in almost all cases but in a few cases takes much more time than an entire average modular exponentiation. The attack proceeds bit-by-bit starting with the leftmost bit,

b_k .

Therefore, if the observed time to execute the decryption algorithm is always slow when this particular iteration is slow with a 1 bit, then this bit is assumed to be 1. If a number of observed execution times for the entire algorithm are fast, then this bit is assumed to be 0.

In practice, modular exponentiation implementations do not have such extreme timing variations, in which the execution time of a single iteration can exceed the mean execution time of the entire algorithm. Nevertheless, there is enough variation to make this attack practical. For details, see [KOCH96].



Timing Attack Countermeasures

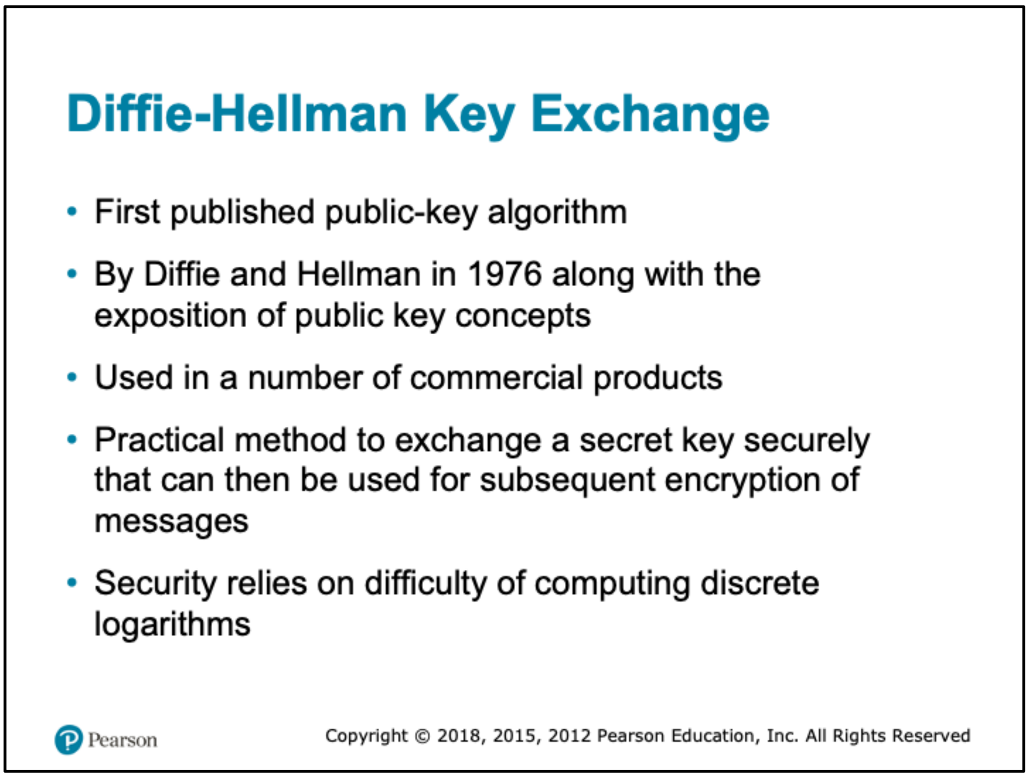
- Constant exponentiation time
  - Ensure that all exponentiations take the same amount of time before returning a result
  - This is a simple fix but does degrade performance
- Random delay
  - Better performance could be achieved by adding a random delay to the exponentiation algorithm to confuse the timing attack
  - If defenders do not add enough noise, attackers could still succeed by collecting additional measurements to compensate for the random delays
- Blinding
  - Multiply the ciphertext by a random number before performing exponentiation
  - This process prevents the attacker from knowing what ciphertext bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack

Pearson

• **Constant exponentiation time:** Ensure that all exponentiations take the same amount of time before returning a result. This is a simple fix but does degrade performance.

• **Random delay:** Better performance could be achieved by adding a random delay to the exponentiation algorithm to confuse the timing attack. Kocher points out that if defenders do not add enough noise, attackers could still succeed by collecting additional measurements to compensate for the random delays.

• **Blinding:** Multiply the ciphertext by a random number before performing exponentiation. This process prevents the attacker from knowing what ciphertext bits are being processed inside the computer and therefore prevents the bit-by-bit analysis essential to the timing attack.



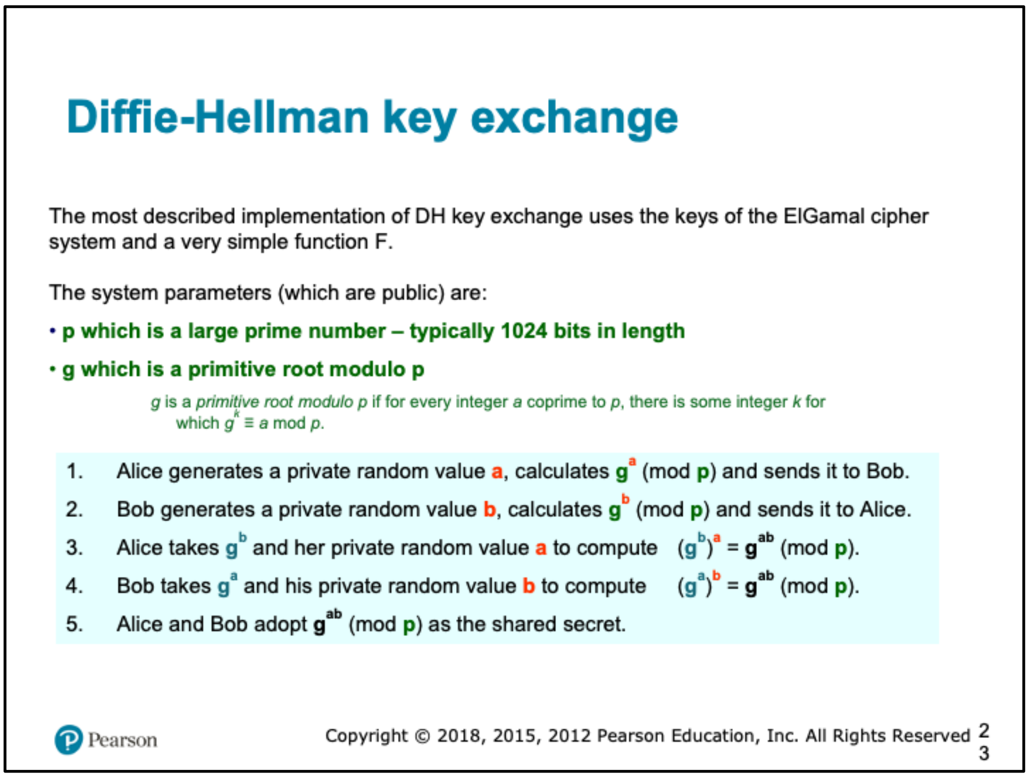
Diffie-Hellman Key Exchange

- First published public-key algorithm
- By Diffie and Hellman in 1976 along with the exposition of public key concepts
- Used in a number of commercial products
- Practical method to exchange a secret key securely that can then be used for subsequent encryption of messages
- Security relies on difficulty of computing discrete logarithms

The first published public-key algorithm appeared in the seminal paper by Diffie and Hellman that defined public-key cryptography [DIFF76] and is generally referred to as Diffie-Hellman key exchange. A number of commercial products employ this key exchange technique.

The purpose of the algorithm is to enable two users to exchange a secret key securely that can then be used for subsequent encryption of messages. The algorithm itself is limited to the exchange of the keys.

The Diffie-Hellman algorithm depends for its effectiveness on the difficulty of computing discrete logarithms.



Diffie-Hellman key exchange

The most described implementation of DH key exchange uses the keys of the ElGamal cipher system and a very simple function F.

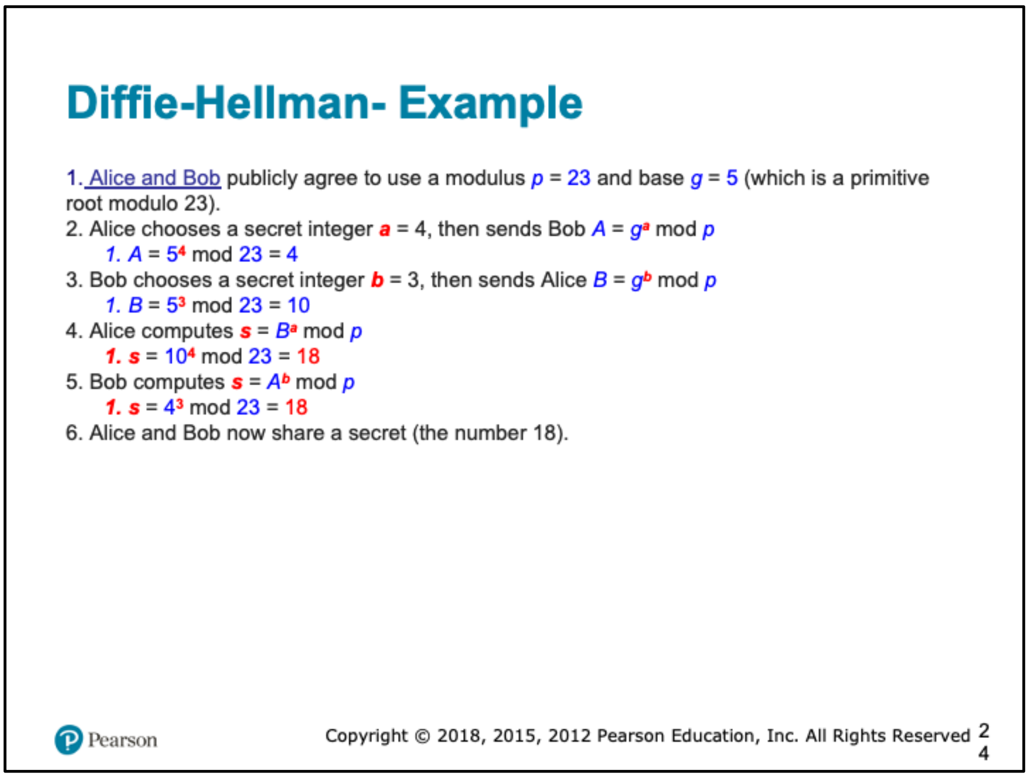
The system parameters (which are public) are:

- **p which is a large prime number – typically 1024 bits in length**
- **g which is a primitive root modulo p**

g is a *primitive root modulo p* if for every integer *a* coprime to *p*, there is some integer *k* for which $g^k \equiv a \bmod p$.

1. Alice generates a private random value **a**, calculates $\mathbf{g}^{a}$ (mod **p**) and sends it to Bob.
2. Bob generates a private random value **b**, calculates $\mathbf{g}^{b}$ (mod **p**) and sends it to Alice.
3. Alice takes $\mathbf{g}^{b}$ and her private random value **a** to compute $(g^{b})^{a} = \mathbf{g}^{ab}$ (mod **p**).
4. Bob takes $\mathbf{g}^{a}$ and his private random value **b** to compute $(g^{a})^{b} = \mathbf{g}^{ab}$ (mod **p**).
5. Alice and Bob adopt $\mathbf{g}^{ab}$ (mod **p**) as the shared secret.

Pearson



Diffie-Hellman- Example

1. Alice and Bob publicly agree to use a modulus p = 23 and base g = 5 (which is a primitive root modulo 23).
2. Alice chooses a secret integer **a** = 4, then sends Bob $A = g^a \bmod p$
    1. $A = 5^4 \bmod 23 = 4$
3. Bob chooses a secret integer **b** = 3, then sends Alice $B = g^b \bmod p$
    1. $B = 5^3 \bmod 23 = 10$
4. Alice computes **s** = $B^a \bmod p$
    1. $s = 10^4 \bmod 23 = 18$
5. Bob computes **s** = $A^b \bmod p$
    1. $s = 4^3 \bmod 23 = 18$
6. Alice and Bob now share a secret (the number 18).

Pearson

24

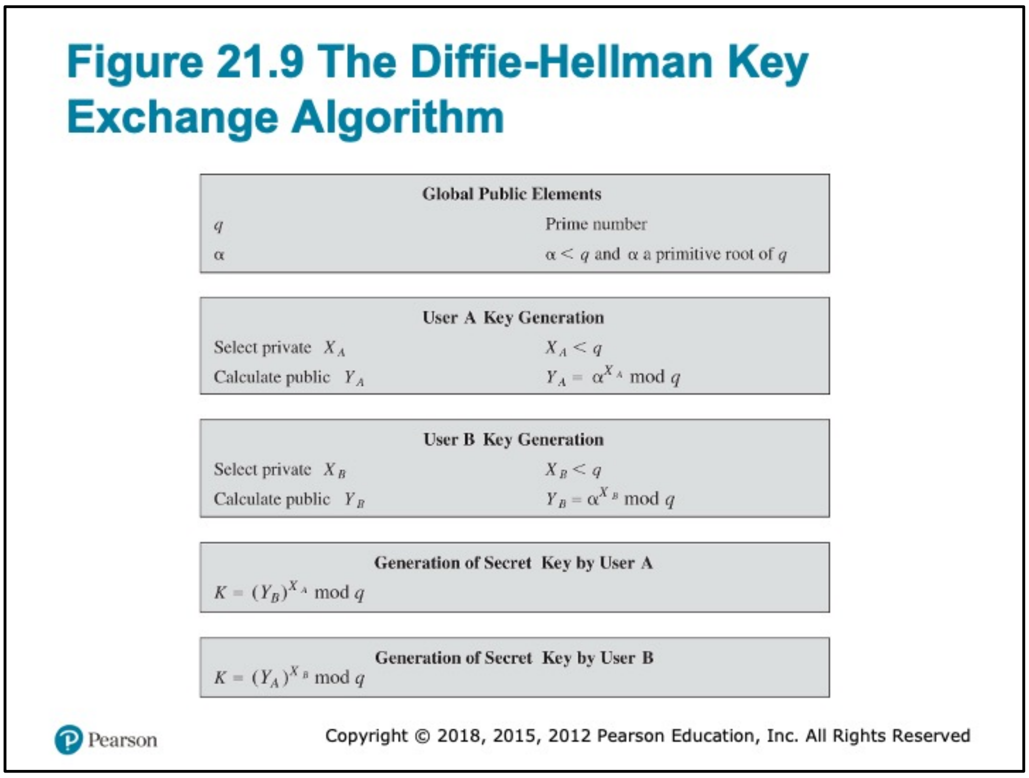


Figure 21.9 The Diffie-Hellman Key Exchange Algorithm

Global Public Elements

q	Prime number
α	$\alpha < q$ and α a primitive root of q

User A Key Generation

Select private X_A	$X_A < q$
Calculate public Y_A	$Y_A = \alpha^{X_A} \bmod q$

User B Key Generation

Select private X_B	$X_B < q$
Calculate public Y_B	$Y_B = \alpha^{X_B} \bmod q$

Generation of Secret Key by User A

$K = (Y_B)^{X_A} \bmod q$

Generation of Secret Key by User B

$K = (Y_A)^{X_B} \bmod q$

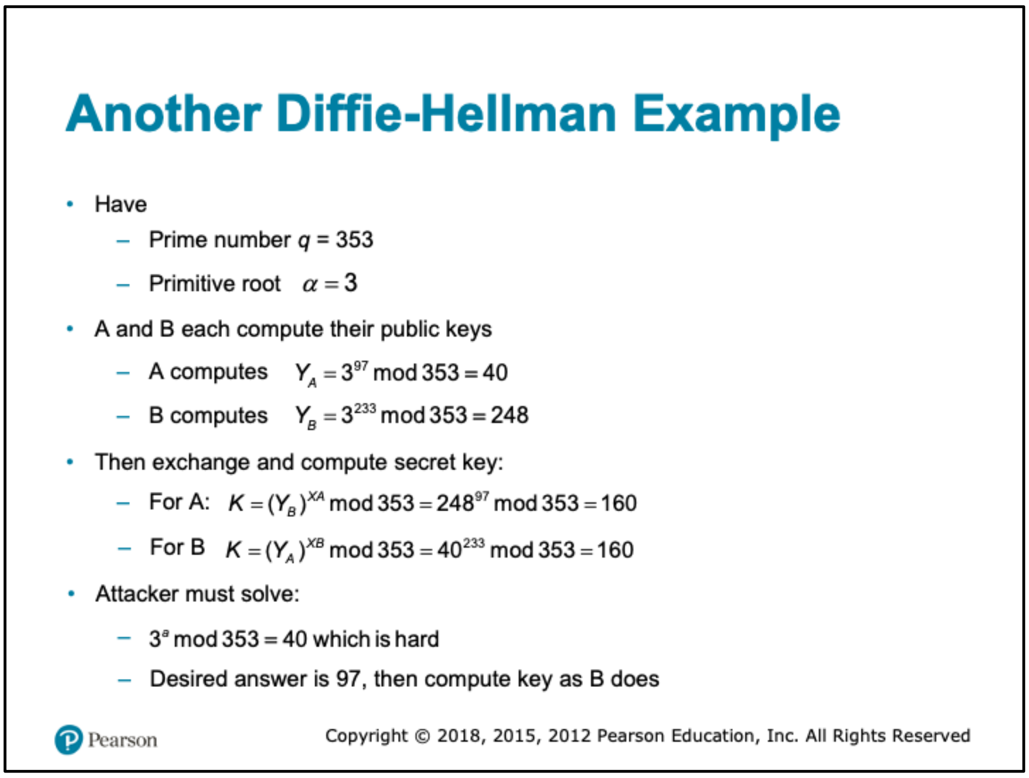
Pearson

The Diffie-Hellman key exchange algorithm is summarized in Figure 21.9. For this scheme, there are two publicly known numbers: a prime number q and an integer α that is a primitive root of q. Suppose the users A and B wish to exchange a key. User A selects a random integer $X_A < q$ and computes . Similarly, user B independently selects a random integer $X_B < q$ and computes . Each side keeps the X value private and makes the Y value available publicly to the other side. Users A and B compute the key as shown. These two calculations produce identical results, as shown in the text. The result is that the two sides have exchanged a secret value.

Furthermore, because X_A and X_B are private, an adversary only has the following ingredients to work with: q, α, Y_A, and Y_B. Thus, the adversary is forced to take a discrete logarithm to determine the key. For example, to determine the private key of user B, an adversary must compute: $X_B = \text{dlog}_{\alpha,q}(Y_B)$. The adversary can then calculate the key K in the same manner as user B calculates it.

The security of the Diffie-Hellman key exchange lies in the fact that, while it is relatively easy to calculate exponentials modulo a prime, it is very difficult to calculate discrete logarithms. For large primes, the latter task is considered infeasible.

Global public elements. q, prime number. alpha, alpha is less than q and alpha a primitive root of q. User A key generation. Select private X sub A, X sub A is less than q. Calculate public Y sub A, Y sub A = alpha to the X sub A power m o d q. User B key generation. Select private X sub B, X sub B is less than q. Calculate public Y sub B, Y sub B = alpha to the X sub B power m o d q. Generation of secret key by User A. k = left parenthesis Y sub B right parenthesis to the X sub A power m o d q. Generation of secret key by User B. k = left parenthesis Y sub A right parenthesis to the X sub B power m o d q.



Another Diffie-Hellman Example

- Have
  - Prime number q = 353
  - Primitive root $\alpha = 3$
- A and B each compute their public keys
  - A computes $Y_A = 3^{97} \bmod 353 = 40$
  - B computes $Y_B = 3^{233} \bmod 353 = 248$
- Then exchange and compute secret key:
  - For A: $K = (Y_B)^{XA} \bmod 353 = 248^{97} \bmod 353 = 160$
  - For B $K = (Y_A)^{XB} \bmod 353 = 40^{233} \bmod 353 = 160$
- Attacker must solve:
  - $3^a \bmod 353 = 40$ which is hard
  - Desired answer is 97, then compute key as B does

Pearson

The security of the Diffie-Hellman key exchange lies in the fact that, while it is relatively easy to calculate exponentials modulo a prime, it is very difficult to calculate discrete logarithms. For large primes, the latter task is considered infeasible

Here is an example. Key exchange is based on the use of the prime number q = 353 and a primitive root of 353, in this case α = 3. A and B select secret keys X_A = 97 and X_B = 233, respectively. Each computes its public key:

A computes $Y_A = 3^{97} \bmod 353 = 40$.
B computes $Y_B = 3^{233} \bmod 353 = 248$.

After they exchange public keys, each can compute the common secret key:

A computes $K = (Y_B)^{XA} \bmod 353 = 248^{97} \bmod 353 = 160$.
B computes $K = (Y_A)^{XB} \bmod 353 = 40^{233} \bmod 353 = 160$.

We assume an attacker would have available the following information:

$q = 353$; $\alpha = 3$; $Y_A = 40$; $Y_B = 248$

In this simple example, it would be possible by brute force to determine the secret key 160. In particular, an attacker E can determine the common key by discovering a solution to the equation 3^a mod 353 = 40 or the equation 3^b mod 353 = 248. The brute-force approach is to calculate powers of 3 modulo 353, stopping when the result equals either 40 or 248. The desired answer is reached with the exponent value of 97, which provides 3^{97} mod 353 = 40.

With larger numbers, the problem becomes impractical.

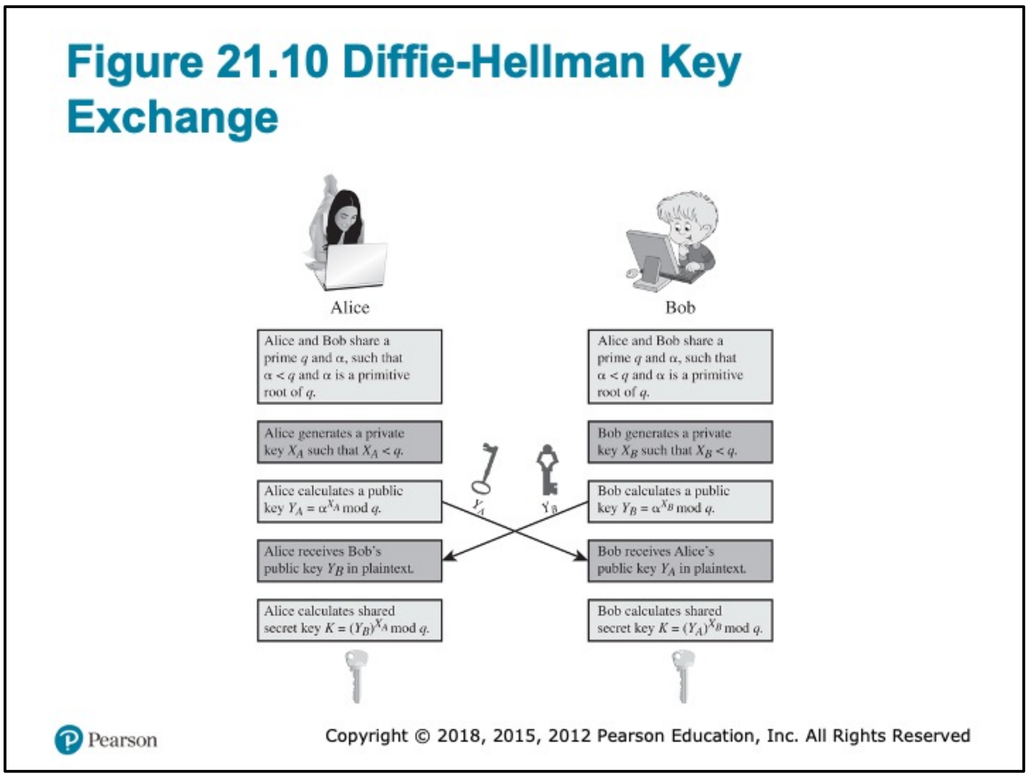


Figure 21.10 Diffie-Hellman Key Exchange

Alice

Bob

Alice and Bob share a prime q and α, such that $\alpha < q$ and α is a primitive root of q.

Alice and Bob share a prime q and α, such that $\alpha < q$ and α is a primitive root of q.

Alice generates a private key X_A such that $X_A < q$.

Bob generates a private key X_B such that $X_B < q$.

Alice calculates a public key $Y_A = \alpha^{X_A} \bmod q$.

Bob calculates a public key $Y_B = \alpha^{X_B} \bmod q$.

Y_A

Y_B

Alice receives Bob's public key Y_B in plaintext.

Bob receives Alice's public key Y_A in plaintext.

Alice calculates shared secret key $K = (Y_B)^{X_A} \bmod q$.

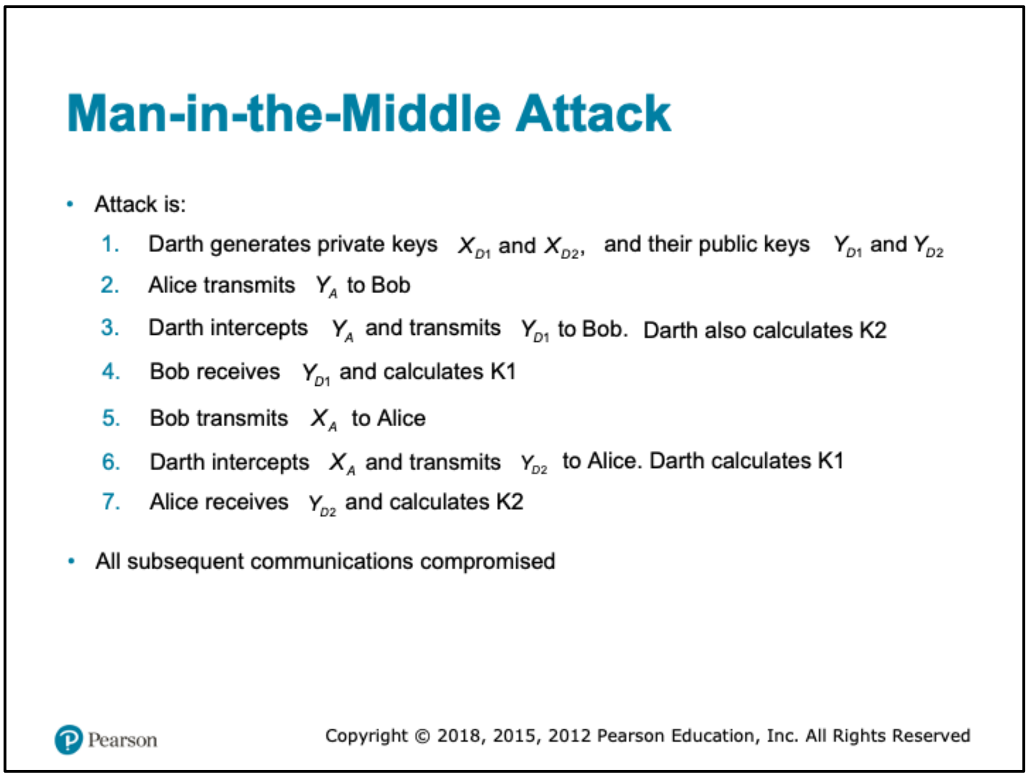
Bob calculates shared secret key $K = (Y_A)^{X_B} \bmod q$.

Pearson

Figure 21.10 shows a simple protocol that makes use of the Diffie-Hellman calculation. Suppose that user A wishes to set up a connection with user B and use a secret key to encrypt messages on that connection. User A can generate a one-time private key X_A, calculate Y_A, and send that to user B. User B responds by generating a private value X_B, calculating Y_B, and sending Y_B to user A. Both users can now calculate the key. The necessary public values q and α would need to be known ahead of time. Alternatively, user A could pick values for q and α and include those in the first message.

As an example of another use of the Diffie-Hellman algorithm, suppose that a group of users (e.g., all users on a LAN) each generate a long-lasting private value X_A and calculate a public value Y_A. These public values, together with global public values for q and α, are stored in some central directory. At any time, user B can access user A's public value, calculate a secret key, and use that to send an encrypted message to user A. If the central directory is trusted, then this form of communication provides both confidentiality and a degree of authentication. Because only A and B can determine the key, no other user can read the message (confidentiality). Recipient A knows that only user B could have created a message using this key (authentication). However, the technique does not protect against replay attacks.

Two users Alice and Bob share a prime q and alpha, such that alpha is less than q and alpha is a primitive root of q. Alice generates a private key X sub A such that X sub A is less than q. Bob generates a private key X sub B such that X sub B is less than q. Bob calculates a public key Y sub B = alpha to the X sub B power m o d q. Alice calculates a public key Y sub A = alpha to the X sub A power m o d q. The keys are exchanged. Alice recieves Bob's public key Y sub B in plaintext. Bob receives Alice's public key Y sub A in plaintext. Alice calculates shared secret key K = left parenthesis Y sub B right parenthesis to the X sub A power m o d q. Bob calculates shared secret key K = left parenthesis Y sub A right parenthesis X sub B m o d q.



Man-in-the-Middle Attack

- Attack is:
  1. Darth generates private keys X_{D1} and X_{D2}, and their public keys Y_{D1} and Y_{D2}
  2. Alice transmits Y_A to Bob
  3. Darth intercepts Y_A and transmits Y_{D1} to Bob. Darth also calculates K2
  4. Bob receives Y_{D1} and calculates K1
  5. Bob transmits X_A to Alice
  6. Darth intercepts X_A and transmits Y_{D2} to Alice. Darth calculates K1
  7. Alice receives Y_{D2} and calculates K2
- All subsequent communications compromised

Pearson

The protocol depicted in Figure 21.10 is insecure against a man-in-the-middle attack. Suppose Alice and Bob wish to exchange keys, and Darth is attacks as follows:

1. Darth generates two private keys X_{D1} and X_{D2}, and public keys Y_{D1} & Y_{D2}.

2. Alice transmits Y_A to Bob.

3. Darth intercepts Y_A and transmits Y_{D1} to Bob. Darth also calculates K2

4. Bob receives Y_{D1} and calculates K1.

5. Bob transmits X_A to Alice.

6. Darth intercepts X_A and transmits Y_{D2} to Alice. Darth calculates .

7. Alice receives Y_{D2} and calculates .

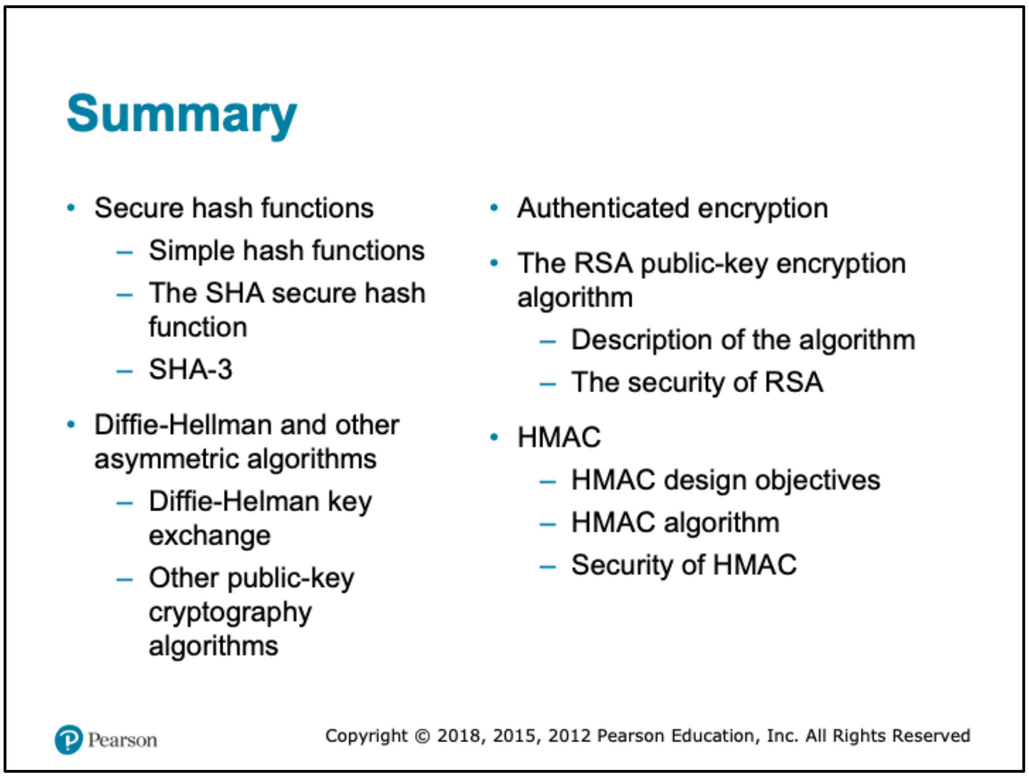
At this point, Bob and Alice think that they share a secret key, but instead Bob and Darth share secret key $K1$ and Alice and Darth share secret key $K2$. All future communication between Bob and Alice is compromised in the following way:

1. Alice sends an encrypted message *M*: E(*K*2, *M*).

2. Darth intercepts the encrypted message and decrypts it, to recover *M*.

3. Darth sends Bob E(*K*1, *M*) or E(*K*1, *M*'), where *M*' is any message. In the first case, Darth simply wants to eavesdrop on the communication without altering it. In the second case, Darth wants to modify the message going to Bob.

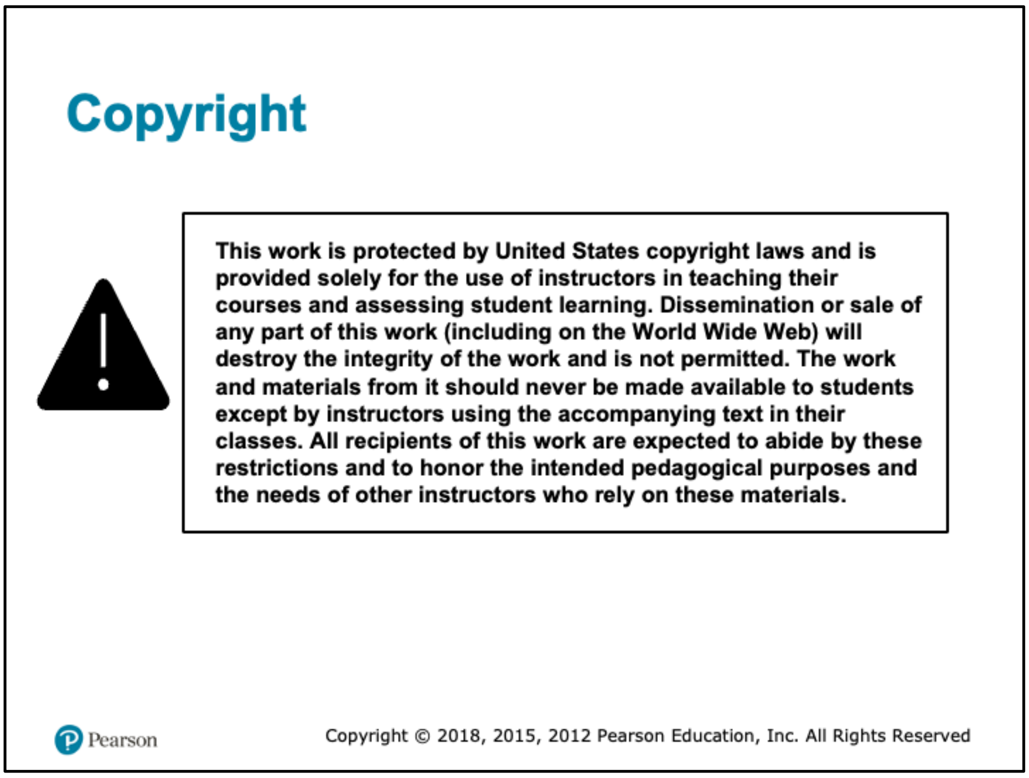
The key exchange protocol is vulnerable to such an attack because it does not authenticate the participants. This vulnerability can be overcome with the use of digital signatures and public-key certificates.



Summary

- Secure hash functions
  - Simple hash functions
  - The SHA secure hash function
  - SHA-3
- Diffie-Hellman and other asymmetric algorithms
  - Diffie-Helman key exchange
  - Other public-key cryptography algorithms
- Authenticated encryption
- The RSA public-key encryption algorithm
  - Description of the algorithm
  - The security of RSA
- HMAC
  - HMAC design objectives
  - HMAC algorithm
  - Security of HMAC

Chapter 21 summary.



Copyright