ECE590
Computer and Information Security

Fall 2018

Cryptography

Tyler Bletsch

Duke University

Some slides adapted from slideware accompanying
“Computer Security: Principles and Practice” by William Stallings and Lawrie Brown

REAL advice for using cryptography

- I’m about to teach cryptography basics, which you should know
- However, you **should not reach for these functions in most real-world programming scenarios!!**
- **Repeat after me:**

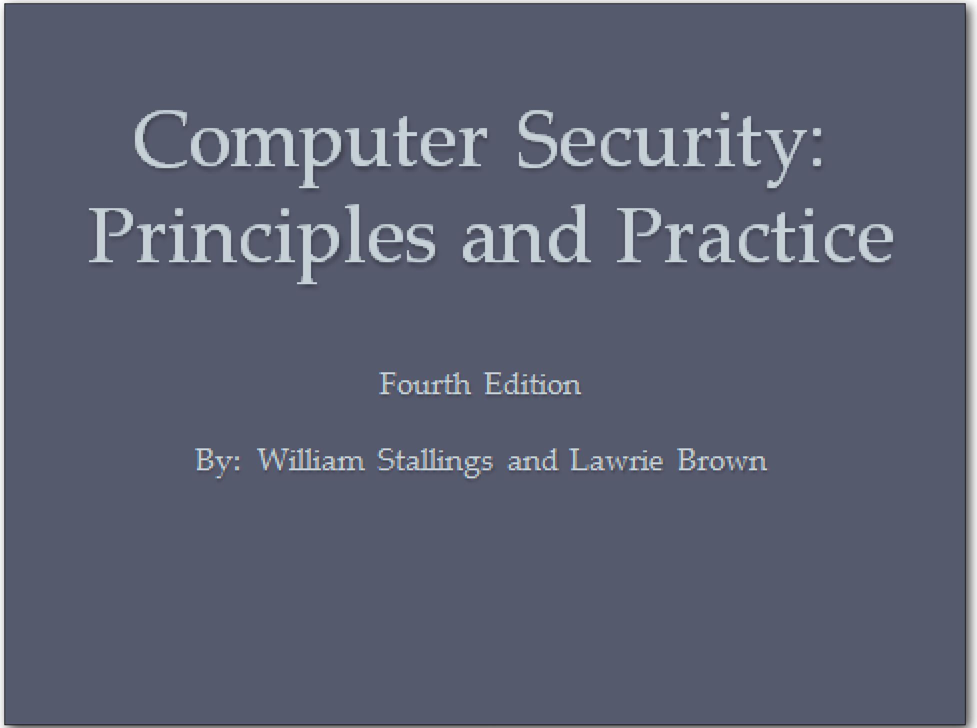
Don’t roll your own crypto!

Don’t roll your own crypto!

Don’t roll your own crypto!

I’ll provide more detailed advice after we understand the theory...

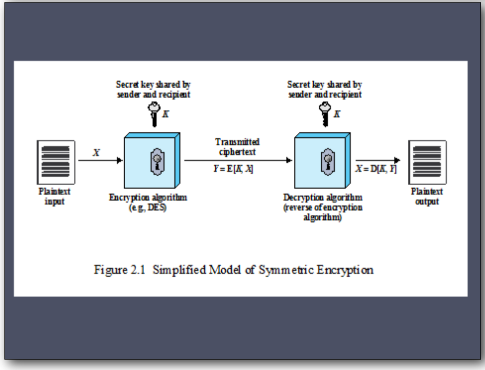
Introducing the “grey slides”



Computer Security: Principles and Practice

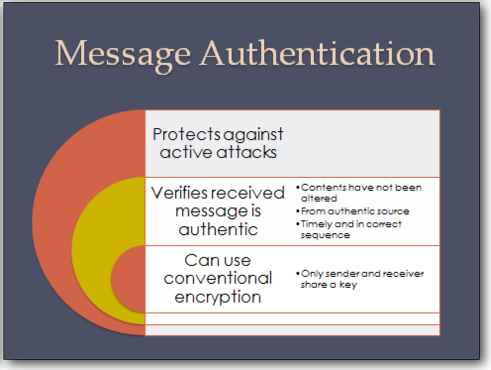
Fourth Edition

By: William Stallings and Lawrie Brown



Secret key shared by sender and recipient
K
Secret key shared by sender and recipient
K
X
Transmitted ciphertext
Y = E[K, X]
X = D[K, Y]
Plaintext input
Encryption algorithm (e.g, DES)
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

Figure 2.1 Simplified Model of Symmetric Encryption



Message Authentication

Protects against active attacks

Verifies received message is authentic

- Contents have not been altered
- From authentic source
- Timely and in correct sequence

Can use conventional encryption

- Only sender and receiver share a key

- From the textbook publisher
- Perfectly fine for the most part, except...
  - A bit out of date (you’ll see me address this with my slides)
  - Diagrams haven’t been updated since the 90s (lol)
  - Randomly wraps words in needless colored shapes like a drunk preshooler (why???)

Crypto basics summary

- Symmetric (secret key) cryptography
  - $c = E_s(p,k)$
  - $p = D_s(c,k)$

c = ciphertext
p = plaintext
k = secret key
E_s = Encryption function (symmetric)
D_s = Decryption function (symmetric)

- Message Authentication Codes (MAC)
  - Generate and append: H(p+k), E(H(p),k), or tail of E(p,k)
  - Check: A match proves recipient knew k

H = Hash function

- Asymmetric (public key) cryptography
  - $c = E_a(p,k_{pub})$
  - $p = D_a(c,k_{priv})$
  - k_{pub} and k_{priv} generated together, mathematically related

E_a = Encryption function (asymmetric)
D_a = Decryption function (asymmetric)
k_{pub} = public key
k_{priv} = private key

- Digital signatures
  - Generate and append: $s = E_a(H(p),k_{priv})$
  - Check: $D_a(s,k_{pub})$==H(p) proves sender knew k_{priv}

s = signature

Symmetric (Secret Key) Encryption

Symmetric Encryption

- The universal technique for providing confidentiality for transmitted or stored data
- Also referred to as conventional encryption or single-key encryption
- Two requirements for secure use:
  - Need a strong encryption algorithm
  - Sender and receiver must have obtained copies of the secret key in a secure fashion and must keep the key secure

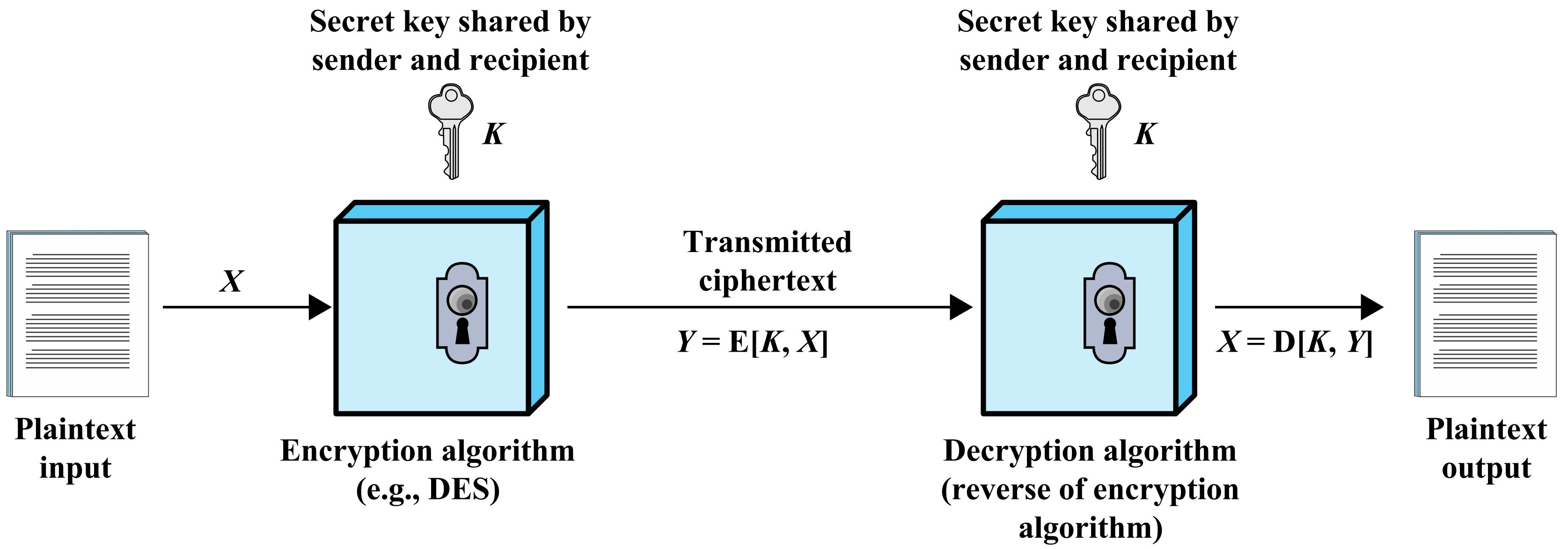

Secret key shared by sender and recipient
K
Secret key shared by sender and recipient
K
X
Transmitted ciphertext
Y = E[K, X]
X = D[K, Y]
Plaintext input
Encryption algorithm (e.g., DES)
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

Figure 2.1 Simplified Model of Symmetric Encryption

Attacking Symmetric Encryption

Cryptanalytic Attacks

- Rely on:
  - Nature of the algorithm
  - Some knowledge of the general characteristics of the plaintext
  - Some sample plaintext-ciphertext pairs
- Exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or the key being used
  - If successful all future and past messages encrypted with that key are compromised

Brute-Force Attacks

- Try all possible keys on some ciphertext until an intelligible translation into plaintext is obtained
  - On average half of all possible keys must be tried to achieve success

Hypothetical *bad* symmetric encryption algorithm: XOR

- A lot of encryption algorithms rely on properties of XOR
  - Can think of A^B as “Flip a bit in A if corresponding bit in B is 1”
  - If you XOR by same thing twice, you get the data back
  - XORing by a random bit string yields NO info about original data
    - Each bit has a 50% chance of having been flipped

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

```
>>> a=501
>>> b=199
>>> a ^= b
>>> print a
306
>>> a ^= b
>>> print a
501
```

- Could consider XOR itself to be a symmetric encryption algorithm (but it sucks at it!) – can be illustrative to explore

- Simple XOR encryption algorithm:
  - E(p,k) = p ^ k (keep repeating k as often as needed to cover p)
  - D(c,k) = c ^ k (same algorithm both ways!)

XOR “encryption” demo

```
Plaintext: 'Hello'
Key      : 'key'

                 H        e        l        l        o
Plaintext : 01001000 01100101 01101100 01101100 01101111
                 k        e        y Key repeats> k        e
Key       : 01101011 01100101 01111001 01101011 01100101
Ciphertext:
   ^ XOR result

Ciphertext: 00100011 00000000 00010101 00000111 00001010
Key       : 01101011 01100101 01111001 01101011 01100101
Decrypted :
   ^ XOR result
```

Table 20.1 Types of Attacks on Encrypted Messages

Type of Attack	Known to Cryptanalyst
Ciphertext only	•Encryption algorithm •Ciphertext to be decoded
Known plaintext	•Encryption algorithm •Ciphertext to be decoded •One or more plaintext-ciphertext pairs formed with the secret key
Chosen plaintext	•Encryption algorithm •Ciphertext to be decoded •Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key
Chosen ciphertext	•Encryption algorithm •Ciphertext to be decoded •Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key
Chosen text	•Encryption algorithm •Ciphertext to be decoded •Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key •Purported ciphertext chosen by cryptanalyst, together with its corresponding decrypted plaintext generated with the secret key

Attacking XOR

- Known plaintext attack:

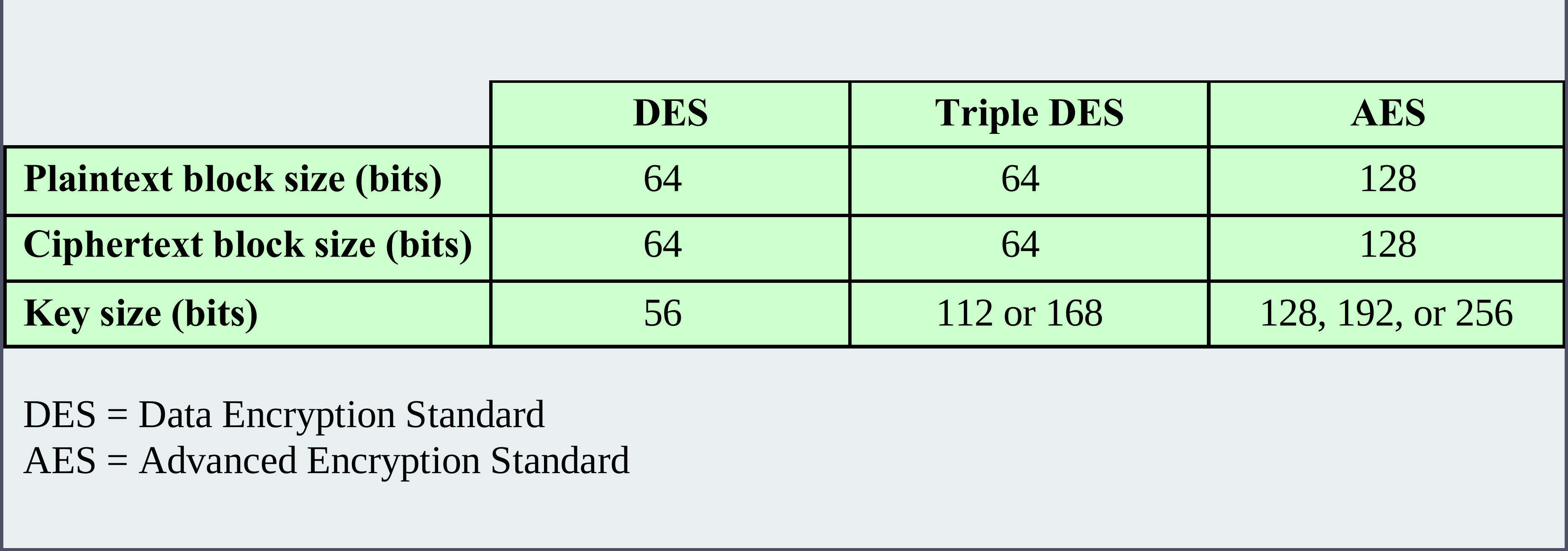
  ```
  Given plaintext  : 01001000 01100101 01101100 01101100 01101111
  Given ciphertext : 00100011 00000000 00010101 00000111 00001010
  XOR result       : 01101011 01100101 01111001 01101011 01100101
                      ^^ it's the key!!!
  ```

- Chosen plaintext attack:

  ```
  Chosen plaintext : 00000000 00000000 00000000 00000000 00000000
  Given ciphertext : 01101011 01100101 01111001 01101011 01100101
  XOR result       : 01101011 01100101 01111001 01101011 01100101
                      ^^ it's the key!!!
  ```

- Ciphertext only attack:
  - `Ciphertext: 00100011 00000000 00010101 00000111 00001010`
  - "I assume the plaintext had ASCII text with lowercase letters, and in all such letters bit 6 is 1, but none of the ciphertext has bit 6 set, so i bet the key is most/all lower case letters"
  - "The second byte is all zeroes, which means the second byte of the key and plaintext are equal"
  - etc....
- **Conclusion: XOR is a sucky encryption algorithm**

Table 2.1



	DES	Triple DES	AES
Plaintext block size (bits)	64	64	128
Ciphertext block size (bits)	64	64	128
Key size (bits)	56	112 or 168	128, 192, or 256

DES = Data Encryption Standard
AES = Advanced Encryption Standard

Comparison of Three Popular Symmetric Encryption Algorithms

Data Encryption Standard (DES)

1999

- Until ~~recently~~ was the most widely used encryption scheme
  - FIPS PUB 46
  - Referred to as the Data Encryption Algorithm (DEA)
  - Uses 64 bit plaintext block and 56 bit key to produce a 64 bit ciphertext block
- Strength concerns:
  - Concerns about the algorithm itself
    - DES is the most studied encryption algorithm in existence
  - Concerns about the use of a 56-bit key
    - The speed of commercial off-the-shelf processors makes this key length woefully inadequate

Table 2.2

Key size (bits)	Cipher	Number of Alternative Keys	Time Required at 10^9 decryptions/s	Time Required at 10^{13} decryptions/s
56	DES	2^{56} ≈ 7.2 ´ 10^{16}	2^{55} ns = 1.125 years	1 hour
128	AES	2^{128} ≈ 3.4 ´ 10^{38}	2^{127} ns = 5.3 ´ 10^{21} years	5.3 ´ 10^{17} years
168	Triple DES	2^{168} ≈ 3.7 ´ 10^{50}	2^{167} ns = 5.8 ´ 10^{33} years	5.8 ´ 10^{29} years
192	AES	2^{192} ≈ 6.3 ´ 10^{57}	2^{191} ns = 9.8 ´ 10^{40} years	9.8 ´ 10^{36} years
256	AES	2^{256} ≈ 1.2 ´ 10^{77}	2^{255} ns = 1.8 ´ 10^{60} years	1.8 ´ 10^{56} years

Average Time Required for Exhaustive Key Search

Triple DES (3DES)

- Repeats basic DES algorithm three times using either two or three unique keys
- First standardized for use in financial applications in ANSI standard X9.17 in 1985
- Attractions:
  - 168-bit key length overcomes the vulnerability to brute-force attack of DES
  - Underlying encryption algorithm is the same as in DES
- Drawbacks:
  - Algorithm is sluggish in software
  - Uses a 64-bit block size

Advanced Encryption Standard (AES)

Needed a replacement for 3DES

3DES was not reasonable for long term use

NIST called for proposals for a new AES in 1997

- Should have a security strength equal to or better than 3DES
- Significantly improved efficiency
- Symmetric block cipher
- 128 bit data and 128/192/256 bit keys

Selected Rijndael in November 2001

Published as FIPS 197

Computationally Secure Encryption Schemes

- Encryption is computationally secure if:
  - Cost of breaking cipher exceeds value of information
  - Time required to break cipher exceeds the useful lifetime of the information
- Usually very difficult to estimate the amount of effort required to break *algorithm* (cryptanalysis)
- Can estimate time/cost of a brute-force attack

Practical Security Issues

- Typically symmetric encryption is applied to a unit of data larger than a single 64-bit or 128-bit block
- Electronic codebook (ECB) mode is the simplest approach to multiple-block encryption
    - Each block of plaintext is encrypted using the same key
    - Cryptanalysts may be able to exploit regularities in the plaintext
- Modes of operation
    - Alternative techniques developed to increase the security of symmetric block encryption for large sequences
    - Overcomes the weaknesses of ECB

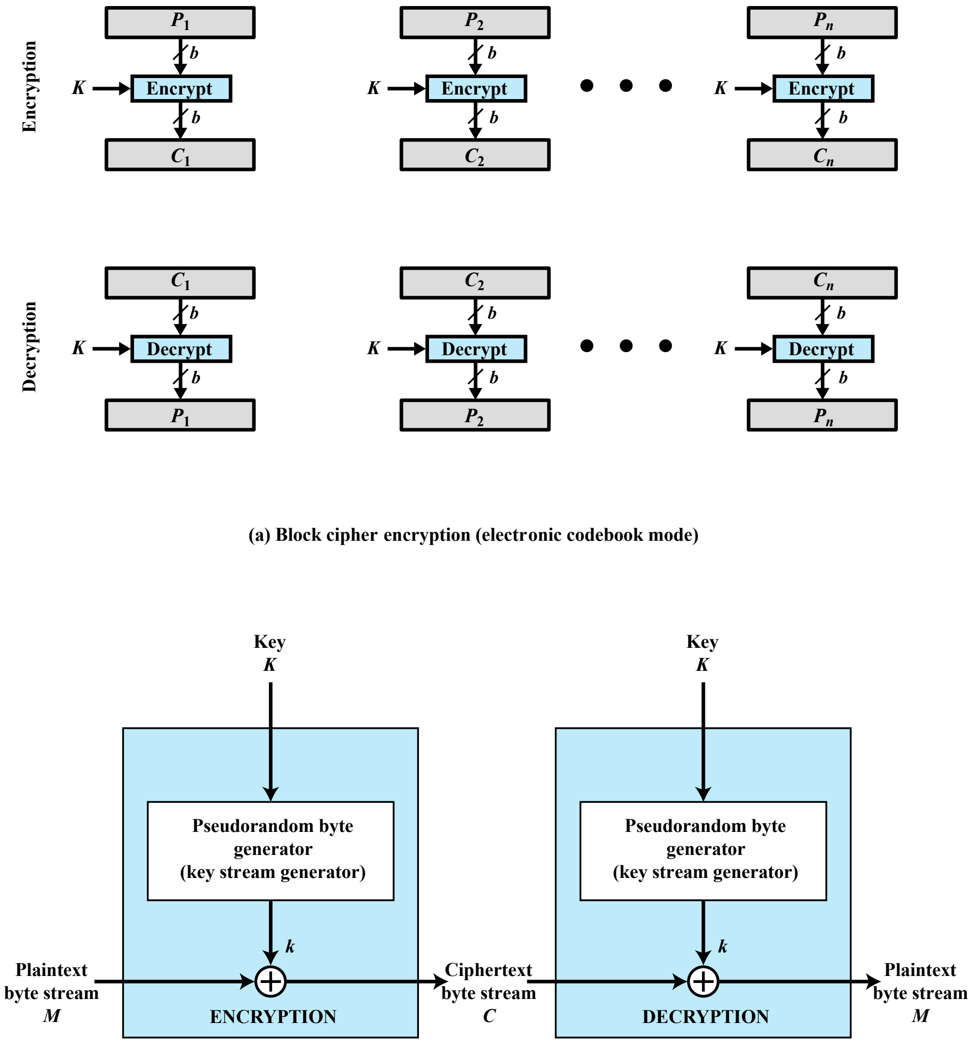
Block & Stream Ciphers

Block Cipher

- Processes the input one block of elements at a time
- Produces an output block for each input block
- Can reuse keys
- More common

Stream Cipher

- Processes the input elements continuously
- Produces output one element at a time
- Primary advantage is that they are almost always faster and use far less code
- Encrypts plaintext one byte at a time
- Pseudorandom stream is one that is unpredictable without knowledge of the input key

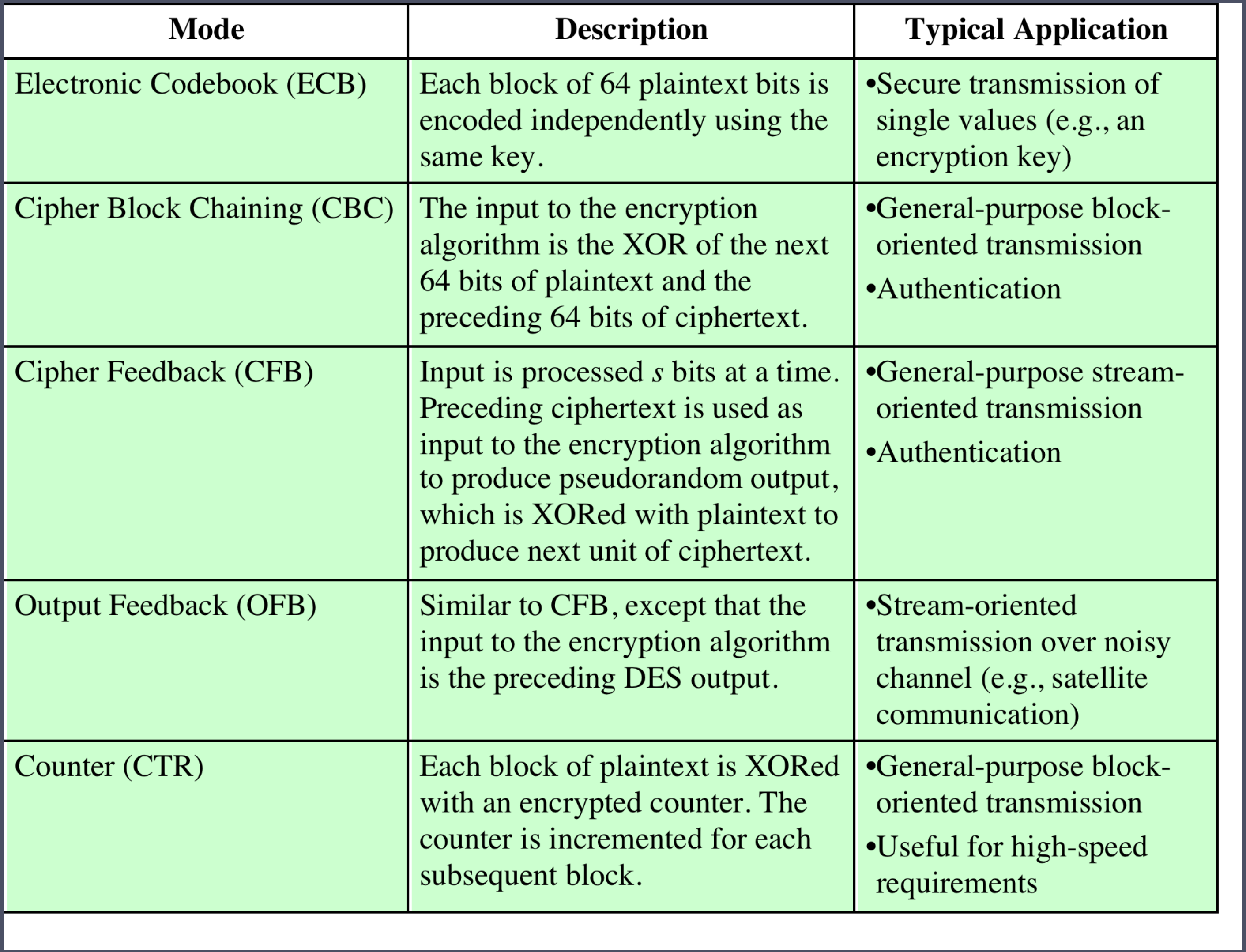


(a) Block cipher encryption (electronic codebook mode)

(b) Stream encryption

Figure 2.2 Types of Symmetric Encryption

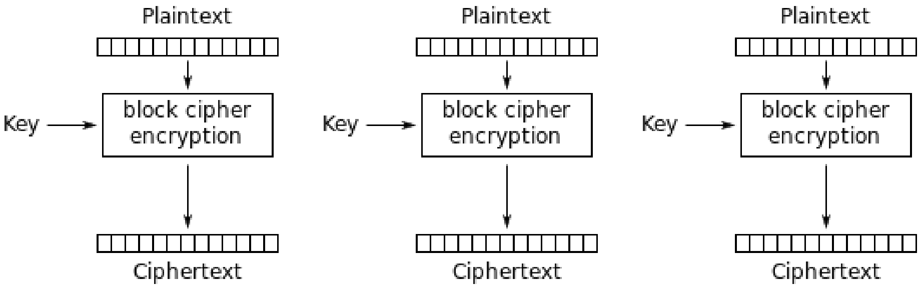
Table 20.3
Block Cipher Modes of Operation



Mode	Description	Typical Application
Electronic Codebook (ECB)	Each block of 64 plaintext bits is encoded independently using the same key.	•Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next 64 bits of plaintext and the preceding 64 bits of ciphertext.	•General-purpose block-oriented transmission •Authentication
Cipher Feedback (CFB)	Input is processed *s* bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	•General-purpose stream-oriented transmission •Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding DES output.	•Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	•General-purpose block-oriented transmission •Useful for high-speed requirements

Modes of operation are critical!

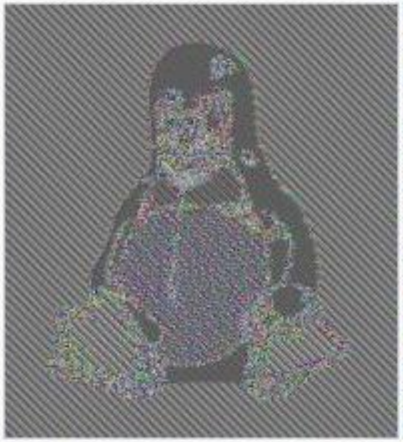
- Electronic Codebook (ECB) is what you’d come up with naively:
  “Just apply the key to each block”



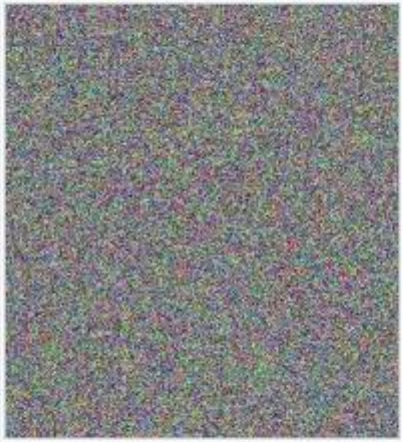
Electronic Codebook (ECB) mode encryption

- But this means that identical blocks give identical ciphertext, which can be informative to an attacker...

Original image



Encrypted using ECB mode ☹



Modes other than ECB result in pseudo-randomness ☺

See PoC||GTFO 4:13
for a poem about this

Figures from Wikipedia “Block cipher mode of operation”

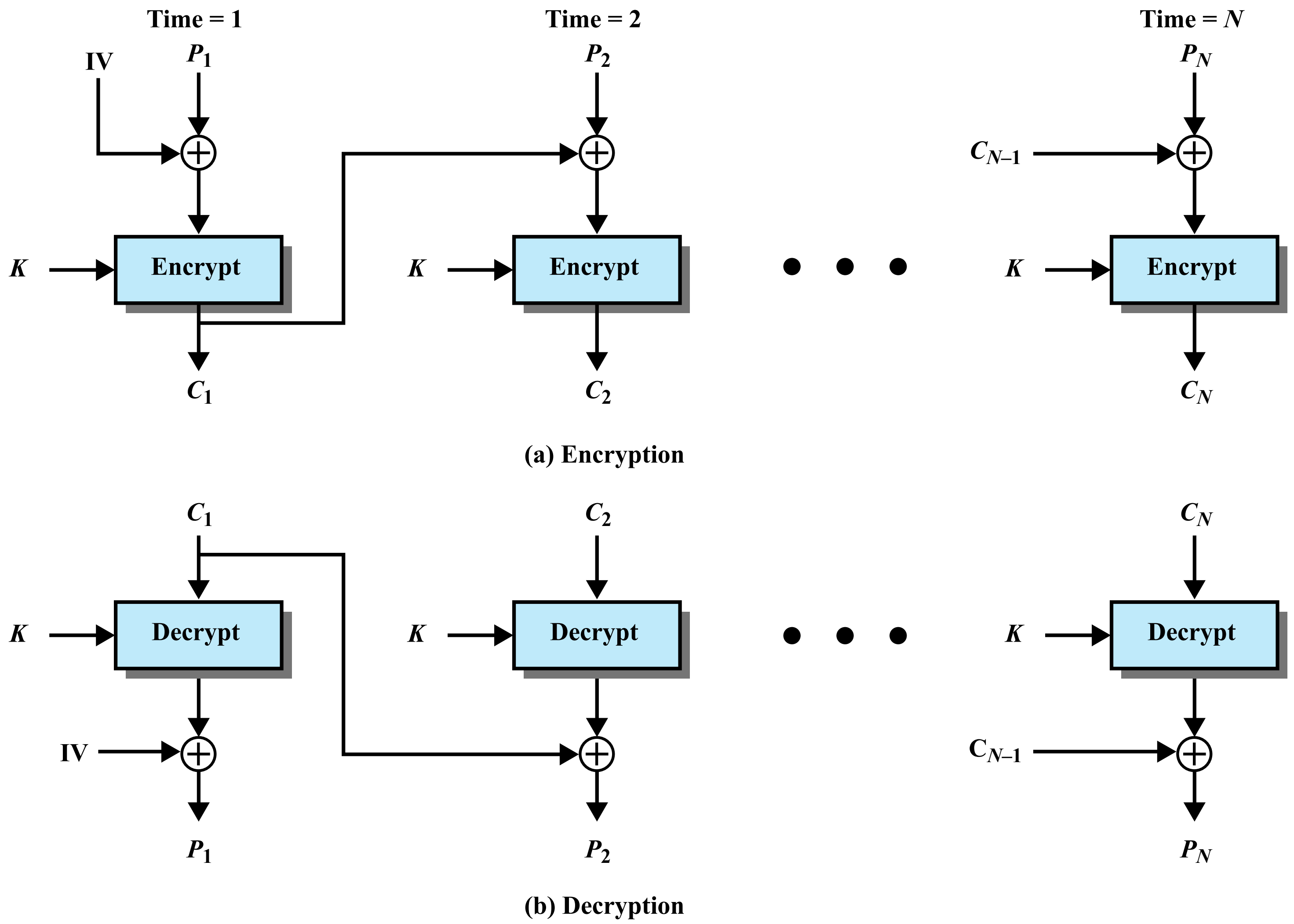


Figure 20.7 Cipher Block Chaining (CBC) Mode

The Initialization Vector

- The previous slide showed an “IV” (Initialization Vector”) used to start the chain (it’s XORed with the first block of plaintext). Something like this is used in many modes.
  - IV is random per-message; ensures first block of two ciphertexts don’t match just because plaintexts match.
- The IV must be known to both the sender and receiver, typically not a secret (often included in the communication).
- IV *integrity* is important: If an opponent is able to fool the receiver into using a different value for IV, then the opponent is able to invert selected bits in the first block of plaintext. Other attacks, too...
  - A more detailed discussion can be found here.

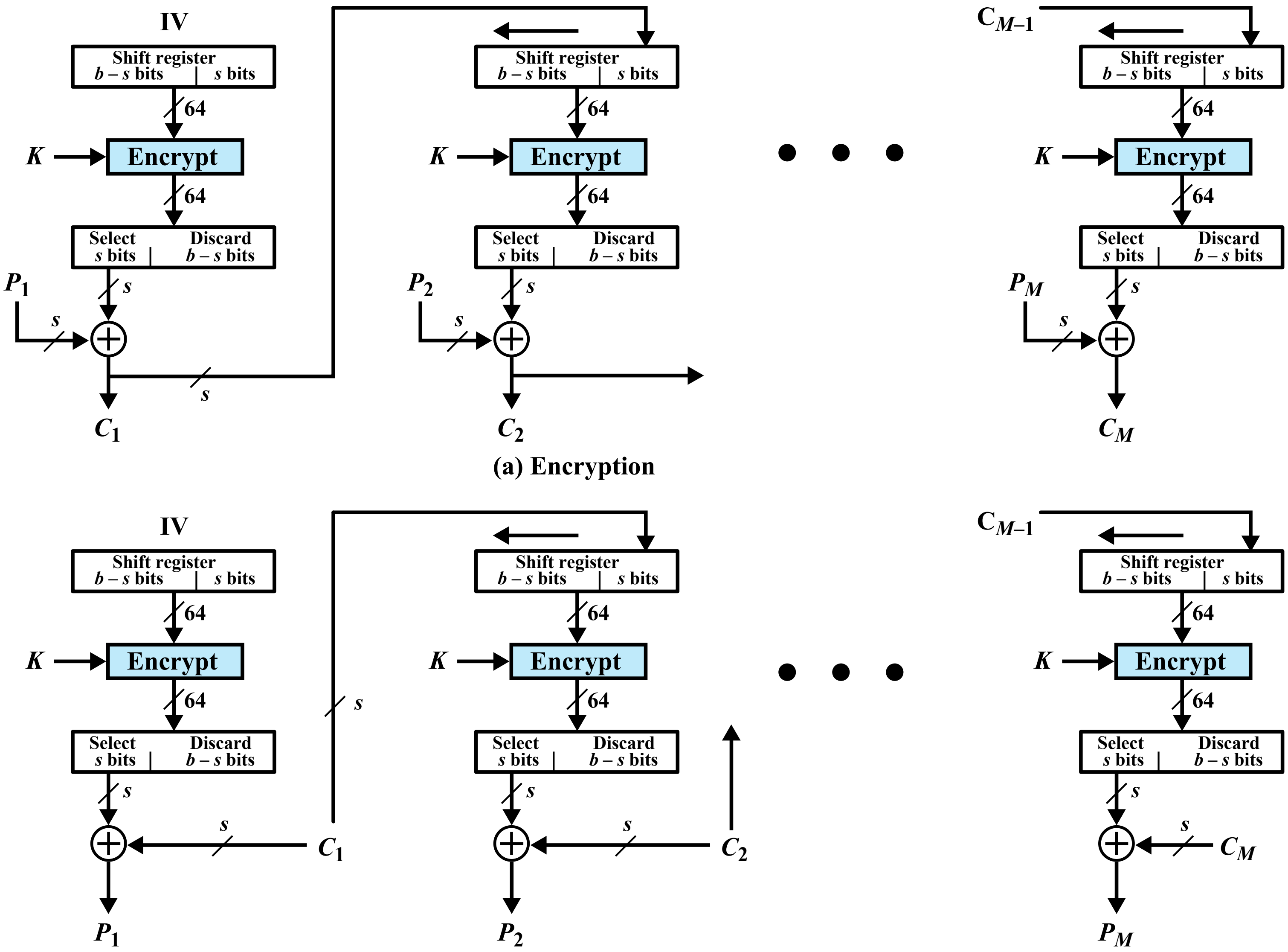


Figure 20.8 ***s*-bit Cipher Feedback (CFB) Mode**

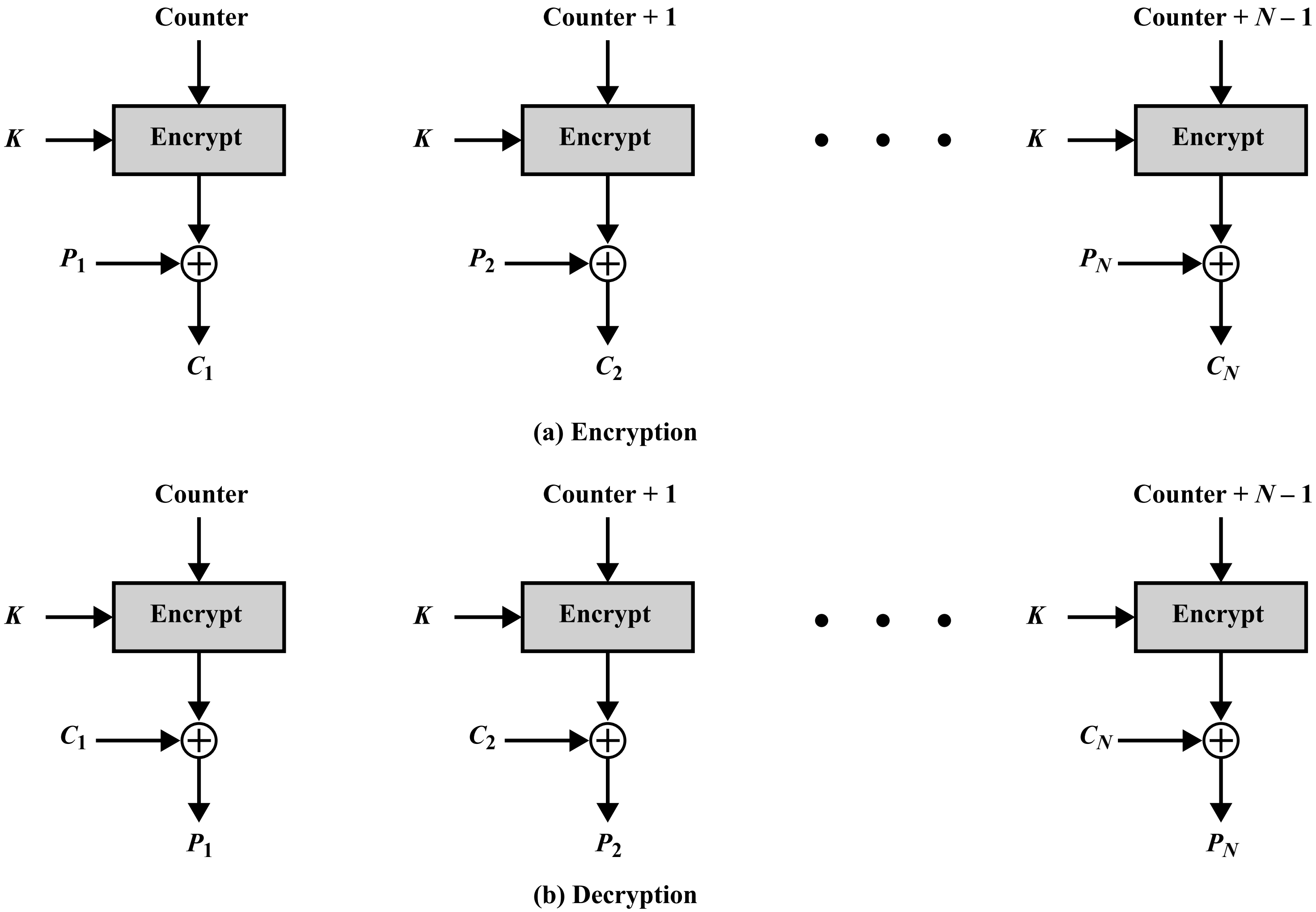
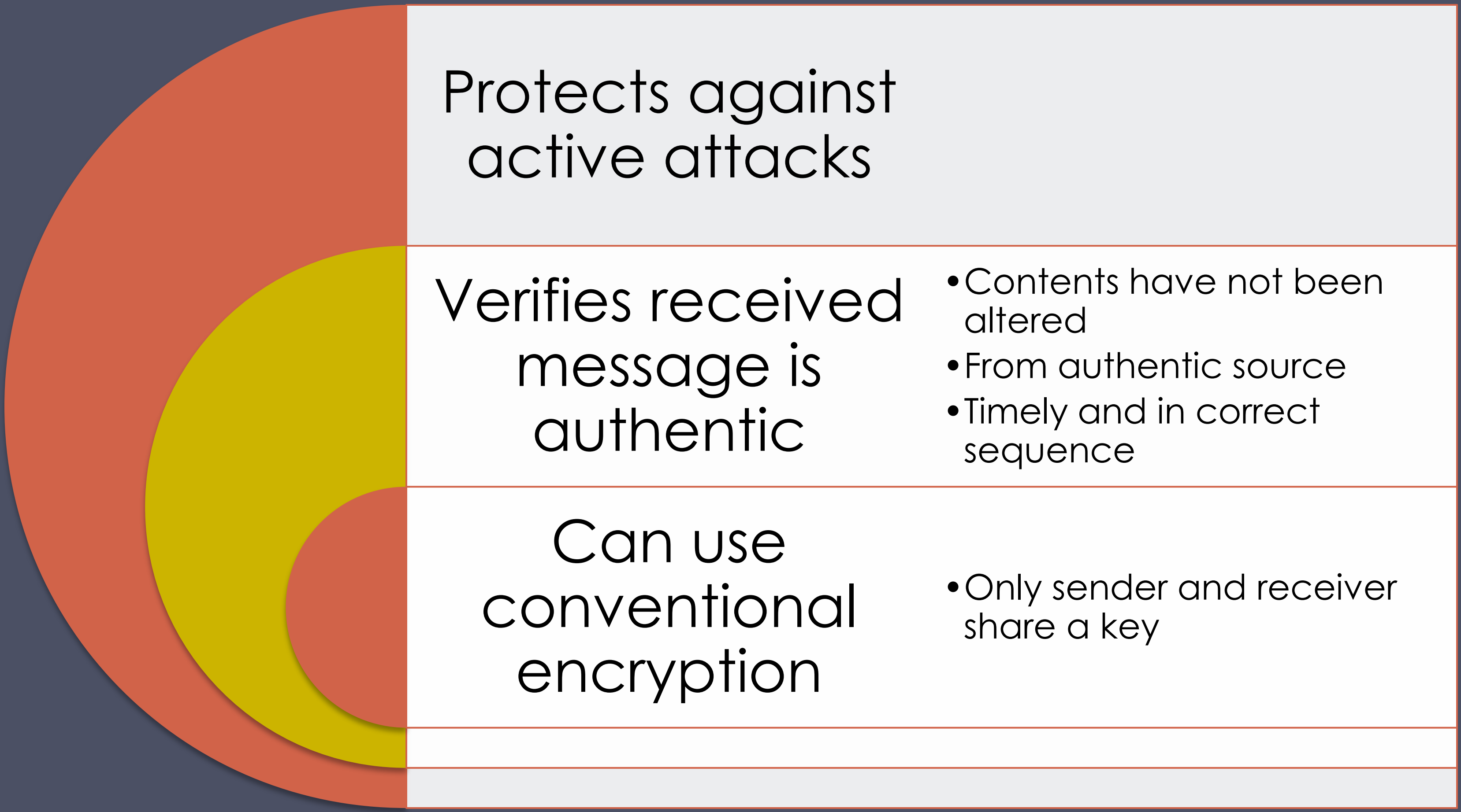


Figure 20.9 Counter (CTR) Mode

Message Authentication Codes (MAC) and secure hash functions

Message Authentication



Protects against active attacks

Verifies received message is authentic

- Contents have not been altered
- From authentic source
- Timely and in correct sequence

Can use conventional encryption

- Only sender and receiver share a key

MAC overview

- Core idea: want to prove that message was sent by Alice
  - Can't do that directly (no time machine)
  - Instead show that whoever *did* send it had access to a secret key

- Can use symmetric encryption:
  - Include last block of E(message,key) in CBC mode – sender could only generate that data if they had the key and message at the same time
  - Shown in next slides

- Can use hash functions:
  - Non-reversible, arbitrary size input to fixed size output
  - Various schemes (shown in slide after next)

Message Authentication Without Confidentiality

- Message encryption by itself does not provide a secure form of authentication
- It is possible to combine authentication and confidentiality in a single algorithm by encrypting a message plus its authentication tag
- Typically message authentication is provided as a separate function from message encryption
- Situations in which message authentication without confidentiality may be preferable include:
    - There are a number of applications in which the same message is broadcast to a number of destinations
    - An exchange in which one side has a heavy load and cannot afford the time to decrypt all incoming messages
    - Authentication of a computer program in plaintext is an attractive service
- Thus, there is a place for both authentication and encryption in meeting security requirements

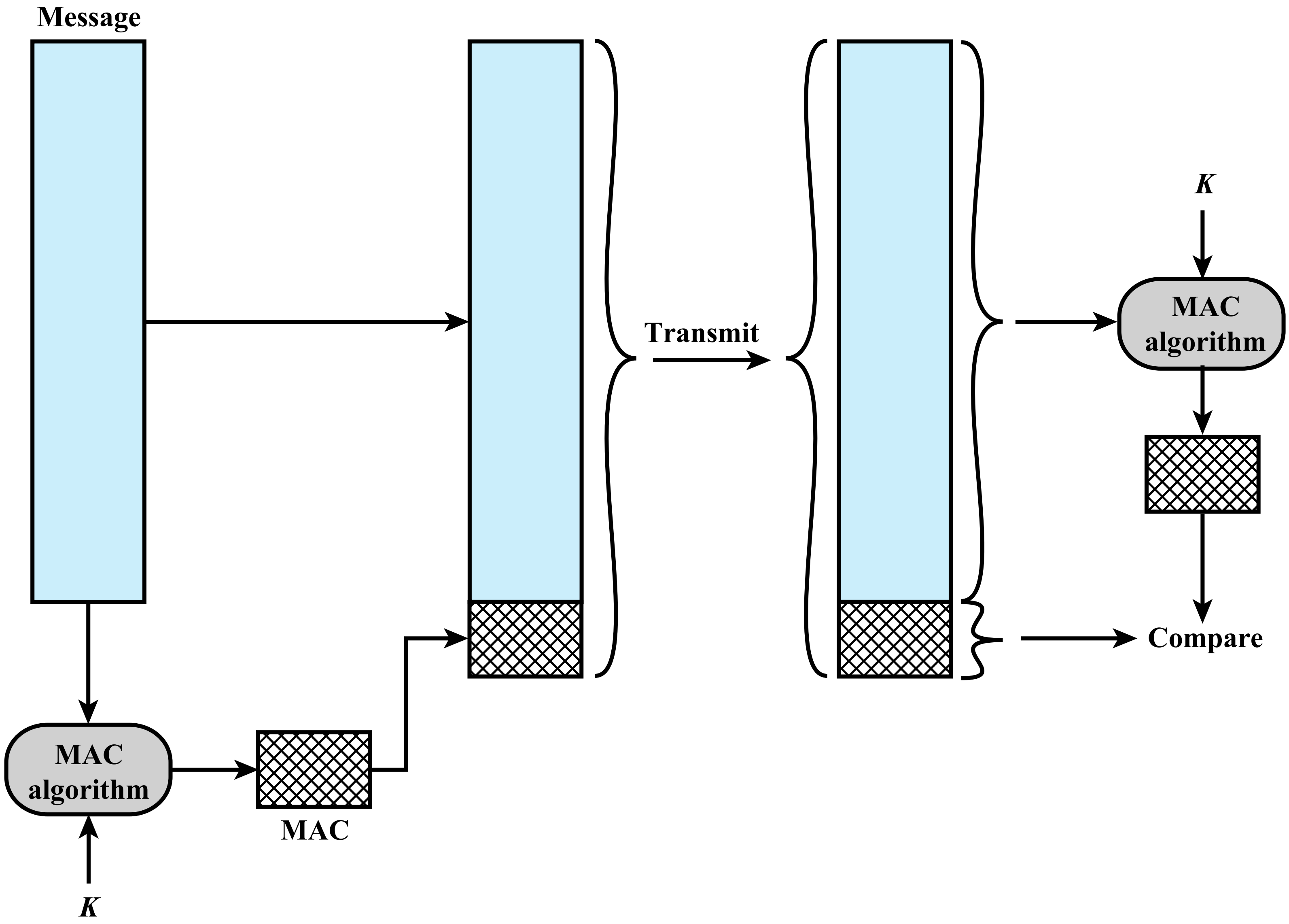
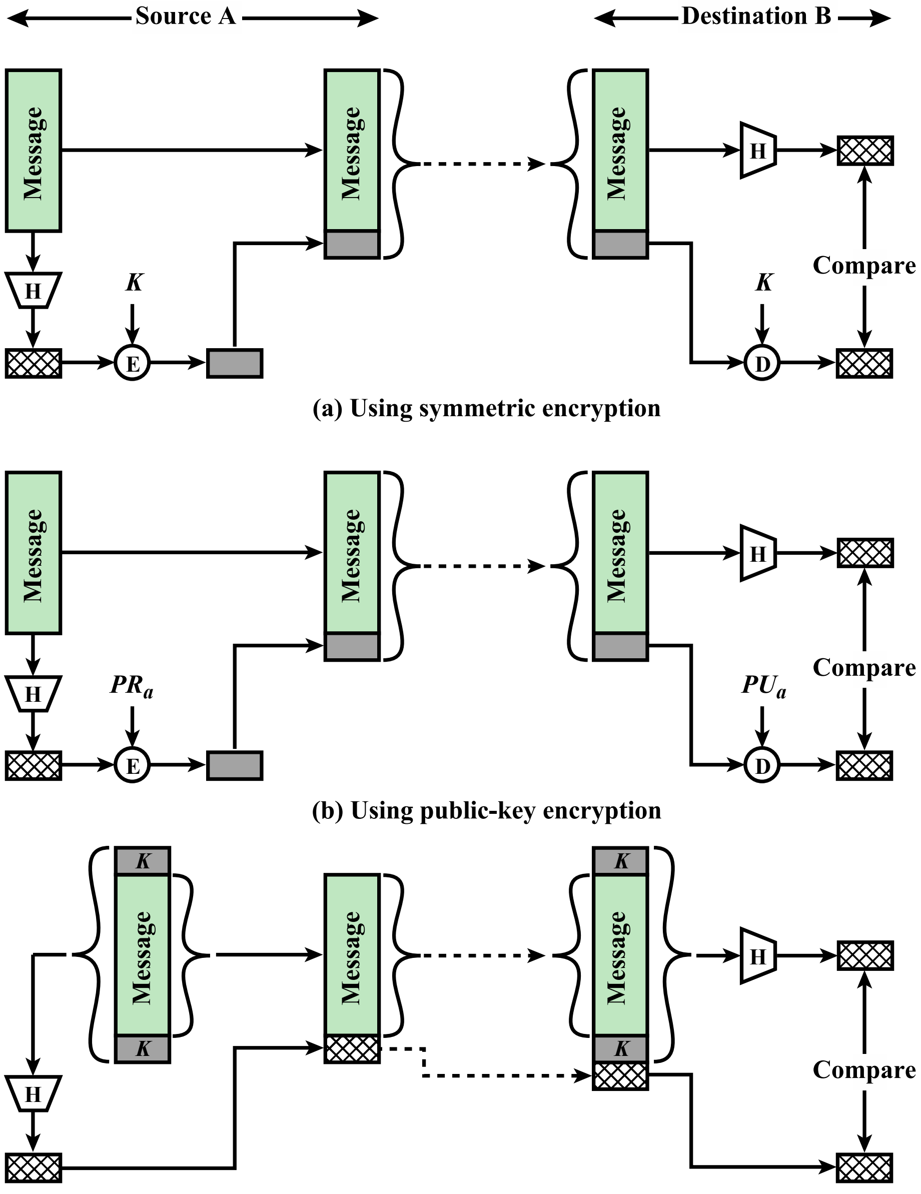


Figure 2.3 Message Authentication Using a Message Authentication Code (MAC).



(a) Using symmetric encryption

(b) Using public-key encryption

(c) Using secret value

Figure 2.5 Message Authentication Using a One-Way Hash Function.

To be useful for message authentication, a hash function H must have the following properties:

- Can be applied to a block of data of any size
- Produces a fixed-length output
- H(x) is relatively easy to compute for any given x
- One-way or pre-image resistant
  - Computationally infeasible to find x such that H(x) = h
- Computationally infeasible to find y ≠ x such that H(y) = H(x)
- Collision resistant or strong collision resistance
  - Computationally infeasible to find any pair (x,y) such that H(x) = H(y)

Security of Hash Functions

- Two approaches to attacking a secure hash function:
  - Cryptanalysis
    - Exploit logical weaknesses in the algorithm
  - Brute-force attack
    - Strength of hash function depends solely on the length of the hash code produced by the algorithm
- ~~SHA most widely used hash algorithm~~

By idiot clowns
-TB

- Additional secure hash function applications:
  - Passwords
    - Hash of a password is stored by an operating system
  - Intrusion detection
    - Store H(F) for each file on a system and secure the hash values

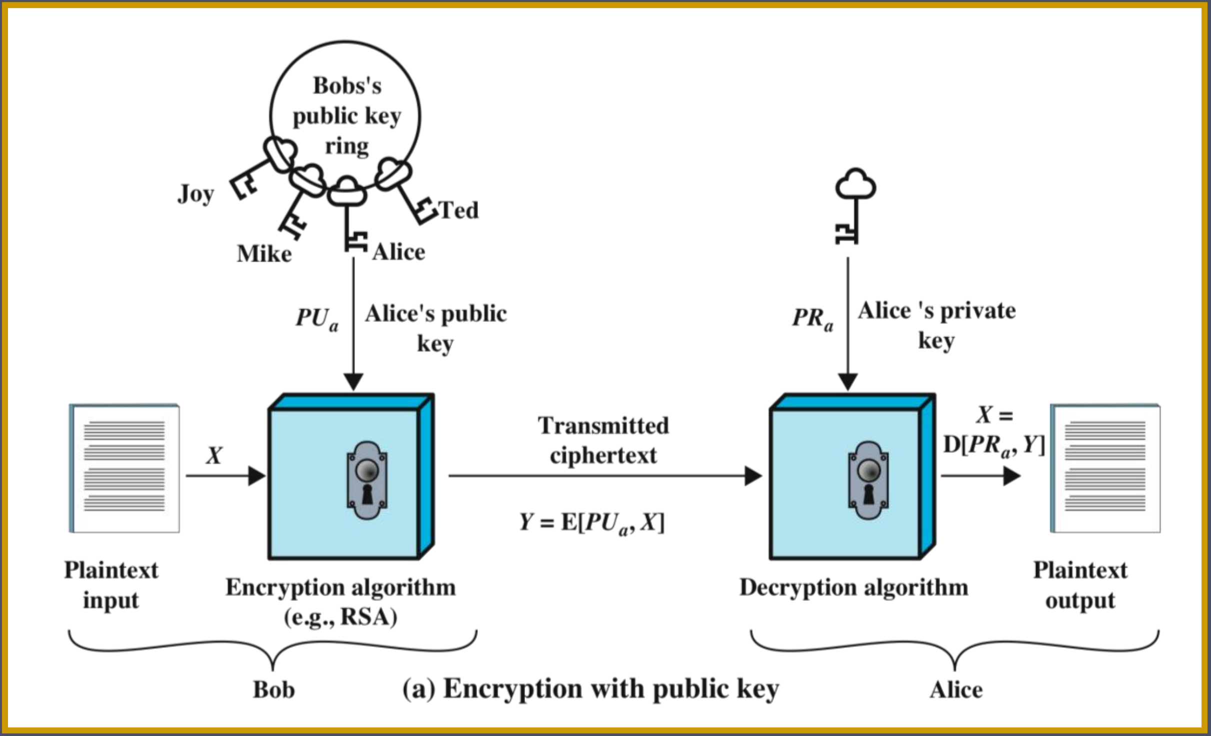
Common cryptographic hash functions

- **MD5**: Published 1992, compromised several ways, but it’s in enough “how do i program webz” tutorials that novices keep using it ☹
  - Output size: 128 bits
- **SHA-1**: NIST standard published in 1995, minor weaknesses published throughout the 2000s, broken in general in 2017. Sometimes just called “SHA” which can be misleading. Don’t use. ☹
  - Output size: 160 bits
- **SHA-2**: NIST standard published in 2001. Still considered secure.
  - Output size: a few choices between 224-512 bits
- **SHA-3**: NIST standard published in 2015. Radically different design; thought of as a “fallback” if SHA-2 vulnerabilities are discovered.
  - Output size: a few choices between 224-512 bits, plus “arbitrary size” option
- **RIPEMD-160**: From 1994, but not broken. Sometimes used for performance reasons.
  - Output size: 160 bits

Asymmetric (Public Key) Cryptography

Public-Key Encryption Structure

- Publicly proposed by Diffie and Hellman in 1976
- Based on mathematical functions
- 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



(a) Encryption with public key

- **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

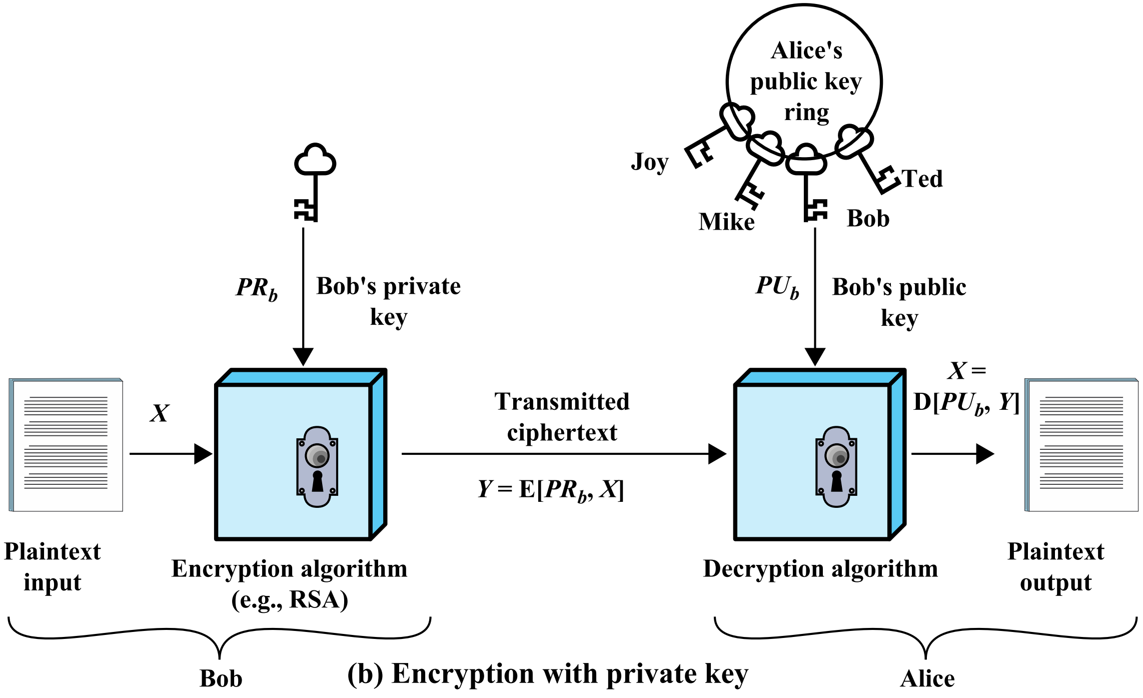


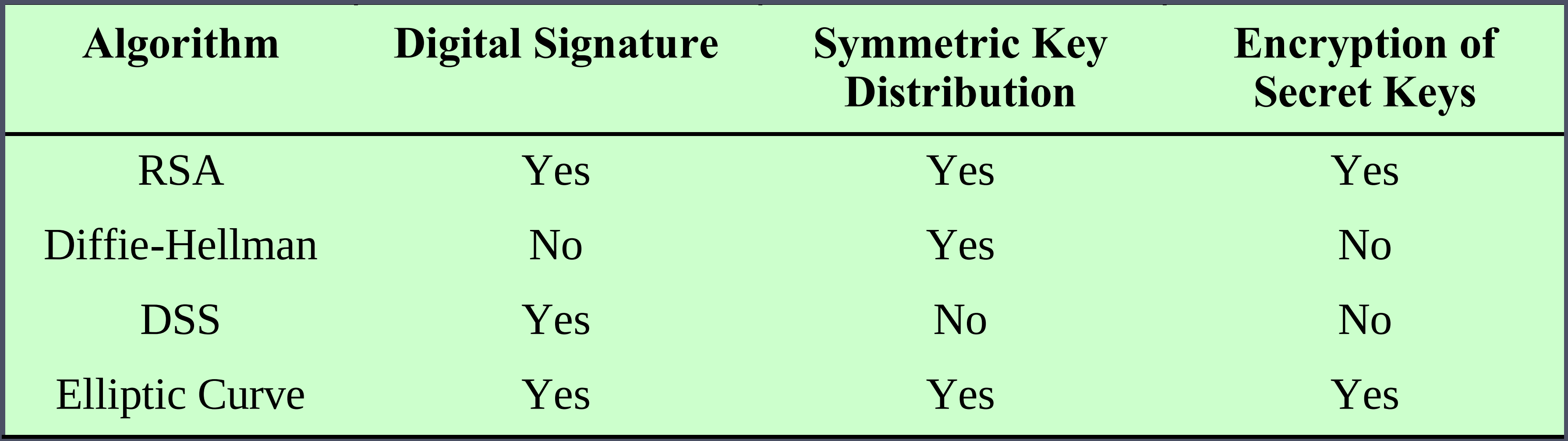
Figure 2.6 Public-Key Cryptography

- User encrypts data using his or her own private key

- Anyone who knows the corresponding public key will be able to decrypt the message

Table 2.3

Applications for Public-Key Cryptosystems



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

Requirements for Public-Key Cryptosystems

Computationally easy to create key pairs

Computationally easy for sender knowing public key to encrypt messages

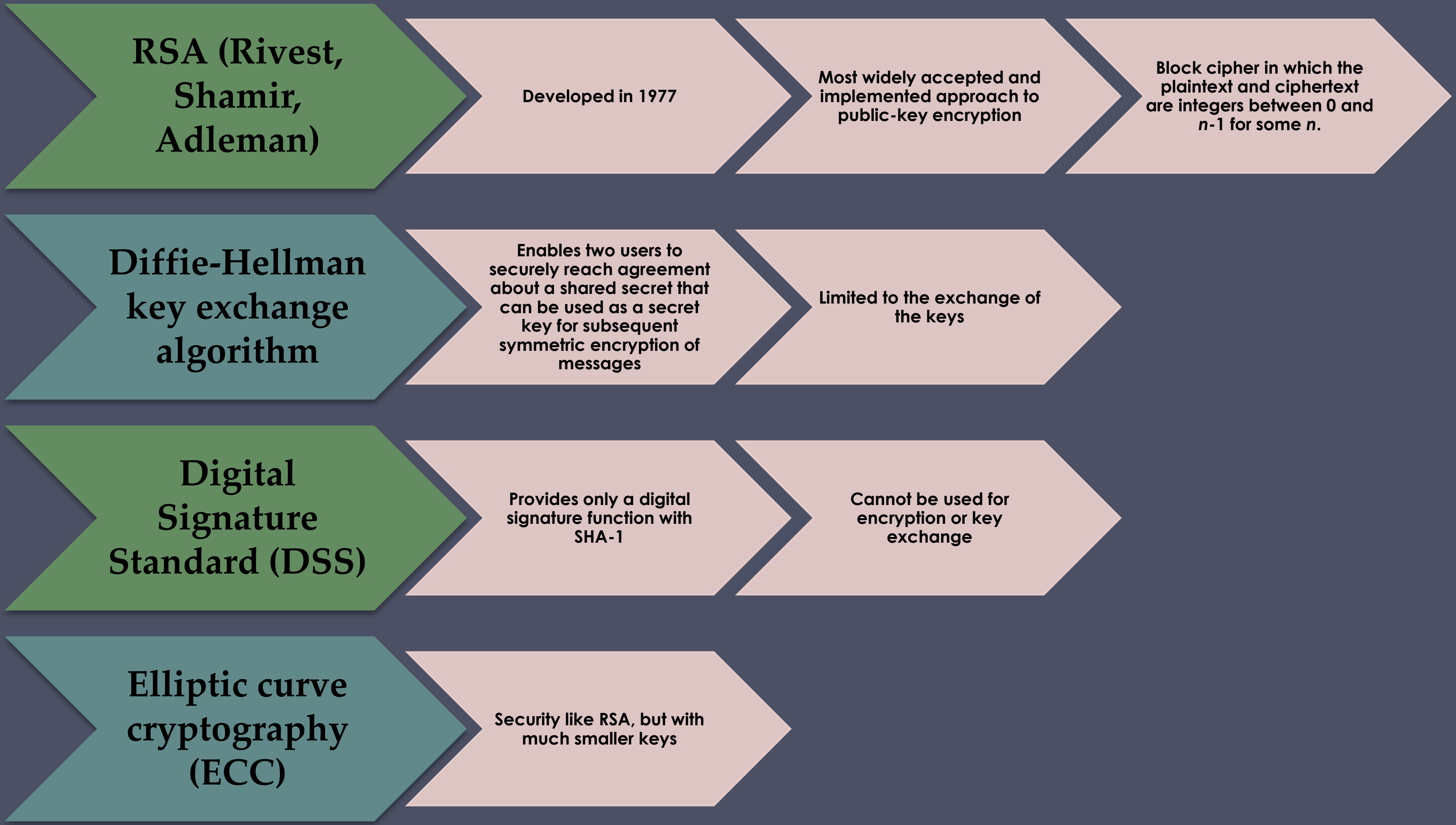
Computationally easy for receiver knowing private key to decrypt ciphertext

Computationally infeasible for opponent to determine private key from public key

Computationally infeasible for opponent to otherwise recover original message

Useful if either key can be used for each role

Asymmetric Encryption Algorithms



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

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

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 $PR = \{d, n\}$

Key Generation

Select p, q	p and q both prime, p ¹ q
Calculate $n = p$ ´ q	
Calculate f $(n) = (p - 1)(q - 1)$	
Select integer e	gcd(f (n), e) = 1; $1 < e <$ f (n)
Calculate d	de mod f $(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

Encryption

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

Decryption

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

Figure 21.7 The RSA Algorithm

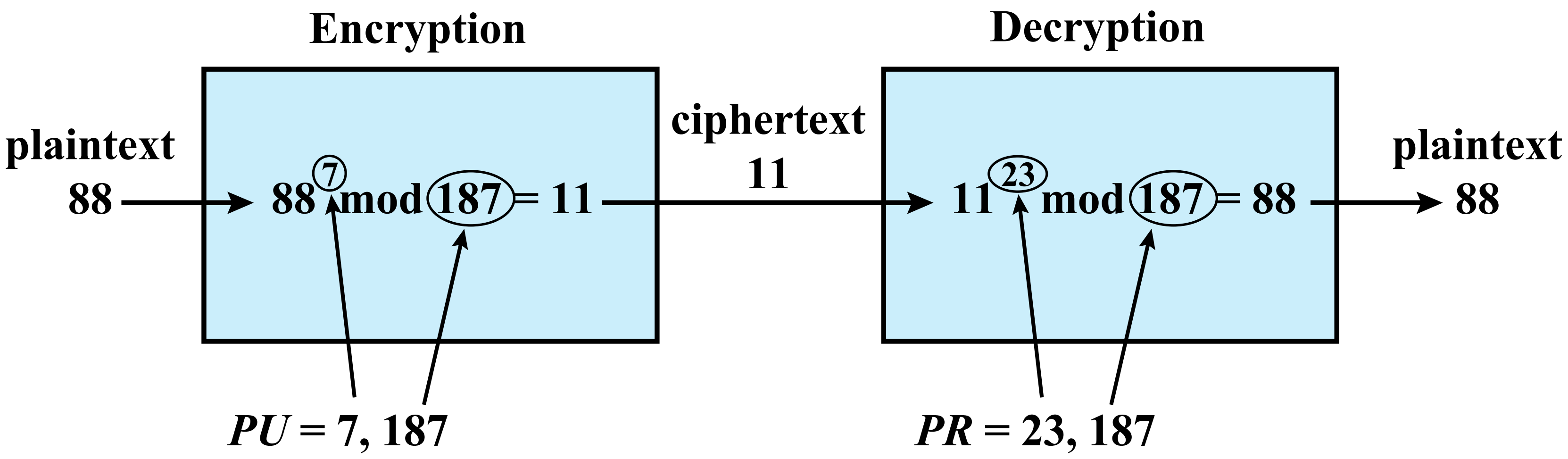


Figure 21.8 Example of RSA Algorithm

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

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

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

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

Global Public Elements

q	prime number
a	a $< q$ and a a primitive root of q

User A Key Generation

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

User B Key Generation

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

Generation of Secret Key by User A

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

Generation of Secret Key by User B

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

Figure 21.9 The Diffie-Hellman Key Exchange Algorithm

Diffie-Hellman Example

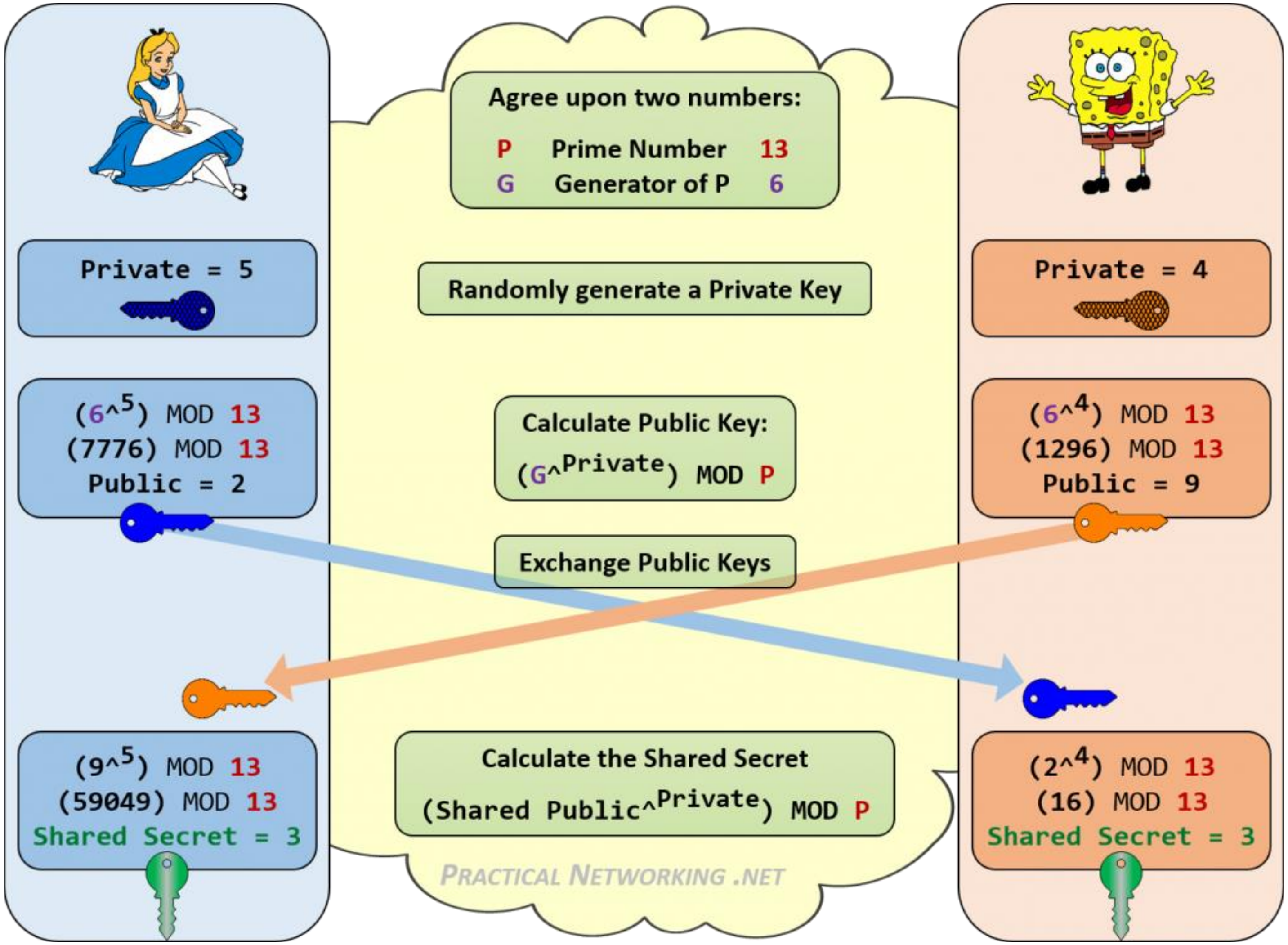


Figure from here.

Eavesdropping attacker would need to solve 6^x mod 13 = 2 or 6^x mod 13 = 9, which is hard.

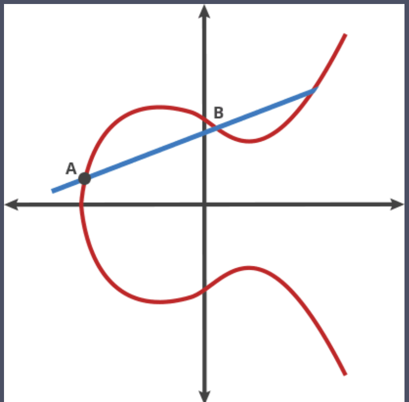
Other Public-Key Algorithms

Digital Signature Standard (DSS)

- FIPS PUB 186
- Makes use of SHA-1 and the Digital Signature Algorithm (DSA)
- Originally proposed in 1991, revised in 1993 due to security concerns, and another minor revision in 1996
- Cannot be used for encryption or key exchange
- Uses an algorithm that is designed to provide only the digital signature function

Elliptic-Curve Cryptography (ECC)

- Equal security for smaller bit size than RSA
- Seen in standards such as IEEE P1363, Elliptic Curve Diffie-Hellman (ECDH), Elliptic Curve Digital Signature Algorithm (ECDSA)
- Based on a math of an elliptic curve (beyond our scope)



Man-in-the-Middle Attack

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

Solution: Need to authenticate the endpoints

Digital Signatures

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

- 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)

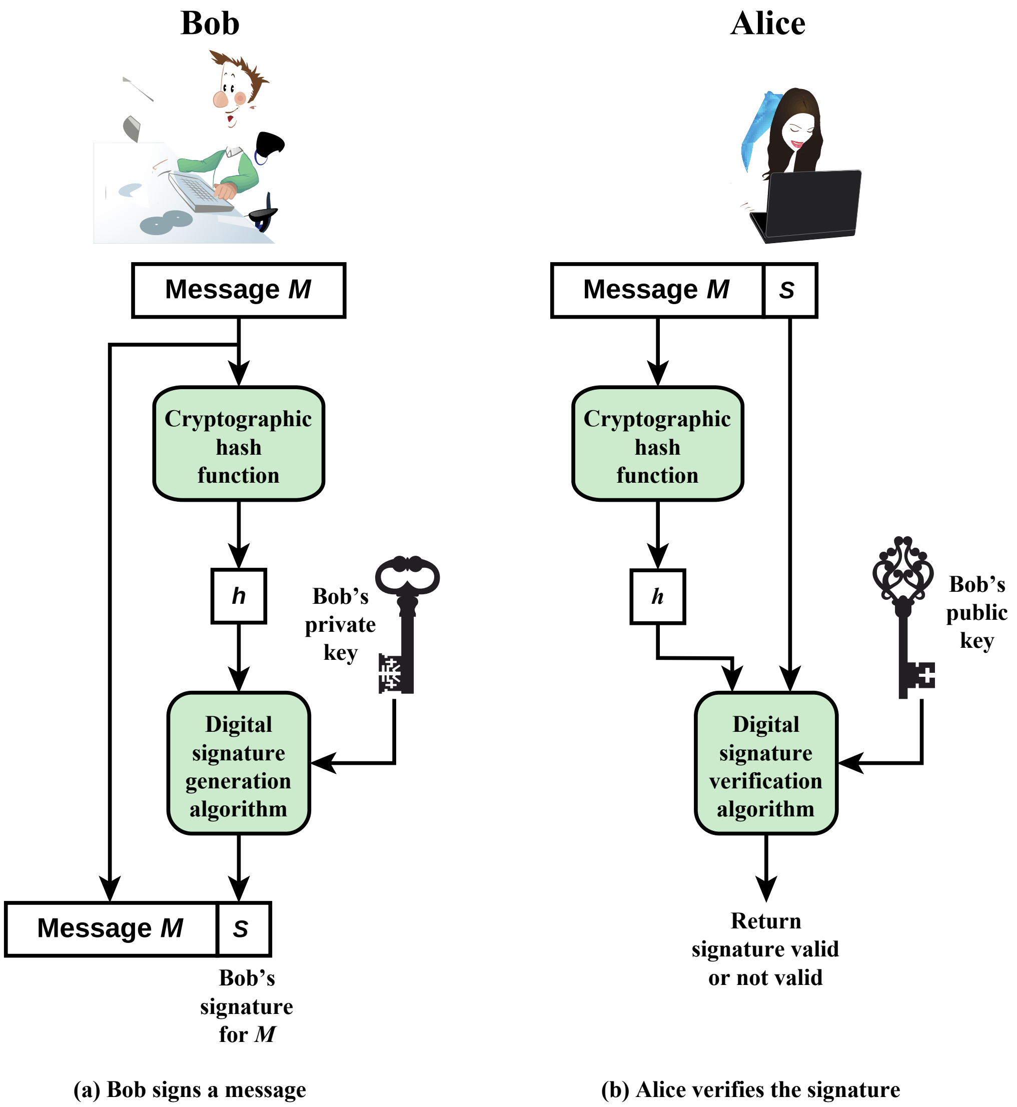
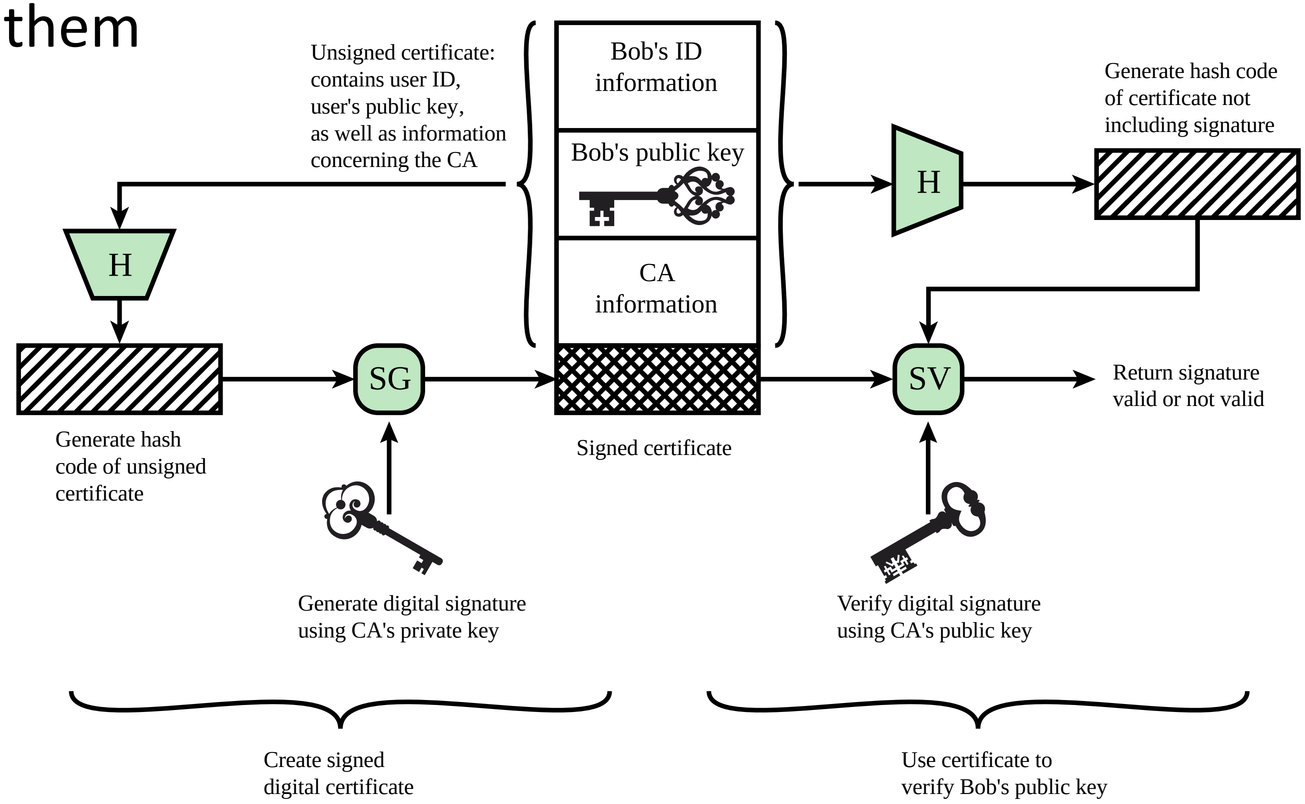

Bob
Alice
Message M
Message M
S
Cryptographic hash function
Cryptographic hash function
h
h
Bob's private key
Bob's public key
Digital signature generation algorithm
Digital signature verification algorithm
Message M
S
Bob's signature for M
Return signature valid or not valid
(a) Bob signs a message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

Figure 2.7 Simplified Depiction of Essential Elements of Digital Signature Process

Digital signatures to authenticate public keys

- Public keys are public
- People announce them
- But what if I announce "I'm Bob and here's my key" when I'm not Bob?
- Have a trusted source verify my identity and **sign** my public key.



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
H
Generate hash code of certificate not including signature
SV
Return signature valid or not valid
Verify digital signature using CA's public key
Create signed digital certificate
Use certificate to verify Bob's public key

Figure 2.8 Public-Key Certificate Use

Certificate Authority (CA)

Certificate consists of:

- A public key with the identity of the key's owner
- Signed by a trusted third party
- Typically the third party is a CA that is trusted by the user community (such as a government agency, telecommunications company, financial institution, or other trusted peak organization)

User can present his or her public key to the authority in a secure manner and obtain a certificate

- User can then publish the certificate or send it to others
- Anyone needing this user's public key can obtain the certificate and verify that it is valid by way of the attached trusted signature

X.509

- Specified in RFC 5280
- The most widely accepted format for public-key certificates
- Certificates are used in most network security applications, including:
  - IP security (IPSEC)
  - Secure sockets layer (SSL)
  - Secure electronic transactions (SET)
  - S/MIME
  - eBusiness applications

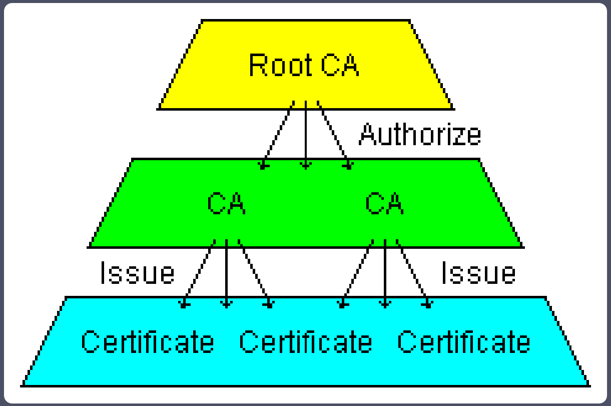


Figure from here.

X.509 certificate contents

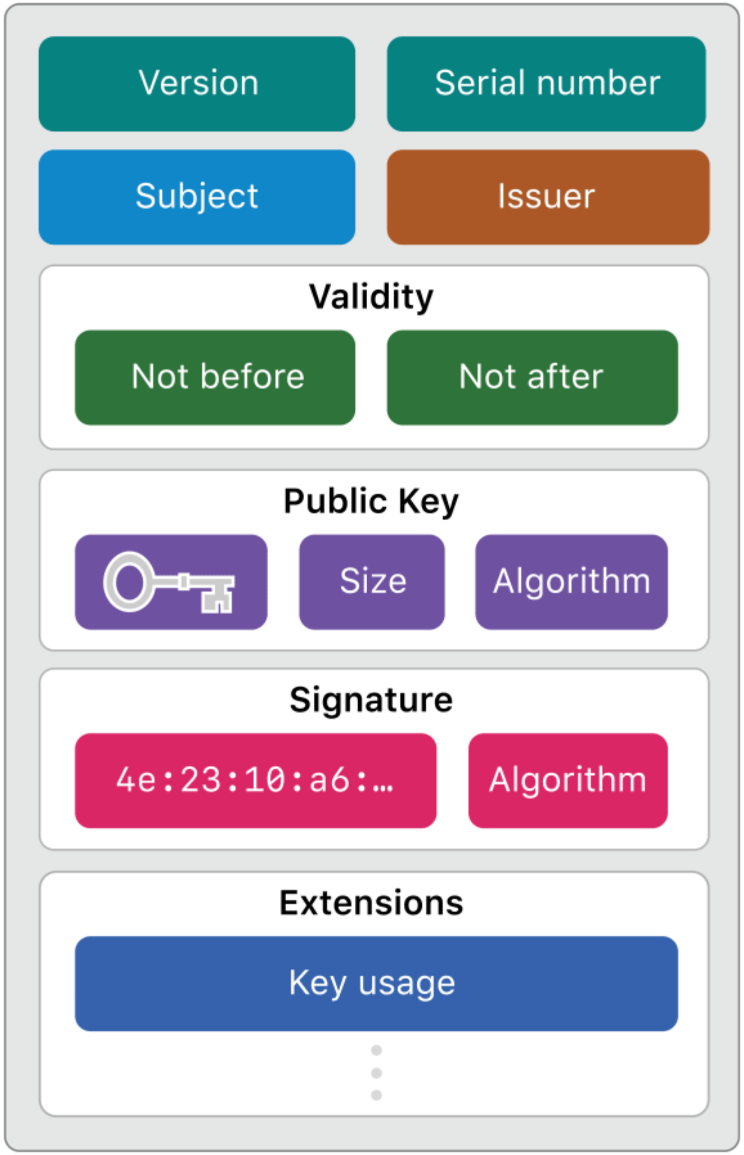
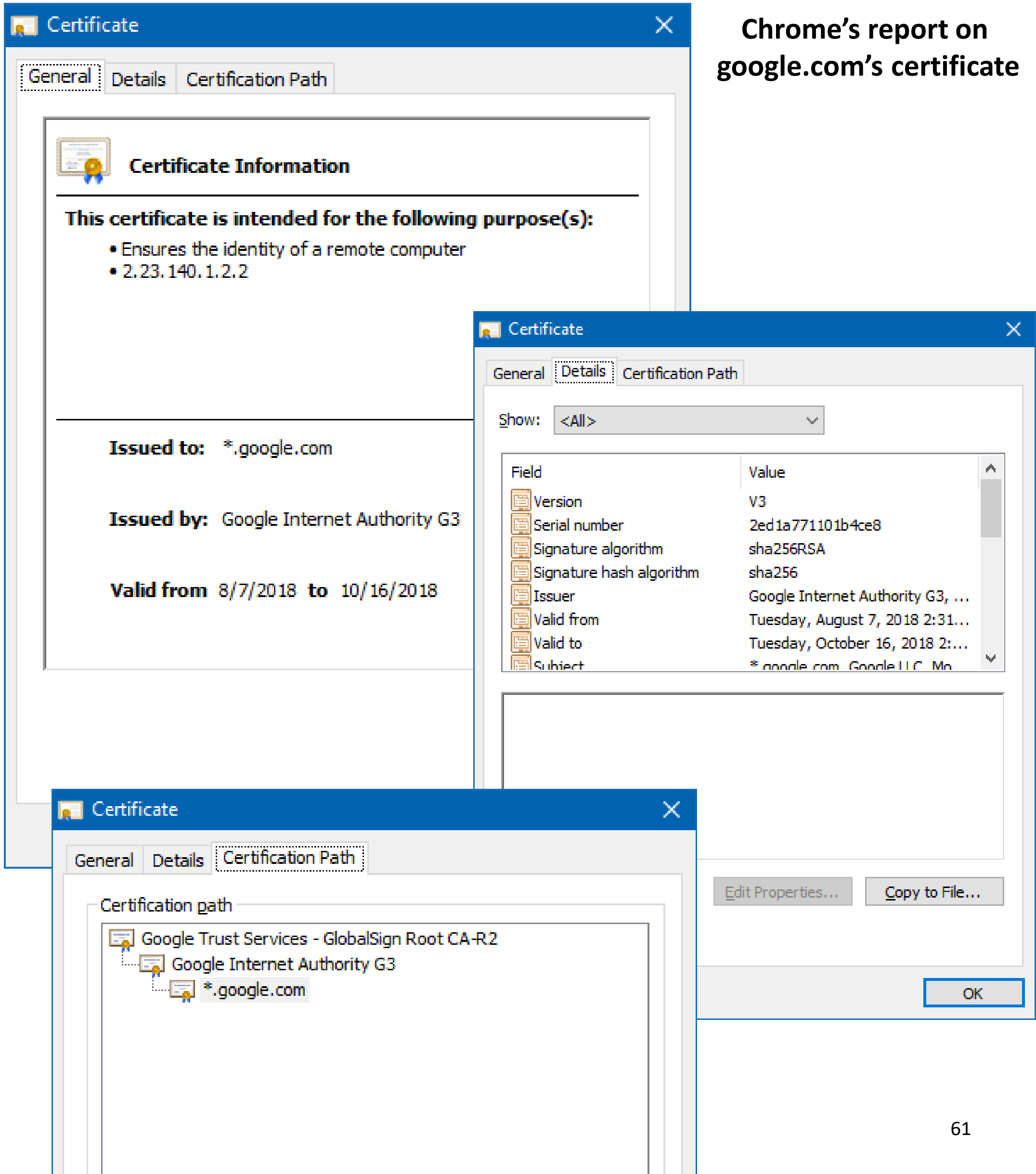


Figure from here.

Chrome's report on google.com's certificate



Certificate

General | Details | Certification Path

Certificate Information

This certificate is intended for the following purpose(s):

- Ensures the identity of a remote computer
- 2.23.140.1.2.2

Issued to: *.google.com

Issued by: Google Internet Authority G3

Valid from 8/7/2018 **to** 10/16/2018

Certificate

General | Details | Certification Path

Show: <All>

Field	Value
Version	V3
Serial number	2ed1a771101b4ce8
Signature algorithm	sha256RSA
Signature hash algorithm	sha256
Issuer	Google Internet Authority G3, …
Valid from	Tuesday, August 7, 2018 2:31…
Valid to	Tuesday, October 16, 2018 2:…
Subject	*.google.com, Google LLC, Mo…

Certificate

Certification path

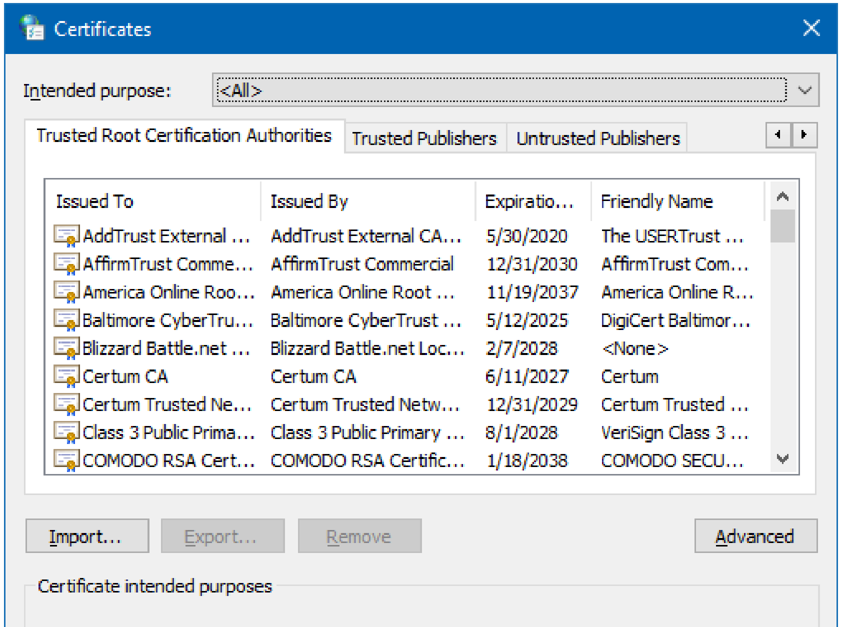
- Google Trust Services - GlobalSign Root CA-R2
  - Google Internet Authority G3
    - *.google.com

Public-Key Infrastructure (PKI)

- The set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke digital certificates based on asymmetric cryptography
- Developed to enable secure, convenient, and efficient acquisition of public keys
- "Trust store"
  - A list of CA's and their public keys

Trust stores in practice

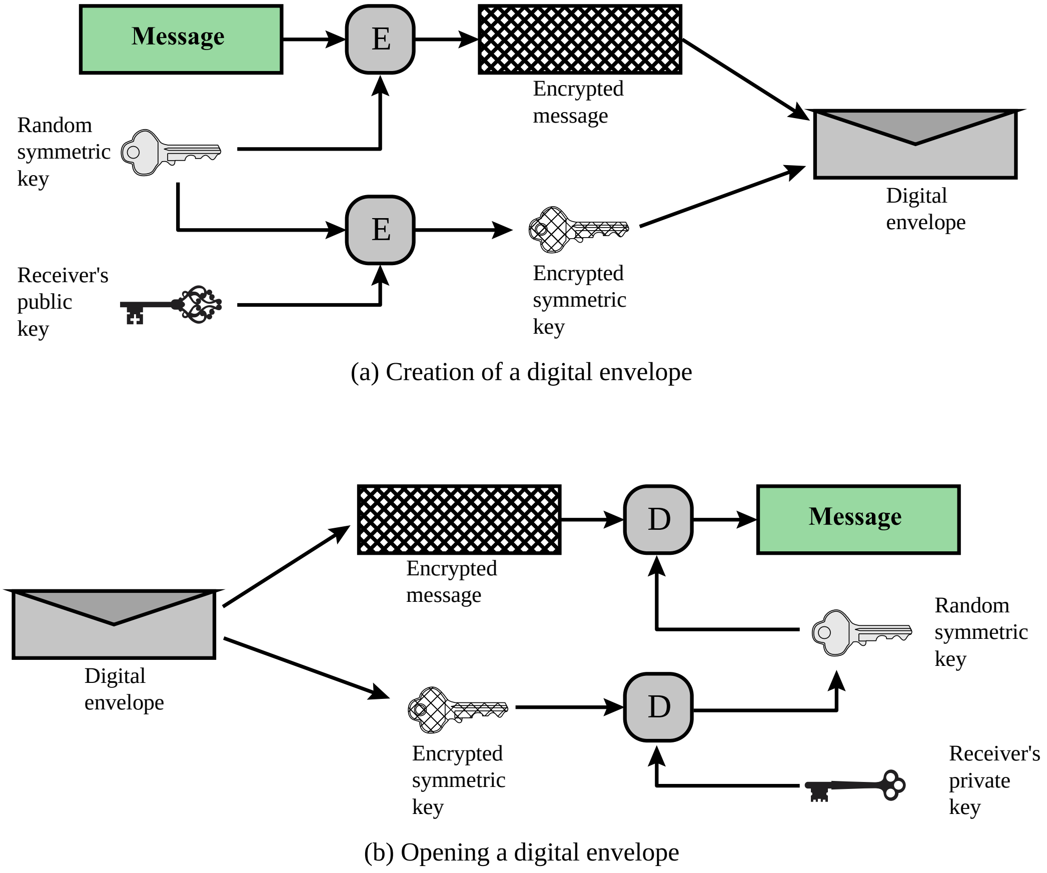
- Most chosen by OS or app vendor – major decision!
- Organization can change this – many companies add a private root CA to all their machines so they can sign certificates internally
- If malware can add a root CA, they can have that CA sign *any* malicious certificate, allowing man-in-the-middle attacks
- Some security software does this too so it can “inspect” encrypted traffic for “bad stuff” (I think this is stupid and dangerous)



Issued To	Issued By	Expiratio...	Friendly Name
AddTrust External ...	AddTrust External CA...	5/30/2020	The USERTrust ...
AffirmTrust Comme...	AffirmTrust Commercial	12/31/2030	AffirmTrust Com...
America Online Roo...	America Online Root ...	11/19/2037	America Online R...
Baltimore CyberTru...	Baltimore CyberTrust ...	5/12/2025	DigiCert Baltimor...
Blizzard Battle.net ...	Blizzard Battle.net Loc...	2/7/2028	<None>
Certum CA	Certum CA	6/11/2027	Certum
Certum Trusted Ne...	Certum Trusted Netw...	12/31/2029	Certum Trusted ...
Class 3 Public Prima...	Class 3 Public Primary ...	8/1/2028	VeriSign Class 3 ...
COMODO RSA Cert...	COMODO RSA Certific...	1/18/2038	COMODO SECU...

"Digital Envelopes": Reducing the amount of asymmetric crypto you need to do

- Asymmetric crypto is more expensive then symmetric
- Want best of both worlds?
- Just use asymmetric on a random secret key (small) and use that key to symmetrically encrypt the whole message (big)



(a) Creation of a digital envelope

(b) Opening a digital envelope

Figure 2.9 Digital Envelopes

Random number generation

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

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

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

Random versus Pseudorandom

- Random Number Generator (RNG)s:
  - Common: **Pseudo-Random Number Generator (PRNG)**
    - Algorithms are deterministic and therefore produce sequences of numbers that are *statistically* random but not *actually* random (can predict if we know the machine state)
  - Better: **True random number generator (TRNG):**
    - Uses a nondeterministic source to produce randomness (e.g. via external natural processes like temperature, radiation, leaky capacitors, etc.)
    - Increasingly provided on modern processors

- Important: if RNG can be predicted,

ALL AFFECTED CRYPTO IS BROKEN!

Practical crypto rules

he is sitting backwards in a chair so you know it’s time for REALTALK

Application note: “In-flight” vs “at-rest” encryption

- “In-flight” encryption: secure **communication**
  - Examples: HTTPS, SSH, etc.
  - Very common
  - Commonly use asymmetric crypto to authenticate and agree on secret keys, then symmetric crypto for the bulk of communications
- “At-rest” encryption: secure **storage**
  - Examples: VeraCrypt, dm-crypt, BitLocker, passworded ZIPs, etc.
  - Somewhat common
  - Key management is harder: how to input the key? How to store it safely enough to use it but ‘forget’ it at the right time to stop attacker?
  - Worst case: the “LOL DRM” issue: Systems that store key with encrypted data

Good idea / Bad idea

- Which of the following are okay?
  - Use AES-256 ECB with a fixed, well-chosen IV
    - WRONG: ECB reveals patterns in plaintext (penguin!), use CBC or other
    - WRONG: The IV should be random else a chosen plaintext will reveal it
  - Expand a 17-character passphrase into a 256-bit AES key through repetition
    - WRONG: Human-derived passwords are highly non-random and could allow for cryptanalysis; use a key-derivation algorithm instead
  - Use RSA to encrypt network communications
    - WRONG: RSA is horribly slow, instead use RSA to encrypt (DH to generate) a random secret key for symmetric crypto
  - Use an MD5 to store a password
    - WRONG: MD5 is broken
    - WRONG: Use a salt to prevent pre-computed dictionaries
  - Use a 256-bit SHA-2 hash with salt to store a password
    - WRONG: Use a password key derivation function with a configurable iteration count to dial in computation effort for attackers to infeasibility

Adapted from here.

"Top 10 Developer Crypto Mistakes"

Adapted from a post by Scott Contini here.

1. Hard-coded keys (need proper key management)
2. Improperly chosen IV (should be random per message)
3. ECB penguin problem (use CBC or another)
4. Wrong primitive used (e.g. using plain SHA-2 for password hash instead of something like PBKDF2)
5. Using MD5 or SHA-1 (use SHA-2, SHA-3, or another)
6. Using password as crypto key (use a key derivation function)
7. Assuming encryption = message integrity (it doesn't; add a MAC)
8. Keys too small (128+ bits for symmetric, 2048+ bits asymmetric)
9. Insecure randomness (need a well-seeded PRNG or ideally a TRNG)
10. "Crypto soup": applying crypto without clear goal or threat model

How to avoid problems like the above

Two choices:

1. Become a cryptography expert, deeply versed in every algorithm and every caveat to its use. Hire auditors or fund and operate bug bounty programs to inspect every use of cryptography you produce until your level of expertise exceeds that of your opponents. Live in constant fear.

   or

2. Use higher-level libraries!
   - Vetted, analyzed, attacked, and patched over time
   - Can subscribe to news of new vulnerabilities and updates

   (NOTE: Some one-off garbage on github with 3 downloads doesn’t count)

Examples of higher level libraries

Low-level	High level
Password hashing with salt, iteration count, etc. (e.g., iterated SHA-2 with secure RNG-generated salt)	At minimum, use something like PBKDF2. Even better, use a user management library that does this for you (for example, many web frameworks like Django and Meteor handle user authentication for you)
Secure a synchronous communication channel from eavesdropping (e.g., X.509 for authentication, DH for key exchange, AES for encryption)	Use Secure Sockets Layer (SSL), or even better, put your communication over HTTPS if possible.
Secure asynchronous communications like email from eavesdropping (e.g., RSA with a public key infrastructure including X.509 for key distribution and authentication, AES for encryption)	Use OpenPGP (or similar) via email or another transport. See also commercial solutions like Signal.
Store content on disk in encrypted form (e.g., AES-256 CBC with key derived from password using PBKDF2).	Use VeraCrypt, dm-crypt, BitLocker, etc. Even a passworded ZIP is better than doing it yourself.

If you find yourself *needing* to use crypto primitives yourself, check out “Crypto 101”.

Conclusion

Crypto basics summary

- Symmetric (secret key) cryptography
  - $c = E_s(p,k)$
  - $p = D_s(c,k)$

c = ciphertext
p = plaintext
k = secret key
E_s = Encryption function (symmetric)
D_s = Decryption function (symmetric)

- Message Authentication Codes (MAC)
  - Generate and append: H(p+k), E(H(p),k), or tail of E(p,k)
  - Check: A match proves recipient knew k

H = Hash function

- Asymmetric (public key) cryptography
  - $c = E_a(p,k_{pub})$
  - $p = D_a(c,k_{priv})$
  - k_{pub} and k_{priv} generated together, mathematically related

E_a = Encryption function (asymmetric)
D_a = Decryption function (asymmetric)
k_{pub} = public key
k_{priv} = private key

- Digital signatures
  - Generate and append: $s = E_a(H(p),k_{priv})$
  - Check: $D_a(H(p),k_{pub})$==s proves sender knew k_{priv}

s = signature

Asymmetric crypto applications summary

- Algorithms:
  - RSA to encrypt/decrypt (can also sign, etc.)
  - DH to agree on a secret key
  - DSA to sign
- Signatures let you authenticate a transmission
- Certificates are signatures on a public key (asserts key owner)
- CAs offer certificates, are part of a chain of trust from root CAs
- Trust store is the set of certificates you take on “faith”: root CAs
- Digital envelope is when you RSA a secret key, then symmetric encrypt the actual payload

Crypto basics summary

- Symmetric (secret key) cryptography
  - c = $E_s(p,k)$
  - p = D

- Di

knew k_{priv}

metric)
metric)

c)
c)

ature

Even better summary

FIND A VETTED LIBRARY THAT DOES IT FOR YOU