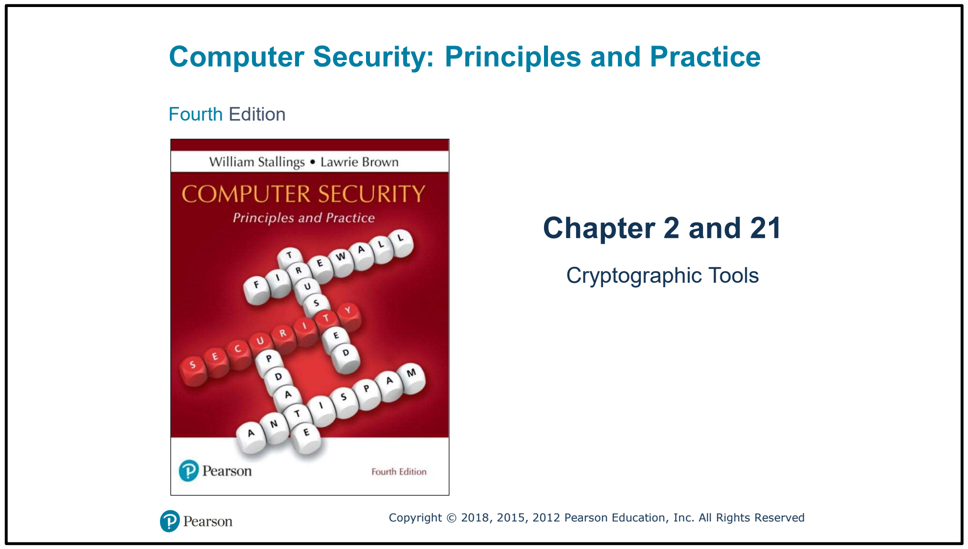
ECE 4309

Basics of cryptography – part 3 - Public crytpography

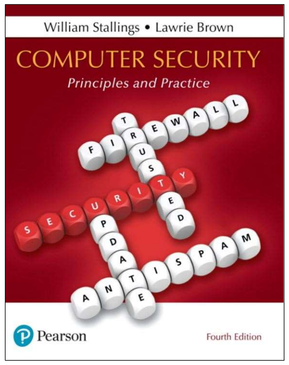
Dr. Valerio Formicola

CalPolyPomona



Computer Security: Principles and Practice

Fourth Edition



Chapter 2 and 21

Cryptographic Tools

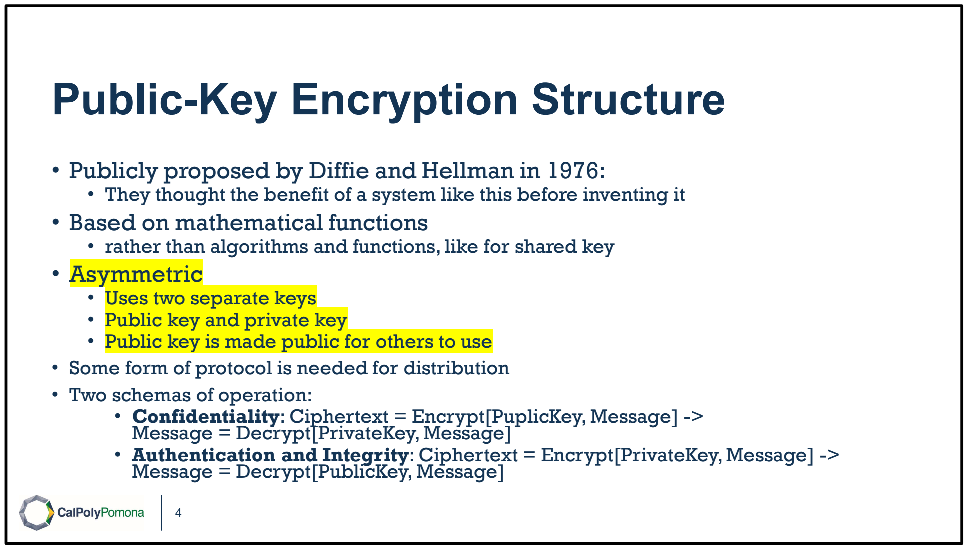
If this PowerPoint presentation contains mathematical equations, you may need to check that your computer has the following installed:
1) MathType Plugin
2) Math Player (free versions available)
3) NVDA Reader (free versions available)

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

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

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

Public key encryption



Public-Key Encryption Structure

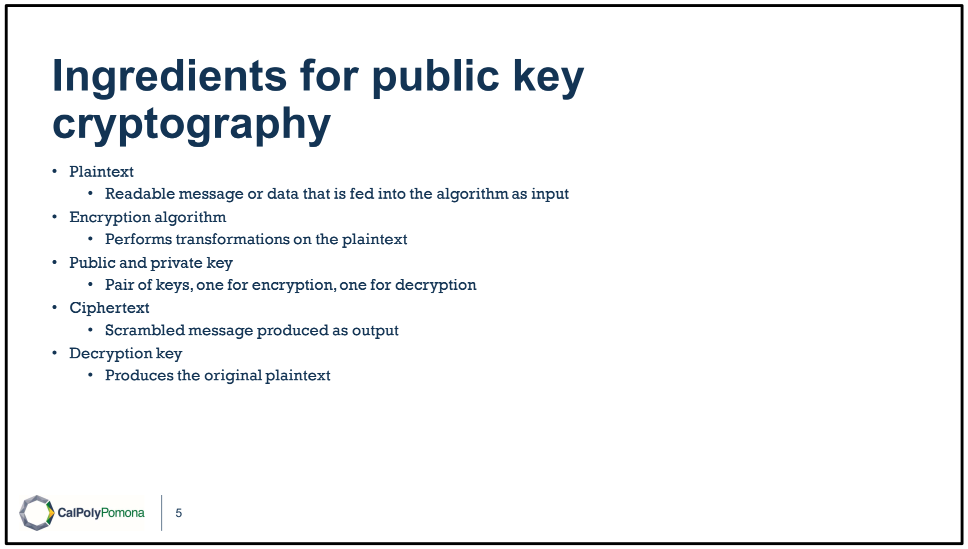
- Publicly proposed by Diffie and Hellman in 1976:
  - They thought the benefit of a system like this before inventing it
- Based on mathematical functions
  - rather than algorithms and functions, like for shared key
- Asymmetric
  - Uses two separate keys
  - Public key and private key
  - Public key is made public for others to use
- Some form of protocol is needed for distribution
- Two schemas of operation:
  - **Confidentiality**: Ciphertext = Encrypt[PuplicKey, Message] -> Message = Decrypt[PrivateKey, Message]
  - **Authentication and Integrity**: Ciphertext = Encrypt[PrivateKey, Message] -> Message = Decrypt[PublicKey, Message]

CalPolyPomona | 4

Public-key encryption, first publicly proposed by Diffie and Hellman in 1976 [DIFF76], is the first truly revolutionary advance in encryption in literally thousands of years. Public-key algorithms are based on mathematical functions rather than on simple operations on bit patterns, such as are used in symmetric encryption algorithms. More important, public-key cryptography is **asymmetric**, involving the use of two separate keys, in contrast to symmetric encryption, which uses only one key. The use of two keys has profound consequences in the areas of confidentiality, key distribution, and authentication.

Before proceeding, we should first mention several common misconceptions concerning public-key encryption. One is that public-key encryption is more secure from cryptanalysis than symmetric encryption. In fact, the security of any encryption scheme depends on (1) the length of the key and (2) the computational work involved in breaking a cipher. There is nothing in principle about either symmetric or public-key encryption that makes one superior to another from the point of view of resisting cryptanalysis. A second misconception is that public-key encryption is a general- purpose technique that has made symmetric encryption obsolete. On the contrary, because of the computational overhead of current public-key encryption schemes, there seems no foreseeable likelihood that symmetric encryption will be abandoned. Finally, there is a feeling that key distribution is trivial when using public-key encryption, compared to the rather cumbersome handshaking involved with key distribution centers for symmetric encryption. For public-key key distribution, some form of protocol is needed, often involving a central agent, and the procedures involved are no simpler or any more efficient than those required for symmetric encryption.

As the names suggest, the public key of the pair is made public for others to use, while the private key is known only to its owner. A general-purpose public-key cryptographic algorithm relies on one key for encryption and a different but related key for decryption.



Ingredients for public key cryptography

- Plaintext
  - Readable message or data that is fed into the algorithm as input
- Encryption algorithm
  - Performs transformations on the plaintext
- Public and private key
  - Pair of keys, one for encryption, one for decryption
- Ciphertext
  - Scrambled message produced as output
- Decryption key
  - Produces the original plaintext

CalPolyPomona 5

A public-key encryption scheme has six ingredients (Figure 2.6a):

• Plaintext: This is the readable message or data that is fed into the algorithm as input.

• Encryption algorithm: The encryption algorithm performs various transformations on the plaintext.

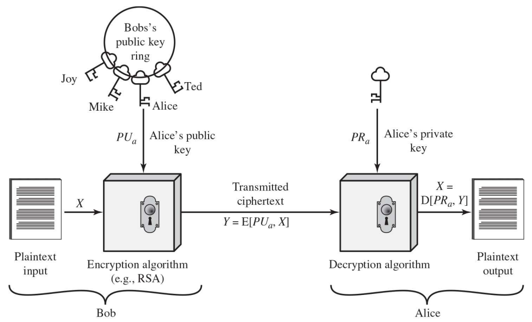
• Public and private key: This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input.

Ciphertext: This is 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: This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

As the names suggest, the public key of the pair is made public for others to use, while the private key is known only to its owner. A general-purpose public-key cryptographic algorithm relies on one key for encryption and a different but related key for decryption.

Figure 2.6 Public-Key Cryptography: schema 1



(a) Encryption with public key

The essential steps are the following:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.

2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private. As Figure 2.6a suggests, each user maintains a collection of public keys obtained from others.

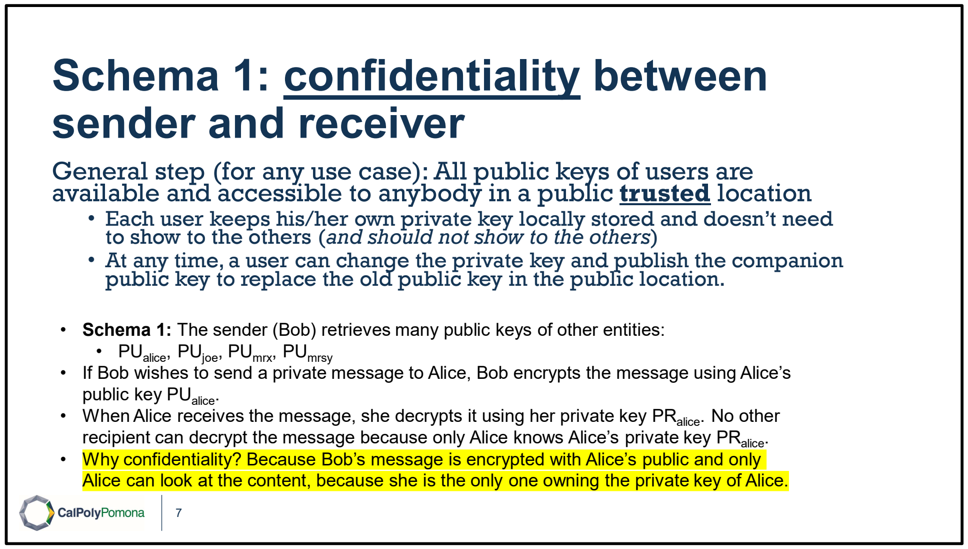
3. If Bob wishes to send a private message to Alice, Bob encrypts the message using Alice's public key.

4. When Alice receives the message, she decrypts it using her private key. No other recipient can decrypt the message because only Alice knows Alice's private key.

With this approach, all participants have access to public keys, and private keys are generated locally by each participant and therefore need never be distributed.
As long as a user protects his or her private key, incoming communication is secure.
At any time, a user can change the private key and publish the companion public key to replace the old public key.

Note that the scheme of Figure 2.6a is directed toward providing confidentiality:
Only the intended recipient should be able to decrypt the ciphertext because only the intended recipient is in possession of the required private key. Whether in fact confidentiality is provided depends on a number of factors, including the security of the algorithm, whether the private key is kept secure, and the security of any protocol of which the encryption function is a part.

Diagram a, encryption with public key. The plain text is fed into the algorithm as input. The plain text produces the output X to encryption Algorithm, example, RSA. The input Bob’s public key ring has pair of keys labeled, Joy, Mike, Alice, and Ted. The public key from Alice, PUa is shared to the encryption algorithm. The encryption algorithm transmits the output as ciphertext, $Y = E [PU_a, X]$. Decryption algorithm, reverse of encryption algorithm. It takes the ciphertext and the private key PR_a, shared by Alice as the input and produces the original plain text by a description key $X = D[PR_a, Y]$.



Schema 1: <u>confidentiality</u> between sender and receiver

General step (for any use case): All public keys of users are available and accessible to anybody in a public **<u>trusted</u>** location

- Each user keeps his/her own private key locally stored and doesn't need to show to the others (*and should not show to the others*)
- At any time, a user can change the private key and publish the companion public key to replace the old public key in the public location.

- **Schema 1:** The sender (Bob) retrieves many public keys of other entities:
  - PU_{alice}, PU_{joe}, PU_{mrx}, PU_{mrsy}
- If Bob wishes to send a private message to Alice, Bob encrypts the message using Alice's public key PU_{alice}.
- When Alice receives the message, she decrypts it using her private key PR_{alice}. No other recipient can decrypt the message because only Alice knows Alice's private key PR_{alice}.
- Why confidentiality? Because Bob's message is encrypted with Alice's public and only Alice can look at the content, because she is the only one owning the private key of Alice.

CalPolyPomona | 7

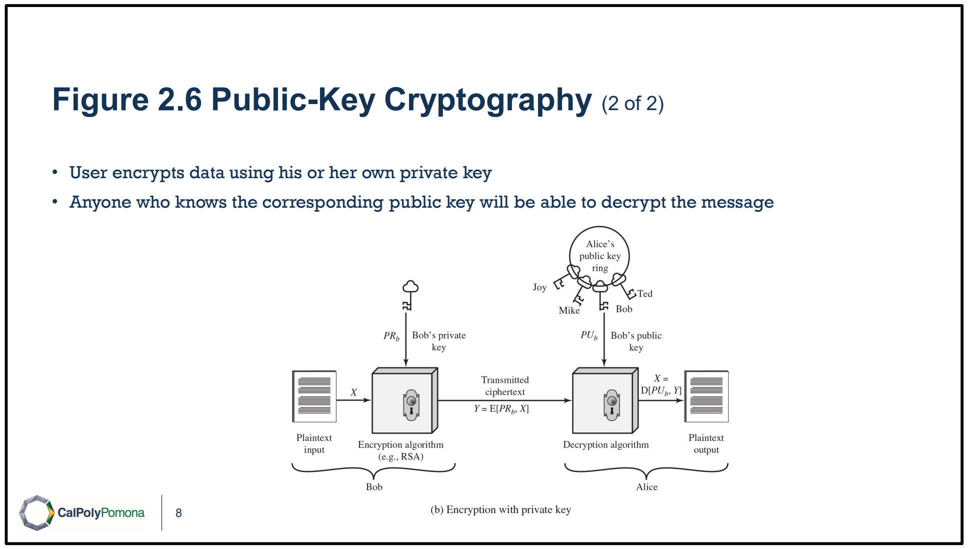


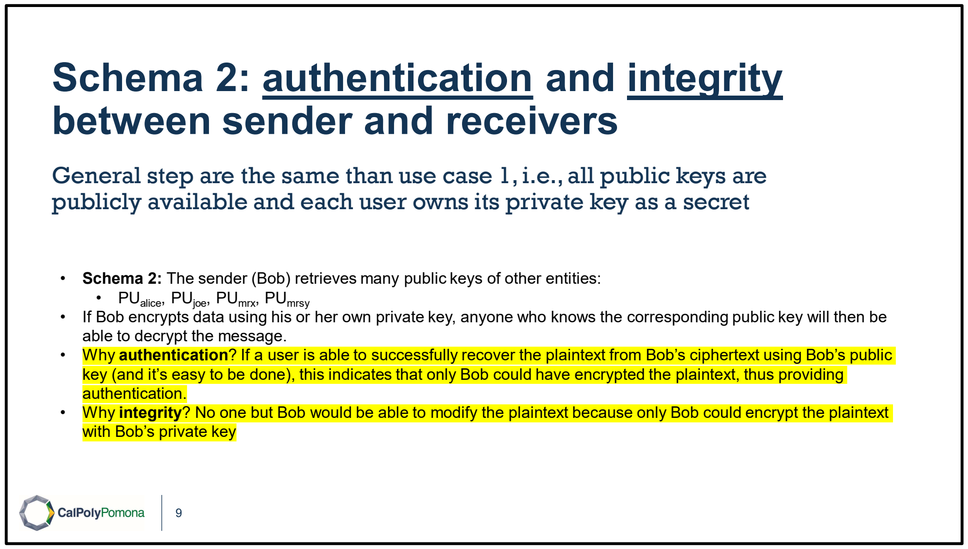
Figure 2.6 Public-Key Cryptography (2 of 2)

- User encrypts data using his or her own private key
- Anyone who knows the corresponding public key will be able to decrypt the message

Figure 2.6b illustrates another mode of operation of public-key cryptography. In this scheme, a user encrypts data using his or her own private key. Anyone who knows the corresponding public key will then be able to decrypt the message.

The scheme of Figure 2.6b is directed toward providing authentication and/or data integrity. If a user is able to successfully recover the plaintext from Bob's ciphertext using Bob's public key, this indicates that only Bob could have encrypted the plaintext, thus providing authentication. Further, no one but Bob would be able to modify the plaintext because only Bob could encrypt the plaintext with Bob's private key. Once again, the actual provision of authentication or data integrity depends on a variety of factors. This issue is addressed primarily in Chapter 21, but other references are made to it where appropriate in this text.

Diagram b, encryption with private key. The plain text is fed into the algorithm as input. The plain text produces the output X to encryption Algorithm, example, RSA. The private key from Bob, PR_b , is only shared to the encryption algorithm. The encryption algorithm transmits the output as ciphertext, Y = E[PR_b , X]. Decryption algorithm, reverse of encryption algorithm. It takes the ciphertext and the Alice public key ring with pair of keys labeled, Joy, Mike, Alice, and Ted. From the ring Bob's public key is shared as the input and produces the original plain text by a description key X = D[PU_b, Y].

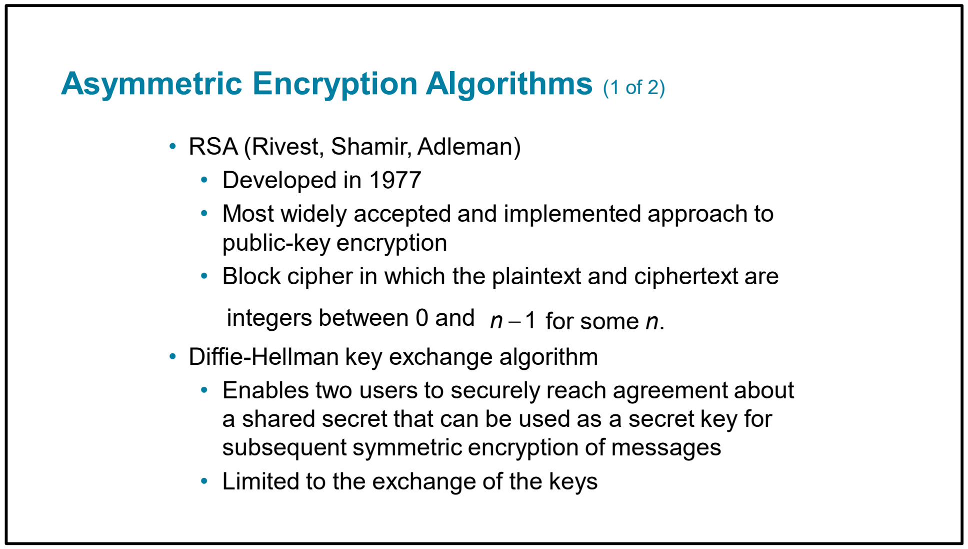


Schema 2: authentication and integrity between sender and receivers

General step are the same than use case 1, i.e., all public keys are publicly available and each user owns its private key as a secret

- **Schema 2:** The sender (Bob) retrieves many public keys of other entities:
  - PU_{alice}, PU_{joe}, PU_{mrx}, PU_{mrsy}
- If Bob encrypts data using his or her own private key, anyone who knows the corresponding public key will then be able to decrypt the message.
- Why **authentication**? If a user is able to successfully recover the plaintext from Bob's ciphertext using Bob's public key (and it's easy to be done), this indicates that only Bob could have encrypted the plaintext, thus providing authentication.
- Why **integrity**? No one but Bob would be able to modify the plaintext because only Bob could encrypt the plaintext with Bob's private key

CalPolyPomona | 9


Asymmetric Encryption Algorithms (1 of 2)

- RSA (Rivest, Shamir, Adleman)
  - Developed in 1977
  - Most widely accepted and implemented approach to public-key encryption
  - Block cipher in which the plaintext and ciphertext are integers between 0 and $n - 1$ for some *n*.
- Diffie-Hellman key exchange algorithm
  - Enables two users to securely reach agreement about a shared secret that can be used as a secret key for subsequent symmetric encryption of messages
  - Limited to the exchange of the keys

RSA 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 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.*

In 1977, the three inventors of RSA dared *Scientific American readers to decode* a cipher they printed in Martin Gardner's "Mathematical Games" column. They offered a $100 reward for the return of a plaintext sentence, an event they predicted might not occur for some 40 quadrillion years. In April of 1994, a group working over the Internet and using over 1600 computers claimed the prize after only eight months of work [LEUT94]. This challenge used a public-key size (length of *n) of 129 decimal* digits, or around 428 bits. This result does not invalidate the use of RSA; it simply means that larger key sizes must be used. Currently, a 1024-bit key size (about 300 decimal digits) is considered strong enough for virtually all applications.

DIFFIE-HELLMAN KEY AGREEMENT 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, or key agreement. A number of commercial products employ this key exchange technique.

The purpose of the algorithm is to enable two users to securely reach agreement about a shared secret that can be used as a secret key for subsequent symmetric encryption of messages. The algorithm itself is limited to the exchange of the keys.

DIGITAL SIGNATURE STANDARD The National Institute of Standards and Technology (NIST) has published Federal Information Processing Standard FIPS PUB 186, known as the *Digital Signature Standard (DSS*). The DSS makes use of SHA-1 and presents a new digital signature technique, the Digital Signature Algorithm (DSA). The DSS was originally proposed in 1991 and revised in 1993 in response to public feedback concerning the security of the scheme. There were further revisions in 1998, 2000, 2009, and most recently in 2013 as FIPS PUB 186–4. The DSS uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange.

ELLIPTIC CURVE CRYPTOGRAPHY The vast majority of the products and standards that use public-key cryptography for encryption and digital signatures use RSA. The bit length for secure RSA use has increased over recent years, and this has put a heavier processing load on applications using RSA. This burden has ramifications, especially for electronic commerce sites that conduct large numbers of secure transactions. Recently, a competing system has begun to challenge RSA: elliptic curve cryptography (ECC). Already, ECC is showing up in standardization efforts, including the IEEE (Institute of Electrical and Electronics Engineers) P1363 Standard for Public-Key Cryptography.

The principal attraction of ECC compared to RSA is that it appears to offer equal security for a far smaller bit size, thereby reducing processing overhead. On the other hand, although the theory of ECC has been around for some time, it is only recently that products have begun to appear and that there has been sustained cryptanalytic interest in probing for weaknesses. Thus, the confidence level in ECC is not yet as high as that in RSA.

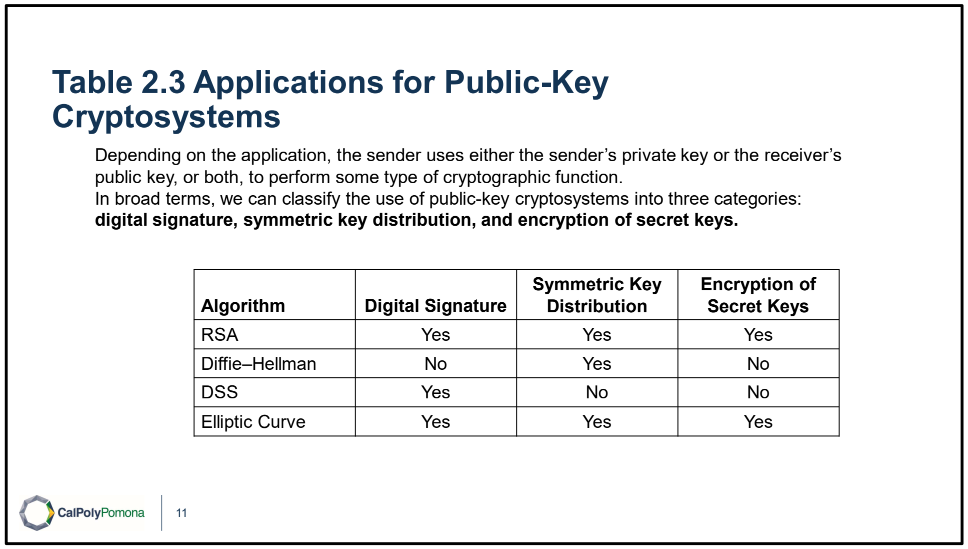


Table 2.3 Applications for Public-Key Cryptosystems

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 three categories: **digital signature, symmetric key distribution, and encryption of secret keys.**

Algorithm	Digital Signature	Symmetric Key Distribution	Encryption of Secret Keys
RSA	Yes	Yes	Yes
Diffie–Hellman	No	Yes	No
DSS	Yes	No	No
Elliptic Curve	Yes	Yes	Yes

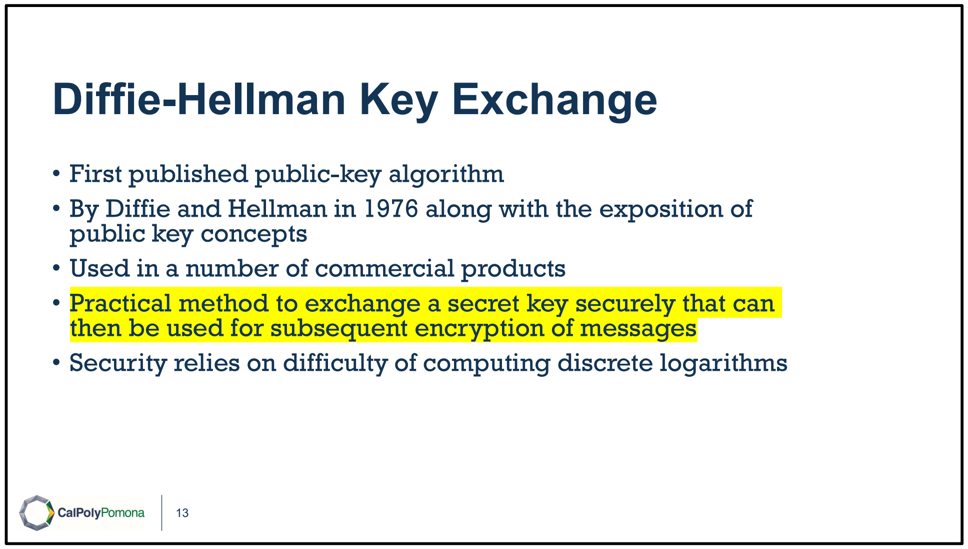
CalPolyPomona 11

Public-key systems are characterized by the use of a cryptographic type of algorithm with two keys, one held private and one available publicly. 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 three categories: digital signature, symmetric key distribution, and encryption of secret keys.

These applications will be discussed in Section 2.4. Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications. Table 2.3 indicates the applications supported by the algorithms discussed in this section.

Diffie-Hellman algorithm for secret key generation and exchange

CalPolyPomona



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

CalPolyPomona | 13

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.

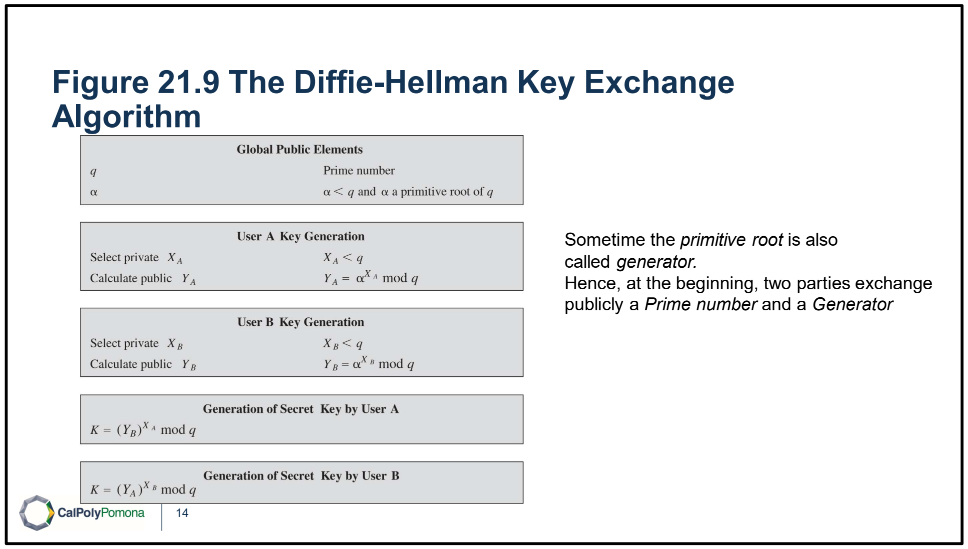


Figure 21.9 The Diffie-Hellman Key Exchange Algorithm

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

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

Sometime the *primitive root* is also called *generator.*
Hence, at the beginning, two parties exchange publicly a *Prime number* and a *Generator*

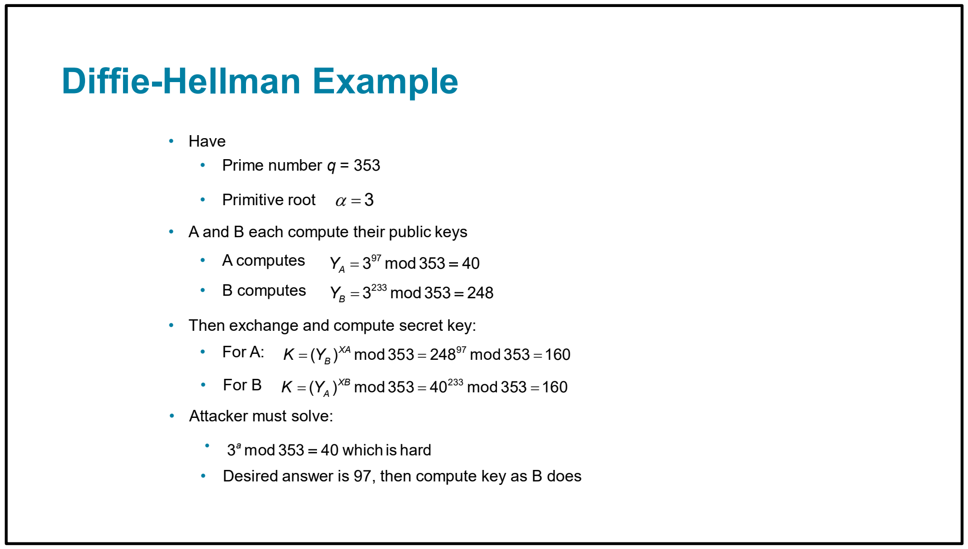
CalPolyPomona 14

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: $X_B = \text{d}\log_{\alpha,q}(Y_B)$. 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.



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

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 $K = (Y_A)^{XB} \bmod 353 = 40^{233} \bmod 353 = 160$.

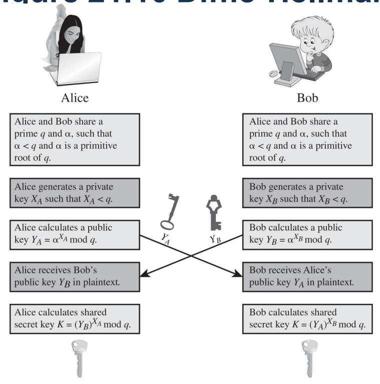
We assume an attacker would have available the following information:

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

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 $3^a \bmod 353 = 40$ 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.

Figure 21.10 Diffie-Hellman Key Exchange Protocol


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

Note 1: $(g^b \bmod p)^a = g^{ba} \bmod p$

i.e., exponentiation does not change module result

Note 2: $g^{ba} \bmod p \bmod p = g^{ba} \bmod p$

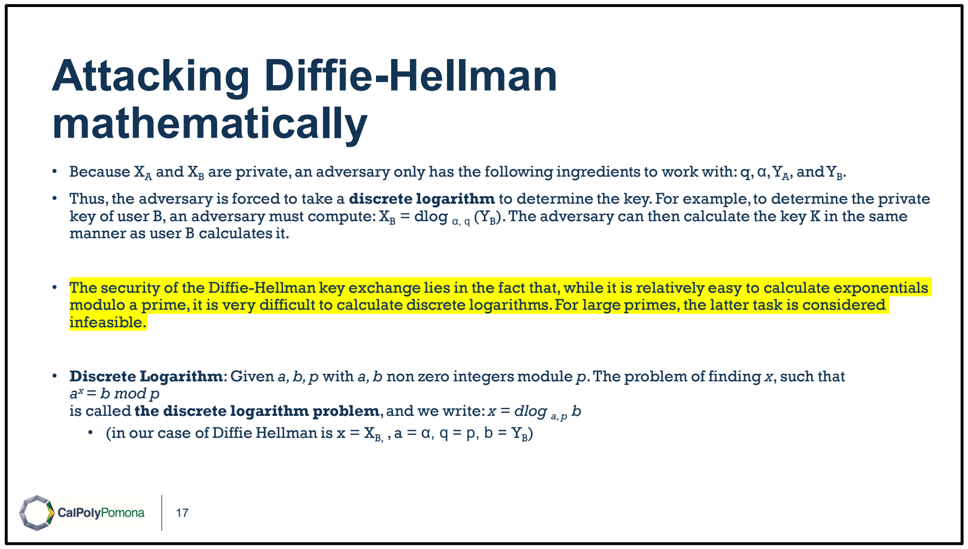
i.e., multiple module operations don't change the result

CalPolyPomona | 16

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.

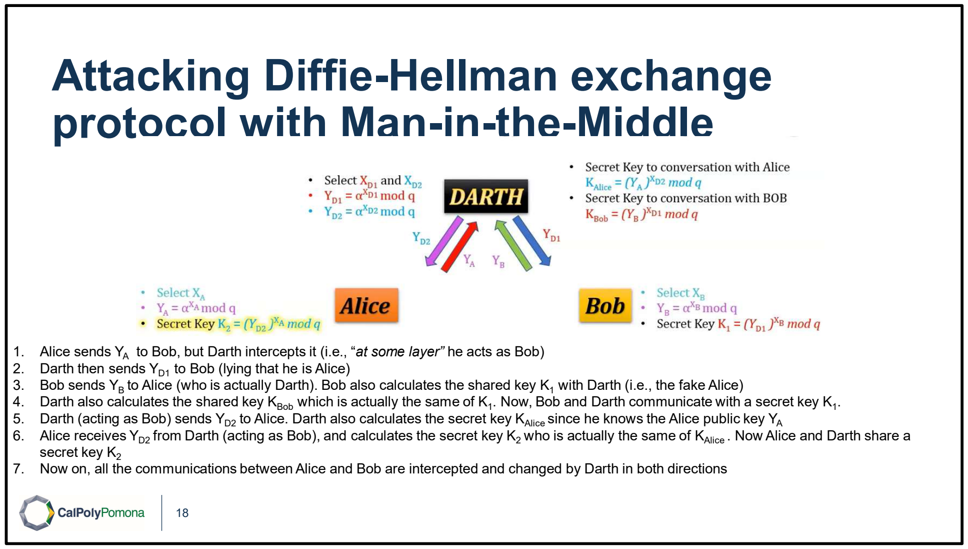


Attacking Diffie-Hellman mathematically

- 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 = dlog $_{α, q}$ (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.

- **Discrete Logarithm**: Given *a, b, p* with *a, b* non zero integers module *p*. The problem of finding *x*, such that
  $a^x = b \bmod p$
  is called **the discrete logarithm problem**, and we write: $x = dlog_{a,p}\ b$
  - (in our case of Diffie Hellman is x = X_B, a = α, q = p, b = Y_B)



Attacking Diffie-Hellman exchange protocol with Man-in-the-Middle

- Select X_{D1} and X_{D2}
- $Y_{D1} = \alpha^{X_{D1}} \bmod q$
- $Y_{D2} = \alpha^{X_{D2}} \bmod q$

DARTH

- Secret Key to conversation with Alice
  $K_{Alice} = (Y_A)^{X_{D2}} \bmod q$
- Secret Key to conversation with BOB
  $K_{Bob} = (Y_B)^{X_{D1}} \bmod q$

Y_{D2}
Y_A
Y_B
Y_{D1}

- Select X_A
- $Y_A = \alpha^{X_A} \bmod q$
- Secret Key $K_2 = (Y_{D2})^{X_A} \bmod q$

Alice

Bob

- Select X_B
- $Y_B = \alpha^{X_B} \bmod q$
- Secret Key $K_1 = (Y_{D1})^{X_B} \bmod q$

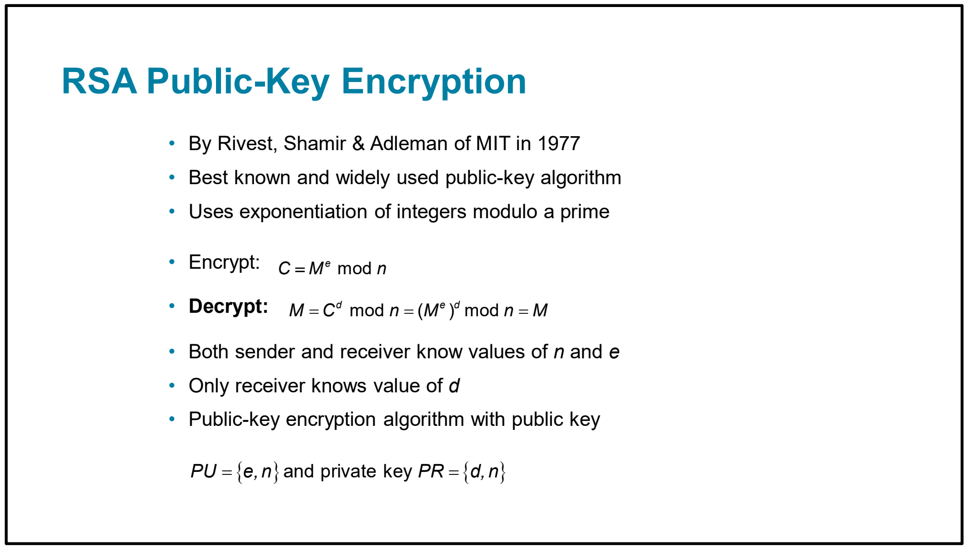
1. Alice sends Y_A to Bob, but Darth intercepts it (i.e., "*at some layer*" he acts as Bob)
2. Darth then sends Y_{D1} to Bob (lying that he is Alice)
3. Bob sends Y_B to Alice (who is actually Darth). Bob also calculates the shared key K_1 with Darth (i.e., the fake Alice)
4. Darth also calculates the shared key K_{Bob} which is actually the same of K_1. Now, Bob and Darth communicate with a secret key K_1.
5. Darth (acting as Bob) sends Y_{D2} to Alice. Darth also calculates the secret key K_{Alice} since he knows the Alice public key Y_A
6. Alice receives Y_{D2} from Darth (acting as Bob), and calculates the secret key K_2 who is actually the same of K_{Alice} . Now Alice and Darth share a secret key K_2
7. Now on, all the communications between Alice and Bob are intercepted and changed by Darth in both directions

CalPolyPomona | 18

In cryptography and computer security, a **man-in-the-middle**[a] (**MITM**) **attack** is a cyberattack where the attacker secretly relays and possibly alters the communications between two parties who believe that they are directly communicating with each other, as the attacker has inserted themselves between the two parties.[9]

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.

RSA



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 $PR = \{d, n\}$

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*.

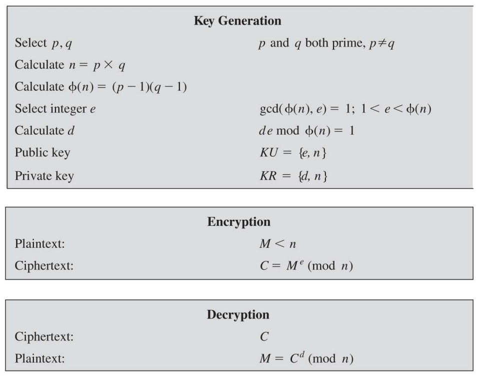
Encryption and decryption are of the following form, for some plaintext block *M* and ciphertext block *C*:

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

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

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
(for generation of a public-private key pair)



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

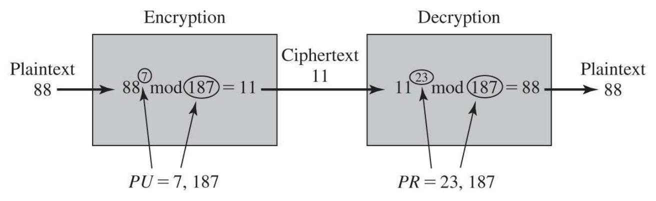
Note: e is coprime of Φ(n)

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.

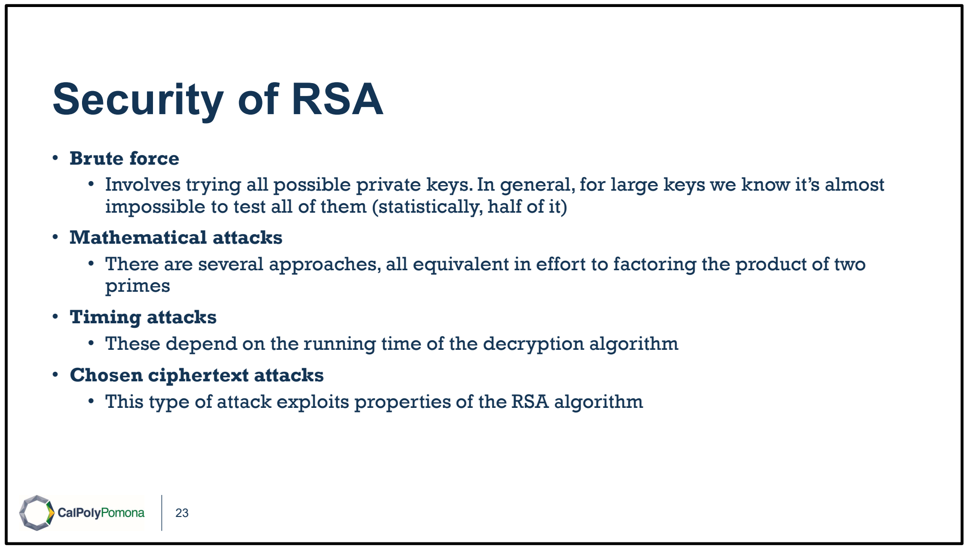
Note: two numbers are coprime if their gcd is 1. Example, 2 and 33 are coprime (2 is prime but, in general, 33 is not prime).

Figure 21.8 Example of RSA Algorithm



Plaintext = 88
Public key = {7, 187}
Private key = {23,187}

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


Security of RSA

- **Brute force**
  - Involves trying all possible private keys. In general, for large keys we know it's almost impossible to test all of them (statistically, half of it)
- **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

CalPolyPomona | 23

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.

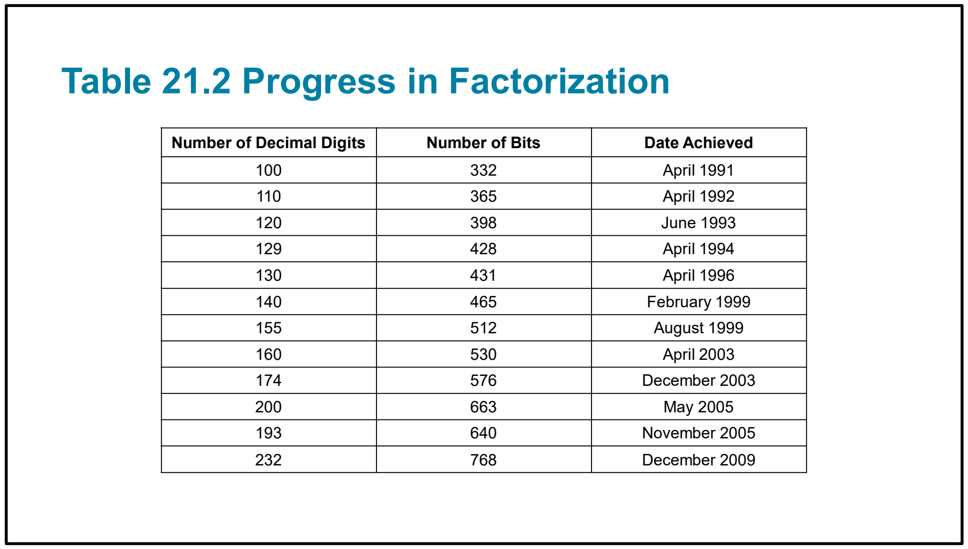


Table 21.2 Progress in Factorization

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.

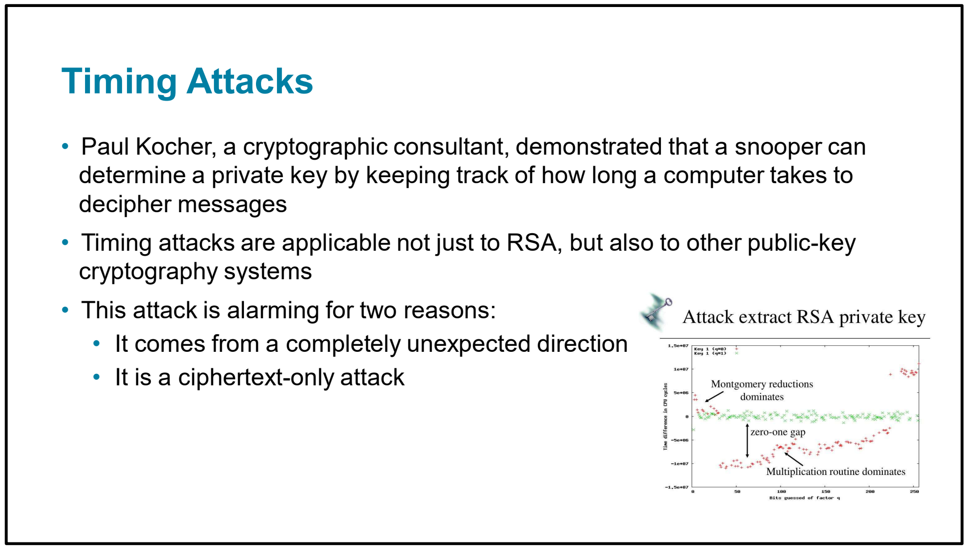
In addition to specifying the size of *n , a number of other constraints have been* suggested by researchers. To avoid values of *n that may be factored more easily, the* algorithm's inventors suggest the following constraints on *p and q :*

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.*

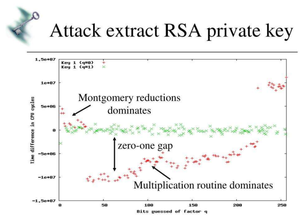
3. gcd (*p - 1, q - 1) should be small.*

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



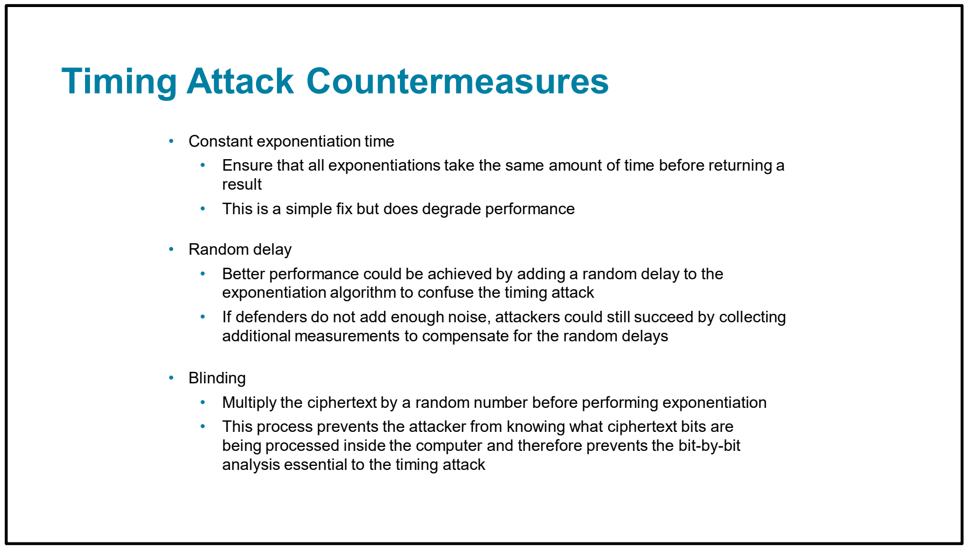
If one needed yet another lesson about how difficult it is to assess the security of a cryptographic algorithm, the appearance of timing attacks provides a stunning one. Paul Kocher, a cryptographic consultant, demonstrated that a snooper can determine a private key by keeping track of how long a computer 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].



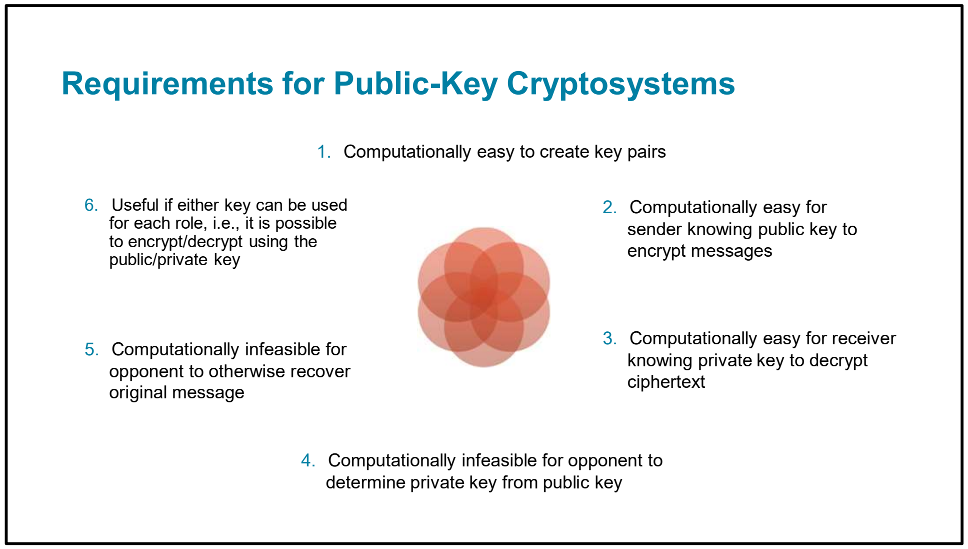
Timing Attack Countermeasures

- 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

• **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.



Requirements for Public-Key Cryptosystems

1. Computationally easy to create key pairs
2. Computationally easy for sender knowing public key to encrypt messages
3. Computationally easy for receiver knowing private key to decrypt ciphertext
4. Computationally infeasible for opponent to determine private key from public key
5. Computationally infeasible for opponent to otherwise recover original message
6. Useful if either key can be used for each role, i.e., it is possible to encrypt/decrypt using the public/private key

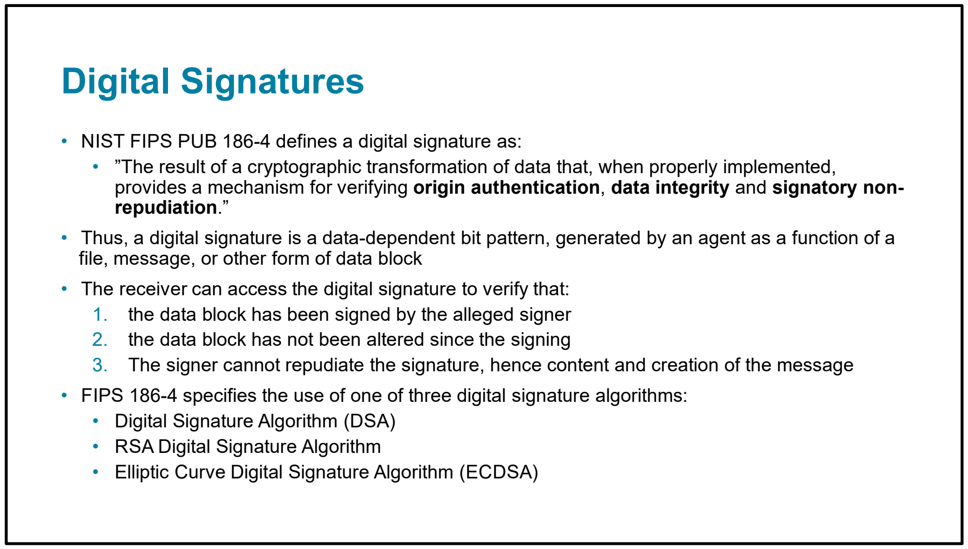
The cryptosystem illustrated in Figure 2.6 depends on a cryptographic algorithm based on two related keys. Diffie and Hellman postulated this system without demonstrating that such algorithms exist. However, they did lay out the conditions that such algorithms must fulfill [DIFF76]:

Other relevant Asymmetric Encryption Algorithms

- Digital Signature Standard (DSS)
  - Provides only a digital signature function with SHA-1
  - Cannot be used for encryption or key exchange
- Elliptic curve cryptography (ECC)
  - Security like RSA, but with much smaller keys

Applications of public key

CalPolyPomona



Digital Signatures

- NIST FIPS PUB 186-4 defines a digital signature as:
  - ”The result of a cryptographic transformation of data that, when properly implemented, provides a mechanism for verifying **origin authentication**, **data integrity** and **signatory non-repudiation**.”
- Thus, a digital signature is a data-dependent bit pattern, generated by an agent as a function of a file, message, or other form of data block
- The receiver can access the digital signature to verify that:
  1. the data block has been signed by the alleged signer
  2. the data block has not been altered since the signing
  3. The signer cannot repudiate the signature, hence content and creation of the message
- FIPS 186-4 specifies the use of one of three digital signature algorithms:
  - Digital Signature Algorithm (DSA)
  - RSA Digital Signature Algorithm
  - Elliptic Curve Digital Signature Algorithm (ECDSA)

Public-key encryption can be used for authentication with a technique known as the digital signature. NIST FIPS PUB 186-4 [*Digital Signature Standard (DSS*) , July 2013] defines a digital signature as follows: The result of a cryptographic transformation of data that, when properly implemented, provides a mechanism for verifying origin authentication, data integrity and signatory non-repudiation.

Thus, a digital signature is a data-dependent bit pattern, generated by an agent as a function of a file, message, or other form of data block. Another agent can access the data block and its associated signature and verify (1) the data block has been signed by the alleged signer, and (2) the data block has not been altered since the signing. Further, the signer cannot repudiate the signature.

FIPS 186-4 specifies the use of one of three digital signature algorithms:

• Digital Signature Algorithm (DSA): The original NIST-approved algorithm, which is based on the difficulty of computing discrete logarithms.

- RSA Digital Signature Algorithm: Based on the RSA public-key algorithm.
- Elliptic Curve Digital Signature Algorithm (ECDSA): Based on elliptic-curve cryptography.

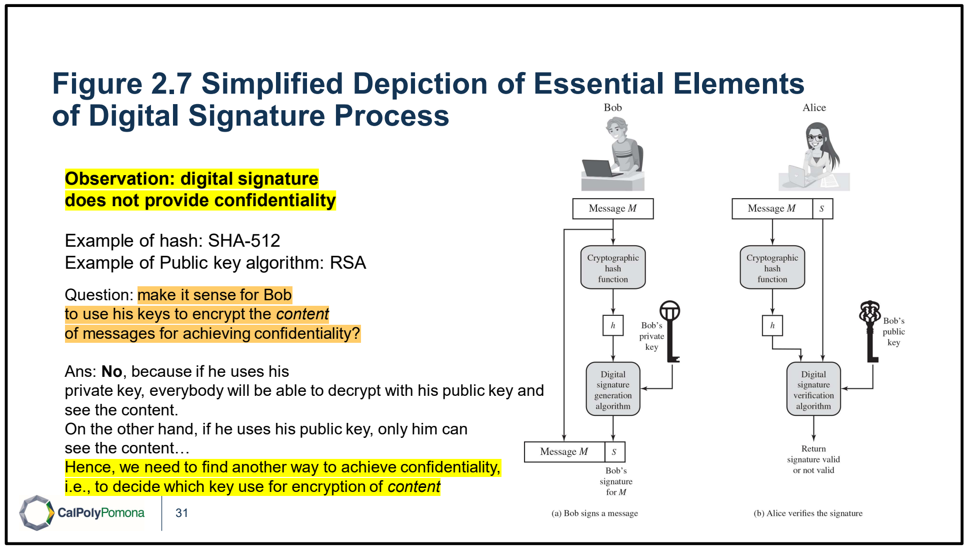


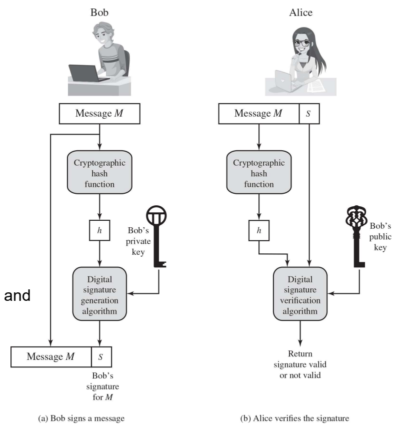
Figure 2.7 Simplified Depiction of Essential Elements of Digital Signature Process

Observation: digital signature does not provide confidentiality

Example of hash: SHA-512
Example of Public key algorithm: RSA

Question: make it sense for Bob to use his keys to encrypt the *content* of messages for achieving confidentiality?

Ans: **No**, because if he uses his private key, everybody will be able to decrypt with his public key and see the content.
On the other hand, if he uses his public key, only him can see the content…
Hence, we need to find another way to achieve confidentiality, i.e., to decide which key use for encryption of *content*



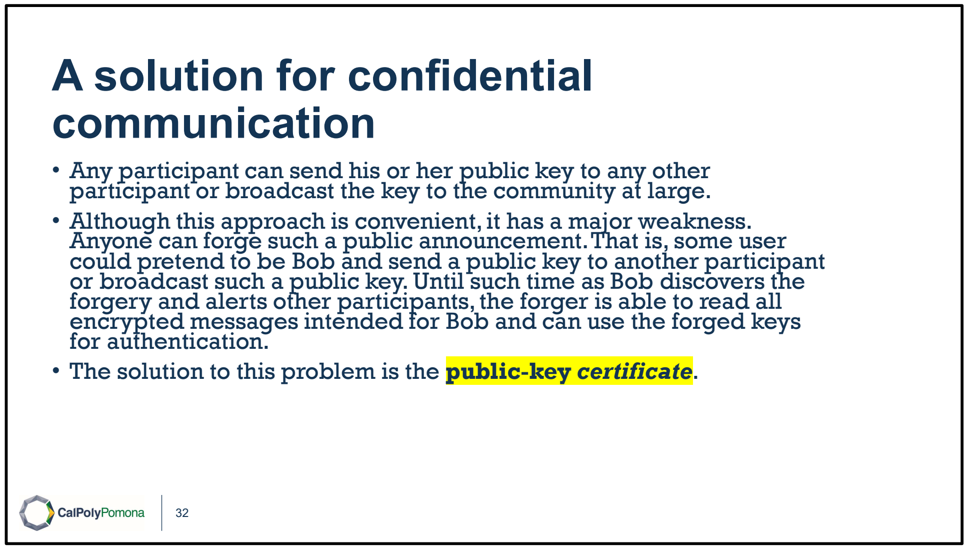
Bob
Alice
Message *M*
Cryptographic hash function
h
Bob's private key
Digital signature generation algorithm
Message *M*
S
Bob's signature for *M*
(a) Bob signs a message
Message *M*
S
Cryptographic hash function
h
Bob's public key
Digital signature verification algorithm
Return signature valid or not valid
(b) Alice verifies the signature

CalPolyPomona | 31

Figure 2.7 is a generic model of the process of making and using digital signatures. All of the digital signature schemes in FIPS 186-4 have this structure. Suppose Bob wants to send a message to Alice. Although it is not important that the message be kept secret, he wants Alice to be certain that the message is indeed from him. For this purpose, Bob uses a secure hash function, such as SHA-512, to generate a hash value for the message. That hash value, together with Bob's private key, serve as input to a digital signature generation algorithm that produces a short block that functions as a digital signature. Bob sends the message with the signature attached. When Alice receives the message plus signature, she (1) calculates a hash value for the message; (2) provides the hash value and Bob's public key as inputs to a digital signature verification algorithm. If the algorithm returns the result that the signature is valid, Alice is assured that the message must have been signed by Bob. No one else has Bob's private key, and therefore no one else could have created a signature that could be verified for this message with Bob's public key. In addition, it is impossible to alter the message without access to Bob's private key, so the message is authenticated both in terms of source and in terms of data integrity.

The digital signature does not provide confidentiality. That is, the message being sent is safe from alteration, but not safe from eavesdropping. This is obvious in the case of a signature based on a portion of the message, because the rest of the message is transmitted in the clear. Even in the case of complete encryption, there is no protection of confidentiality because any observer can decrypt the message by using the sender’s public key.

Flow diagram a, Bob signs a message. Step 1. Message M is fed to the final step of message M with attached signature. Step 2. Cryptographic has function. Step 3. Hash value, h. Step 4. Digital signature generation algorithm, Bob’s private key is shared to the digital signature. Step 5. Bob’s signature for M with attached message M. Flow diagram b, Alice verifies the signature. Step 1. Message M with attached signature is fed to the final step, Cryptographic has function. Step 2. Cryptographic has function. Step 3. Hash value, h. Step 4. Digital signature verification algorithm, Bob’s public key is shared to the digital signature. Step 4 produces the output to check, return signature valid or not valid.



A solution for confidential communication

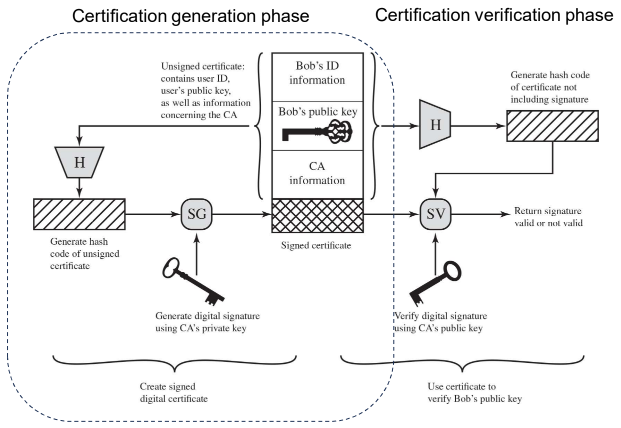
- Any participant can send his or her public key to any other participant or broadcast the key to the community at large.
- Although this approach is convenient, it has a major weakness. Anyone can forge such a public announcement. That is, some user could pretend to be Bob and send a public key to another participant or broadcast such a public key. Until such time as Bob discovers the forgery and alerts other participants, the forger is able to read all encrypted messages intended for Bob and can use the forged keys for authentication.
- The solution to this problem is the **public-key *certificate***.

CalPolyPomona | 32

On the face of it, the point of public-key encryption is that the public key is public. Thus, if there is some broadly accepted public-key algorithm, such as RSA, any participant can send his or her public key to any other participant or broadcast the key to the community at large. Although this approach is convenient, it has a major weakness. Anyone can forge such a public announcement. That is, some user could pretend to be Bob and send a public key to another participant or broadcast such a public key. Until such time as Bob discovers the forgery and alerts other participants, the forger is able to read all encrypted messages intended for Bob and can use the forged keys for authentication.

Figure 2.8 Public-Key Certificate Use

Note:
generation
and *verification*
phases happen usually in two different moments
(even if, here, represented in the same figure)


Certification generation phase
Certification verification phase
Unsigned certificate: contains user ID, user's public key, as well as information concerning the CA
Bob's ID information
Bob's public key
CA information
H
Generate hash code of unsigned certificate
SG
Signed certificate
Generate digital signature using CA's private key
Create signed digital certificate
H
Generate hash code of certificate not including signature
SV
Return signature valid or not valid
Verify digital signature using CA's public key
Use certificate to verify Bob's public key

In essence, a certificate consists of a public key plus a user ID of the key owner, with the whole block signed by a trusted third party. The certificate also includes some information about the third party plus an indication of the period of validity of the certificate. Typically, the third party is a certificate authority (CA) that is trusted by the user community, such as a government agency or a financial institution. A user can present his or her public key to the authority in a secure manner and obtain a signed certificate. The user can then publish the certificate. Anyone needing this user's public key can obtain the certificate and verify that it is valid by means of the attached trusted signature.
Figure 2.8 illustrates the process.

The key steps can be summarized as follows:

1. User software (client) creates a pair of keys: one public and one private.

2. Client prepares an unsigned certificate that includes the user ID and user's public key.

3. User provides the unsigned certificate to a CA in some secure manner. This might require a face-to-face meeting, the use of registered e-mail, or happen via a Web form with e-mail verification.

4. CA creates a signature as follows:

    a. CA uses a hash function to calculate the hash code of the unsigned certificate.
    A hash function is one that maps a variable-length data block or message into a fixed-length value called a hash code, such as SHA family that we will discuss in Sections 2.2 and 21.1.

    b. CA generates digital signature using the CA's private key and a signature generation algorithm.

5. CA attaches the signature to the unsigned certificate to create a signed certificate.

6. CA returns the signed certificate to client.

7. Client may provide the signed certificate to any other user.

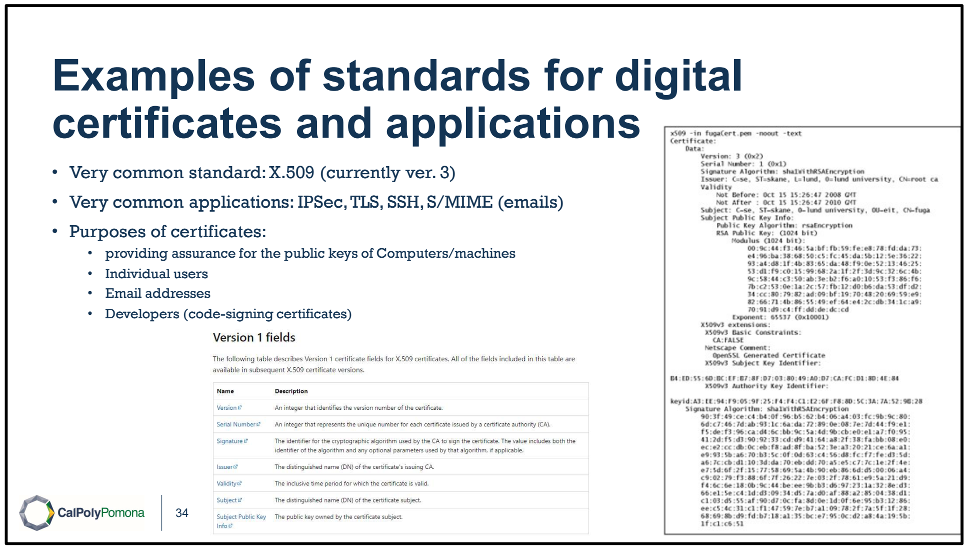
8. Any user may verify that the certificate is valid as follows:

    a. User calculates the hash code of certificate (not including signature).

    b. User verifies digital signature using CA's public key and the signature verification algorithm. The algorithm returns a result of either signature valid or invalid.

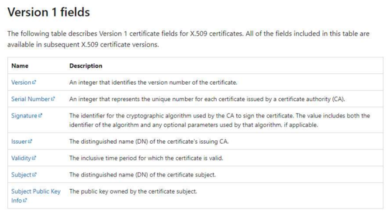
One scheme has become universally accepted for formatting public-key certificates: the X.509 standard. X.509 certificates are used in most network security applications, including IP Security (IPsec), Transport Layer Security (TLS), Secure Shell (SSH), and Secure/Multipurpose Internet Mail Extension (S/MIME). We will examine most of these applications in Part Five.

The diagram is as follows. User software creates a pair of keys, one public and one private. User prepares an unsigned certificate that includes the Bob’s I D and Bob’s public key as well as information concerning the C A. C A uses a hash function to calculate the hash code of the unsigned certificate. C A generates digital signature using the CA’s private key and a signature generation algorithm. C A attaches the signature to the unsigned certificate to create a signed certificate. C A returns the signed certificate to Bob. User calculates the hash code of certificate, not including signature. User verifies digital signature using C A’s public key and the signature verification algorithm. The algorithm returns a result of either signature valid or invalid.



Examples of standards for digital certificates and applications

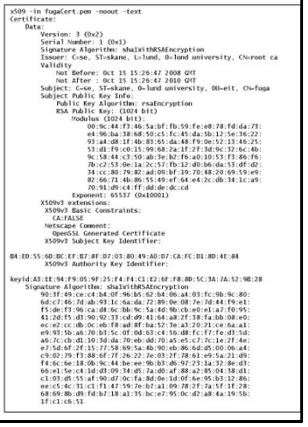
- Very common standard: X.509 (currently ver. 3)
- Very common applications: IPSec, TLS, SSH, S/MIME (emails)
- Purposes of certificates:
  - providing assurance for the public keys of Computers/machines
  - Individual users
  - Email addresses
  - Developers (code-signing certificates)



Version 1 fields

The following table describes Version 1 certificate fields for X.509 certificates. All of the fields included in this table are available in subsequent X.509 certificate versions.

Name	Description
Version	An integer that identifies the version number of the certificate.
Serial Number	An integer that represents the unique number for each certificate issued by a certificate authority (CA).
Signature	The identifier for the cryptographic algorithm used by the CA to sign the certificate. The value includes both the identifier of the algorithm and any optional parameters used by that algorithm, if applicable.
Issuer	The distinguished name (DN) of the certificate's issuing CA.
Validity	The inclusive time period for which the certificate is valid.
Subject	The distinguished name (DN) of the certificate subject.
Subject Public Key Info	The public key owned by the certificate subject.

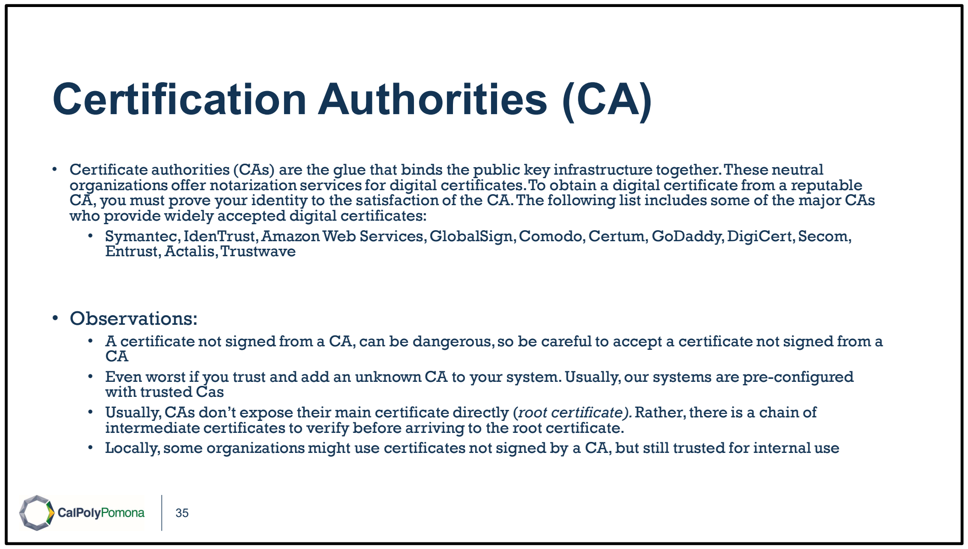


```
x509 -in fugaCert.pem -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=se, ST=skane, L=lund, O=lund university, CN=root ca
        Validity
            Not Before: Oct 15 15:26:47 2008 GMT
            Not After : Oct 15 15:26:47 2010 GMT
        Subject: C=se, ST=skane, O=lund university, OU=eit, CN=fuga
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:9c:44:f3:46:5a:bf:fb:59:fe:e8:78:fd:da:73:
                    e4:96:ba:38:68:50:c5:fc:45:da:5b:12:5e:36:22:
                    93:a4:d8:1f:4b:83:65:da:48:f9:0e:52:13:46:25:
                    53:d1:f9:c0:15:99:68:2a:1f:2f:3d:9c:32:6c:4b:
                    9c:58:44:c3:50:ab:3e:b2:f6:a0:10:53:f3:86:f6:
                    7b:c2:53:0e:1a:2c:57:fb:12:d0:b6:da:53:df:d2:
                    34:cc:80:79:82:ad:09:bf:19:70:48:20:69:59:e9:
                    82:66:71:4b:86:55:49:ef:64:e4:2c:db:34:1c:a9:
                    70:91:d9:c4:ff:dd:de:dc:cd
                Exponent: 65537 (0x10001)
        X509v3 extensions:
         X509v3 Basic Constraints:
           CA:FALSE
         Netscape Comment:
           OpenSSL Generated Certificate
         X509v3 Subject Key Identifier:

B4:ED:55:6D:BC:EF:B7:8F:D7:03:80:49:A0:D7:CA:FC:D1:8D:4E:84
         X509v3 Authority Key Identifier:

keyid:A3:EE:94:F9:05:9F:25:F4:F4:C1:E2:6F:F8:8D:5C:3A:7A:52:9B:28
    Signature Algorithm: sha1WithRSAEncryption
        90:3f:49:ce:c4:b4:0f:96:b5:62:b4:06:a4:03:fc:9b:9c:80:
        6d:c7:46:7d:ab:93:1c:6a:da:72:89:0e:08:7e:7d:44:f9:e1:
        f5:de:f3:96:ca:d4:6c:bb:9c:5a:4d:9b:cb:e0:e1:a7:f0:95:
        41:2d:f5:d3:90:92:33:cd:d9:41:64:a8:2f:38:fa:bb:08:e0:
        ec:e2:cc:db:0c:eb:f8:ad:8f:ba:52:3e:a3:20:21:ce:6a:a1:
        e9:93:5b:a6:70:b3:5c:0f:0d:63:c4:56:d8:fc:f7:fe:d3:5d:
        a6:7c:cb:d1:10:3d:da:70:eb:dd:70:a5:e5:c7:7c:1e:2f:4e:
        e7:5d:6f:2f:15:77:58:69:5a:4b:90:eb:86:6d:d5:00:06:a4:
        c9:02:79:f3:88:6f:7f:26:22:7e:03:2f:78:61:e9:5a:21:d9:
        f4:6c:6e:18:0b:9c:44:be:ee:9b:b3:d6:97:23:1a:32:8e:d3:
        66:e1:5e:c4:1d:d3:09:34:d5:7a:d0:af:88:a2:85:04:38:d1:
        c1:03:d5:55:af:90:d7:0c:fa:8d:0e:1d:0f:6e:95:b3:12:86:
        ee:c5:4c:31:c1:f1:47:59:7e:b7:a1:09:78:2f:7a:5f:1f:28:
        68:69:8b:d9:fd:b7:18:a1:35:bc:e7:95:0c:d2:a8:4a:19:5b:
        1f:c1:c6:51
```

CalPolyPomona | 34



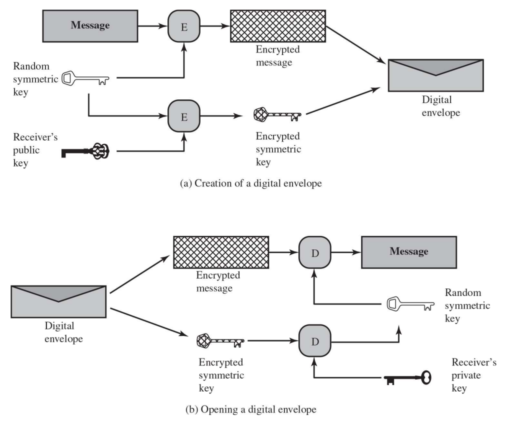
Certification Authorities (CA)

- Certificate authorities (CAs) are the glue that binds the public key infrastructure together. These neutral organizations offer notarization services for digital certificates. To obtain a digital certificate from a reputable CA, you must prove your identity to the satisfaction of the CA. The following list includes some of the major CAs who provide widely accepted digital certificates:
  - Symantec, IdenTrust, Amazon Web Services, GlobalSign, Comodo, Certum, GoDaddy, DigiCert, Secom, Entrust, Actalis, Trustwave
- Observations:
  - A certificate not signed from a CA, can be dangerous, so be careful to accept a certificate not signed from a CA
  - Even worst if you trust and add an unknown CA to your system. Usually, our systems are pre-configured with trusted Cas
  - Usually, CAs don't expose their main certificate directly (*root certificate*). Rather, there is a chain of intermediate certificates to verify before arriving to the root certificate.
  - Locally, some organizations might use certificates not signed by a CA, but still trusted for internal use

CalPolyPomona | 35

Nothing is preventing any organization from simply setting up shop as a CA. However, the certificates issued by a CA are only as good as the trust placed in the CA that issued them. This is an important item to consider when receiving a digital certificate from a third party. If you don't recognize and trust the name of the CA that issued the certificate, you shouldn't place any trust in the certificate at all. PKI relies on a hierarchy of trust relationships. If you configure your browser to trust a CA, it will automatically trust all of the digital certificates issued by that CA. Browser developers preconfigure browsers to trust the major CAs to avoid placing this burden on users. Registration authorities (RAs) assist CAs with the burden of verifying users' identities prior to issuing digital certificates. They do not directly issue certificates themselves, but they play an important role in the certification process, allowing CAs to remotely validate user identities. Certificate authorities must carefully protect their own private keys to preserve their trust relationships. To do this, they often use an offline CA to protect their root certificate, the top-level certificate for their entire PKI. This offline CA is disconnected from networks and powered down until it is needed. The offline CA uses the root certificate to create subordinate intermediate CAs that serve as the online CAs used to issue certificates on a routine basis. In the CA trust model, the use of a series of intermediate CAs is known as certificate chaining. To validate a certificate, the browser verifies the identity of the intermediate CA(s) first and then traces the path of trust back to a known root CA, verifying the identity of each link in the chain of trust. Certificate authorities do not need to be third-party service providers. Many organizations operate internal CAs that provide self-signed certificates for use inside an organization. These certificates won't be trusted by the browsers of external users, but internal systems may be configured to trust the internal CA, saving the expense of obtaining certificates from a third-party CA.

Figure 2.9 Digital Envelopes



Example of digital envelope:
PGP – Pretty Good Privacy
OpenPGP

PGP is an encryption program that provides cryptographic privacy and authentication for data communication. PGP is used for signing, encrypting, and decrypting texts, e-mails, files, directories, and whole disk partitions and to increase the security of e-mail communications.

Another application in which public-key encryption is used to protect a symmetric key is the digital envelope, which can be used to protect a message without needing to first arrange for sender and receiver to have the same secret key. The technique is referred to as a digital envelope, which is the equivalent of a sealed envelope containing an unsigned letter. The general approach is shown in Figure 2.9. Suppose Bob wishes to send a confidential message to Alice, but they do not share a symmetric secret key. Bob does the following:

1. Prepare a message.

2. Generate a random symmetric key that will be used this one time only.

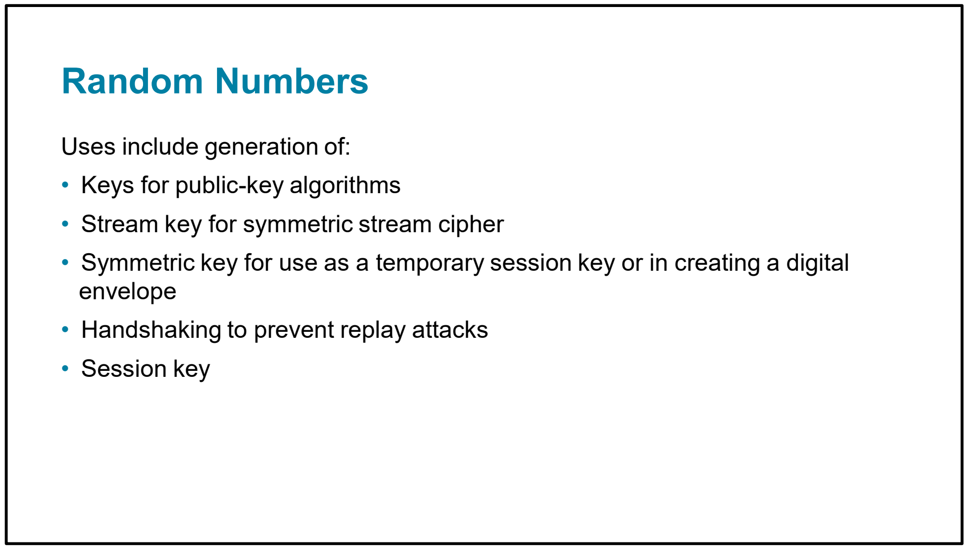
3. Encrypt that message using symmetric encryption the one-time key.

4. Encrypt the one-time key using public-key encryption with Alice’s public key.

5. Attach the encrypted one-time key to the encrypted message and send it to Alice.

Only Alice is capable of decrypting the one-time key and therefore of recovering the original message. If Bob obtained Alice’s public key by means of Alice’s public-key certificate, then Bob is assured that it is a valid key.

Diagram a, creation of digital envelope. Diagram a, creation of digital envelope. Step 1. Prepare a message. Step 2. Generate a random symmetric key. Step 3. Encrypt the message using symmetric encryption key. Step 4. Encrypt the key using public-key encryption with receiver’s public key. Step 5. The encrypted key to the encrypted message are attached and provides the output as digit envelope. Diagram b, opening a digital envelope. Step 1. Digital envelope is generated as encrypted message and encrypted symmetric key. Step 2. The encrypted message and encrypted key is fed into decryption algorithm. The decryption algorithm gives the output as message. Step 4. The random symmetric key and receiver’s private key are decrypted.



Random Numbers

Uses include generation of:

- Keys for public-key algorithms
- Stream key for symmetric stream cipher
- Symmetric key for use as a temporary session key or in creating a digital envelope
- Handshaking to prevent replay attacks
- Session key

A number of network security algorithms based on cryptography make use of random numbers. For example,

• Generation of keys for the RSA public-key encryption algorithm (described in Chapter 21) and other public-key algorithms.

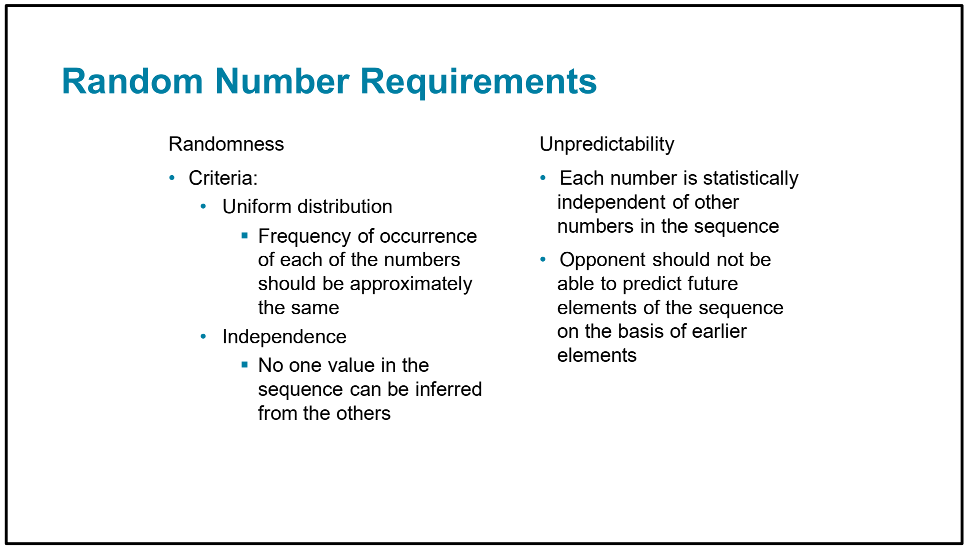
• Generation of a stream key for symmetric stream cipher.

• Generation of a symmetric key for use as a temporary session key or in creating a digital envelope.

• In a number of key distribution scenarios, such as Kerberos (described in Chapter 23), random numbers are used for handshaking to prevent replay attacks.

• Session key generation, whether done by a key distribution center or by one of the principals.

These applications give rise to two distinct and not necessarily compatible requirements for a sequence of random numbers: randomness and unpredictability.



Random Number Requirements

Randomness

- Criteria:
  - Uniform distribution
    - Frequency of occurrence of each of the numbers should be approximately the same
  - Independence
    - No one value in the sequence can be inferred from the others

Unpredictability

- Each number is statistically independent of other numbers in the sequence
- Opponent should not be able to predict future elements of the sequence on the basis of earlier elements

Traditionally, the concern in the generation of a sequence of allegedly random numbers has been that the sequence of numbers be random in some well-defined statistical sense. The following two criteria are used to validate that a sequence of numbers is random:

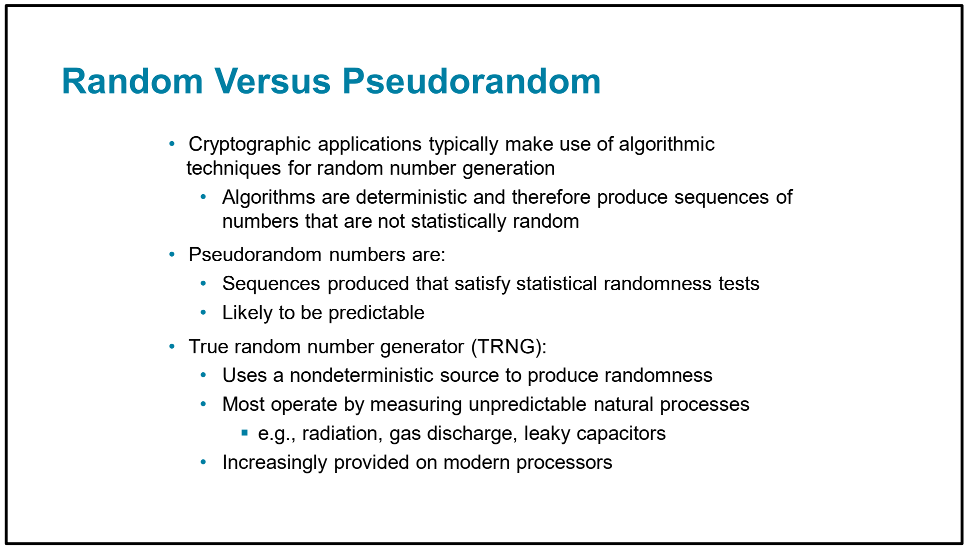
• Uniform distribution: The distribution of numbers in the sequence should be uniform; that is, the frequency of occurrence of each of the numbers should be approximately the same.

• Independence: No one value in the sequence can be inferred from the others.

Although there are well-defined tests for determining that a sequence of numbers matches a particular distribution, such as the uniform distribution, there is no such test to "prove" independence. Rather, a number of tests can be applied to demonstrate if a sequence does not exhibit independence. The general strategy is to apply a number of such tests until the confidence that independence exists is sufficiently strong.

In the context of our discussion, the use of a sequence of numbers that appear statistically random often occurs in the design of algorithms related to cryptography.
In essence, if a problem is too hard or time-consuming to solve exactly, a simpler, shorter approach based on randomization is used to provide an answer with any desired level of confidence.

UNPREDICTABILITY
In applications such as reciprocal authentication and session key generation, the requirement is not so much that the sequence of numbers be statistically random but that the successive members of the sequence are unpredictable. With “true” random sequences, each number is statistically independent of other numbers in the sequence and therefore unpredictable. However, as is discussed shortly, true random numbers are not always used; rather, sequences of numbers that appear to be random are generated by some algorithm. In this latter case, care must be taken that an opponent not be able to predict future elements of the sequence on the basis of earlier elements.



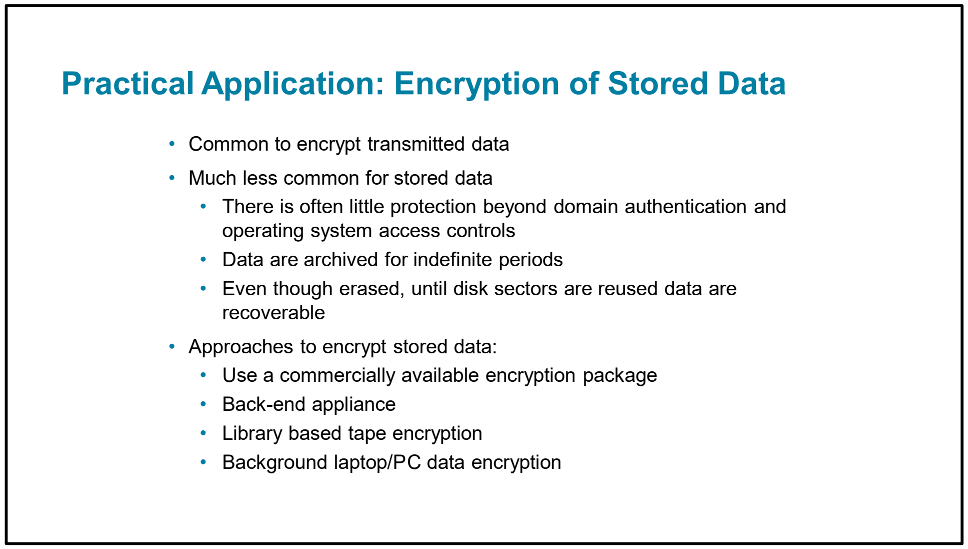
Random Versus Pseudorandom

- Cryptographic applications typically make use of algorithmic techniques for random number generation
  - Algorithms are deterministic and therefore produce sequences of numbers that are not statistically random
- Pseudorandom numbers are:
  - Sequences produced that satisfy statistical randomness tests
  - Likely to be predictable
- True random number generator (TRNG):
  - Uses a nondeterministic source to produce randomness
  - Most operate by measuring unpredictable natural processes
    - e.g., radiation, gas discharge, leaky capacitors
  - Increasingly provided on modern processors

Cryptographic applications typically make use of algorithmic techniques for random number generation. These algorithms are deterministic and therefore produce sequences of numbers that are not statistically random. However, if the algorithm is good, the resulting sequences will pass many reasonable tests of randomness. Such numbers are referred to as **pseudorandom numbers.**

You may be somewhat uneasy about the concept of using numbers generated by a deterministic algorithm as if they were random numbers. Despite what might be called philosophical objections to such a practice, it generally works. That is, under most circumstances, pseudorandom numbers will perform as well as if they were random for a given use. The phrase “as well as” is unfortunately subjective, but the use of pseudorandom numbers is widely accepted. The same principle applies in statistical applications, in which a statistician takes a sample of a population and assumes the results will be approximately the same as if the whole population were measured.

A true random number generator (TRNG) uses a nondeterministic source to produce randomness. Most operate by measuring unpredictable natural processes, such as pulse detectors of ionizing radiation events, gas discharge tubes, and leaky capacitors. Intel has developed a commercially available chip that samples thermal noise by amplifying the voltage measured across undriven resistors [JUN99]. A group at Bell Labs has developed a technique that uses the variations in the response time of raw read requests for one disk sector of a hard disk [JAKO98]. LavaRnd is an open source project for creating truly random numbers using inexpensive cameras, open source code, and inexpensive hardware. The system uses a saturated charge- coupled device (CCD) in a light-tight can as a chaotic source to produce the seed. Software processes the result into truly random numbers in a variety of formats. The first commercially available TRNG that achieves bit production rates comparable with that of PRNGs is the Intel digital random number generator (DRNG) [TAYL11], offered on new multicore chips since May 2012.



Practical Application: Encryption of Stored Data

- Common to encrypt transmitted data
- Much less common for stored data
  - There is often little protection beyond domain authentication and operating system access controls
  - Data are archived for indefinite periods
  - Even though erased, until disk sectors are reused data are recoverable
- Approaches to encrypt stored data:
  - Use a commercially available encryption package
  - Back-end appliance
  - Library based tape encryption
  - Background laptop/PC data encryption

One of the principal security requirements of a computer system is the protection of stored data. Security mechanisms to provide such protection include access control, intrusion detection, and intrusion prevention schemes, all of which are discussed in this book. The book also describes a number of technical means by which these various security mechanisms can be made vulnerable. But beyond technical approaches, these approaches can become vulnerable because of human factors. We list a few examples here, based on [ROTH05].

• In December of 2004, Bank of America employees backed up and sent to its backup data center tapes containing the names, addresses, bank account numbers, and Social Security numbers of 1.2 million government workers enrolled in a charge-card account. None of the data were encrypted. The tapes never arrived and indeed have never been found. Sadly, this method of backing up and shipping data is all too common. As an another example, in April of 2005, Ameritrade blamed its shipping vendor for losing a backup tape containing unencrypted information on 200,000 clients.

• In April of 2005, San Jose Medical group announced that someone had physically stolen one of its computers and potentially gained access to 185,000 unencrypted patient records.

• There have been countless examples of laptops lost at airports, stolen from a parked car, or taken while the user is away from his or her desk. If the data on the laptop's hard drive are unencrypted, all of the data are available to the thief.

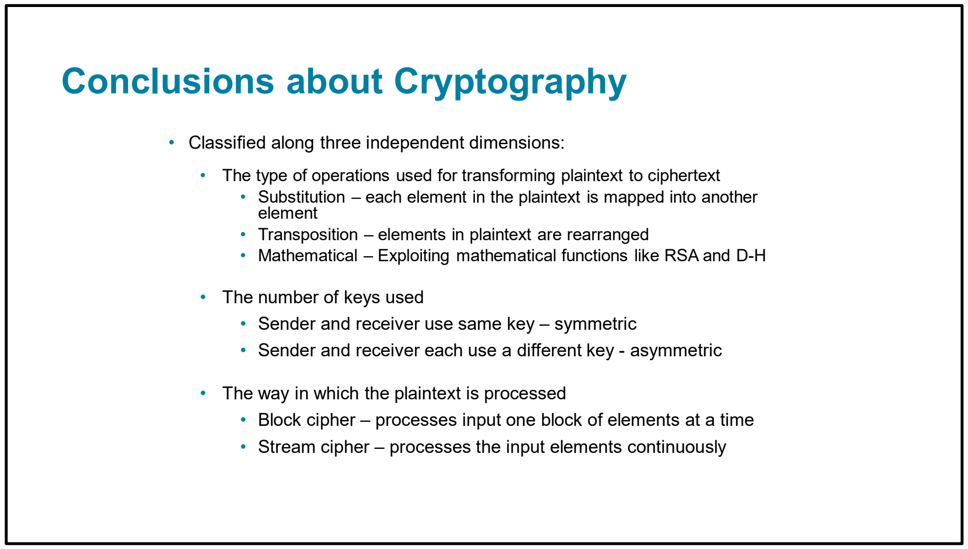
Although it is now routine for businesses to provide a variety of protections, including encryption, for information that is transmitted across networks, via the Internet, or via wireless devices, once data are stored locally (referred to as *data at rest),* there is often little protection beyond domain authentication and operating system access controls. Data at rest are often routinely backed up to secondary storage such as CDROM or tape, archived for indefinite periods. Further, even when data are erased from a hard disk, until the relevant disk sectors are reused, the data are recoverable. Thus it becomes attractive, and indeed should be mandatory, to encrypt data at rest and combine this with an effective encryption key management
scheme.

There are a variety of ways to provide encryption services. A simple approach available for use on a laptop is to use a commercially available encryption package such as Pretty Good Privacy (PGP). PGP enables a user to generate a key from a password and then use that key to encrypt selected files on the hard disk. The PGP package does not store the password. To recover a file, the user enters the password, PGP generates the password, and PGP decrypts the file. So long as the user protects his or her password and does not use an easily guessable password, the files are fully protected while at rest. Some more recent approaches are listed in [COLL06]:

• **Back-end appliance**: This is a hardware device that sits between servers and storage systems and encrypts all data going from the server to the storage system and decrypts data going in the opposite direction. These devices encrypt data at close to wire speed, with very little latency. In contrast, encryption software on servers and storage systems slows backups. A system man ager configures the appliance to accept requests from specified clients, for which unencrypted data are supplied.

• **Library-based tape encryption**: This is provided by means of a co-processor board embedded in the tape drive and tape library hardware. The co-processor encrypts data using a nonreadable key configured into the board. The tapes can then be sent off-site to a facility that has the same tape drive hardware. The key can be exported via secure e-mail or a small flash drive that is transported securely. If the matching tape drive hardware co-processor is not available at the other site, the target facility can use the key in a software decryption package to recover the data.

• **Background laptop and PC data encryption**: A number of vendors offer software products that provide encryption that is transparent to the application and the user. Some products encrypt all or designated files and folders. Other products, such as Windows BitLocker and MacOS FileVault, encrypt an entire disk or disk image located on either the user's hard drive or maintained on a network storage device, with all data on the virtual disk encrypted. Various key management solutions are offered to restrict access to the owner of the data.



Conclusions about Cryptography

- Classified along three independent dimensions:
  - The type of operations used for transforming plaintext to ciphertext
    - Substitution – each element in the plaintext is mapped into another element
    - Transposition – elements in plaintext are rearranged
    - Mathematical – Exploiting mathematical functions like RSA and D-H
  - The number of keys used
    - Sender and receiver use same key – symmetric
    - Sender and receiver each use a different key - asymmetric
  - The way in which the plaintext is processed
    - Block cipher – processes input one block of elements at a time
    - Stream cipher – processes the input elements continuously

Cryptographic systems are generically classified along three independent dimensions:

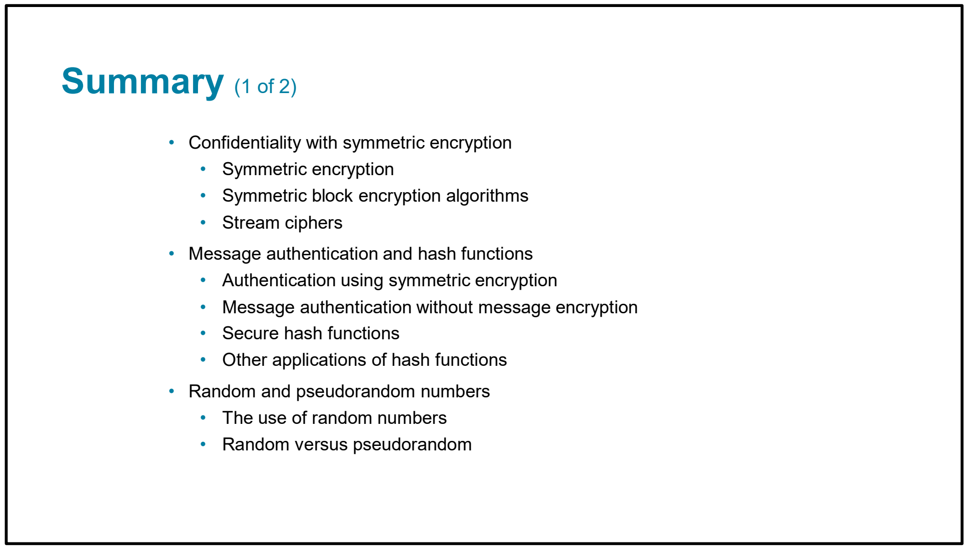
1. The type of operations used for transforming plaintext to ciphertext. All encryption algorithms are based on two general principles: substitution, in which each element in the plaintext (bit, letter, group of bits or letters) is mapped into another element, and transposition, in which elements in the plaintext are rearranged. The fundamental requirement is that no information be lost (i.e., that all operations be reversible). Most systems, referred to as product systems, involve multiple stages of substitutions and transpositions.

2. The number of keys used. If both sender and receiver use the same key, the system is referred to as symmetric, single-key, secret-key, or conventional encryption. If the sender and receiver each use a different key, the system is referred to as asymmetric, two-key, or public-key encryption.

3. The way in which the plaintext is processed. A *block cipher* processes the input one block of elements at a time, producing an output block for each input block. A *stream cipher* processes the input elements continuously, producing output one element at a time, as it goes along.

Observations and misconceptions

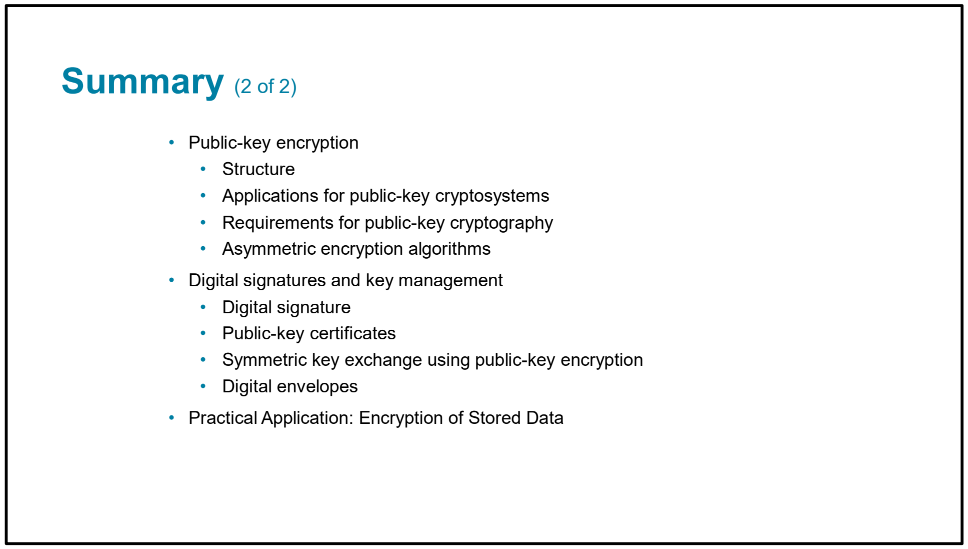
- Symmetric key is not necessarily worst than a public key algorithm, it mostly depends on the length of the key and the mathematical functions required to be broken
- Because of the computational overhead of current public-key encryption schemes, there seems no foreseeable likelihood that symmetric encryption will be abandoned
- For public-key key distribution, some form of protocol is needed, often involving a central agent, and the procedures involved are no simpler or any more efficient than those required for symmetric encryption.



Summary (1 of 2)

- Confidentiality with symmetric encryption
  - Symmetric encryption
  - Symmetric block encryption algorithms
  - Stream ciphers
- Message authentication and hash functions
  - Authentication using symmetric encryption
  - Message authentication without message encryption
  - Secure hash functions
  - Other applications of hash functions
- Random and pseudorandom numbers
  - The use of random numbers
  - Random versus pseudorandom

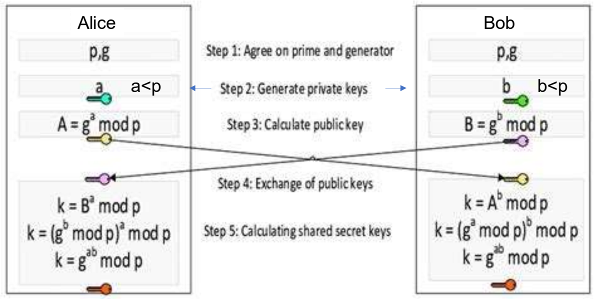
Chapter 2 summary.



Summary (2 of 2)

- Public-key encryption
  - Structure
  - Applications for public-key cryptosystems
  - Requirements for public-key cryptography
  - Asymmetric encryption algorithms
- Digital signatures and key management
  - Digital signature
  - Public-key certificates
  - Symmetric key exchange using public-key encryption
  - Digital envelopes
- Practical Application: Encryption of Stored Data

Appendix: Another example of Diffie-Hellman



Alice
Bob
p,g
Step 1: Agree on prime and generator
p,g
a
a<p
Step 2: Generate private keys
b
b<p
$A = g^a \bmod p$
Step 3: Calculate public key
$B = g^b \bmod p$
Step 4: Exchange of public keys
$k = B^a \bmod p$
$k = (g^b \bmod p)^a \bmod p$
$k = g^{ab} \bmod p$
Step 5: Calculating shared secret keys
$k = A^b \bmod p$
$k = (g^a \bmod p)^b \bmod p$
$k = g^{ab} \bmod p$

Note 1: $(g^b \bmod p)^a = g^{ba} \bmod p$

i.e., exponentiation does not change module result

Note 2: $g^{ba} \bmod p \bmod p = g^{ba} \bmod p$

i.e., multiple module operations don't change the result