CS 6530 Applied Cryptography

July-Nov 2025

Introduction to Cryptography and Data Security

18th August 2025 – S1, S2, 19th August 2025

Source: Cryptography and Network Security – William Stallings

Dr. Manikantan Srinivasan

Cryptographic Hash Functions

Hash Functions

- condenses arbitrary message to fixed size

  `h = H(M)`
- usually assume hash function is public
- hash used to detect changes to message
- want a cryptographic hash function
  - computationally infeasible to find data mapping to specific hash (one-way property)
  - computationally infeasible to find two data to same hash (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. 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 (because no attack is significantly more efficient than brute force) to find either (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 (the collision-free property). Because of these characteristics, hash functions are often used to determine whether or not data has changed.

Cryptographic Hash Function

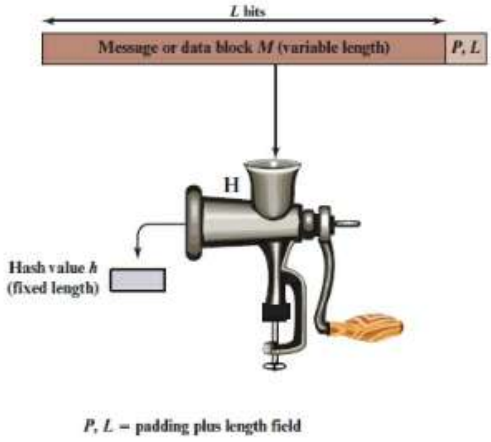
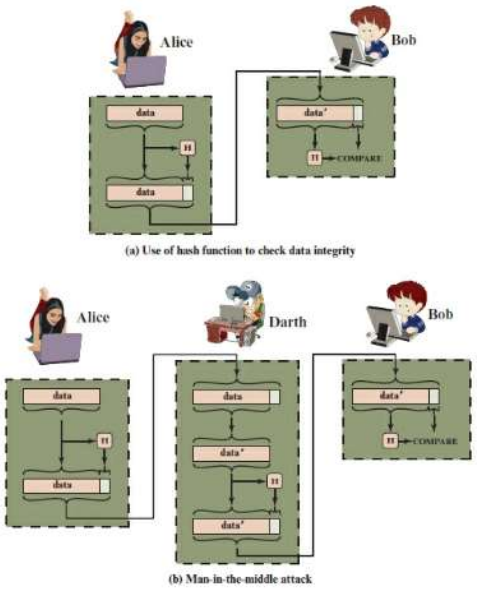


Figure 11.1 Cryptographic Hash Function; $h = \mathrm{H}(M)$

Stallings Figure 11.1 depicts the general operation of a cryptographic hash function. Typically, the input is padded out to an integer multiple of some fixed length (e.g., 1024 bits) and the padding includes the value of the length of the original message in bits. The length field is a security measure to increase the difficulty for an attacker to produce an alternative message with the same hash value.

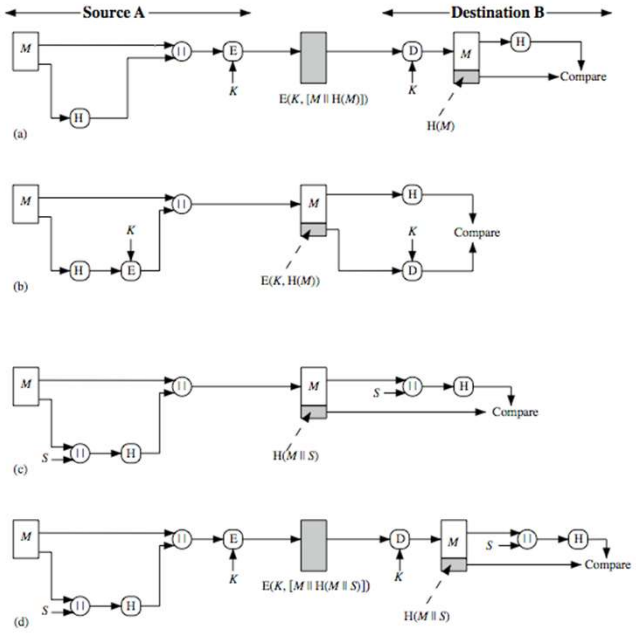
Cryptographic Hash Function



The hash value must be transmitted in a secure fashion. That is, the hash value must be protected so that if an adversary alters or replaces the message, it is not feasible for adversary to also alter the hash value to fool the receiver.

Use of hash function to check the integrity of the communication. Possible Man in the middle attack scenario.

Hash Functions & Message Authent-ication



Message authentication is a mechanism or service used to verify the integrity of a message, by assuring that the data received are exactly as sent. Stallings Figure 11.2 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows:

a.The message plus concatenated hash code is encrypted using symmetric encryption. Since only A and B share the secret key, the message must have come from A and has not been altered. The hash code provides the structure or redundancy required to achieve authentication.

b.Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden for those applications not requiring confidentiality.

c.Shows the use of a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value S. A computes the hash value over the concatenation of M and S and appends the resulting hash value to M. Because B possesses S, it can recompute the hash value to verify. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.

d.Confidentiality can be added to the approach of (c) by encrypting the entire message plus the hash code.

When confidentiality is not required, method (b) has an advantage over methods (a) and (d), which encrypts the entire message, in that less computation is required.

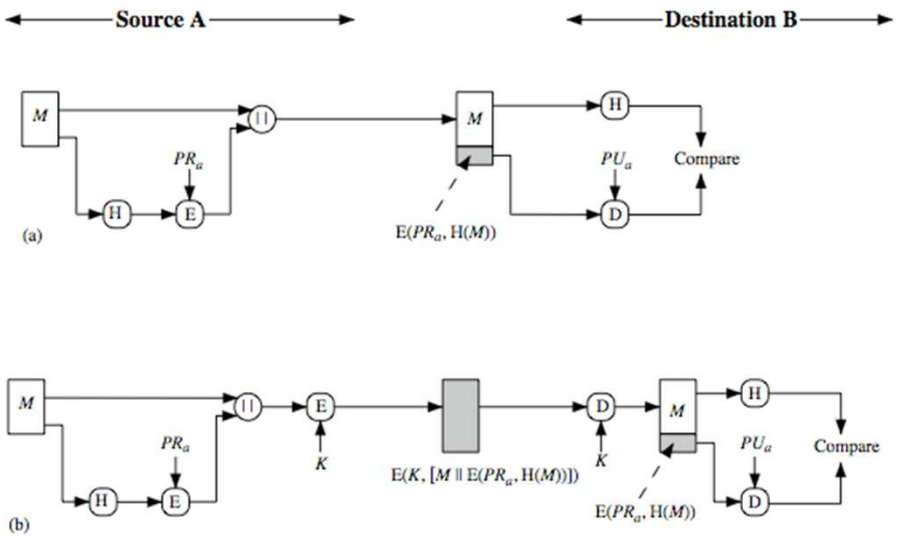
Message Authentication Code (MAC)

- Message authentication is achieved using a message authentication code (MAC), also known as a keyed hash function
- Typically, MACs are used between two parties that share a secret key to authenticate information exchanged between those parties.
- A MAC function takes as input a secret key and a data block and produces a hash value, referred to as the MAC, which is associated with the protected message.
- E(K, H(M)) is a function of a variable-length message M and a secret key K, and it produces a fixed-size output that is secure against an opponent who does not know the secret key.

Digital Signature

- The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, 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 that is associated with the digital signature. In this case, an attacker who wishes to alter the message would need to know the user's private key.
- The implications of digital signatures go beyond just message authentication.

Hash Functions & Digital Signatures



Another important application, which is similar to the message authentication application, is the digital signature. The operation of the digital signature is similar to that of the MAC. In the case of the digital signature, 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 that is associated with the digital signature. In this case an attacker who wishes to alter the message would need to know the user's private key. As we shall see in Chapter 14, the implications of digital signatures go beyond just message authentication.
Stallings Figure 11.3 illustrates, in a simplified fashion, how a hash code is used to provide a digital signature:
a.The hash code is encrypted, using public-key encryption and using the sender's private key. As with Figure 11.2b, this provides authentication. It also provides a digital signature, because only the sender could have produced the encrypted hash code. In fact, this is the essence of the digital signature technique.
b.If confidentiality as well as a digital signature is desired, then the message plus the private-key-encrypted hash code can be encrypted using a symmetric secret key. This is a common technique.

Other Hash Function Uses

- to create a one-way password file
  - store hash of password not actual password
- for intrusion detection and virus detection
  - keep & check hash of files on system
- pseudorandom function (PRF) or pseudorandom number generator (PRNG)

Hash functions are commonly used to create a one-way password file. Chapter 20 explains a scheme in which a hash of a password is stored by an operating system rather than the password itself. Thus, the actual password is not retrievable by a hacker who gains access to the password file. In simple terms, when a user enters a password, the hash of that password is compared to the stored hash value for verification. This approach to password protection is used by most operating systems.

Hash functions can be used for intrusion detection and virus detection. Store H(F) for each file on a system and secure the hash values (e.g., on a CD-R that is kept secure). One can later determine if a file has been modified by recomputing H(F). An intruder would need to change F without changing H(F).

A cryptographic hash function 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. We discuss this application in Chapter 12.

Two Simple Insecure Hash Functions

- consider two simple insecure hash functions
- bit-by-bit exclusive-OR (XOR) of every block
  - $C_i = b_{i1}$ *xor* b_{i2} *xor . . . xor* b_{im}
  - a longitudinal redundancy check
  - reasonably effective as data integrity check
- one-bit circular shift on hash value
  - for each successive *n-bit* block
    - rotate current hash value to left by1bit and XOR block
  - good for data integrity but useless for security

To get some feel for the security considerations involved in cryptographic hash functions, we present two simple, insecure hash functions in this section. One of the simplest hash functions is the bit-by-bit exclusive-OR (XOR) of every block, which can be expressed as shown. This operation 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}.

A simple way to improve matters is to perform a one-bit circular shift, or rotation, on the hash value after each block is processed. Although this second procedure provides a good measure of data integrity, it is virtually useless for data security when an encrypted hash code is used with a plaintext message. Given a message, it is an easy matter to produce a new message that yields that hash code: Simply prepare the desired alternate message and then append an *n-bit* block that forces the new message plus block to yield the

desired hash code.

A Definition

- For a hash value h = H(x), we say that x is the **preimage** of h. That is, x is a data block whose hash value, 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.
- A **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.

Hash functions are commonly used to create a one-way password file. Chapter 20 explains a scheme in which a hash of a password is stored by an operating system rather than the password itself. Thus, the actual password is not retrievable by a hacker who gains access to the password file. In simple terms, when a user enters a password, the hash of that password is compared to the stored hash value for verification. This approach to password protection is used by most operating systems.

Hash functions can be used for intrusion detection and virus detection. Store H(F) for each file on a system and secure the hash values (e.g., on a CD-R that is kept secure). One can later determine if a file has been modified by recomputing H(F). An intruder would need to change F without changing H(F).

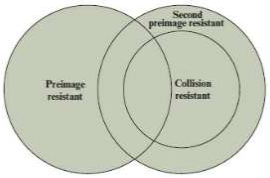
A cryptographic hash function can be used to construct a pseudorandom function (PRF) or a pseudorandom number generator (PRNG). A common application for a hash-based PRF is for the generation of symmetric keys. We discuss this application in Chapter 12.

Hash Function Requirements

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) with $x \neq y$, such that $H(x) = H(y)$.
Pseudorandomness	Output of H meets standard tests for pseudorandomness.

Stallings Table 11.1 lists the generally accepted requirements for a cryptographic hash function. The first three properties are requirements for the practical application of a hash function. The fourth property, preimage (for a hash value *h = H(x),* we say that x is the **preimage** of ***h***) resistant, is the one-way property: it is easy to generate a code given a message, but virtually impossible to generate a message given a code. This property is important if the authentication technique involves the use of a secret value (Figure 11.2c). The fifth property, second preimage resistant, guarantees that it is impossible to find an alternative message with the same hash value as a given message. This prevents forgery when an encrypted hash code is used (Figure 11.2b and Figure 11.3a). A hash function that satisfies the first five properties in Table 11.1 is referred to as a weak hash function. If the sixth property, collision resistant, is also satisfied, then it is referred to as a strong hash function. A strong hash function protects against an attack in which one party generates a message for another party to sign. The final requirement, **pseudorandomness**, has not traditionally been listed as a requirement of cryptographic hash functions, but is more or less implied.

Hash Function Requirements



Relationship Among Hash Function Properties

	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

Hash Function Resistance Properties Required for Various Data Integrity Applications

Stallings Table 11.1 lists the generally accepted requirements for a cryptographic hash function. The first three properties are requirements for the practical application of a hash function. The fourth property, preimage (for a hash value $h = H(x)$, we say that x is the **preimage** of ***h***) resistant, is the one-way property: it is easy to generate a code given a message, but virtually impossible to generate a message given a code. This property is important if the authentication technique involves the use of a secret value (Figure 11.2c). The fifth property, second preimage resistant, guarantees that it is impossible to find an alternative message with the same hash value as a given message. This prevents forgery when an encrypted hash code is used (Figure 11.2b and Figure 11.3a). A hash function that satisfies the first five properties in Table 11.1 is referred to as a weak hash function. If the sixth property, collision resistant, is also satisfied, then it is referred to as a strong hash function. A strong hash function protects against an attack in which one party generates a message for another party to sign. The final requirement, **pseudorandomness**, has not traditionally been listed as a requirement of cryptographic hash functions, but is more or less implied.

Secure Hash Algorithm

- SHA originally designed by NIST & NSA in 1993
- was revised in 1995 as SHA-1
- US standard for use with DSA signature scheme
  - standard is FIPS 180-1 1995, also Internet RFC3174
  - nb. the algorithm is SHA, the standard is SHS
- based on design of MD4 with key differences
- produces 160-bit hash values
- recent 2005 results on security of SHA-1 have raised concerns on its use in future applications

In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). The Secure Hash Algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993; a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as SHA-1. The actual standards document is entitled Secure Hash Standard. SHA is based on the hash function MD4 and its design closely models MD4. SHA-1 produces a hash value of 160 bits. In 2005, 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 newer, longer versions of SHA.

Revised Secure Hash Standard

- NIST issued revision FIPS 180-2 in 2002
- adds 3 additional versions of SHA
  - SHA-256, SHA-384, SHA-512
- designed for compatibility with increased security provided by the AES cipher
- structure & detail is similar to SHA-1
- hence analysis should be similar
- but security levels are rather higher

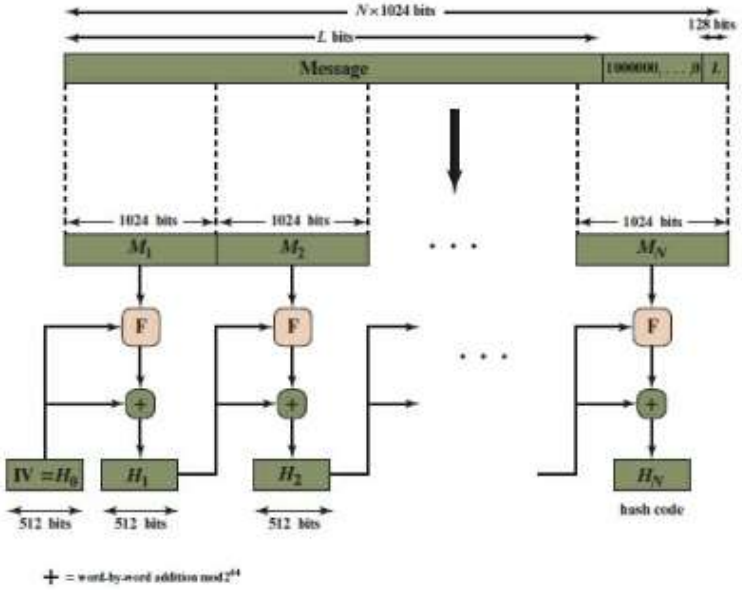
In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512. Collectively, these hash algorithms are known as SHA-2. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1, hence analyses should be similar. A revised document was issued as FIP PUB 180-3 in 2008, which added a 224-bit version. SHA-2 is also specified in RFC 4634, which essentially duplicates the material in FIPS 180-3, but adds a C code implementation.
In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on the other SHA versions by 2010.

SHA Versions

	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

Stallings Table 11.3 provides a comparison of the various parameters for the SHA hash functions.

SHA-512 Overview



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.
Step 1 Append padding bits.
Step 2 Append length.
Step 3 Initialize hash buffer.
Step 4 Process message in 1024-bit (128-byte) blocks.

Message Digest Generation Using SHA-512

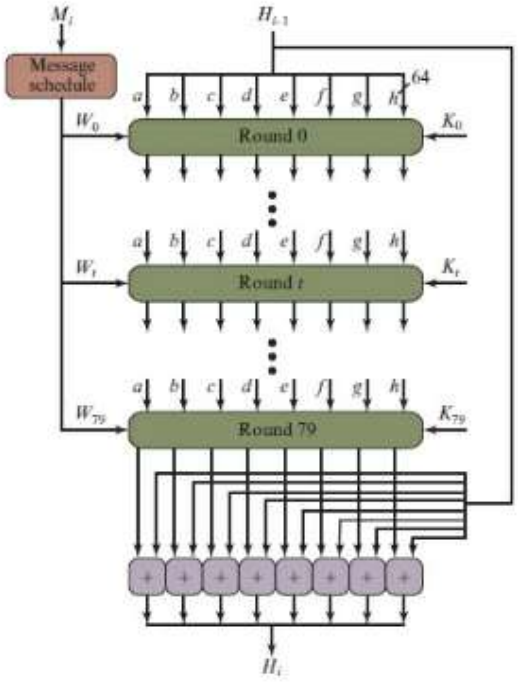
Now examine the structure of SHA-512, noting that the other versions are quite similar.
SHA-512 follows the structure depicted in Stallings Figure 11.8. The processing consists of the following steps:
• Step 1: Append padding bits, consists of a single 1-bit followed by the necessary number of 0-bits, so that its length is congruent to 896 modulo 1024
• Step 2: Append length as an (big-endian) unsigned 128-bit integer
• Step 3: Initialize hash buffer to a set of 64-bit integer constants (see text)
• Step 4: Process the message in 1024-bit (128-word) blocks, which forms the heart of the algorithm. Each round takes as input the 512-bit buffer value H_i, and updates the contents of that buffer.
• Step 5: Output the final state value as the resulting hash
See text for more details.

SHA-512 Compression Function

- heart of the algorithm
- processing message in 1024-bit blocks
- consists of 80 rounds
  - updating a 512-bit buffer
  - using a 64-bit value Wt derived from the current message block
  - and a round constant based on cube root of first 80 prime numbers

The SHA-512 Compression Function is the heart of the algorithm. In this Step 4, it processes the message in 1024-bit (128-word) blocks, using a module that consists of 80 rounds, labeled F in Stallings Figure 11.8, and is shown in detail in Figure 11.9. Each round takes as input the 512-bit buffer value, and updates the contents of the buffer. At input to the first round, the buffer has the value of the intermediate hash value. Each round *t* makes use of a 64-bit value *Wt* derived using a message schedule from the current 1024-bit block being processed. Each round also makes use of an additive constant *Kt*, based on 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 output of the eightieth round is added to the input to the first round to produce the final hash value for this message block, which forms the input to the next iteration of this compression function, as shown on the previous slide.

SHA-512 Overview



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.

SHA-512 Processing of a Single 1024-Bit Block

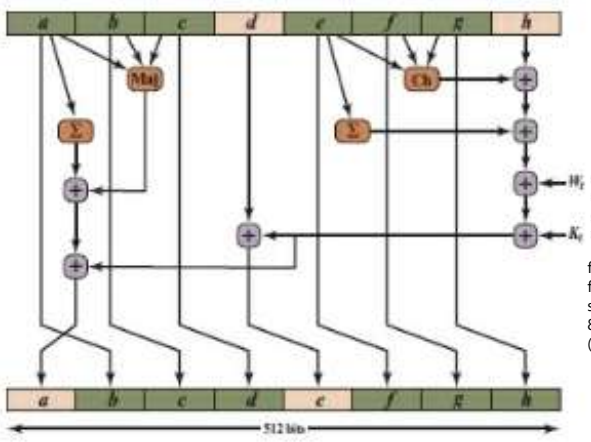
Now examine the structure of SHA-512, noting that the other versions are quite similar.
SHA-512 follows the structure depicted in Stallings Figure 11.8. The processing consists of the following steps:

- Step 1: Append padding bits, consists of a single 1-bit followed by the necessary number of 0-bits, so that its length is congruent to 896 modulo 1024
- Step 2: Append length as an (big-endian) unsigned 128-bit integer
- Step 3: Initialize hash buffer to a set of 64-bit integer constants (see text)
- Step 4: Process the message in 1024-bit (128-word) blocks, which forms the heart of the algorithm. Each round takes as input the 512-bit buffer value H_i, and updates the contents of that buffer.
- Step 5: Output the final state value as the resulting hash

See text for more details.

SHA-512 Round Function

Elementary SHA-512 Operation (single round)



Hash Buffer

Register a	Register b
Register c	Register d
Register e	Register f
Register g	Register h

Initialization Vector

a = 0x6A09E667F3BCC908	b = 0xBB67AE8584CAA73B
c = 0x3C6EF372FE94F82B	d = 0xA54FF53A5F1D36F1
e = 0x510E527FADE682D1	f = 0x9B05688C2B3E6C1F
g = 0x1F83D9ABFB41BD6B	h = 0x5BE0CD19137E2179

first 64 bits of the fractional parts of the square roots of the first 8 prime numbers (2,3,5,7,11,13,17,19)

Each round is defined by the following set of equations:

$$T_1 = h + \mathrm{Ch}(e,f,g) + (\sum_1^{512} e) + W_t + K_t$$
$$T_2 = (\sum_0^{512} a) + \mathrm{Maj}(a,b,c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

t = step number; $0 \le t \le 79$

$\mathrm{Ch}(e,f,g)$ = $(e \text{ AND } f) \oplus (\text{NOT } e \text{ AND } g)$
the conditional function: If e then f else g

$\mathrm{Maj}(a,b,c)$ = $(a \text{ AND } b) \oplus (a \text{ AND } c) \oplus (b \text{ AND } c)$
the function is true only of the majority (two or three) of the arguments are true

$(\Sigma_0^{512} a)$ = $\mathrm{ROTR}^{28}(a) \oplus \mathrm{ROTR}^{34}(a) \oplus \mathrm{ROTR}^{39}(a)$

$(\Sigma_1^{512} e)$ = $\mathrm{ROTR}^{14}(e) \oplus \mathrm{ROTR}^{18}(e) \oplus \mathrm{ROTR}^{41}(e)$

$\mathrm{ROTR}^n(x)$ = circular right shift (rotation) of the 64-bit argument x by n bits

K_t = first 64 bits from the fractional part of the cube roots of the first 80 prime numbers

W_t = a 64-bit word derived from the current 1024-bit input block

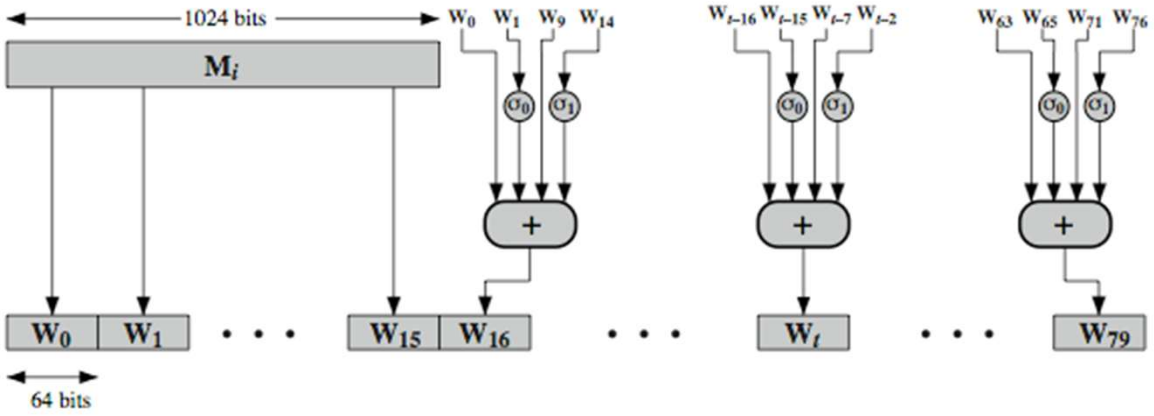
K_t = a 64-bit additive constant

+ = addition modulo 2^{64}

The structure of each of the 80 rounds is shown in Stallings Figure 11.10. Each 64-bit word is shuffled along one place, and in some cases manipulated using a series of simple logical functions (ANDs, NOTs, ORs, XORs, ROTates), in order to provide the avalanche & completeness properties of the hash function. The elements are:
Ch(e,f,g) = (e AND f) XOR (NOT e AND g)
Maj(a,b,c) = (a AND b) XOR (a AND c) XOR (b AND c)
∑(a) = ROTR(a,28) XOR ROTR(a,34) XOR ROTR(a,39)
∑(e) = ROTR(e,14) XOR ROTR(e,18) XOR ROTR(e,41)
+ = addition modulo 2^64
Kt = a 64-bit additive constant
Wt = a 64-bit word derived from the current 512-bit input block.

Six of the eight words of the output of the round function involve simply permutation (*b, c, d, f, g, h*) by means of rotation. This is indicated by shading in Figure 11.10. Only two of the output words (*a, e)* are generated by substitution. Word e is a function of input variables *d, e, f, g, h,* as well as the round word W t and the constant Kt. Word a is a function of all of the input variables, as well as the round word W t and the constant Kt.

SHA-512 Round Function



Wt are derived from the 1024-bit message. The first 16 values of Wt are taken directly from the 16 words of the current block. The remaining values are defined as a function of the earlier values using ROTates, SHIFTs and XORs as shown. The function elements are:
∂0(x) = ROTR(x,1) XOR ROTR(x,8) XOR SHR(x,7)
∂1(x) = ROTR(x,19) XOR ROTR(x,61) XOR SHR(x,6)

Stallings Figure 11.11 illustrates how the 64-bit word values Wt are derived from the 1024-bit message. The first 16 values of Wt are taken directly from the 16 words of the current block. The remaining values are defined as a function of the earlier values using ROTates, SHIFTs and XORs as shown. The function elements are:
∂0(x) = ROTR(x,1) XOR ROTR(x,8) XOR SHR(x,7)
∂1(x) = ROTR(x,19) XOR ROTR(x,61) XOR SHR(x,6)
Thus, in the first 16 steps of processing, the value of W_t is equal to the corresponding word in the message block. For the remaining 64 steps, the value of W_t consists of the circular left shift by one bit of the XOR of four of the preceding values of W_t, with two of those values subjected to shift and rotate operations. This introduces a great deal of redundancy and interdependence into the message blocks that are compressed, which complicates the task of finding a different message block that maps to the same compression function output.
See text for details of an example based on one in FIPS 180.

SHA-3

- SHA-1 is broken
  - indicated in 2011.
- SHA-2 (esp. SHA-512) seems secure
  - shares same structure and mathematical operations as predecessors so have concern
- NIST announced in 2007 a competition for the SHA-3 next gen NIST hash function
  - goal to have in place by 2012 but not fixed

SHA-1 is broken and is considered insecure and has been phased out for SHA-2.
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 decided to begin the process of developing a new hash standard. Accordingly, NIST announced in 2007 a competition to produce the next generation NIST hash function, to be called SHA-3. NIST would like to have a new standard in place by the end of 2012, but emphasizes that this is not a fixed timeline.

SHA-3 Requirements

- replace SHA-2 with SHA-3 in any use
  - so use same hash sizes
- preserve the online nature of SHA-2
  - so must process small blocks (512 / 1024 bits)
- evaluation criteria
  - security close to theoretical max for hash sizes
  - cost in time & memory
  - characteristics: such as flexibility & simplicity

The basic requirements that must be satisfied by any candidate for SHA-3 are:
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
Beyond these basic requirements, NIST has defined a set of evaluation criteria. These criteria are designed to reflect the requirements for the main applications supported by SHA-2, and are:
• Security: The strength of SHA-3 should be close to the theoretical maximum for the different required hash sizes, and for both preimage resistance and collision resistance. SHA-3 algorithms must be designed to resist any potentially successful attack on SHA-2 functions
• Cost: be both time and memory efficient over a range of hardware platforms.
• Algorithm and implementation characteristics: such as flexibility (e.g., tunable parameters for security/performance tradeoffs, opportunity for parallelization, and so on), and simplicity (which makes it easier to analyze the security properties of the algorithm)

Summary

- have considered:
  - hash functions
    - uses, requirements, security
  - hash functions based on block ciphers
  - SHA-1, SHA-2, SHA-3

Public Key Cryptography and RSA

Private-Key Cryptography

- traditional private/secret/single key cryptography uses one key
- shared by both sender and receiver
- if this key is disclosed communications are compromised
- also is symmetric, parties are equal
- hence does not protect sender from receiver forging a message & claiming is sent by sender

The development of public-key cryptography is the greatest and perhaps the only true revolution in the entire history of cryptography. From its earliest beginnings to modern times, virtually all cryptographic systems have been based on the elementary tools of substitution and permutation, and can be classed as private/secret/single key (symmetric) systems. All classical, and modern block and stream ciphers are of this form.

Public-Key Cryptography

- probably most significant advance in the 3000 year history of cryptography
- uses **two** keys – a public & a private key
- **asymmetric** since parties are **not** equal
- uses clever application of number theoretic concepts to function
- complements **rather than** replaces private key crypto

Will now discuss the radically different **public key** systems, in which **two keys** are used. Public-key cryptography provides a radical departure from all that has gone before. The development of public-key cryptography is the greatest and perhaps the only true revolution in the entire history of cryptography. It is asymmetric, involving the use of two separate keys, in contrast to symmetric encryption, that uses only one key. Anyone knowing the public key can encrypt messages or verify signatures, but **cannot** decrypt messages or create signatures, counter-intuitive though this may seem. The use of two keys has profound consequences in the areas of confidentiality, key distribution, and authentication. It works by the clever use of number theory problems that are easy one way but hard the other. Note that public key schemes are neither more nor less secure than private key (security depends on the key size for both), nor do they replace private key schemes (they are too slow to do so), rather they complement them. Both also have issues with key distribution, requiring the use of some suitable protocol.

Why Public-Key Cryptography?

- developed to address two key issues:
  - **key distribution** – how to have secure communications in general without having to trust a KDC with your key
  - **digital signatures** – how to verify a message comes intact from the claimed sender
- public invention due to Whitfield Diffie & Martin Hellman at Stanford Uni in 1976
  - known earlier in classified community

The concept of public-key cryptography evolved from an attempt to attack two of the most difficult problems associated with symmetric encryption: key distribution and digital signatures. The first problem is that of key distribution, which under symmetric encryption requires either (1) that two communicants already share a key, which somehow has been distributed to them; or (2) the use of a key distribution center. This seemed to negated the very essence of cryptography: the ability to maintain total secrecy over your own communication. The second was that of "digital signatures." If the use of cryptography was to become widespread, not just in military situations but for commercial and private purposes, then electronic messages and documents would need the equivalent of signatures used in paper documents.
The idea of public key schemes, and the first practical scheme, which was for key distribution only, was published in 1976 by Diffie & Hellman. The concept had been previously described in a classified report in 1970 by James Ellis (UK CESG) - and subsequently declassified [ELLI99]. Its interesting to note that they discovered RSA first, then Diffie-Hellman, opposite to the order of public discovery! There is also a claim that the NSA knew of the concept in the mid-60's [SIMM93].

Public-Key Cryptography

- **public-key/two-key/asymmetric** cryptography involves the use of **two** keys:
  - a **public-key**, which may be known by anybody, and can be used to **encrypt messages**, and **verify signatures**
  - a related **private-key**, known only to the recipient, used to **decrypt messages**, and **sign** (create) **signatures**
- **infeasible to determine private key from public**
- is **asymmetric** because
  - those who encrypt messages or verify signatures **cannot** decrypt messages or create signatures

Asymmetric algorithms rely on one key for encryption and a different but related key for decryption. These algorithms have the following important characteristic:

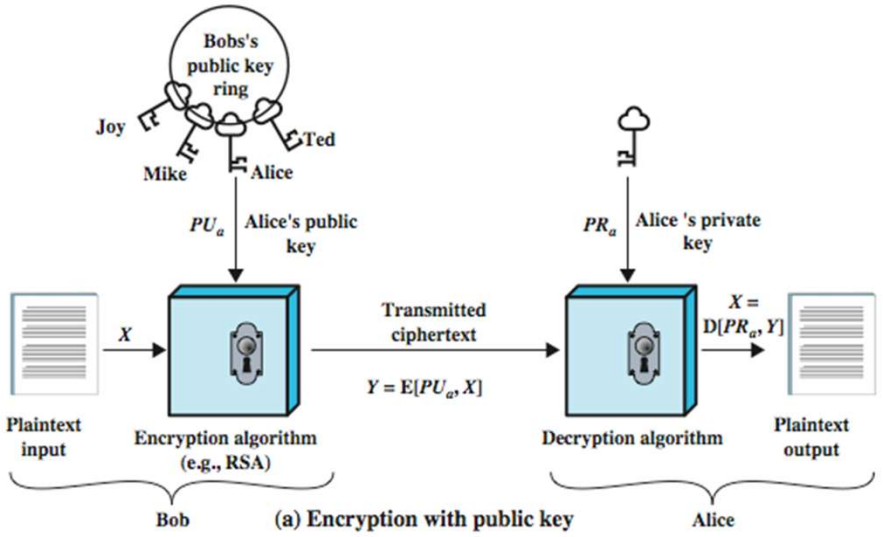
• It is computationally infeasible to determine the decryption key given only knowledge of the cryptographic algorithm and the encryption key.

In addition, some algorithms, such as RSA, also exhibit the following characteristic:

• Either of the two related keys can be used for encryption, with the other used for decryption.

Anyone knowing the public key can encrypt messages or verify signatures, but **cannot** decrypt messages or create signatures, thanks to some clever use of number theory.

Public-Key Cryptography



Stallings Figure 9.1a "Public-Key Cryptography", shows that a public-key encryption scheme has six ingredients:

• Plaintext: the readable message /data fed into the algorithm as input.

• Encryption algorithm: performs various transformations on the plaintext.

• Public and private keys: a pair of keys selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the algorithm depend on the public or private key that is provided as input.

• Ciphertext: the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.

• Decryption algorithm: accepts the ciphertext and matching key and produces the original plaintext.

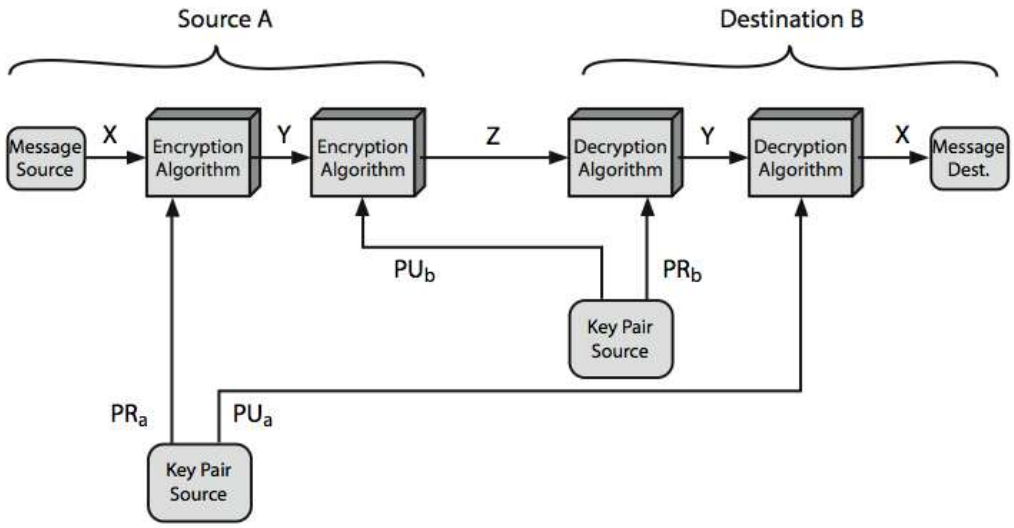
Consider the following analogy using padlocked boxes: traditional schemes involve the sender putting a message in a box and locking it, sending that to the receiver, and somehow securely also sending them the key to unlock the box. The radical advance in public key schemes was to turn this around, the receiver sends an **unlocked box** (their public key) to the sender, who puts the message in the box and locks it (easy - and having locked it cannot get at the message), and sends the locked box to the receiver who can unlock it (also easy), having the (private) key. An attacker would have to pick the lock on the box (hard).

Symmetric vs Public-Key

Conventional Encryption	Public-Key Encryption
Needed to Work: 1. The same algorithm with the same key is used for encryption and decryption. 2. The sender and receiver must share the algorithm and the key. *Needed for Security:* 1. The key must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.	*Needed to Work:* 1. One algorithm is used for encryption and decryption with a pair of keys, one for encryption and one for decryption. 2. The sender and receiver must each have one of the matched pair of keys (not the same one). *Needed for Security:* 1. One of the two keys must be kept secret. 2. It must be impossible or at least impractical to decipher a message if no other information is available. 3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.

Stallings Table 9.2 summarizes some of the important aspects of symmetric and public-key encryption. To discriminate between the two, we refer to the key used in symmetric encryption as a **secret key**. The two keys used for asymmetric encryption are referred to as the **public key** and the **private key.** Invariably, the private key is kept secret, but it is referred to as a private key rather than a secret key to avoid confusion with symmetric encryption.

Public-Key Cryptosystems



Stallings Figure 9.4 "Public-Key Cryptosystems: Secrecy and Authentication" illustrates the essential elements of a public-key encryption scheme.
Note that public-key schemes can be used for either secrecy or authentication, or both (as shown here). There is some source A that produces a message in plaintext X The M elements of X are letters in some finite alphabet. The message is intended for destination B. B generates a related pair of keys: a public key, PUb, and a private key, PRb. PRb is known only to B, whereas PUb is publicly available and therefore accessible by A. With the message X and the encryption key PUb as input, A forms the ciphertext $Y = E(PUb, X)$ The intended receiver, in possession of the matching private key, is able to invert the transformation: $X = D(PRb, Y)$ An adversary, observing Y and having access to PUb, but not having access to PRb or X, must attempt to recover X and/or PRb. This provides confidentiality. Can also use a public-key encryption to provide authentication: $Y = E(PRa, X);\ X = D(PUa, Y)$ To provide both the authentication function and confidentiality have a double use of the public-key scheme (as shown here): $Z = E(PUb, E(PRa, X))\ X = D(PUa, D(PRb, Z))$ In this case, separate key pairs are used for each of these purposes. The receiver owns and creates secrecy keys, sender owns and creates authentication keys.

In practice typically DO NOT do this, because of the computational cost of public-key schemes. Rather encrypt a session key which is then used with a block cipher to encrypt the actual message, and separately sign a hash of the message as a digital signature - this will be discussed more later.

Public-Key Applications

- can classify uses into 3 categories:
  - **encryption/decryption** (provide secrecy)
  - **digital signatures** (provide authentication)
  - **key exchange** (of session keys)
- some algorithms are suitable for all uses, others are specific to one

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Elliptic Curve	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Public-key systems are characterized by the use of a cryptographic type of algorithm with two keys. Depending on the application, the sender uses either the sender's private key or the receiver's public key, or both, to perform some type of cryptographic function. In broad terms, we can classify the use of public-key cryptosystems into the three categories:

• Encryption/decryption: The sender encrypts a message with the recipient's public key.

• Digital signature: The sender "signs" a message with its private key, either to the whole message or to a small block of data that is a function of the message.

• Key exchange: Two sides cooperate to exchange a session key. Several different approaches are possible, involving the private key(s) of one or both parties.

Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications. Stallings Table 9.3 (shown here) indicates the applications supported by the algorithms discussed in this book.

Public-Key Requirements

- Public-Key algorithms rely on two keys where:
  - it is computationally infeasible to find decryption key knowing only algorithm & encryption key
  - it is computationally easy to en/decrypt messages when the relevant (en/decrypt) key is known
  - either of the two related keys can be used for encryption, with the other used for decryption (for some algorithms)
- these are formidable requirements which only a few algorithms have satisfied

The cryptosystem illustrated in Figures 9.2 through 9.4 depends on a cryptographic algorithm based on two related keys. Diffie and Hellman postulated this system without demonstrating that such algorithms exist. However, they did lay out the conditions that such algorithms must fulfill:

1.It is computationally easy for a party B to generate a pair (public key *PUb,* private key PRb*).*

2.It is computationally easy for a sender A, knowing the public key and the message to be encrypted, *M*, to generate the corresponding ciphertext: C = E(PUb, M)

3.It is computationally easy for the receiver B to decrypt the resulting ciphertext using the private key to recover the original message: M = D(PRb, C) = D[PRb, E(PUb, M)

4.It is computationally infeasible for an adversary, knowing the public key, Pb, to determine the private key, PRb

5.It is computationally infeasible for an adversary, knowing the public key, Pb, and a ciphertext, C, to recover the original message, M.

6.(optional) The two keys can be applied in either order:

M = D[PU , E(PR, M)] = D[PR, E(PU, M)]

These are formidable requirements, as evidenced by the fact that only a few

algorithms (RSA, elliptic curve cryptography, Diffie-Hellman, DSS) have received widespread acceptance in the several decades since the concept of public-key cryptography was proposed.

Public-Key Requirements

- need a trapdoor one-way function
- one-way function has
  - $Y = f(X)$ easy
  - $X = f^{-1}(Y)$ infeasible
- a trap-door one-way function has
  - $Y = f_k(X)$ easy, if k and X are known
  - $X = f_k^{-1}(Y)$ easy, if k and Y are known
  - $X = f_k^{-1}(Y)$ infeasible, if Y known but k not known
- a practical public-key scheme depends on a suitable trap-door one-way function

The requirements boil down to the need for a trap-door one-way function. A one-way function is one that maps a domain into a range such that every function value has a unique inverse, with the condition that the calculation of the function is easy whereas the calculation of the inverse is infeasible:

$Y = f(X)$ easy

$X = f^{-1}(Y)$ infeasible

Generally, *easy* is defined to mean a problem that can be solved in polynomial time as a function of input length. The term *infeasible* is a much fuzzier concept. In general, we can say a problem

Now consider a trap-door one-way function, which is easy to calculate in one direction and infeasible to calculate in the other direction unless certain additional information is known. With the additional information the inverse can be calculated in polynomial time. We can summarize as follows: A trap-door one-way function is a family of invertible functions f_k, such that:

$Y = f_k(X)$ easy, if k and X are known

$X = f_k^{-1}(Y)$ easy, if k and Y are known

$X = f_k^{-1}(Y)$ infeasible, if Y known but k not known

Thus, the development of a practical public-key scheme depends on discovery of a suitable trap-door one-way function.

Security of Public Key Schemes

- like private key schemes brute force exhaustive search attack is always theoretically possible
- but keys used are too large (>512bits)
- security relies on a large enough difference in difficulty between easy (en/decrypt) and hard (cryptanalyse) problems
- more generally the hard problem is known, but is made hard enough to be impractical to break
- requires the use of very large numbers
- hence is slow compared to private key schemes

Public key schemes are no more or less secure than private key schemes - in both cases the size of the key determines the security. As with symmetric encryption, a public-key encryption scheme is vulnerable to a brute-force attack. The countermeasure is the same: Use large keys. However, there is a tradeoff to be considered. Public-key systems depend on the use of some sort of invertible mathematical function. The complexity of calculating these functions may not scale linearly with the number of bits in the key but grow more rapidly than that. Thus, the key size must be large enough to make brute-force attack impractical but small enough for practical encryption and decryption. In practice, the key sizes that have been proposed do make brute-force attack impractical but result in encryption/decryption speeds that are too slow for general-purpose use. Instead, as was mentioned earlier, public-key encryption is currently confined to key management and signature applications. Another form of attack is to find some way to compute the private key given the public key. To date, it has not been mathematically proven that this form of attack is infeasible for a particular public-key algorithm. Note also that you can't compare key sizes - a 64-bit private key scheme has very roughly similar security to a 512-bit RSA - both could be broken given sufficient resources. But with public key schemes at least there is usually a

firmer theoretical basis for determining the security since its based on well-known and well studied number theory problems.

RSA

- by Rivest, Shamir & Adleman of MIT in 1977
- best known & widely used public-key scheme
- based on exponentiation in a finite (Galois) field over integers modulo a prime
  - nb. exponentiation takes $O((\log n)^3)$ operations (easy)
- uses large integers (eg. 1024 bits)
- security due to cost of factoring large numbers
  - nb. factorization takes $O(e^{\log n \log \log n})$ operations (hard)

RSA is the best known, and by far the most widely used general public key encryption algorithm, and was first published by Rivest, Shamir & Adleman of MIT in 1978 [RIVE78]. The Rivest-Shamir-Adleman (RSA) scheme has since that time reigned supreme as the most widely accepted and implemented general-purpose approach to public-key encryption. It is based on exponentiation in a finite (Galois) field over integers modulo a prime, using large integers (eg. 1024 bits). Its security is due to the cost of factoring large numbers.

RSA En/decryption

- to encrypt a message M the sender:
  - obtains **public key** of recipient $PU=\{e,n\}$
  - computes: $C = M^e \bmod n$, where $0 \leq M < n$
- to decrypt the ciphertext C the owner:
  - uses their private key $PR=\{d,n\}$
  - computes: $M = C^d \bmod n$
- note that the message M must be smaller than the modulus n (block if needed)

The scheme developed by Rivest, Shamir, and Adleman makes use of an expression with exponentials. Plaintext is encrypted in blocks, with each block having a binary value less than some number n. The actual RSA encryption and decryption computations are each simply a single exponentiation mod (n). Both sender and receiver must know the value of n. The sender knows the value of e, and only the receiver knows the value of d. Thus, this is a public-key encryption algorithm with a public key of PU = {e, n} and a private key of PR = {d, n}. Note that the message must be smaller than the modulus. The "magic" is in the choice of the modulus and exponents which makes the system work.

RSA Key Setup

- each user generates a public/private key pair by:
- selecting two large primes at random: p, q
- computing their system modulus $n=p.q$
  - note $\phi(n)=(p-1)(q-1)$
- selecting at random the encryption key e
  - where $1<e<\phi(n)$, $\gcd(e,\phi(n))=1$
- solve following equation to find decryption key d
  - $e.d=1 \bmod \phi(n)$ and $0 \le d \le n$
- publish their public encryption key: PU={e,n}
- keep secret private decryption key: PR={d,n}

The required moduls and exponent values are chosen during key setup. RSA key setup is done once (rarely) when a user establishes (or replaces) their public key, using the steps as shown. The exponent e is usually fairly small, just must be relatively prime to ø(n). Need to compute its inverse mod ø(n) to find d. It is critically important that the factors p & q of the modulus n are kept secret, since if they become known, the system can be broken. Note that different users will have different moduli n.

Why RSA Works

- because of Euler's Theorem:
  - $a^{\phi(n)} \bmod n = 1$ where $\gcd(a,n)=1$
- in RSA have:
  - $n=p.q$
  - $\phi(n)=(p-1)(q-1)$
  - carefully chose e & d to be inverses $\bmod \phi(n)$
  - hence $e.d=1+k.\phi(n)$ for some k
- hence :

$$C^d = M^{e.d} = M^{1+k.\phi(n)} = M^1.(M^{\phi(n)})^k$$
$$= M^1.(1)^k = M^1 = M \bmod n$$

For this algorithm to be satisfactory for public-key encryption, it must be possible to find values of e, d, n such that $M^{ed} \bmod n = M$ for all $M < n$. We need to find a relationship of the form $M^{ed} \bmod n = M$ The preceding relationship holds if *e and d* are multiplicative inverses modulo $\phi(n)$, where $\phi(n)$ is the Euler totient function. This is a direct consequence of Euler's Theorem, so that raising a number to power e then d (or vica versa) results in the original number!

RSA Example - Key Setup

1. Select primes: $p=17$ & $q=11$
2. Calculate $n = pq = 17 \times 11 = 187$
3. Calculate $\phi(n) = (p-1)(q-1) = 16 \times 10 = 160$
4. Select e: $\gcd(e,160)=1$; choose $e=7$
5. Determine d: $de=1 \bmod 160$ and $d < 160$ Value is $d=23$ since $23 \times 7 = 161 = 1 \times 160 + 1$
6. Publish public key $PU=\{7,187\}$
7. Keep secret private key $PR=\{23,187\}$

Stallings provides an example of RSA key generation using “trivial” sized numbers.
Selecting primes requires the use of a primality test.
Finding d as inverse of e mod $\phi(n)$ requires use of Euclid’s Inverse algorithm (see Ch4)

RSA Example - En/Decryption

- sample RSA encryption/decryption is:
- given message M = 88 (nb. 88<187)
- encryption:
  - $C = 88^7 \bmod 187 = 11$
- decryption:
  - $M = 11^{23} \bmod 187 = 88$

Then show that the encryption and decryption operations are simple exponentiations mod 187.
Rather than having to laborious repeatedly multiply, can use the "square and multiply" algorithm with modulo reductions to implement all exponentiations quickly and efficiently (see next).

RSA Another Example

```
sage: p = random_prime(2^32); p
1103222539
sage: q = random_prime(2^32); q
17870599
sage: n = p*q; n
19715247602230861
sage: phi = (p-1)*(q-1); phi
19715246481137724
sage: e = random_prime(phi); e
13771927877214701
sage: d = xgcd(e, phi)[1]; d = mod(d, phi)
11417851791646385
sage: mod(d*e, phi)
1
```

Listing 10-1: Generating RSA parameters using SageMath

Example from Serious Cryptography book

Exponentiation

- can use the Square and Multiply Algorithm
- a fast, efficient algorithm for exponentiation
- concept is based on repeatedly squaring base
- and multiplying in the ones that are needed to compute the result
- look at binary representation of exponent
- only takes $O(\log_2 n)$ multiples for number n
  - eg. $7^5 = 7^4.7^1 = 3.7 = 10 \bmod 11$
  - eg. $3^{129} = 3^{128}.3^1 = 5.3 = 4 \bmod 11$

To perform the modular exponentiations, you can use the “Square and Multiply Algorithm”, a fast, efficient algorithm for doing exponentiation, which has a long history. The idea is to repeatedly square the base, and multiply in the ones that are needed to compute the result, as found by examining the binary representation of the exponent..

Exponentiation

```
c = 0; f = 1
for i = k downto 0
    do c = 2 x c
       f = (f x f) mod n
    if b_i == 1 then
       c = c + 1
       f = (f x a) mod n
 return f
```

State here one version of the “Square and Multiply Algorithm”, from Stallings Figure 9.8.

RSA Key Generation

- users of RSA must:
  - determine two primes at random - `p, q`
  - select either `e` or `d` and compute the other
- primes `p,q` must not be easily derived from modulus `n=p.q`
  - means must be sufficiently large
  - typically guess and use probabilistic test
- exponents `e, d` are inverses, so use Inverse algorithm to compute the other

Before the application of the public-key cryptosystem, each participant must generate a pair of keys, which requires finding primes and computing inverses. Both the prime generation and the derivation of a suitable pair of inverse exponents may involve trying a number of alternatives. Typically make random guesses for a possible p or q, and check using a probabalistic primality test whether the guessed number is indeed prime. If not, try again. Note that the prime number theorem shows that the average number of guesses needed is not too large. Then compute decryption exponent d using Euclid's Inverse Algorithm, which is quite efficient.

RSA Security

- possible approaches to attacking RSA are:
  - brute force key search - infeasible given size of numbers
  - mathematical attacks - based on difficulty of computing ø(n), by factoring modulus n
  - timing attacks - on running of decryption
  - chosen ciphertext attacks - given properties of RSA

Note some possible possible approaches to attacking the RSA algorithm, as shown.
The defense against the brute-force approach is the same for RSA as for other cryptosystems, namely, use a large key space. Thus the larger the number of bits in d, the better. However because the calculations involved both in key generation and in encryption/decryption are complex, the larger the size of the key, the slower the system will run.
Will now review the other possible types of attacks.

Factoring Problem

- mathematical approach takes 3 forms:
  - factor $n=p.q$, hence compute $\varnothing(n)$ and then d
  - determine $\varnothing(n)$ directly and compute d
  - find d directly
- currently believe all equivalent to factoring
  - have seen slow improvements over the years
    - as of May-05 best is 200 decimal digits (663) bit with LS
  - biggest improvement comes from improved algorithm
    - cf QS to GHFS to LS
  - currently assume 1024-2048 bit RSA is secure
    - ensure p, q of similar size and matching other constraints

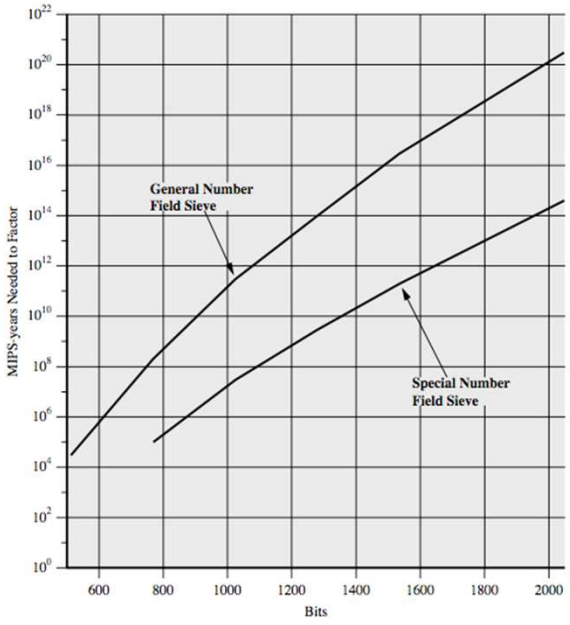
We can identify three approaches to attacking RSA mathematically, as shown. Mathematicians currently believe all equivalent to factoring.
See Stallings Table 9.4 (next slide) for progress in factoring, where see slow improvements over the years, with the biggest improvements coming from improved algorithms. The best current algorithm is the “Lattice Sieve” (LS), which replaced the “Generalized Number Field Sieve” (GNFS), which replaced the “Quadratic Sieve”(QS).
Have to assume computers will continue to get faster, and that better factoring algorithms may yet be found. 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 reasonable. 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 order 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.

Progress in Factoring

Number of Decimal Digits	Approximate Number of Bits	Date Achieved	MIPS-years	Algorithm
100	332	April 1991	7	quadratic sieve
110	365	April 1992	75	quadratic sieve
120	398	June 1993	830	quadratic sieve
129	428	April 1994	5000	quadratic sieve
130	431	April 1996	1000	generalized number field sieve
140	465	February 1999	2000	generalized number field sieve
155	512	August 1999	8000	generalized number field sieve
160	530	April 2003	—	Lattice sieve
174	576	December 2003	—	Lattice sieve
200	663	May 2005	—	Lattice sieve

Stallings Table 9.5 shows the progress in factoring 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 x 10^{13} instructions executed. A 1 GHz Pentium is about a 250-MIPS machine.

Progress in Factoring



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. Stallings Figure 9.9 compares the performance of the two algorithms. It is reasonable to expect a breakthrough that would enable a general factoring performance in about the same time as SNFS, or even better.

Timing Attacks

- developed by Paul Kocher in mid-1990's
- exploit timing variations in operations
  - eg. multiplying by small vs large number
  - or IF's varying which instructions executed
- infer operand size based on time taken
- RSA exploits time taken in exponentiation
- countermeasures
  - use constant exponentiation time
  - add random delays
  - blind values used in calculations

Have a radical new category of attacks developed by Paul Kocher in mid-1990's, based on observing how long it takes to compute the cryptographic operations. Timing attacks are applicable not just to RSA, but 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.
Although the timing attack is a serious threat, there are simple countermeasures that can be used, including using constant exponentiation time algorithms, adding random delays, or using blind values in calculations.

Chosen Ciphertext Attacks

- RSA is vulnerable to a Chosen Ciphertext Attack (CCA)
- attackers chooses ciphertexts & gets decrypted plaintext back
- choose ciphertext to exploit properties of RSA to provide info to help cryptanalysis
- can counter with random pad of plaintext
- or use Optimal Asymmetric Encryption Padding (OASP)

The RSA algorithm is vulnerable to a chosen ciphertext attack (CCA). CCA is defined as an attack in which adversary chooses a number of ciphertexts and is then given the corresponding plaintexts, decrypted with the target's private key. The adversary exploits properties of RSA and selects blocks of data that, when processed using the target's private key, yield information needed for cryptanalysis. Can counter simple attacks with random pad of plaintext. More sophisticated variants need to modify the plaintext using a procedure known as optimal asymmetric encryption padding (OAEP).

Summary

- have considered:
  - principles of public-key cryptography
  - RSA algorithm, implementation, security

RSA and public key cryptography summary

Diffie Hellman Key Exchange

Diffie-Hellman Key Exchange

- first public-key type scheme proposed
- by Diffie & Hellman in 1976 along with the exposition of public key concepts
  - note: now know that Williamson (UK CESG) secretly proposed the concept in 1970
- is a practical method for public exchange of a secret key
- used in a number of commercial products

This chapter continues our overview of public-key cryptography systems (PKCSs), and begins with a description of one of the earliest and simplest PKCS, Diffie-Hellman key exchange. This first published public-key algorithm appeared in the seminal paper by Diffie and Hellman that defined public-key cryptography [DIFF76b] and is generally referred to as Diffie-Hellman key exchange. The concept had been previously described in a classified report in 1970 by Williamson (UK CESG) - and subsequently declassified in 1987, see [ELLI99]. The purpose of the algorithm is to enable two users to securely exchange a key that can then be used for subsequent encryption of messages. The algorithm itself is limited to the exchange of secret values. A number of commercial products employ this key exchange technique.

Diffie-Hellman Key Exchange

- a public-key distribution scheme
  - cannot be used to exchange an arbitrary message
  - rather it can establish a common key
  - known only to the two participants
- value of key depends on the participants (and their private and public key information)
- based on exponentiation in a finite (Galois) field (modulo a prime or a polynomial) - easy
- security relies on the difficulty of computing discrete logarithms (similar to factoring) – hard

The purpose of the algorithm is to enable two users to securely exchange a key that can then be used for subsequent encryption of messages. The algorithm itself is limited to the exchange of secret values, which depends on the value of the public/private keys of the participants. The Diffie-Hellman algorithm uses exponentiation in a finite (Galois) field (modulo a prime or a polynomial), and depends for its effectiveness on the difficulty of computing discrete logarithms.

Diffie-Hellman Setup

- all users agree on global parameters:
  - large prime integer or polynomial q
  - a being a primitive root mod q
- each user (eg. A) generates their key
  - chooses a secret key (number): $x_A < q$
  - compute their public key: $Y_A = a^{x}_A \bmod q$
- each user makes public that key Y_A

In the Diffie-Hellman key exchange algorithm, there are two publicly known numbers: a prime number q and an integer a that is a primitive root of q. The prime q and primitive root a can be common to all using some instance of the D-H scheme. Note that the primitive root a is a number whose powers successively generate all the elements mod q. Users Alice and Bob choose random secrets x's, and then "protect" them using exponentiation to create their public y's. For an attacker monitoring the exchange of the y's to recover either of the x's, they'd need to solve the discrete logarithm problem, which is hard.

Diffie-Hellman Key Exchange

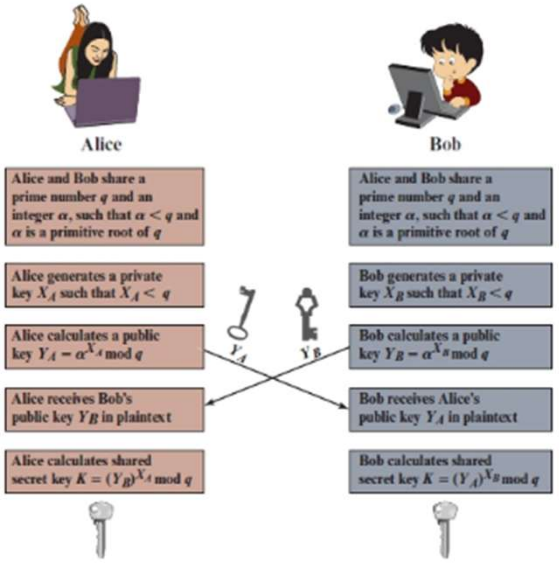
- shared session key for users A & B is KAB:
  - $K_{AB} = a^{x_A \cdot x_B} \bmod q$
  - $= Y_A^{x_B} \bmod q$ (which B can compute)
  - $= Y_B^{x_A} \bmod q$ (which A can compute)
- K_{AB} is used as session key in private-key encryption scheme between Alice and Bob
- if Alice and Bob subsequently communicate, they will have the same key as before, unless they choose new public-keys
- attacker needs an x, must solve discrete log

The actual key exchange for either party consists of raising the others "public key' to power of their private key. The resulting number (or as much of as is necessary) is used as the key for a block cipher or other private key scheme. For an attacker to obtain the same value they need at least one of the secret numbers, which means solving a discrete log, which is computationally infeasible given large enough numbers. Note that if Alice and Bob subsequently communicate, they will have the **same** key as before, unless they choose new public-keys.

Key Exchange Protocols

- users could create random private/public D-H keys each time they communicate
- users could create a known private/public D-H key and publish in a directory, then consulted and used to securely communicate with them
- both of these are vulnerable to a Man-in-the-Middle Attack
- authentication of the keys is needed

Key Exchange Protocols



Now consider 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 *a* would need to be known ahead of time. Alternatively, user *A* could pick values for *q* and *a* and include those in the first message.

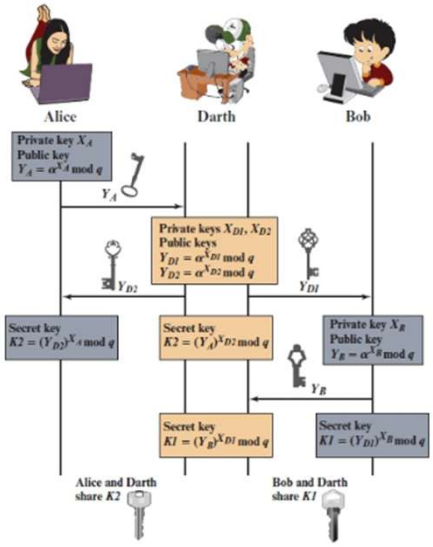
Diffie-Hellman Example

- users Alice & Bob who wish to swap keys:
- agree on prime q=353 and a=3
- select random secret keys:
  - A chooses x_A=97, B chooses x_B=233
- compute respective public keys:
  - $y_A=3^{97} \bmod 353 = 40$ (Alice)
  - $y_B=3^{233} \bmod 353 = 248$ (Bob)
- compute shared session key as:
  - $K_{AB}= y_B^{x_A} \bmod 353 = 248^{97} = 160$ (Alice)
  - $K_{AB}= y_A^{x_B} \bmod 353 = 40^{233} = 160$ (Bob)

Here is an example of Diffie-Hellman from the text using prime q=353, showing how each computes its public key, and then how after they exchange public keys, each can compute the common secret key.I n 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 $3^a \bmod 353 = 40$ or the equation $3^b \bmod 353 = 248$. 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 $3^{97} \bmod 353 = 40$. With larger numbers, the problem becomes impractical.

Man-in-the-Middle Attack

- Darth prepares by creating two private / public keys
- Alice transmits her public key to Bob
- Darth intercepts this and transmits his first public key to Bob. Darth also calculates a shared key with Alice
- Bob receives the public key and calculates the shared key (with Darth instead of Alice)
- Bob transmits his public key to Alice
- Darth intercepts this and transmits his second public key to Alice. Darth calculates a shared key with Bob
- Alice receives the key and calculates the shared key (with Darth instead of Bob)
- Darth can then intercept, decrypt, re-encrypt, forward all messages between Alice & Bob



The protocol described on the previous slide is insecure against a man-in-the-middle attack. Suppose Alice and Bob wish to exchange keys, and Darth is the adversary. The attack proceeds as follows:

1.Darth prepares for the attack by generating two random private keys X_{D1} and X_{D2} and then computing the corresponding 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 = $(Y_A)^{\wedge} X_{D2} \bmod q$

4.Bob receives Y_{D1} and calculates K1=$(Y_{D1})^{\wedge} X_B \bmod q$

5.Bob transmits Y_B to Alice.

6.Darth intercepts Y_B and transmits Y_{D2} to Alice. Darth calculates K1=$(Y_B)^{\wedge} X_{D1} \bmod q$

7.Alice receives Y_{D2} and calculates K2=$(Y_{D2})^{\wedge} X_A \bmod q$.

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(K2, M).

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

3.Darth sends Bob E(K1, M) or E(K1, 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.

Thank you – Q&A