ECE560
Computer and Information Security

Fall 2023

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 **<u>should not reach for these functions in most real-world programming scenarios!!</u>**
- **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...

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)

- 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

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

H = Hash function

- 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}

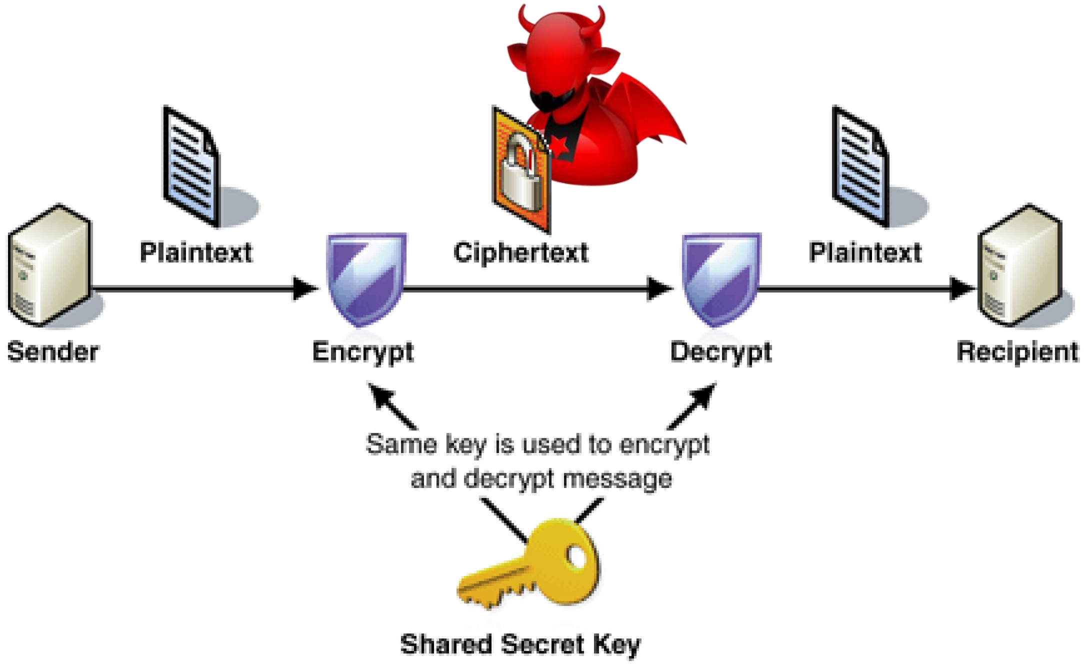
s = signature

Symmetric (Secret Key) Encryption

Symmetric cryptography

- The primary method for providing **confidentiality** of data *transmitted* ("in-flight") or *stored* ("at-rest encryption")

- Uses one key for both encryption and decryption.
  - Sender/receiver must already have copies of this key.



Given:

Plaintext **p** (arbitrary size)
Secret key **k** (fixed size)
Encryption function **E**
Decryption function **D**

Can produce ciphertext **c**:

c = E(p,k)

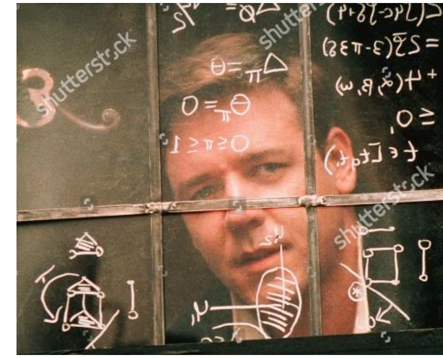
Can recover plaintext:

p = D(c,k)

How to attack cryptography

- **<u>Cryptanalysis</u>** – *apply cleverness*
  - Exploit weaknesses in algorithm or manner of its use
  - May leverage existing plaintext, ciphertext, or pairs of each
  - KEY ISSUE: Even if algorithm is "perfect" (unprovable), you might *use* the algorithm incorrectly.

    (This is why you don't roll your own crypto)



- **<u>Brute-force attack</u>** – *apply money*
  - Try all possible keys, stop when decrypted result seems readable
    - Need to try half of all keys on average
  - Can be done in parallel (i.e., using compute cluster)
  - Always possible, but if the number of possible keys is large enough, *cost to crack > value of info obtained*
    - This is called being **computationally secure**

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					

Types of cryptanalysis attacks

- Given the encryption algorithm and ciphertext under attack, attacks we can do:

Type of attack	Things known to cryptanalyst
Ciphertext only	(just the ciphertext under attack)
Known plaintext	• One or more plaintext-ciphertext pairs using same key
Chosen plaintext	• Plaintext chosen by attacker + ciphertext encrypted with same key
Chosen ciphertext	• Ciphertext chosen by attacker + “plaintext” decrypted with same key
Chosen text	• Plaintext chosen by attacker + ciphertext encrypted with same key • Ciphertext chosen by attacker + “plaintext” decrypted with same key

Attacking XOR (1)

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

Attacking XOR (2)

- 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**

Symmetric ciphers in common use

Cipher	Key size	Block size	Year introduced
DES	56	64	1975
3DES	112/168	64	1995
Twofish	128/192/256	128	1998
Serpent	128/192/256	128	1998
AES	128/192/256	128	1998

(1975) Now ATMs can exist thanks to this Data Encryption Standard (DES)!

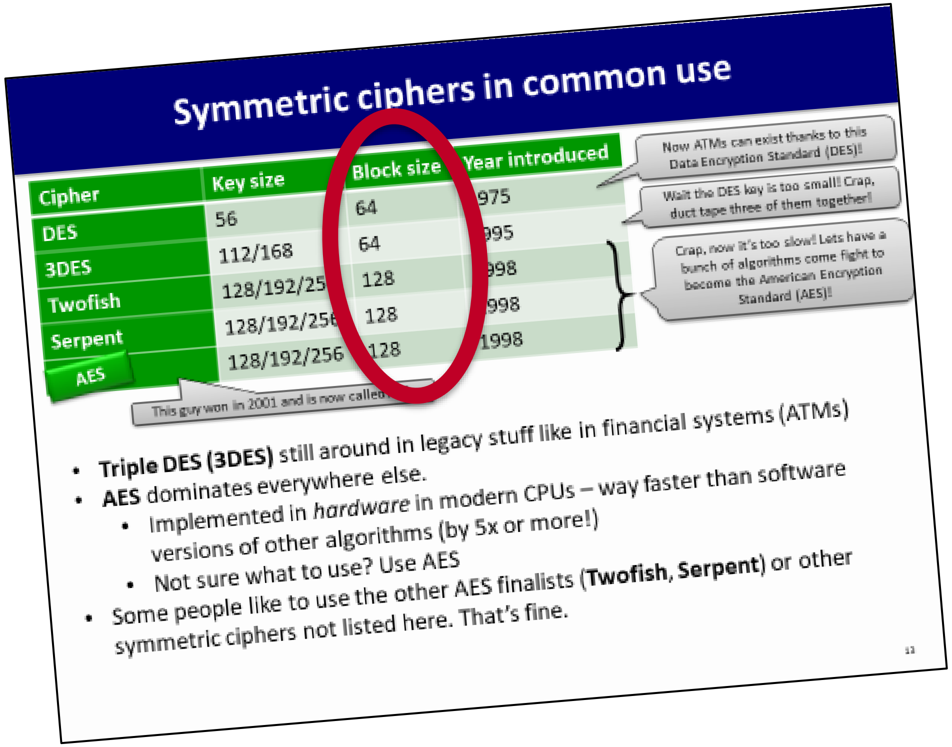
(1995) Ahh, the DES key is too small! It can be brute forced really fast! Hurry, duct tape three of them together!

(1998) Crap, now it's too slow! Lets have a bunch of algorithms come fight to become the American Encryption Standard (AES)!

(2001) This guy won and is now called AES.

- **Triple DES (3DES)** still around in legacy stuff like in financial systems (ATMs)
- **AES** dominates everywhere else.
  - Implemented in *hardware* in modern CPUs – way faster than software versions of other algorithms (by 5x or more!)
  - Not sure what to use? Use AES
- Some people like to use the other AES finalists (**Twofish**, **Serpent**) or other symmetric ciphers not listed here. That's fine.

Okay, but what about that "block size" thing?



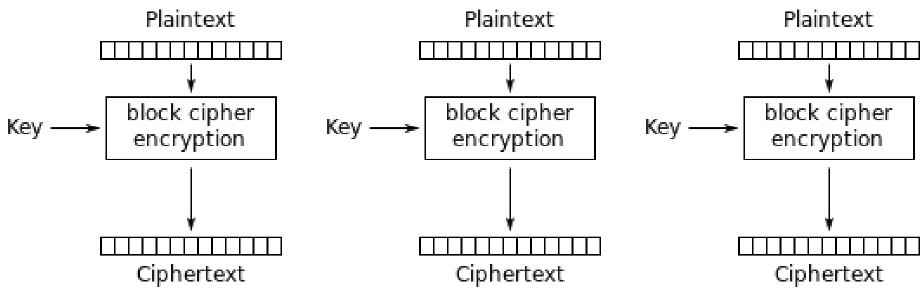
- These are **block ciphers** – they just encrypt a block of bits.
- How do you apply the cipher to data that's bigger than just one block?
- Answer: **modes of operation**

- Simplest mode of operation: **Electronic Code Book (ECB)**
  - Each block of plaintext is encrypted with the same key

Problem?

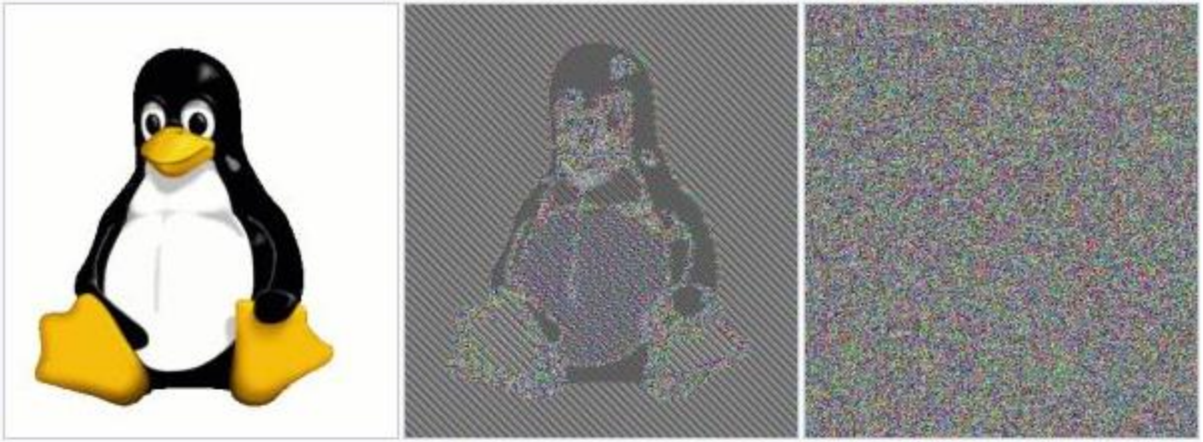
Demonstrating the danger of ECB

- 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"

Solution to the "ECB problem"

- Develop more sophisticated **modes of operation** for use with our block cipher.
  - We'll see several of these – there's tradeoffs to different techniques

- *Some* will convert our block cipher to a **stream cipher**
  - Stream cipher: A cipher where the plaintext is XOR'd with a pseudorandom bit stream derived from the key

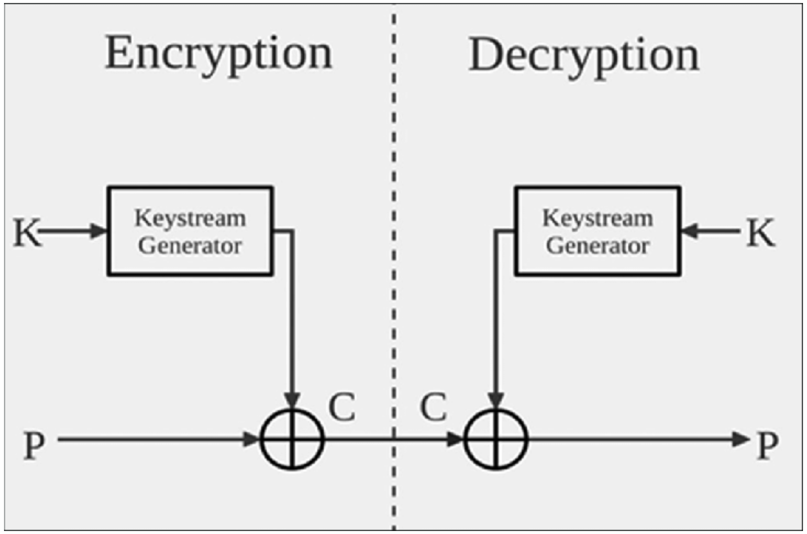


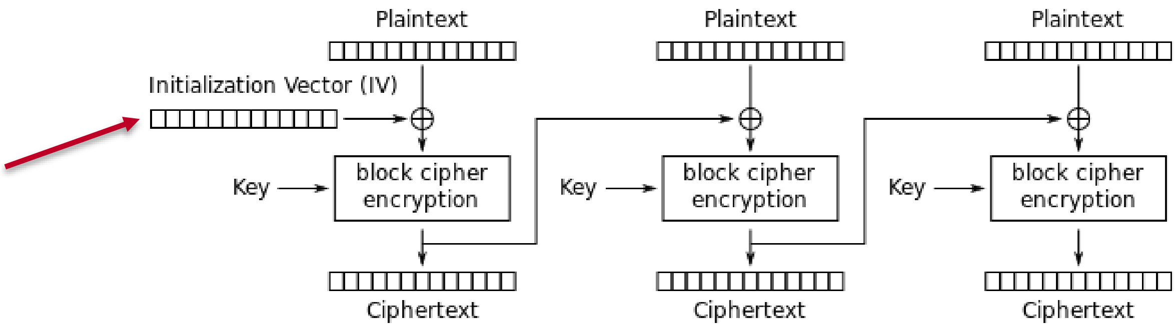
Figure from https://www.researchgate.net/figure/Stream-cipher-diagram_fig2_318517979

Modes of operation: CBC

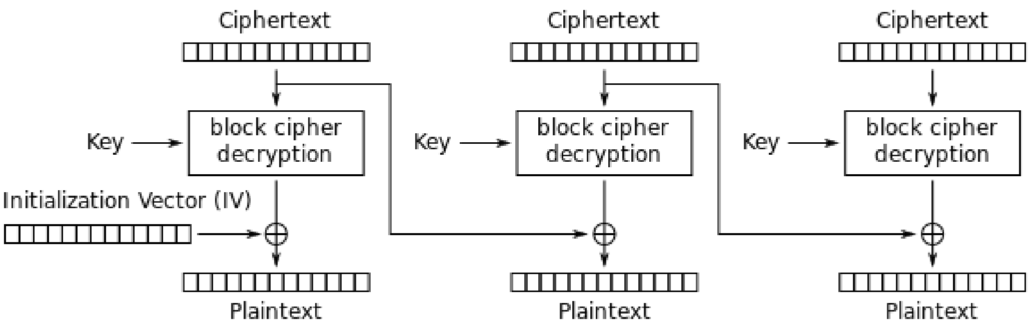
- **Cipher Block Chaining (CBC):**
  - Each block of plaintext is XOR'd with previous block ciphertext
  - Prevents patterns from being visible even in regular data

Initialization Vector (IV)

- Random
- Not secret (transmitted in clear)
- Prevents cryptanalysis, e.g. by preventing a past-seen plaintext from producing the same ciphertext in the first block



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Encryption parallelizable?	No
Decryption parallelizable?	Yes
Random read access?	Yes

Figures from Wikipedia "Block cipher mode of operation"

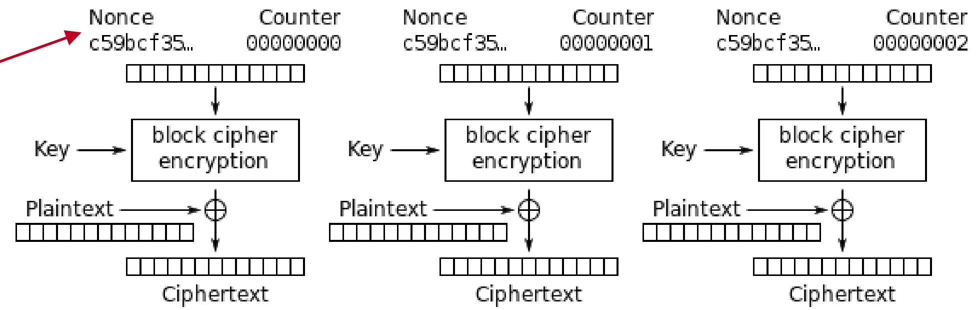
More about the Initialization Vector

- 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.

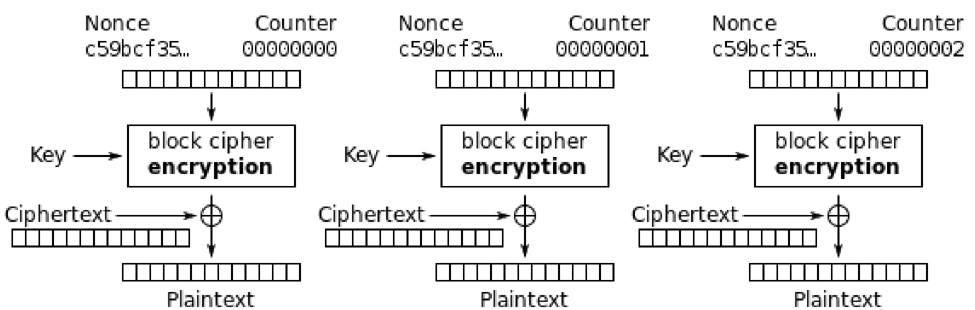
Modes of operation: CTR

- **Counter (CTR):**
  - Encrypt an incrementing list of integers to make a **keystream**: turns a block cipher into a stream cipher!
  - Allows full parallelization and random access

Note: A nonce is basically like an initialization vector.



Counter (CTR) mode encryption



Counter (CTR) mode decryption

Encryption parallelizable?	Yes
Decryption parallelizable?	Yes
Random read access?	Yes

Figures from Wikipedia "Block cipher mode of operation"

Further modes of operation

- **Cipher feedback (CFB)**: Chained encryption similar to CBC, but just used to produce a keystream – works as a stream cipher

- **Output feedback (OFB)**: Very similar to CBC. Neat property: Bitflips in ciphertext become bitflips in plaintext, so error correcting codes work transparently.

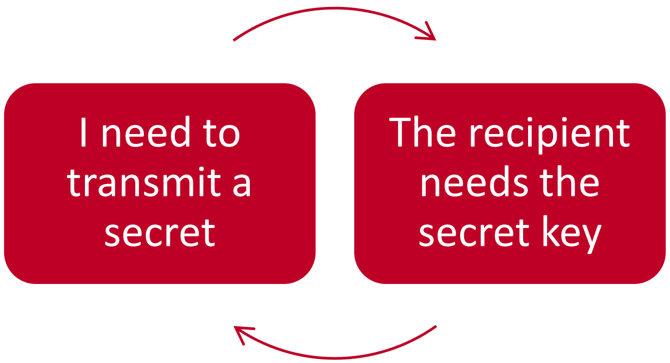
- **Galois/counter mode (GCM)**: Applies lessons from finite field theory (out of scope for us). This mode performs **authentication** as well as providing confidentiality. *Very common on the modern web.*

- Many more! See “Block Cipher Modes of Operation” on Wikipedia.

Asymmetric (Public Key) Cryptography

The problem

- Problem with symmetric crypto:

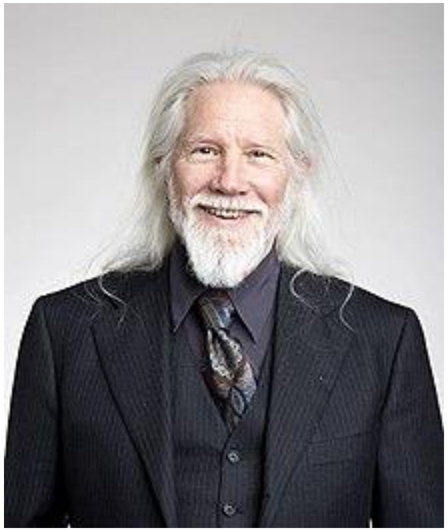


- Want to be able to send a message without having pre-shared a key

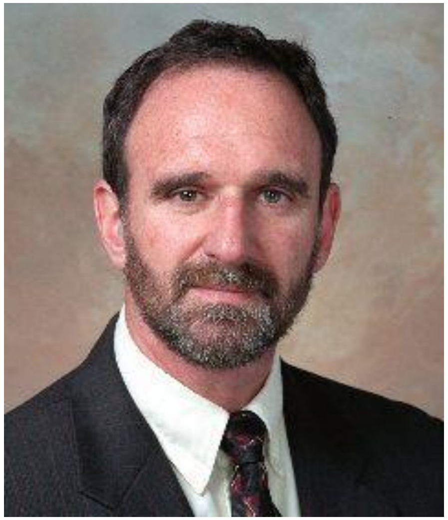
- Solution:
  What if the key to *decrypt* was different than the one to *encrypt*?

Asymmetric (Public Key) Cryptography

- Proposed by Diffie and Hellman in 1976
- Based on math; asymmetric:
  - Uses *two separate keys*
  - **Public key** and **private key**
  - Public key is made public for others to use, private key must be kept secret
- I just need to distribute my public key to anyone who wants to send me stuff.
  - And prove that it's *my* key (covered later)



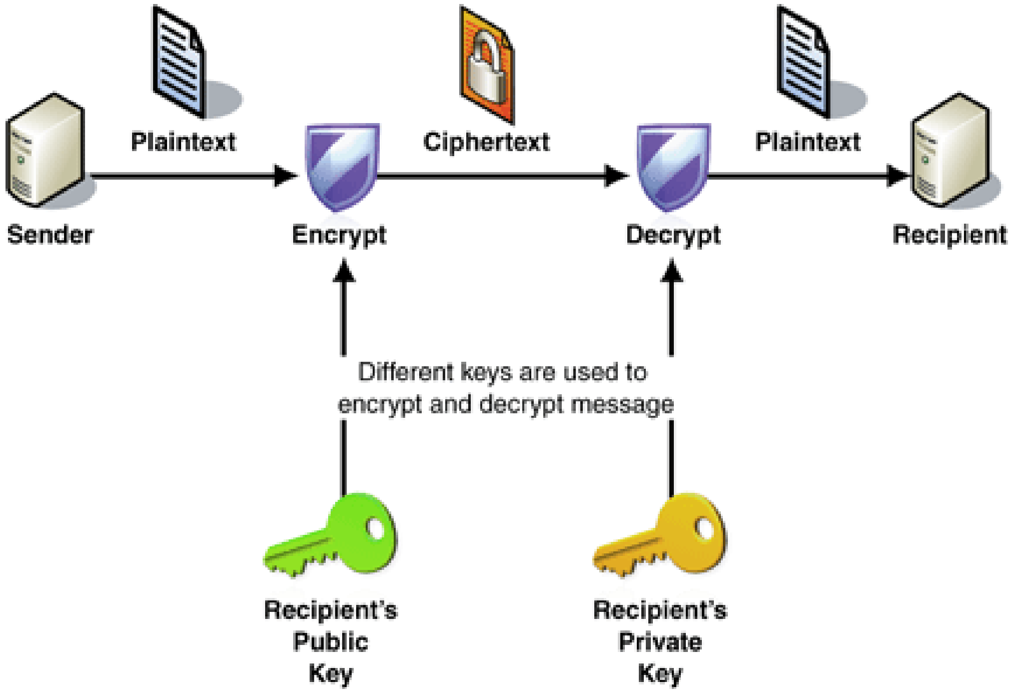
Whitfield Diffie



Martin Hellman

Asymmetric cryptography

- Public and private keys mathematically related, but one cannot be determined from the other
- Far slower than symmetric encryption
  (but there's tricks to get around that – covered later)



Sender has:

Plaintext **p** (arbitrary size)

Recipient's public $\mathbf{k_{pub}}$ (fixed size)

Encryption function **E**

Decryption function **D**

Can produce ciphertext **c**:

$\mathbf{c = E(p,k_{pub})}$

Can recover plaintext:

Need recipient private key $\mathbf{k_{priv}}$

$\mathbf{p = D(c,k_{priv})}$

Also works if you reverse the keys:

$\mathbf{D(E(p,k_{priv}),k_{pub}) == p}$

Asymmetric crypto can also authenticate…

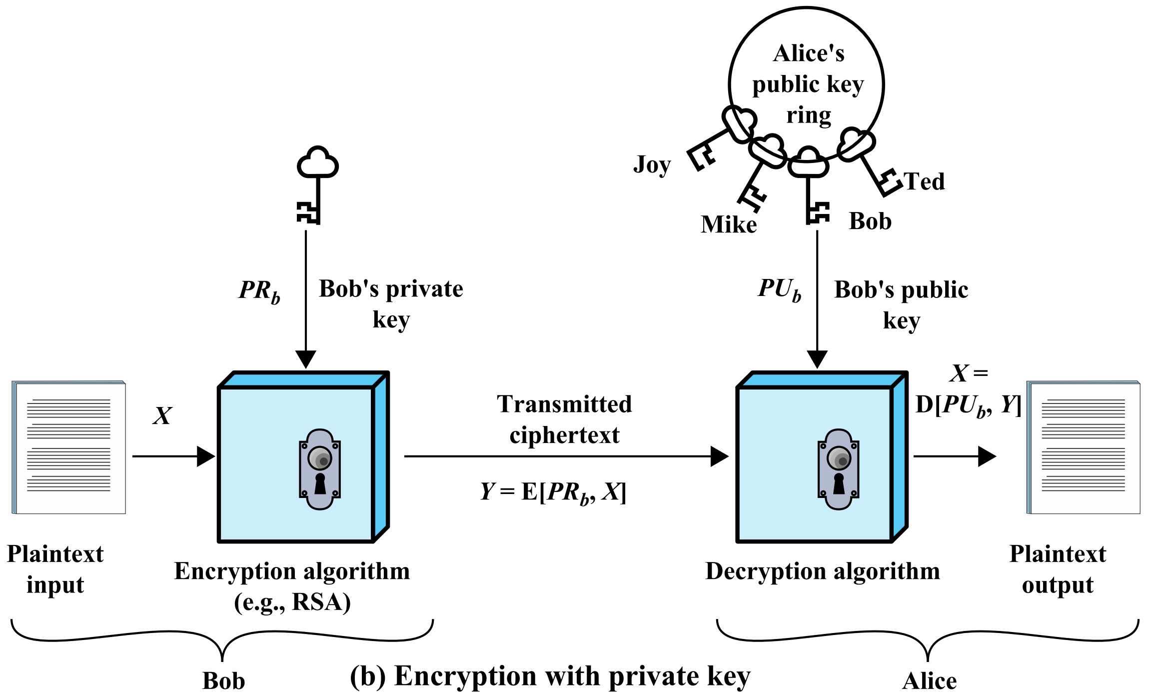


Figure 2.6 Public-Key Cryptography

- Bob encrypts data using his own ***private*** key
- Anyone who knows the corresponding ***public*** key will be able to decrypt the message
  - Proves it was encrypted with Bob's private key → Bob produced this!

Properties of asymmetric crypto systems

- It must be computationally easy:
  - To create key pairs
  - For a sender knowing the public key to encrypt messages
  - For a receiver knowing the private key to decrypt ciphertext
- It must be computationally infeasible:
  - To determine the private key from the public key (or vice versa)
  - To otherwise recover original message (duh)

Asymmetric crypto algorithms

- In symmetric crypto, the list of algorithms just differed in bit sizes and implementation details
- *Asymmetric* algorithms differ in *fundamental method of use*

- Key algorithms:
  - **Diffie-Hellman**: *Just* solves the problem of <u>agreeing to a secret symmetric key</u> over an open communication channel. Doesn't encrypt/decrypt or authenticate on its own.
  - **DSS**: Digital Signature Standard – *Just* able to provide <u>authentication</u>. Doesn't encrypt/decrypt on its own.
  - **RSA**: The original <u>general purpose asymmetric algorithm</u> – able to do encryption/decryption (shown 3 slides ago) and signatures (2 slides ago)
  - **Elliptic Curve (e.g., X25519)**: Uses different fundamental math than the above (smaller keys, more efficient) but <u>achieves the same goals</u> (encryption/decryption and signatures)

RSA Public-Key Encryption

- Developed by <u>R</u>ivest, <u>S</u>hamir & <u>A</u>dleman in 1977
  - Best known and widely used public-key algorithm
- Uses exponentiation of integers modulo a prime
- Given *integers*:
  - Plaintext $\boldsymbol{p}$
  - Public key $\mathbf{k_{pub}} = \{e, n\}$ (Known to sender)
  - Private key $\mathbf{k_{priv}} = \{d, n\}$ (Known to receiver)
- Encrypt: $\boldsymbol{c} = \boldsymbol{p}^e \,\%\, n$
- Decrypt: $\boldsymbol{p} = \boldsymbol{c}^d \,\%\, n$

Where do you get the numbers: Key generation

- Choose two distinct prime numbers $\boldsymbol{p}$ and $\boldsymbol{q}$.
  - Secret, random, similar in magnitude, chosen to make factoring hard.
- Get product $\boldsymbol{n = pq}$
  - Used in modulus in decrypt/encrypt. Length in bits is the “key length”. Part of both keys.
- Compute $\boldsymbol{\varphi = (p-1)(q-1)}$
- Choose an integer $\boldsymbol{e}$ that’s coprime with $\boldsymbol{\varphi}$ (no common factors)
  - $\mathbf{gcd}(e, \varphi) = 1$ and $1 < e < \varphi$
  - $\boldsymbol{e}$ being not very secret is okay; it’s often $2^{16} + 1 = 65537$
  - $\boldsymbol{e}$ is part of the public key
- Determine $\boldsymbol{d}$ by solving $\boldsymbol{de \ \% \ \varphi = 1}$
  - There’s an efficient algorithm for this, since we know $\boldsymbol{\varphi}$ (but $\boldsymbol{\varphi}$ will be discarded later, and it’s not efficient to solve without it)
  - $\boldsymbol{d}$ is part of the private key

Q: Hey this is a lot of math. Do I have to care?
A: Nah

RSA example

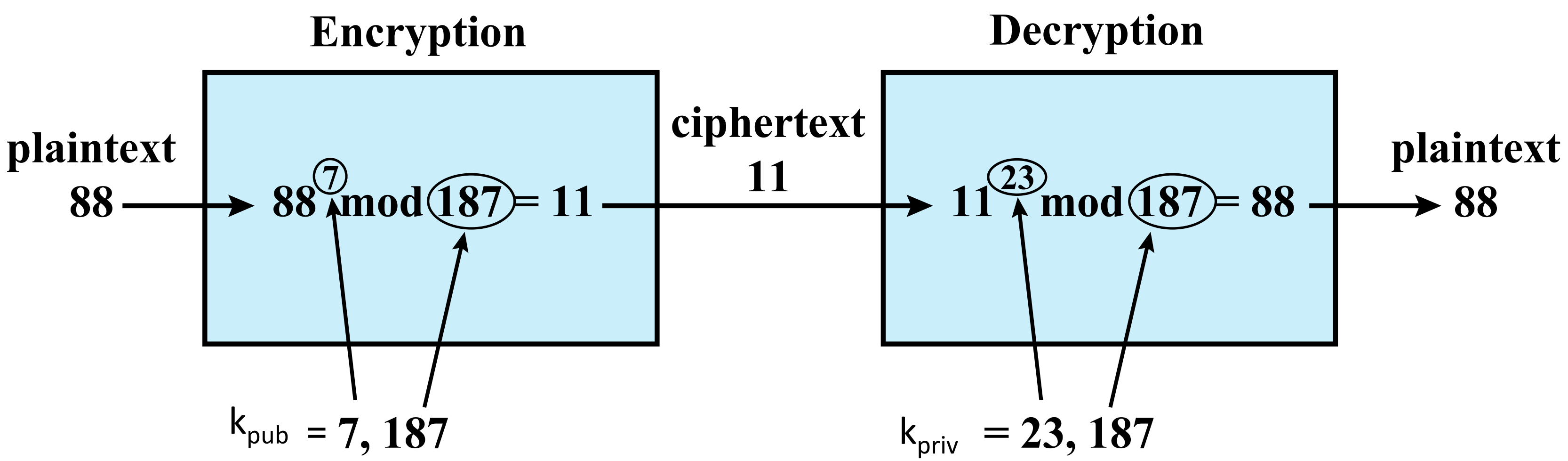


Figure 21.8 Example of RSA Algorithm

- See? It works.
- In practice, all the numbers are muuuuuuuch bigger.

How long of a key do you need?
Or, How good are we at factoring RSA keys?

- RSA Factoring Challenge – cash prizes for factoring big n values
- Computing gets faster/cheaper, *and* algorithms are getting better
  - 1024-bit keys are out there…
  - 2048-bit keys are the default now
- The RSA algorithm is cool, but slow, and uses giant keys
- Waning in popularity, being replaced by Elliptic Curve algorithms (covered later)

RSA number	Decimal digits	Binary digits	Factored on
RSA-100	100	330	Apr 1991
RSA-110	110	364	Apr 1992
RSA-120	120	397	Jul 1993
RSA-129	129	426	Apr 1994
RSA-130	130	430	Apr 1996
RSA-140	140	463	Feb 1999
RSA-150	150	496	Apr 2004
RSA-155	155	512	Aug 1999
RSA-160	160	530	Apr 2003
RSA-170	170	563	Dec 2009
RSA-576	174	576	Dec 2003
RSA-180	180	596	May 2010
RSA-190	190	629	Nov 2010
RSA-640	193	640	Nov 2005
RSA-200	200	663	May 2005
RSA-210	210	696	Sep 2013
RSA-704	212	704	Jul 2012
RSA-220	220	729	May 2016
RSA-230	230	762	Aug 2018
RSA-232	232	768	Feb 2020
RSA-768	232	768	Dec 2009
RSA-240	240	795	Dec 2019
RSA-250	250	829	Feb 2020
RSA-260	260	862	
RSA-270	270	895	
RSA-896	270	896	
RSA-280	280	928	
RSA-290	290	962	
RSA-300	300	995	
RSA-309	309	1024	
RSA-1024	309	1024	

From https://en.wikipedia.org/wiki/RSA_Factoring_Challenge

Timing: Another avenue of attack…

- In crypto, must also be concerned with **side-channel attacks**: Looking at "side info" or "meta info" to cheat and learn secrets
- Example: If you have accurate <u>time measurement</u> of decryption in a naïve RSA implementation, you can determine the private key!
  - This is a **timing attack**, a form of side-channel attack
  - Applicable not just to RSA, but also to other public-key crypto systems
- Countermeasures:
  - **Constant exponentiation time**: Ensure that all exponentiations take the same amount of time before returning a result – simple but slows things down
  - **Random delay**: Better performance, but attacker could do many measurements and statistically tease out actual delay
  - **Blinding**: Multiply the ciphertext by a random number before performing exponentiation then divide out - prevents attacker from knowing the actual ciphertext bits

Diffie-Hellman Key Exchange

- RSA is so slow! I want to use fast symmetric crypto…
  - Could use RSA to send a random secret key, but we can do better!

*Introducing **Diffie-Hellman Key Exchange**!*

- Uses similar mathematical foundations as RSA,
  but <u>just</u> for efficient key exchange.
  - Actually the first published public-key algorithm (1976)
- Two parties on an *open channel* can agree on a secret! Wow!!!
- Used lots of places (including the web in HTTPS – except it’s being replaced by an Elliptic Curve equivalent now)
- Relies on difficulty of computing discrete logarithms

Diffie-Hellman in operation

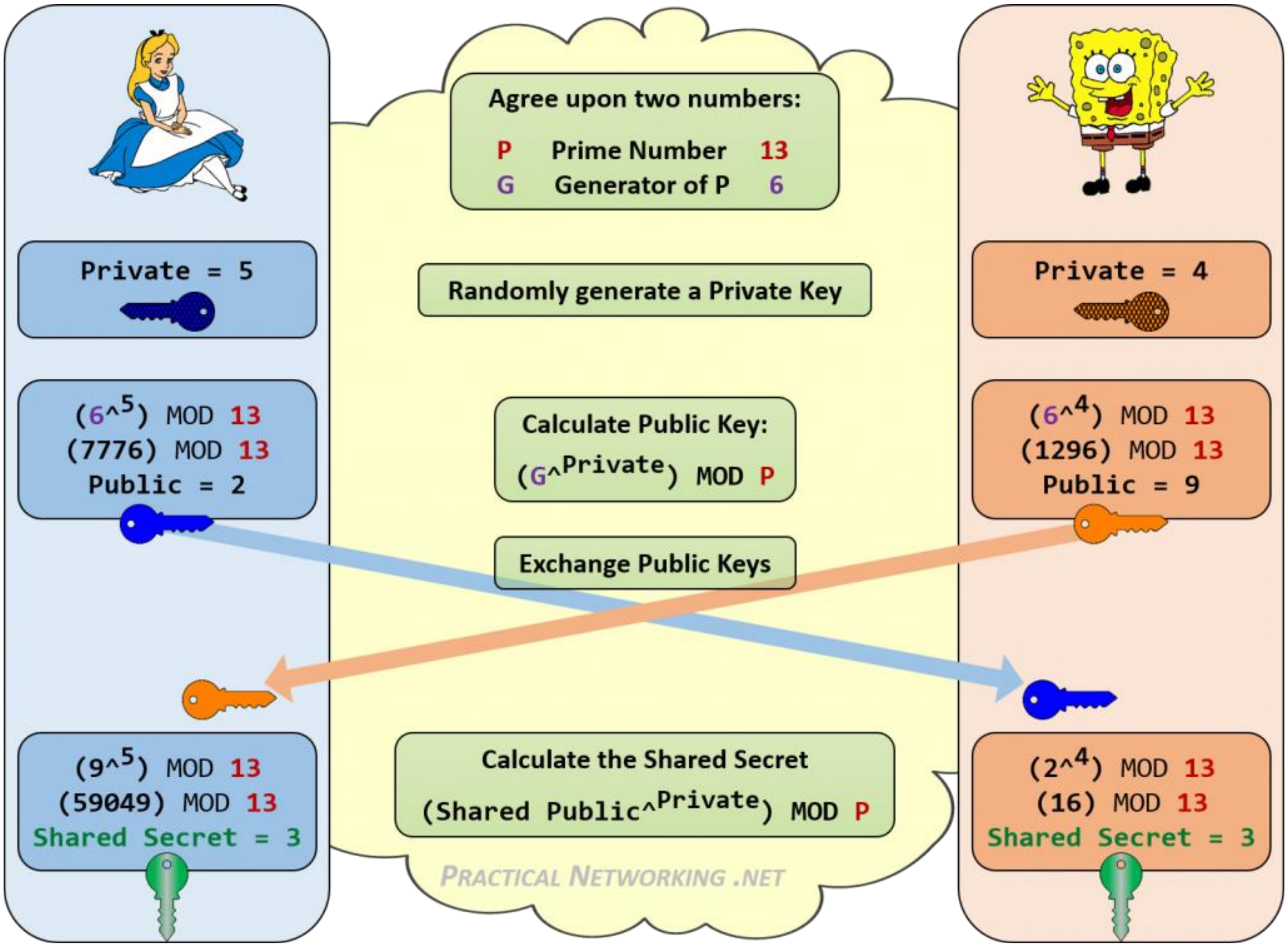


Figure from here.

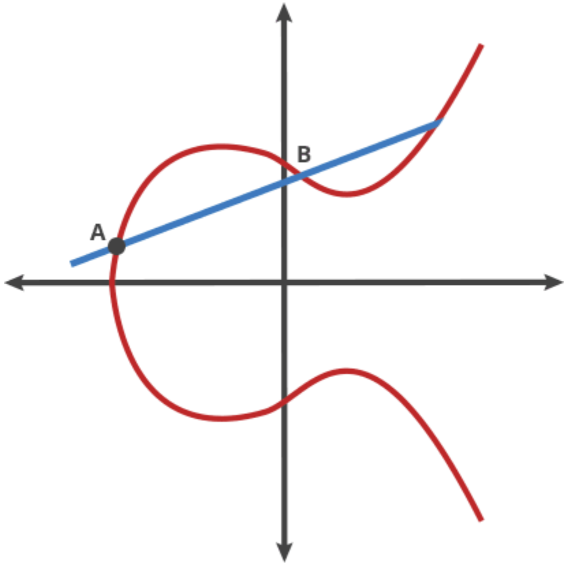
Eavesdropping attacker would need to solve $6^x \bmod 13 = 2$ or $6^x \bmod 13 = 9$, which is hard.

Elliptic Curve (EC) cryptography

- RSA and Diffie-Hellman both rely on basic discrete math
  - Relatively large key sizes, relatively slow operation

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

  - Need a specific curve equation to use
  - Various competing ones, including some weakened by the NSA and/or covered by patents – we don't like those
  - Resulting favorite: **Curve25519** aka **X25519**



"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)

This is crucial.
It's so common that most asymmetric crypto implementations literally *can't* work on large data.

There's no such thing as a "mode of operation" for RSA.

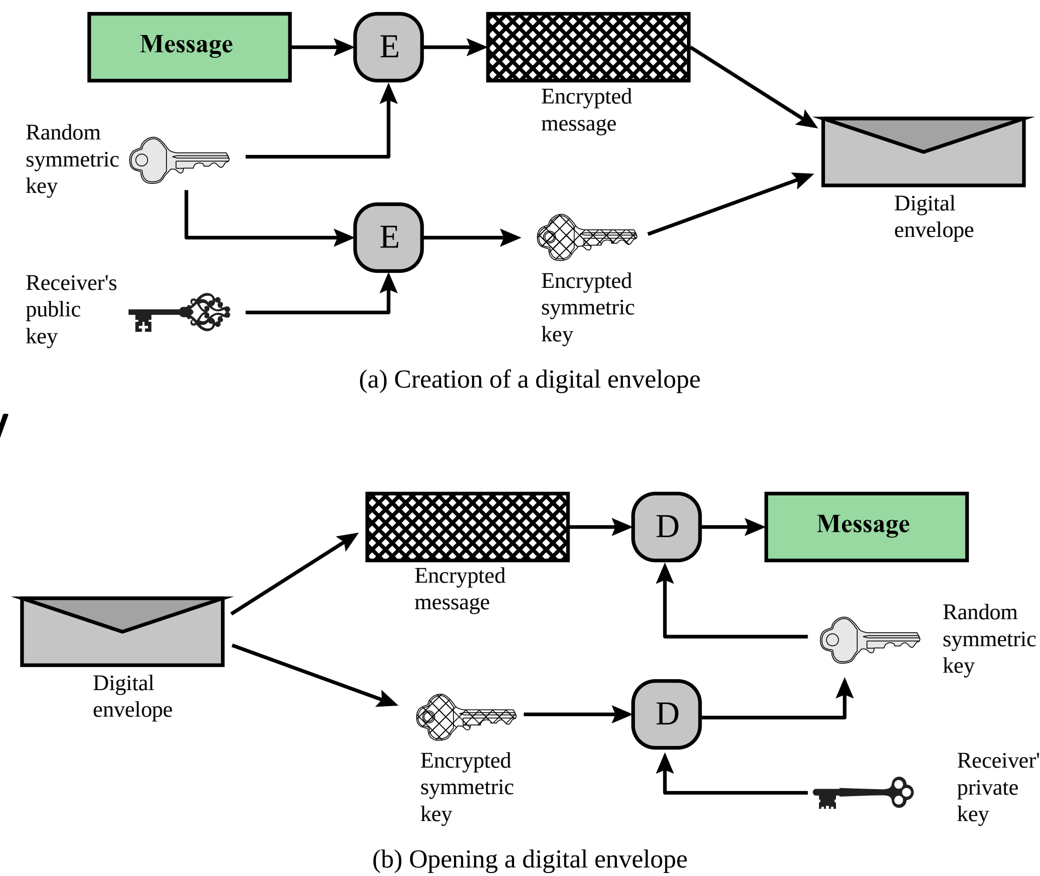
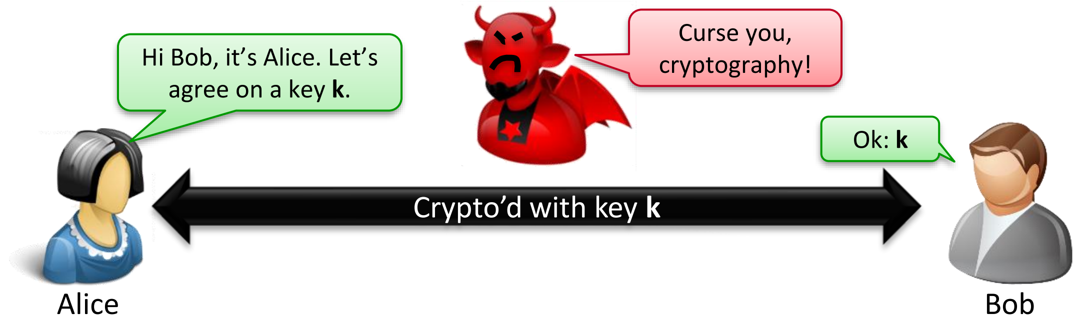


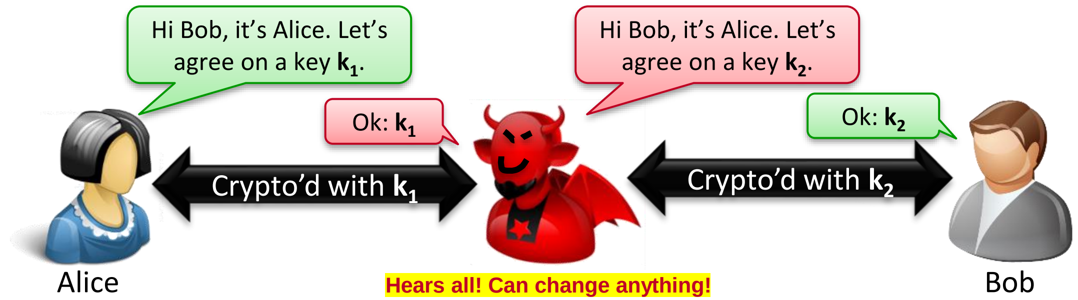
Figure 2.9 Digital Envelopes

So where are we now?

- We can use symmetric or asymmetric cryptography to get confidentiality even if there's an eavesdropper. Yay!



- But what what if the attacker is a **man in the middle**?
  - Can intercept/alter communications



- Need to **authenticate** endpoints!

Establishing Authenticity

Secure Hash Functions,
Message Authentication Codes (MAC), and
Digital Signatures

The Authenticity Problem

- Problem: who sent this message?

- Best solution: the actual person appears to confirm it
  - Not feasible.
- Best practical solution: Sender includes some data that *only* they could have created
  - But how could *only* they have created it?
    Because only they had the key to do so!
  - Attacker: Get that key! Or fool you into validating against the wrong key!

Techniques to authenticate

Two broad approaches similar to crypto:

- **Message Authentication Codes (MACs)** – based on a secret key (symmetric)
  - Sender and receiver need to agree on a shared secret to authenticate
- **Digital signatures** – based on asymmetric crypto
  - Sender uses their *private* key;
    Receiver uses sender's *public* key to authenticate
- Either way, **confidentiality** (from crypto) and **authenticity** (from the above) are separate things.
  - Confidentiality without authenticity? Secrets sent anonymously
  - Authenticity without confidentiality? Public info from a trusted source
  - Confidentiality + authenticity? Secure communication

MAC concept

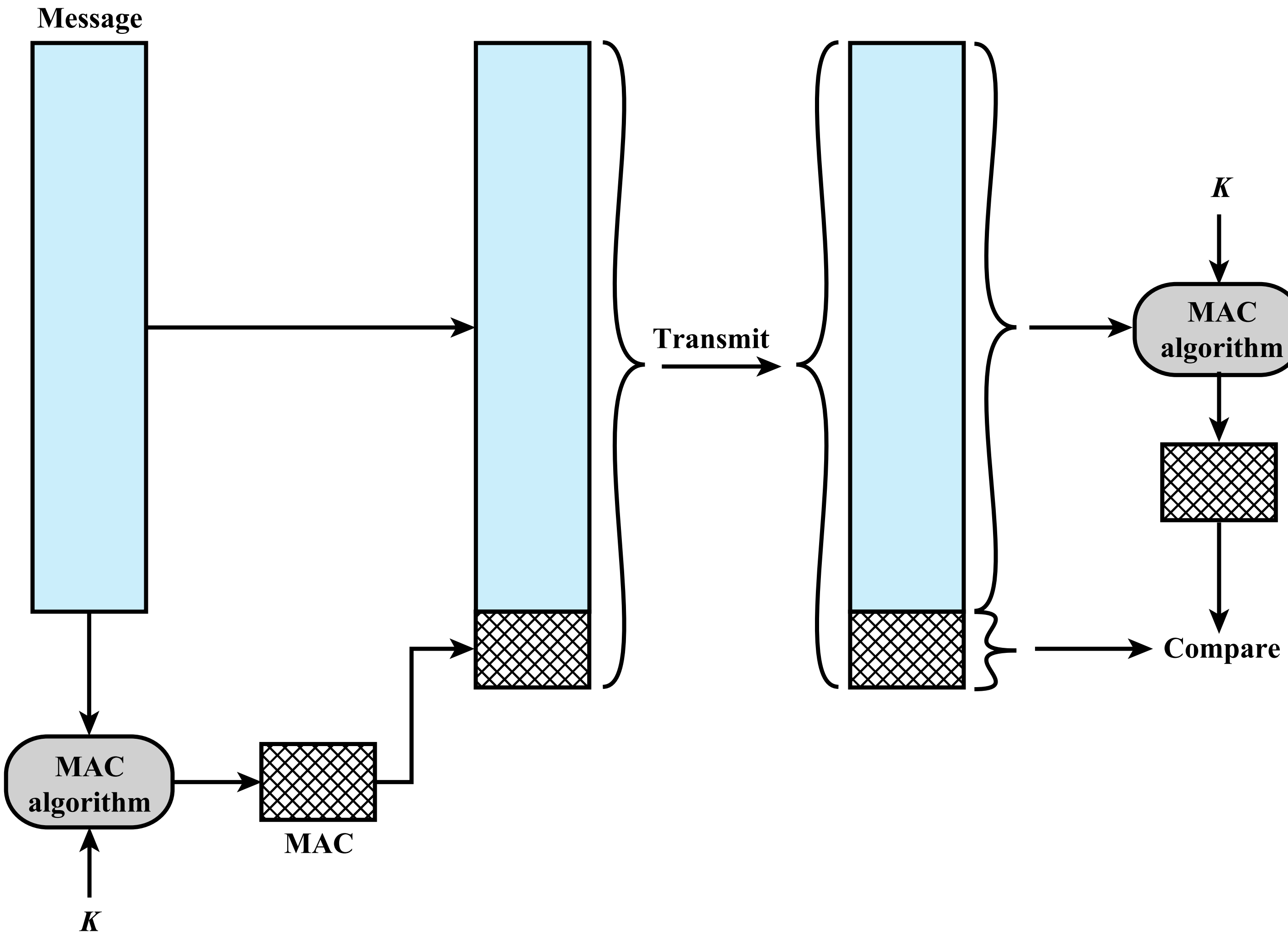


Figure 2.3 Message Authentication Using a Message Authentication Code (MAC).

Methods of implementing a MAC

- 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
  - Kinda expensive – we can do better

- Can use **hash functions**:
  - Non-reversible, arbitrary size input to fixed size output
  - Various schemes for how to use a hash function for this
  - Hash functions used for this have more requirements than ones used for data structures like a hash table – they're called **cryptographic hash functions**

Cryptographic Hash Functions

A cryptographic hash function H(x) must:

- Eat data of any size and give fixed-length output
- Be easy to compute for any given input
- Be **one-way** (a.k.a. pre-image resistant):

  Computationally infeasible to find x from H(x)
- Have **weak collision resistance**:

  Given x, computationally infeasible to find $y \neq x$ such that H(x) = H(y)
- Have **strong collision resistance**:

  Computationally infeasible to find any pair (x, y) such that H(x) = H(y)
- Have the **avalanche effect**:

  A small change to the input should totally change the output

How to **attack** a hash to violate the above?
Same as crypto: either **cryptanalysis** (find algorithm weaknesses) or **brute force** (try all possible inputs)

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

Ways of using a hash to authenticate

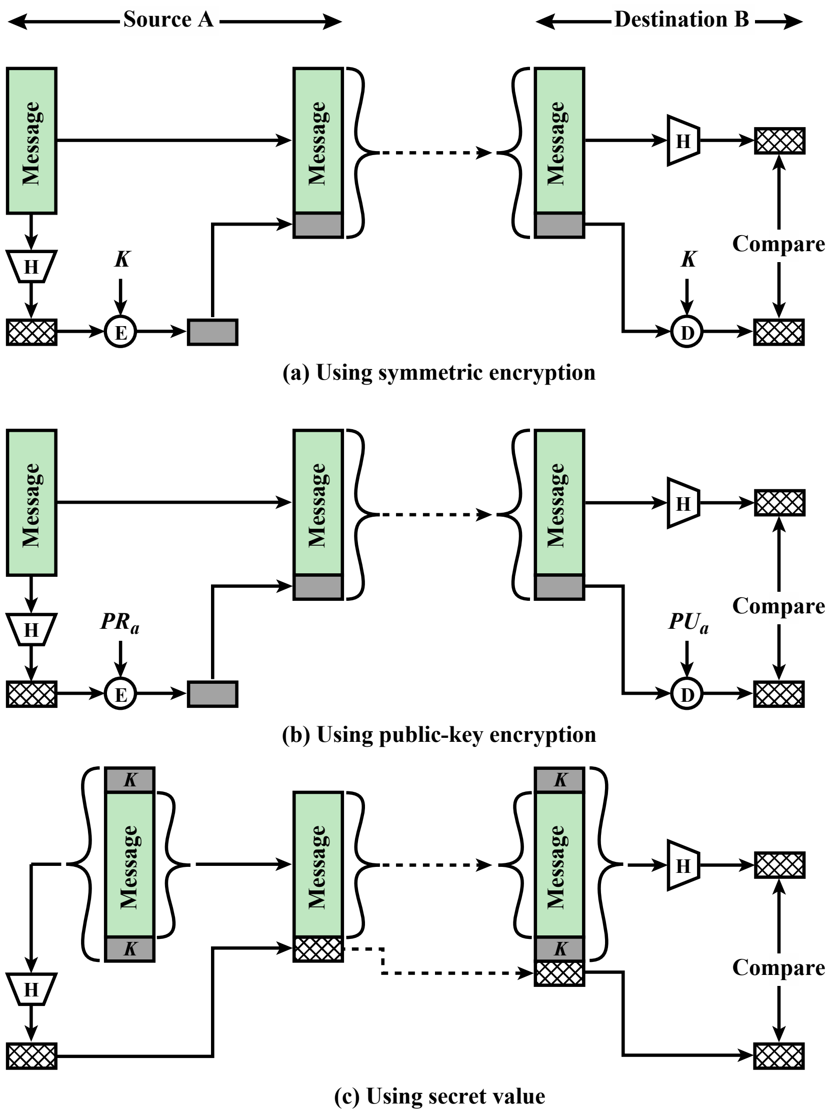


Figure 2.5 Message Authentication Using a One-Way Hash Function.

Uses **hash + symmetric crypto.** Gives a **MAC** – sender and receiver need same key.

Uses **hash + asymmetric crypto.** Gives a **Digital Signature!** *So important we're going to dig deeper...*

Uses **just a hash.** Neat! Gives a **MAC** – sender and receiver need same key.

Digital Signatures

Digital Signatures

- Digital signature provides (per NIST FIPS PUB 186-4):
  - origin authentication,
  - data integrity, and
  - signatory non-repudiation

The notion that you can't deny it was you that sent the message.

- Common algorithms:
  - Digital Signature Algorithm (DSA)
  - RSA Digital Signature Algorithm
  - Elliptic Curve Digital Signature Algorithm (ECDSA)

  (All based on asymmetric cryptography – public and private keys!)

- Advantage over MAC:
  - We don't have to pre-share the key!
    (same advantage as asymmetric crypto)

Digital Signature overview

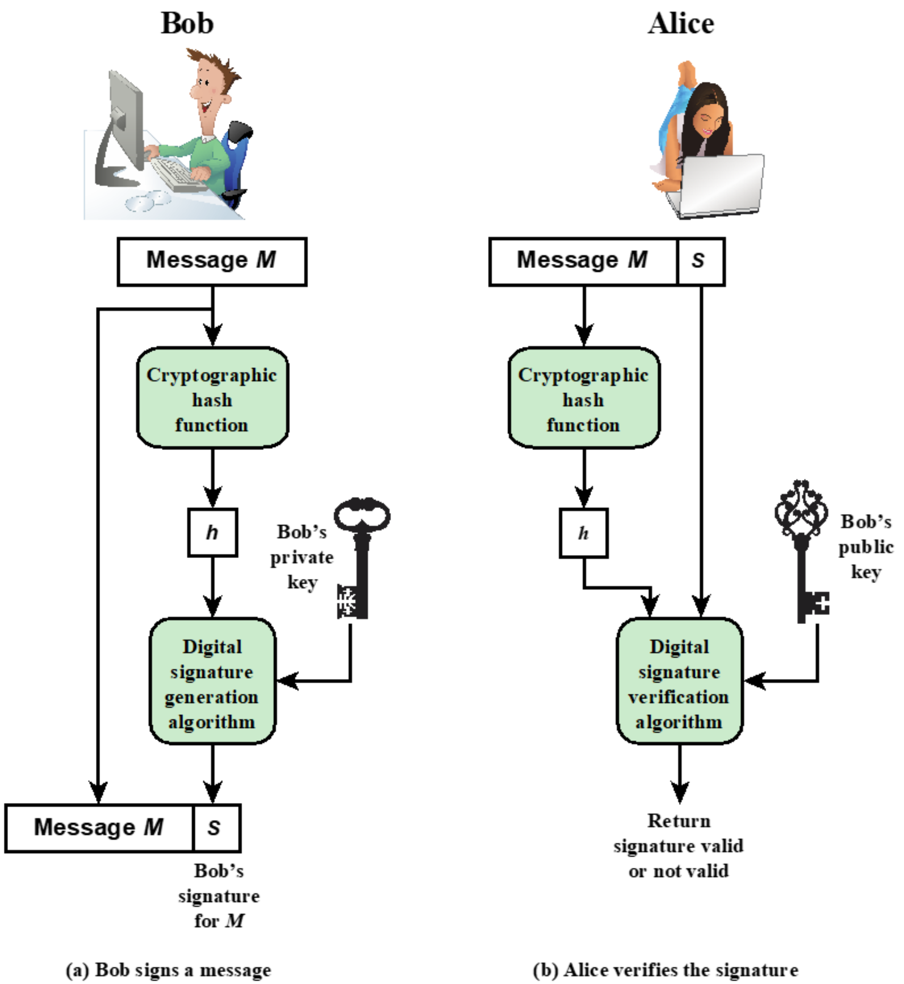


Figure 2.7 Simplified Depiction of Essential Elements of Digital Signature Process

The recursive problem of signatures

Alice can't remotely prove to you that a given key is hers on her own. ☹

Recurse! (1)

Proposed solution:

We have someone ELSE sign a message containing Alice's key.

Recurse (2)

Proposed solution:

We have someone ELSE sign a message containing Bob's key.

Recurse (3)

Proposed solution:

We have someone ELSE sign a message containing Clara's key.

The tiny text says "Don" is the next guy.

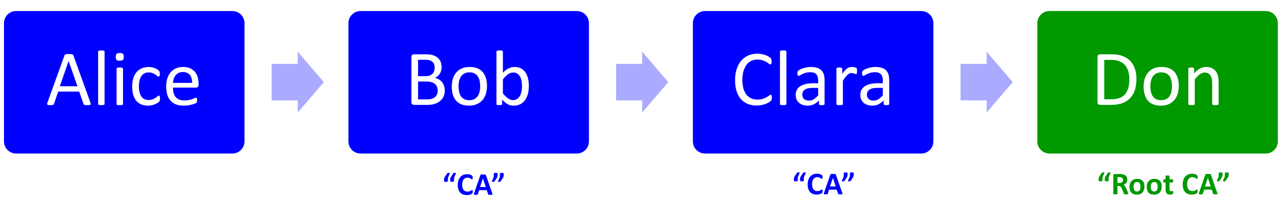
Recurse (4)
The base case

What about Don's key?

We got it shipped to us just for this. We trust it implicitly.

Certificates and the chain of trust

- A **certificate** is a message that:
  - **Contains someone's identity** and **their public key**, and
  - **Is signed by someone else (usually*)**.
- Each message on the previous slide was a certificate, and everyone signing a certificate was a **certificate authority (CA)**.
- The entity that we trust implicitly is a **root certificate authority**.
- Together, they had this **chain of trust**:



---

* It is possible to sign your own certificate. This is called a "**self-signed certificate**", and is used when you want to manually import or approve the key without using the chain of trust.

How to get a certificate

1. Generate a **public/private key pair**.

   Keep the private key secret forever!!

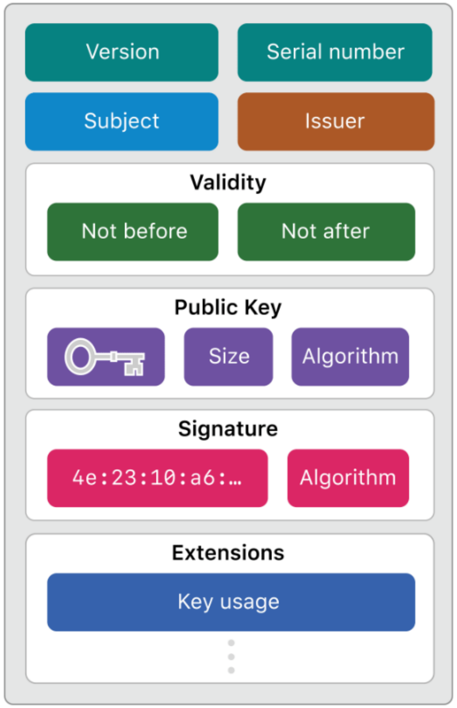
2. In a secure manner, present your public key to the CA in the form of a **certificate signing request (CSR)** – basically everything of the certificate except the signature. Provide some proof of identity.
3. The CA verifies your identity and signs the CSR, thus creating the **certificate**, which you are given.

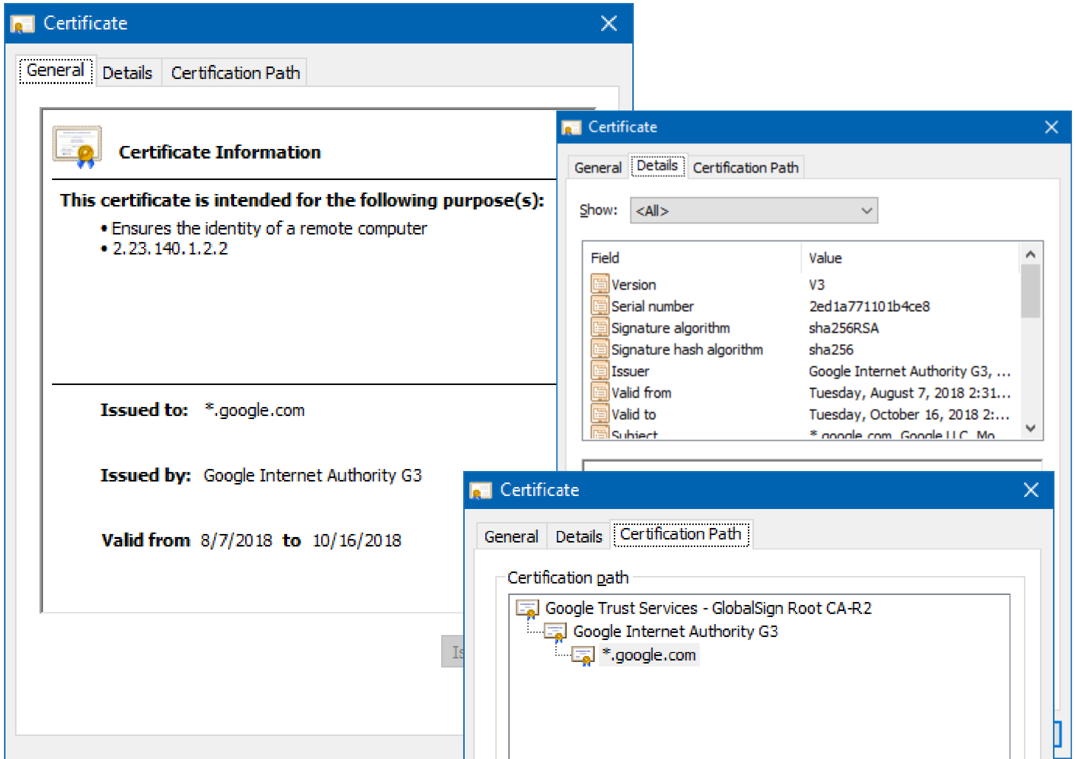
You can now show the certificate to anyone who asks, and as long as they trust your CA (either directly or recursively), they trust that the key shown is yours.

Now we can advertise public keys with confidence!

Certificates in practice: X.509

- Most certificates are in **X.509 format** specified in RFC 5280.
- Used in many contexts, including:
  - **IP security (IPSEC)**: Used in Virtual Private Networks (VPNs)
  - **S/MIME**: Encrypted/authenticated email
  - **Secure sockets layer (SSL)** and its successor **Transport Layer Security (TLS)**
    - This includes **HTTPS**!



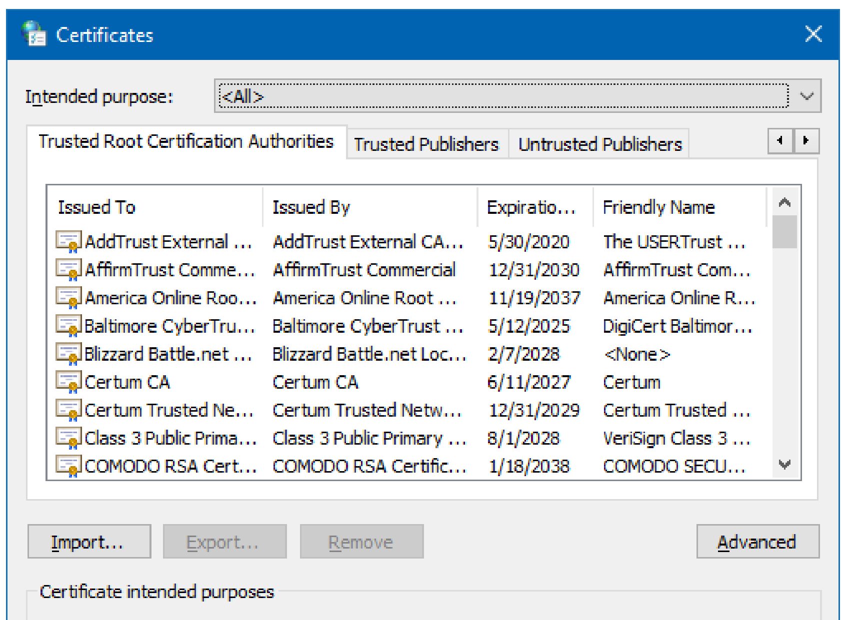


Public-Key Infrastructure (PKI)

- All of this certificate and chain-of-trust stuff is part called **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." -- RFC 4949
- Includes a **trust store**: A list of CA's and their public keys
  - But how do Root CA certificates make it into the trust store?

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)



Random number generation

Wait, how do we make keys again?

- Symmetric crypto:
  - Generate **random** bits.
- Asymmetric crypto:
  - Choose two **random** prime numbers p and q, then do more stuff

- **Randoms** also used for nonces, initialization vectors, and more!

Fundamental problem

Computers are deterministic,
true randomness is almost impossible for them

So we fake it with **Pseudo-Random Number Generators (PRNGs)**
or
Add hardware **True Random Number Generators (TRNGs)**

Requirements for random number generator

- PRNG: Algorithm to do a bunch of math, update **internal state,** and kick out a random. Requirements:
  - **Uniform distribution:** Frequency of occurrence of each of the numbers should be approximately the same
    - In binary, 0's and 1's occur with equal chance.
  - **Independence**: Different sequences are entirely distinct (no patterns)
  - **Uninformative sequences:** Impossible to tell from a given sequence any previous or future values or any inner state of the generator.
  - **Uninformative state:** Impossible to tell, if given the inner state of the generator, any previous numbers in the sequence.

Plain PRNG (uniform distribution, independence)
Good enough for games.

Cryptographically-secure PRNG (CPRNG) (uninformative sequences, uninformative state)
Good enough to make keys!

U.S. FIPS publication 140-1 specifies precise tests of the above

Dangers when seeding the PRNG

- All PRNGs take in a seed (initial state)
  - Given the same seed, it will generate the same sequence
  - For basic randomness (video game junk), you can seed with the **current time**
    - Guaranteed unique sequence ☺
    - For crypto purposes, current time is a super bad choice:
      If I know when you made your keys, then I can figure out your keys ☹
  - Instead, feed in a large amount of external **entropy** (true randomness)
    - IO device delays, mouse movements, keyboard timing
    - Modern kernels are always recording that stuff into an **entropy pool**
    - Read from **`/dev/random`**: pull from this pool and it's used up (rate limit)
      Read from **`/dev/urandom`**: not crypto-secure, but not rate limited

- Given the initial state, the PRNG's full sequence is predictable,
  so don't leak the initial state!

Random versus Pseudorandom

- What's better than a **Pseudo-Random Number Generator?**
- A **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
  - Intel x86: The `RDRAND` instruction uses onboard TRNG. Available on Intel and AMD chips since 2015.
  - Similar features available in many other architectures.
  - You can even get a USB one if you really need it

TrueRNG V3 - USB Hardware Random Number Generator

Brand: ubld.it

28 ratings | 17 answered questions

Price: $59.95 prime & FREE Returns

- High Output Speed: >400 kilobits / second
- Internal Whitening
- Native Windows (XP/8/8.1/10) and Linux Support (CDC Virtual Serial Port)
- Passes all the industry standard tests (Dieharder, ENT, Rngtest, etc.)
- Compatible with embedded environments such as Beaglebone, Raspberry PI, and UDOO

Report incorrect product information.

The quantum computing threat to cryptography

- **Quantum computing** uses quantum mechanical properties like superposition and entanglement to do computing
  - Can perform algorithms in entirely better time domains! $O(2^n)$ might become $O(n^2)$!
- **A problem for cryptography!**
  - **Asymmetric encryption *breaks***: Shor's Algorithm can factor integers in polynomial time! **RSA** and **Elliptic Curve** will be dead. ☹
    - Standardization underway *now* for replacements (link)
    - NIST just picked their fourth round candidates in July 2022
    - New standards expected in 2022-2024 timeframe
  - **Symmetric encryption is *weakened***: Grover's Algorithm makes a brute-force search for a key $\sqrt{n}$ faster, so:
    - 128-bit keys are like 64-bit keys (broken)
    - 256-bit keys are like 128-bit keys (still okay!)
    - So AES-256 will survive. ☺

Want to learn more?
Duke is a world leader in quantum computing! Check our variety of quantum computing courses!

The current state of quantum-proof crypto

- We are currently living through cryptography standards history!
- NIST is currently running a competition for the first standard quantum-resistant public key cryptography algorithm
  - December 2016: Contest started (link to request for proposals)
    - Researchers from around the world submit proposed algorithms
    - Multiple rounds of evaluation and elimination occur
  - July 2022: Four candidate winners selected (announcement)
    - General encryption: CRYSTALS-Kyber
    - Digital signatures: CRYSTALS-Dilithium, FALCON and SPHINCS+
    - Four separate runners up also selected as backups
  - August 2022: One of the runner up algorithms, SIKE, is defeated (article)
    - This algorithm survived scrutiny since 2017 before being defeated!
    - Shows why the selection process is long and methodical…

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 can reveal key; also, ECB mode doesn't use an IV!
  - 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 (or Diffie-Hellman 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

Note: We'll cover password storage at length later when we cover Authentication.

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 (256+ 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 Transport Layer Security (TLS), 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)

- 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

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

H = Hash function

- 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

Crypto applications summary

- Most common algorithms:
  - **Symmetric crypto**: **AES**
  - **Asymmetric crypto**: classic **RSA**, modern **Elliptic Curve**
  - **Secret key generation**: classic **Diffie-Hellman**, modern **Elliptic Curve Diffie-Hellman (ECDH)**
  - **Signature**: classic **RSA** or **DSA**, modern **ECDSA**
  - **Hash**: obsolete **MD5** and **SHA-1**, modern **SHA-2** and **SHA-3**
- **Encryption** provides confidentiality, **MACs/Signatures** provide integrity & authenticity
- **Public Key Infrastructure**:
  - **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 asymmetrically encrypt a secret key, then symmetrically encrypt the actual payload

Crypto applications summary

- Most common algorithms:
  - **Symmetric crypto**: **AES**
  - **Asym**

- **I**
  M
- **P**
  -
  -
  -
- **Dig** encrypt a secret key, then actual payload

Even better summary

FIND A VETTED LIBRARY THAT DOES IT FOR YOU