

Lecture 12 – Public key Crypto

Andrew Miller / Michael Bailey

University of Illinois

ECE 422/CS 461

Review: Integrity

Problem: Sending a message over an **untrusted channel** without being changed

Provably-secure solution: **Random function**

Practical solution:



Pseudorandom function (PRF)

Input: arbitrary-length **k** *Output:* fixed-length value

Secure if practically indistinguishable from a random function, unless know **k**

Real-world use: **Message authentication codes (MACs)**

built on cryptographic hash functions

Popular example: **HMAC-SHA256**_k(m)

[Cautions31]

Review: Confidentiality

Problem: Sending message in the presence of an **eavesdropper** without revealing it

Provably-secure solution: **One-time pad**

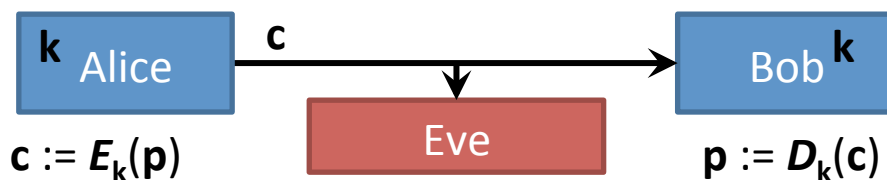
Practical solution:

Pseudorandom generator (PRG)

Input: fixed-length k

Output: arbitrary-length stream

Secure if practically indistinguishable from a random stream, unless know k



Real-world use: **Stream ciphers** (can't reuse k)

Popular example: **AES-128 + CTR mode**

Block ciphers (need **padding/IV**) Popular example: **AES-128 + CBC mode**

[Cautions?!]

Using OpenSSL to do AES encryption from the command line

\$ KEY=\$(openssl rand -hex 16) Generates a random string

**\$ openssl aes-256-cbc -in mymsg.txt -out mymsg.enc
-p -K \${KEY} -iv \$(openssl rand -hex 16)
key=8582D9E1A36DA4DB065394FB1F401DB3
iv =DBB272FE6486C4D9B09DBE464E080468**

Prints the key and IV

**\$ openssl aes-256-cbc -d -in mymsg.enc -out mymsg.txt
-K \${KEY} -iv <iv from above>**

- By default, uses the standard padding described earlier
- Unfortunately, you have to handle prepending/extracting the IV on your own

Key Exchange

Issue: How do we get a shared key?



Amazing fact:

Alice and Bob can have a public conversation to derive a shared key!

Diffie-Hellman (D-H) key exchange

1976: Whit Diffie, Marty Hellman, improving partial solution from Ralph Merkle (earlier, in secret, by Malcolm Williamson of British intelligence agency)

Relies on a mathematical hardness assumption called *discrete log problem* (a problem believed to be hard)



IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. IT-22, NO. 6, NOVEMBER 1976

New Directions in Cryptography

Invited Paper

WHITFIELD DIFFIE AND MARTIN E. HELLMAN, MEMBER, IEEE

Group Theory Basics

Schnorr groups

A Schnorr group **G** is a subset of numbers, under **multiplication**, modulo a prime **p**. (a “safe prime”)

- We can check if a number **x** is an element of the group
- If **x** and **y** are in the group, then **x y** is in the group too
(**x y** means **x** times **y mod p**)
- **g** is a **generator** of the group if every element of the group can be written as **g^x** for some exponent **x**.

g^x — Exponent, $0 \leq x < (p - 1)/2$
— Generator, an element of the group

What is a Group?

A class of mathematical objects (it generalizes “numbers mod p ”)

Definition: A group $(\mathbf{G}, *)$ is a set of elements \mathbf{G} , and a binary operation $*$

- (*Closed*): for any $\mathbf{x}, \mathbf{y} \in \mathbf{G}$, we know $\mathbf{x} \mathbf{y} \in \mathbf{G}$
- (*Identity*): we know the identity \mathbf{e} in \mathbf{G}
for any $\mathbf{x} \in \mathbf{G}$, we have $\mathbf{e} \mathbf{x} = \mathbf{x} = \mathbf{x} \mathbf{e}$
- (*Inverses*): for any \mathbf{x} , we can compute $\mathbf{x}^{-1} \mathbf{x} = \mathbf{e}$
- (*Associative*): For $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbf{G}$, $\mathbf{x} (\mathbf{y} \mathbf{z}) = (\mathbf{x} \mathbf{y}) \mathbf{z}$

Schnorr Groups in more detail

To generate a Schnorr group:

1. Pick a random, large, (e.g. 2048 bits) “safe prime” p
 p is a “safe prime” if $(p - 1) / 2$ is also prime
2. Pick a random number g_0 in the range 2 to $(p - 1)$
3. Let $g = (g_0)^2 \bmod p$. If $g = 1$, loop at step 2

This is the “generator” of the group.

- A number x is in the group if $x^2 \neq 1 \bmod p$
- The order of each element is $(p - 1) / 2$.

$$g^{(p-1)/2} = 1 \bmod p$$

- We can compute inverses x^{-1} s.t. $x^{-1} x = 1 \bmod p$

Problems assumed “hard” in Schnorr groups:

- Discrete logarithm problem

Given g^x for some random x , find x

- Diffie Hellman problem (computational)

Given g^a, g^b for random a, b compute g^{ab}

- Diffie Hellman problem (decisional)

Flip a bit c , generate random exponents

a, b, r

Given (g^a, g^b, g^{ab}) if $c=0$, or (g^a, g^b, g^r) if $c=1$,

Guess c

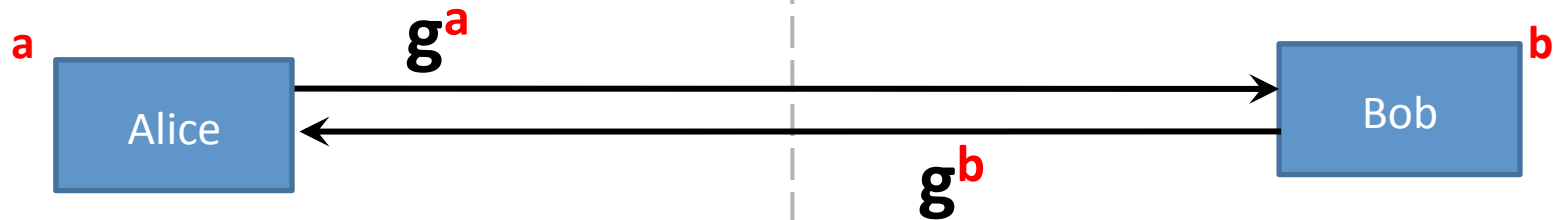
*These problems are thought to be hard in other groups too,
e.g. some Elliptic Curves

Diffie-Hellman protocol

1. Alice and Bob agree on public parameters (maybe in standards doc)

2. **Alice**
Generates random secret exponent **a**.

Bob
Generates random secret value **b**.

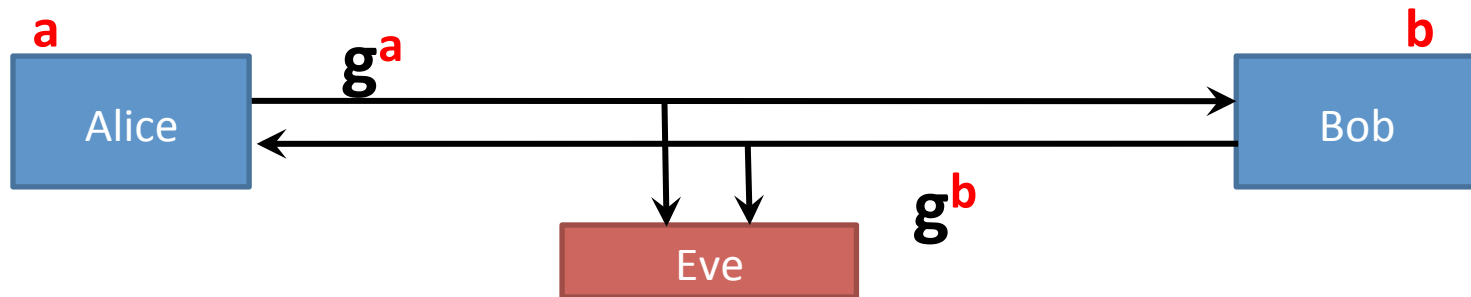


3. **Alice**
Computes x
 $= (g^b)^a$
 $= g^{ba}$

Bob
Computes x'
 $= (g^a)^b$
 $= g^{ab}$

(Notice that $x == x'$)
Can use $k := \text{hash}(x)$ as a shared key.

Passive eavesdropping attack



Eve knows: g, g^a, g^b

Eve wants to compute $x = g^{ab}$

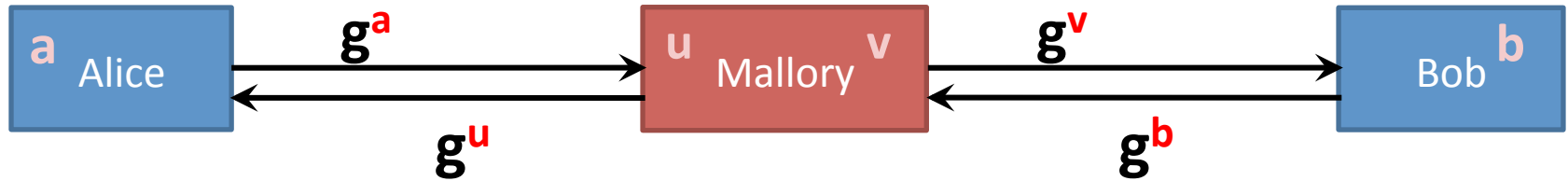
Best known approach:

Find a or b , by solving **discrete log**, then compute x

No known efficient algorithm.

[What's D-H's big weakness?]

Man-in-the-middle (MITM) attack



Alice does D-H exchange, *really with Mallory*, ends up with g^{au}

Bob does D-H exchange, *really with Mallory*, ends up with g^{bv}

Alice and Bob each think they are talking with the other, but really Mallory is between them and knows both secrets

Bottom line: D-H gives you secure connection, but you don't know who's on the other end!

Defending D-H against MITM attacks:

- Cross your fingers and hope there isn't an active adversary.
- Rely on out-of-band communication between users. [\[Examples?\]](#)
- Rely on physical contact to make sure there's no MITM. [\[Examples?\]](#)
- Integrate D-H with user authentication.
 - If Alice is using a password to log in to Bob, leverage the password:
 - Instead of a fixed g , derive g from the password – Mallory can't participate w/o knowing password.
- Use digital signatures. [\[More next week.\]](#)

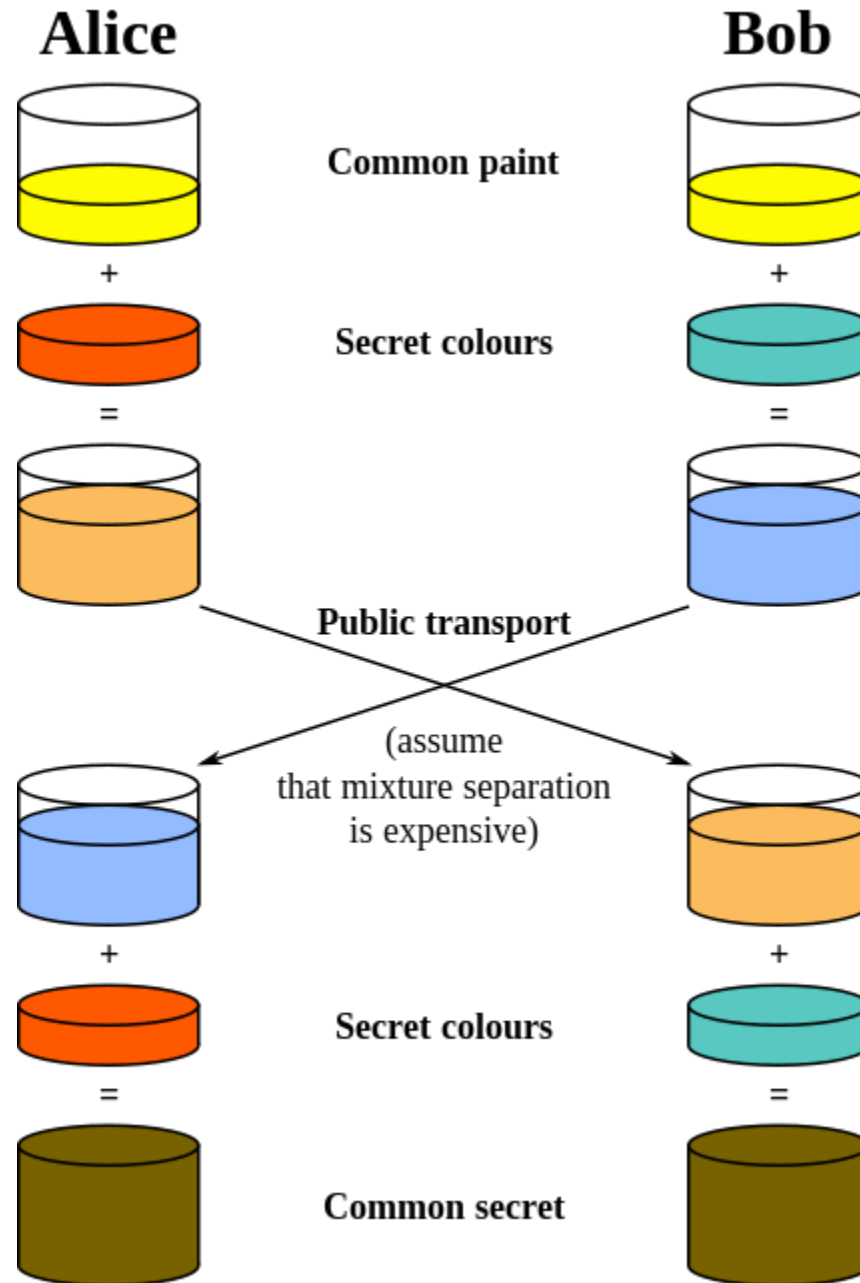
A visual analogy:

“Mixing paints”

Mixing in a new color is a little bit like exponentiation.

Hard to invert?

Two different ways at arriving at the same final result.



Public Key Encryption

Suppose Bob wants to receive data from lots of people, confidentially...

Schemes we've discussed would require a separate key shared with each person

Example: a journalist who wishes to receive secret tips

Public Key Encryption

- **Key generation:** Bob generates a keypair
public key, k_{pub} and private key, k_{priv}
- **Encrypt:** Anyone can encrypt the message M ,
resulting in ciphertext $C = \text{Enc}(k_{\text{pub}}, M)$
- **Decrypt:** Only Bob has the private key needed
to decrypt the ciphertext: $M = \text{Dec}(k_{\text{priv}}, C)$
- **Security:** Infeasible to guess M or k_{priv} , even
knowing k_{pub} and seeing ciphertexts

Public Key Encryption w/ ephemeral key exchange

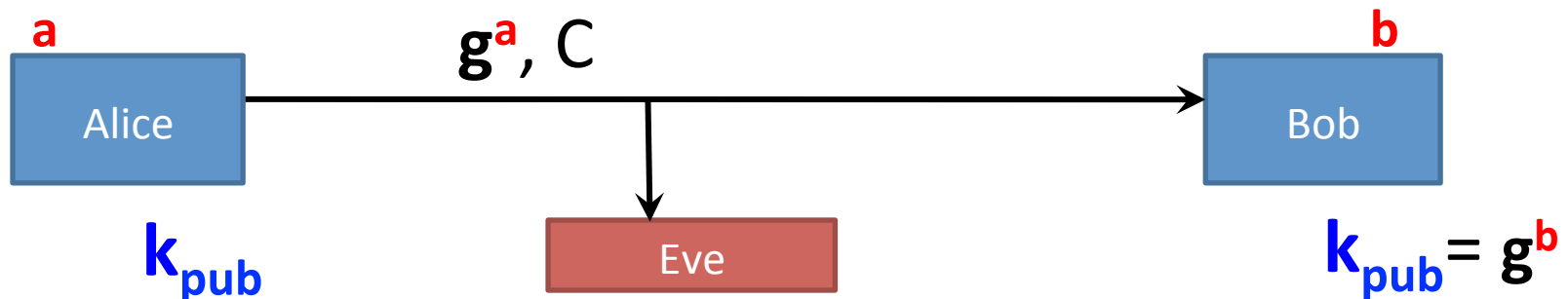
Key generation:

$k_{\text{priv}} := b$ generated randomly, and $k_{\text{pub}} := g^b$

Encrypt:

Generate random a , set $x := \text{hash}(k_{\text{pub}}^a)$, encrypt C using AES. Send (g^a, C) as ciphertext

Decrypt(g^a, C): compute $x = \text{hash}((g^a)^b)$, decrypt



Public Key Digital Signatures

Suppose Alice publishes data to lots of people, and they all want to verify integrity...

Can't share an integrity key with *everybody*, or else *anybody* could forge messages

Example: administrator of a source code repository

Public Key Digital Signature

- Key generation: Bob generates a keypair
public key, k_{pub} and private key, k_{priv}
- Bob can sign a message M , resulting in
signature $S = \text{Sign}(k_{priv}, M)$
- Anyone who knows k_{pub} can check the
signature: $\text{Verify}(k_{pub}, M, S) = ? 1$
- “Unforgeable”: Computationally infeasible to
guess S or k_{priv} , even knowing k_{pub} and seeing
signatures on other messages

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman*



Best known, most common public-key algorithm: **RSA**

Rivest, Shamir, and Adleman 1978

(earlier by Clifford Cocks of British intelligence, in secret)

How RSA works

Key generation:

1. Pick large (say, 1024 bits) random primes **p** and **q**
2. Compute **N** := **pq** (RSA uses multiplication mod **N**)
3. Pick **e** to be relatively prime to $(p-1)(q-1)$
4. Find **d** so that **ed** mod $(p-1)(q-1) = 1$
5. Finally:

Public key is **(e,N)**

Private key is **(d,N)**

To sign: $S = \text{Sign}(x) = x^d \bmod N$

To verify: $\text{Verif}(S) = S^e \bmod N$ Check $\text{Verif}(S)$
=? M

Why RSA works

“Completeness” theorem:

For all $0 < \mathbf{x} < \mathbf{N}$, we can show that $\mathbf{Verif}(\mathbf{Sign}(\mathbf{x})) = \mathbf{x}$

Proof:

$$\mathbf{Verif}(\mathbf{Sign}(\mathbf{x})) = (\mathbf{x}^d \bmod \mathbf{pq})^e \bmod \mathbf{pq}$$

$$= \mathbf{x}^{ed} \bmod \mathbf{pq}$$

$$= \mathbf{x}^{a(p-1)(q-1)+1} \bmod \mathbf{pq} \text{ for some } \mathbf{a}$$

$$(\text{because } \mathbf{ed} \bmod (p-1)(q-1) = 1)$$

$$= (\mathbf{x}^{(p-1)(q-1)})^a \mathbf{x} \bmod \mathbf{pq}$$

$$= (\mathbf{x}^{(p-1)(q-1)} \bmod \mathbf{pq})^a \mathbf{x} \bmod \mathbf{pq}$$

$$= 1^a \mathbf{x} \bmod \mathbf{pq}$$

(because of the fact that if \mathbf{p}, \mathbf{q} are prime, then
for all $0 < \mathbf{x} < \mathbf{N}$,

$$\mathbf{x}^{(p-1)(q-1)} \bmod \mathbf{pq} = 1)$$

Fermat's little theorem

$$= \mathbf{x}$$

Is RSA secure?

Best known way to compute d from e
is factoring N into p and q .

Best known factoring algorithm:

General number field sieve

Takes more than polynomial time
but less than exponential time
to factor n -bit number.

(Still takes way too long if p, q
are large enough and random.)

Fingers crossed...

but can't rule out a breakthrough!

To generate an RSA keypair:

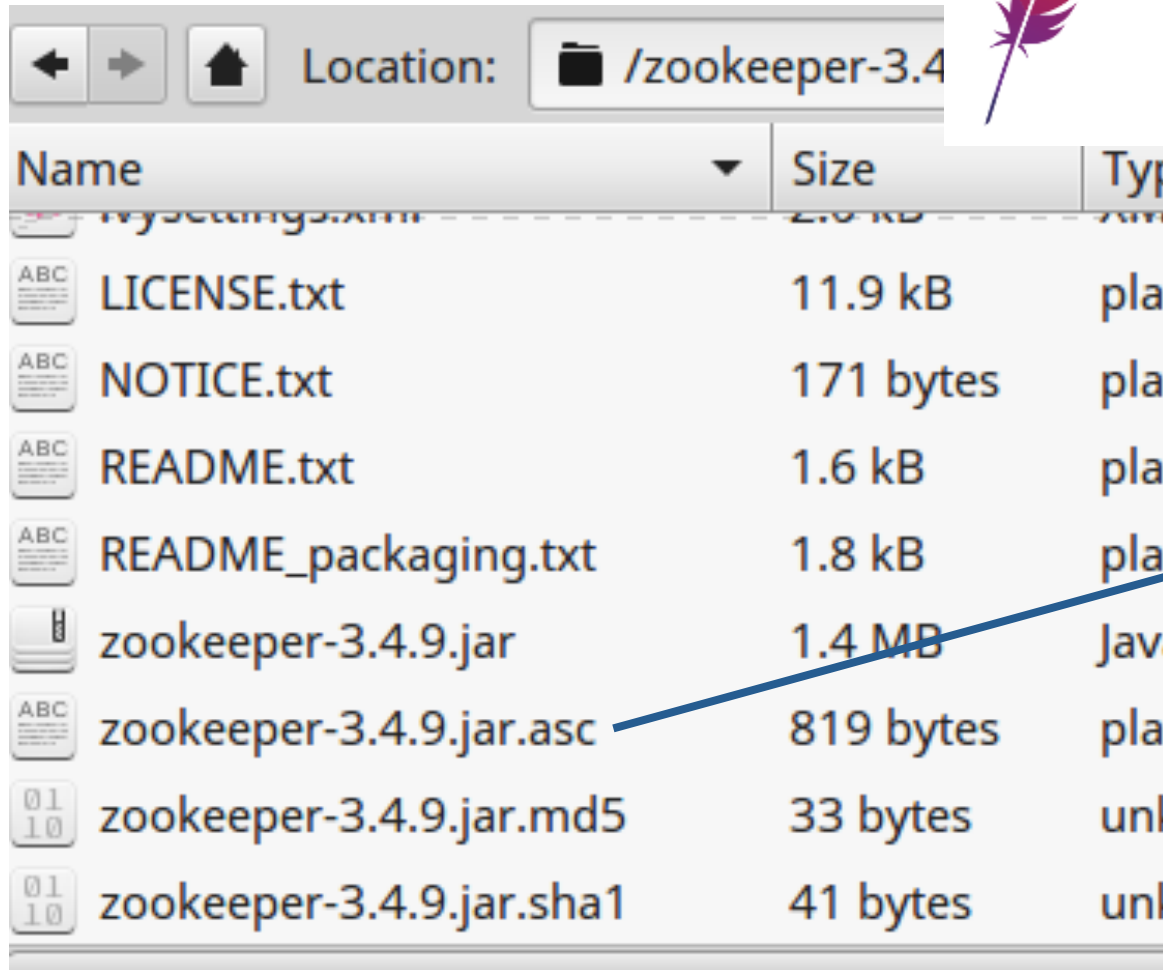
```
$ openssl genrsa -out private.pem 1024  
$ openssl rsa -pubout -in private.pem > public.pem
```

To sign a message with RSA:

```
$ openssl rsautl -sign -inkey private.pem -in a.txt > sig
```

To verify a signed message with RSA:

```
$ openssl rsautl -verify -pubin -inkey public.pem -in sig
```



Name	Size	Type
ABC LICENSE.txt	11.9 kB	plain text
ABC NOTICE.txt	171 bytes	plain text
ABC README.txt	1.6 kB	plain text
ABC README_packaging.txt	1.8 kB	plain text
zookeeper-3.4.9.jar	1.4 MB	Java archive
ABC zookeeper-3.4.9.jar.asc	819 bytes	plain text
01 10 zookeeper-3.4.9.jar.md5	33 bytes	unknown
01 10 zookeeper-3.4.9.jar.sha1	41 bytes	unknown

Public key digital
signatures on
hashes of code
releases

“Pretty Good Privacy”
tool

- alternate command line

HOW TO USE PGP TO VERIFY THAT AN EMAIL IS AUTHENTIC:

LOOK FOR THIS
TEXT AT THE TOP.



IF IT'S THERE, THE EMAIL IS PROBABLY FINE.

If you want to be extra safe, check that there's a big block of jumbled characters at the bottom.

Subtle fact: RSA can be used for either confidentiality or integrity

RSA for confidentiality:

Encrypt with public key, Decrypt with private key

Public key is (e, N)

Private key is (d, N)

To encrypt: $E(x) = x^e \bmod N$

To decrypt: $D(x) = x^d \bmod N$

RSA for integrity:

Encrypt (“sign”) with private key

Decrypt (“verify”) with public key

RSA drawback: Performance

Factor of 1000 or more slower than AES.

Dominated by exponentiation – cost goes up (roughly) as cube of key size.

Message must be shorter than **N**.

Use in practice:

Hybrid Encryption (similar to key exchange):

Use RSA to encrypt a random key $k < N$, then use AES

Signing:

Compute $v := \text{hash}(m)$, use RSA to sign the hash

Should always use crypto libraries to get details right

What can go wrong with RSA?

Twenty Years of Attacks on the RSA Cryptosystem

Dan Boneh
dabo@cs.stanford.edu

Hundreds of things!!

Many have a common theme: tweaking the protocol for efficiency (e.g., small exponents) leads to a compromise.

One example of a failure: Common P's and Q's

Individually, $N = pq$ is very hard to factor.

Turns out, due to poor entropy, many pairs of RSA keys are generated with same p

$$N_1 = p q_1$$

$$N_2 = p q_2$$

Given two products with a common factor, easy to compute $\text{GCD}(N_1, N_2)$ with Euclid's algorithm.

Key Management

The hard part of crypto: **Key-management**

Principles:

0. Always remember, key management is the hard part!

1. Each key should have only one purpose
(in general, no guarantees when keys reused elsewhere)

1. Vulnerability of a key increases:

- a. The more you use it.
- b. The more places you store it.
- c. The longer you have it.

2. Keep your keys far from the attacker.

3. Protect yourself against compromise of old keys.

Goal: **forward secrecy** — learning old key shouldn't help adversary learn new key.

[How can we get this?]

Building a **secure channel**

What if you want confidentiality and integrity at the same time?

Encrypt, then MAC

not the other way around

Use separate keys for confidentiality and integrity.

Need two shared keys,
but only have one?
That's what PRGs are for!

If there's a reverse (Bob to Alice) channel, use separate keys for that too

Issue: How big should keys be?

Want prob. of guessing to be infinitesimal... but watch out for
Moore's law – safe size gets 1 bit larger every 18 months
128 bits usually safe for ciphers/PRGs

Need larger values for MACs/PRFs
due to **birthday attack**

Often trouble if adversary can find
any two messages with same MAC

Attack: Generate random values,
look for coincidence.

Requires $O(2^{|k|/2})$ time, $O(2^{|k|/2})$ space.

For 128-bit output, takes 2^{64} steps: doable!

Upshot: Want output of MACs/PRFs to be twice as big
as cipher keys e.g. use HMAC-SHA256 alongside AES-128

Key Type <i>Move the cursor over a type for description</i>	Cryptoperiod	
	Originator Usage Period (OUP)	Recipient Usage Period
Private Signature Key	1-3 years	-
Public Signature Key	Several years (depends on key size)	
Symmetric Authentication Key	≤ 2 years	$\leq \text{OUP} + 3$ years
Private Authentication Key		1-2 years
Public Authentication Key		1-2 years
Symmetric Data Encryption Key	≤ 2 years	$\leq \text{OUP} + 3$ years
Symmetric Key Wrapping Key	≤ 2 years	$\leq \text{OUP} + 3$ years
Symmetric RBG keys	Determined by design	-
Symmetric Master Key	About 1 year	-
Private Key Transport Key		≤ 2 years ⁽¹⁾
Public Key Transport Key		1-2 years
Symmetric Key Agreement Key		1-2 years ⁽²⁾
Private Static Key Agreement Key		1-2 years ⁽³⁾
Public Static Key Agreement Key		1-2 years
Private Ephemeral Key Agreement Key		One key agreement transaction
Public Ephemeral Key Agreement Key		One key agreement transaction
Symmetric Authorization Key		≤ 2 years
Private Authorization Key		≤ 2 years
Public Authorization Key		≤ 2 years

Date	Minimum of Strength	Symmetric Algorithms	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash (A)	Hash (B)
(Legacy)	80	2TDEA*	1024	160	1024	160	SHA-1**	
2016 - 2030	112	3TDEA	2048	224	2048	224	SHA-224 SHA-512/224 SHA3-224	
2016 - 2030 & beyond	128	AES-128	3072	256	3072	256	SHA-256 SHA-512/256 SHA3-256	SHA-1
2016 - 2030 & beyond	192	AES-192	7680	384	7680	384	SHA-384 SHA3-384	SHA-224 SHA-512/224
2016 - 2030 & beyond	256	AES-256	15360	512	15360	512	SHA-512 SHA3-512	SHA-256 SHA-512/256 SHA-384 SHA-512 SHA3-512

Attacks against Crypto

1. Brute force: trying all possible private keys
2. Mathematical attacks: factoring
3. Timing attacks: using the running time of decryption
4. Hardware-based fault attack: induce faults in hardware to generate digital signatures
5. Chosen ciphertext attack
6. Architectural Changes

So Far:

Message Integrity

Confidentiality

Key Exchange

Public Key Crypto

Next Week:

HTTPS and TLS: Secure channels for the web

Bitcoin and Cryptocurrencies

Post Quantum:

When will a quantum computer be built?

15 years, \$1 billion USD, nuclear power plant (PQCrypto 2014, Matteo Mariantoni)

What will be impacted?

Public key crypto:

~~RSA~~

~~-Elliptic Curve Cryptography (ECDSA)~~

~~-Finite Field Cryptography (DSA)~~

~~-Diffie-Hellman key exchange~~

Symmetric key crypto:

AES, Triple DES



Need Larger Keys

Hash functions:

~~-SHA-1~~, SHA-2 and SHA-3



Use longer output