Lecture 20 – Public key Crypto

Stephen Checkoway
University of Illinois at Chicago
CS 487 – Fall 2017
Slides from Miller and Bailey's ECE 422

Review: Integrity

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

Provably-secure solution: **Random function**

Practical solution:

e.g. "Attack at dawn", 628369867…

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**)

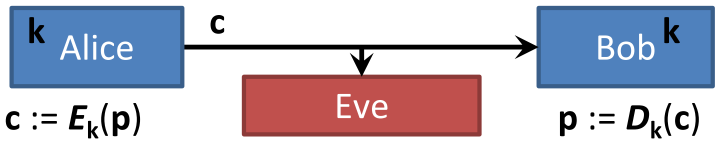
Review: Confidentiality

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**

Common theme: Key

Requirements

- Must be known by both Alice and Bob
- Must be unknown by anyone else
- Must be infeasible to guess

We’d like Alice and Bob to agree on a key that satisfies those properties by sending public messages to each other

Key Exchange

Issue: How do we get a shared key?

Amazing fact:

Alice and Bob can have a <u>public</u> conversation to derive a shared key!

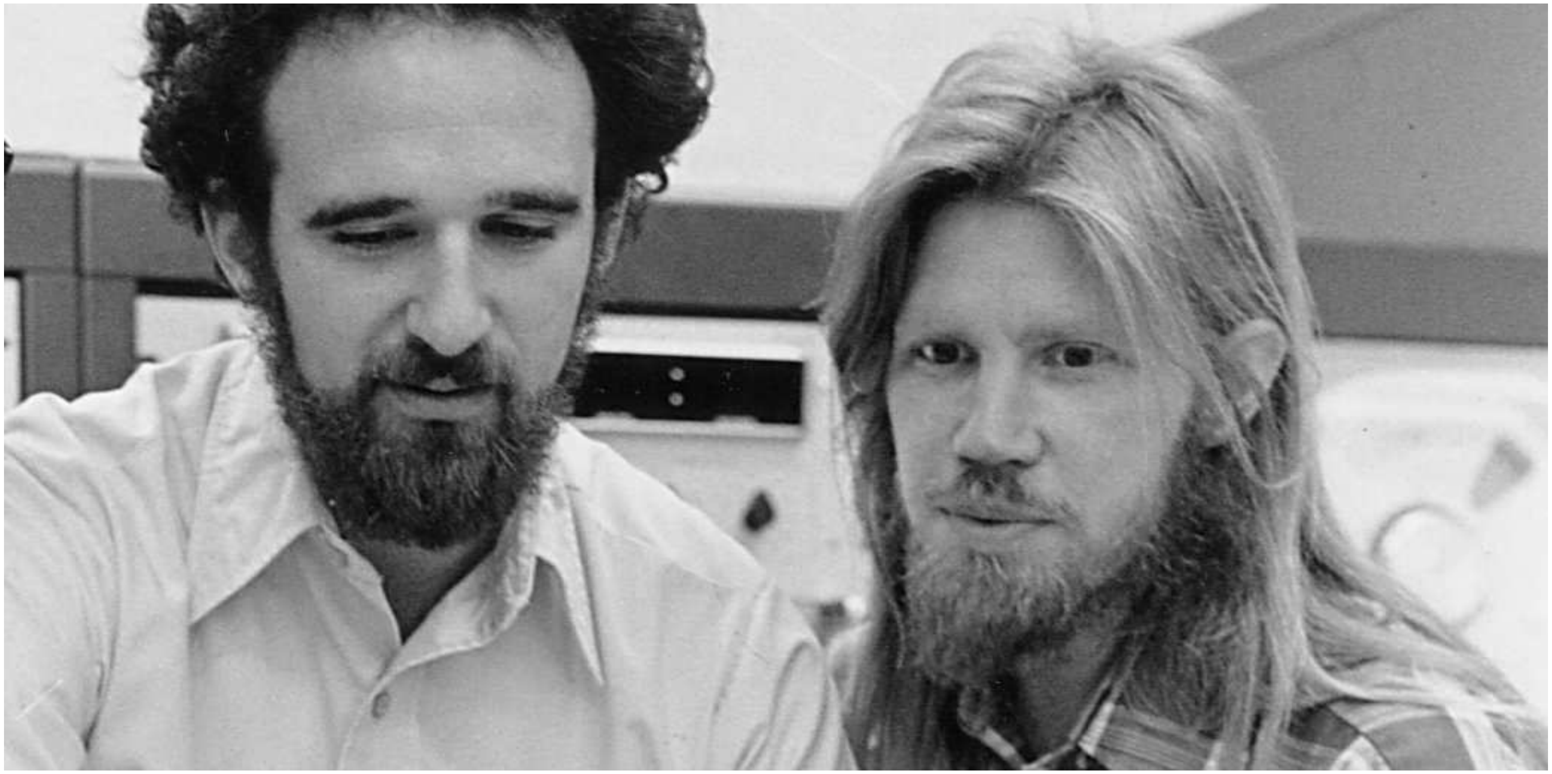
Diffie-Hellman (**D-H**) **key exchange**

1976: Whit Diffie, Marty Hellman, improving partial solution from Ralph Merkle

(earlier, in secret, by Malcolm Williamson of UK’s GCHQ)

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 $\mathbf{g}^{x}$ for some exponent x.

$$\mathbf{g}^{\mathbf{x}}$$

Exponent, $0 <= \mathbf{x} < (p - 1)/2$

Generator, an element of the group

What is a Group?

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

Definition: A group (**G**,*) is a set of elements **G**, and a binary operation *

- *(Closed)*: for any **x, y** $\in$ **G**, we know **x*y** $\in$ **G**
- *(Identity)*: we know the identity e in **G**
  for any **x** $\in$ **G**, we have **e*x** = **x** = **x*e**
- *(Inverses)*: for any **x**, we can compute $\mathbf{x}^{-1}$***x** = **e**
- *(Associative)*: For **x**, **y**, **z** $\in$ **G**, **x***(**y*z**) = (**x*y**)***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 $\mathbf{g_0}$ in the range 2 to (**p** - 1)

3. Let $\mathbf{g} = (\mathbf{g_0})^2$ mod **p.** If **g** = 1, goto step 2

    This is the “generator” of the group.

- A number **x** > 0 is in the group if $\mathbf{x}^2 \neq 1$ mod **p**
- The order of each element is (p - 1) / 2.

    $\mathbf{g}^{(p-1)/2} = 1$ mod **p**
- We can compute inverses $\mathbf{x}^{-1}$ s.t. $\mathbf{x}^{-1}\mathbf{x} = 1$ mod **p**

Problems assumed “hard” in Schnorr groups:

- Discrete logarithm problem

Given $\mathbf{g}^{\mathbf{x}}$ for some random $\mathbf{x}$, find $\mathbf{x}$

- Diffie Hellman problem (computational)

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

- Diffie Hellman problem (decisional)

Flip a bit *c*, generate random exponents $\mathbf{a}$,$\mathbf{b}$,$\mathbf{r}$

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

Guess *c*

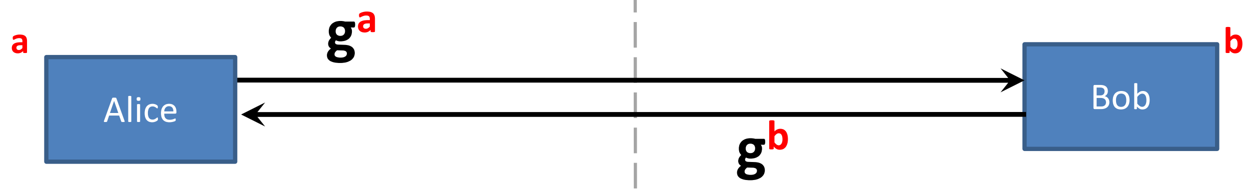
*These problems are thought to be hard in other groups too,
e.g. some Elliptic Curves

Diffie-Hellman protocol

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

Alice
Generates random
secret exponent a.

Bob
Generates random
secret value b.

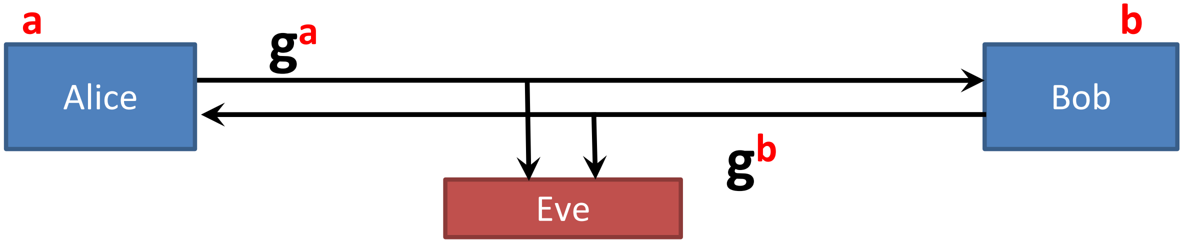


Computes **x**
$= (g^b)^a$
$= g^{ba}$

Computes **x′**
$= (g^a)^b$
$= g^{ab}$

(Notice that **x** = **x′**)
Can use **k** = hash(**x**) as a shared key.

Passive eavesdropping attack



Eve knows: $\mathbf{g}$, $\mathbf{g}^{a}$, $\mathbf{g}^{b}$

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

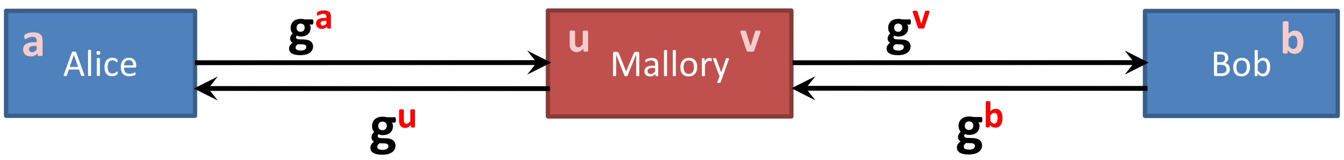
Best known approach:

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 $\mathbf{g}^{au}$

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

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 later.]

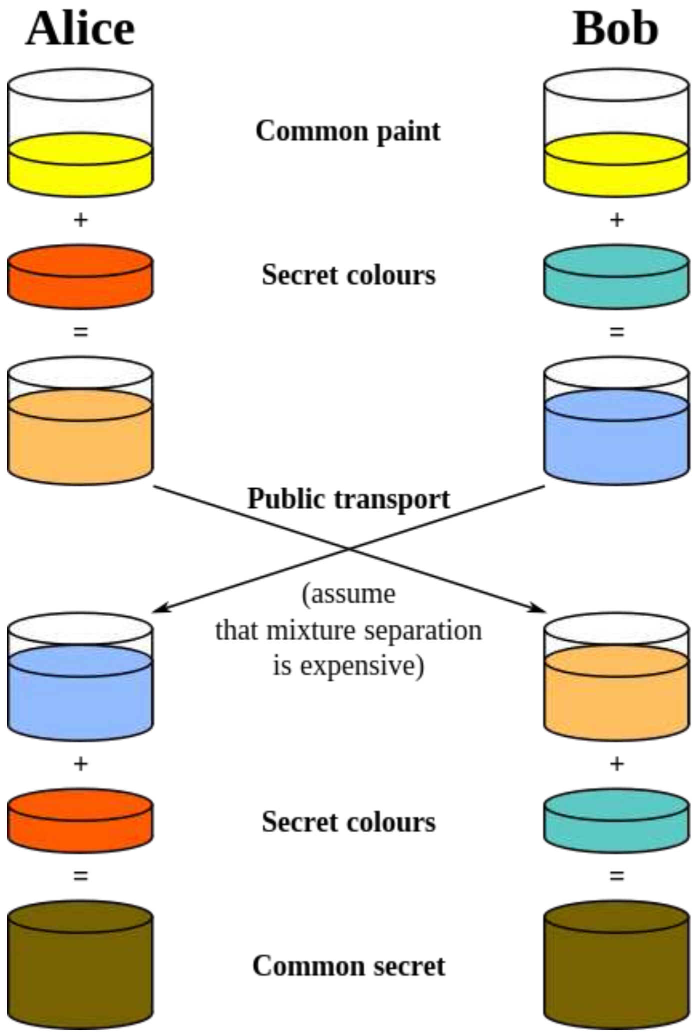
A visual analogy:

"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, $\mathbf{k_{pub}}$ and private key, $\mathbf{k_{priv}}$

- ***Encrypt:*** Anyone can encrypt the message M, resulting in ciphertext C = **Enc**($\mathbf{k_{pub}}$, M)

- ***Decrypt:*** Only Bob has the private key needed to decrypt the ciphertext: M=**Dec**($\mathbf{k_{priv}}$, C)

- ***Security***: Infeasible to guess M or $\mathbf{k_{priv}}$, even knowing $\mathbf{k_{pub}}$and seeing ciphertexts

Public Key Encryption w/ ephemeral key exchange

Key generation:

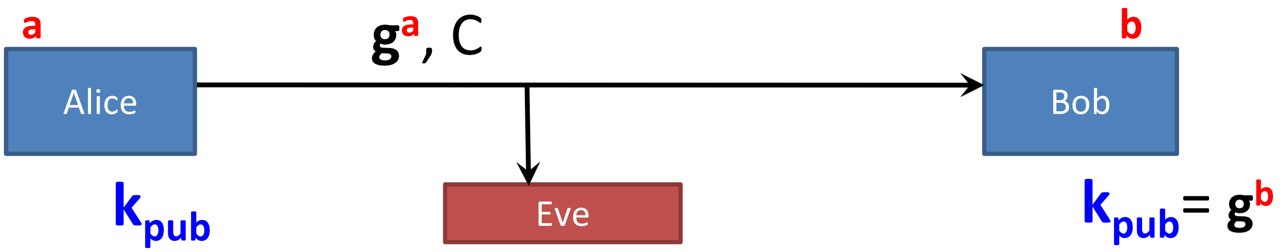
k_{priv} := b generated randomly, and k_{pub} := g^b

Encrypt(M):

Generate random a, set k := hash($k_{pub}{}^a$), encrypt C = AES-enc(k, M)
Send (g^a, C) as ciphertext

Decrypt(g^a, C):

compute k = hash($(g^a)^b$),
decrypt M = AES-dec(k, C)



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, $\mathbf{k_{pub}}$ and private key, $\mathbf{k_{priv}}$

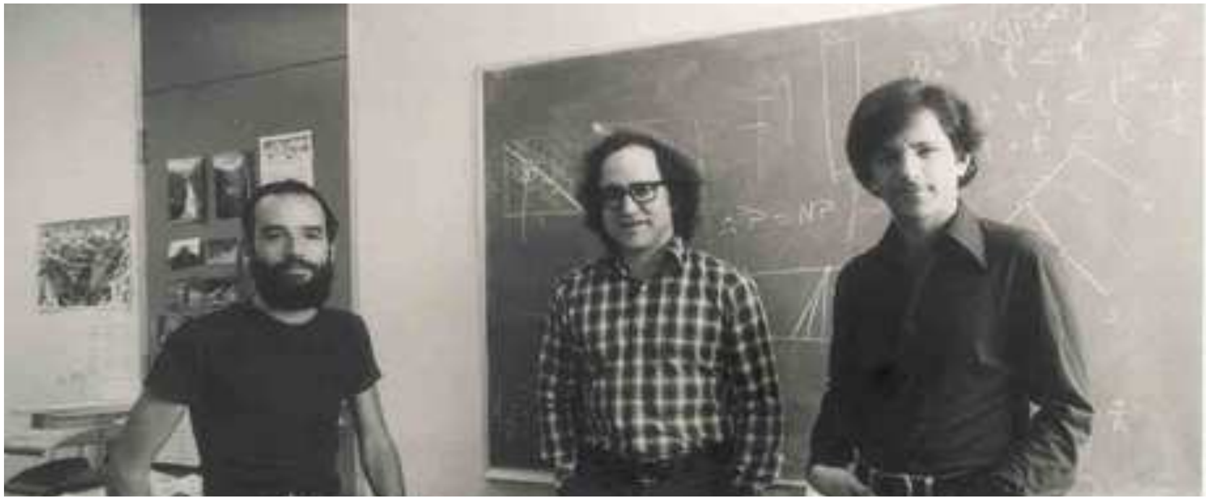
- Bob can sign a message M, resulting in signature
  S = Sign($\mathbf{k_{priv}}$, M)

- Anyone who knows $\mathbf{k_{pub}}$ can check the signature:
  Verify($\mathbf{k_{pub}}$, M, S) $\overset{?}{=}$ 1

- "Unforgeable": Computationally infeasible to guess S or $\mathbf{k_{priv}}$, even knowing $\mathbf{k_{pub}}$ 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 UK's GCHQ, in secret)

How RSA signatures work

Key generation:

1. Pick large (say, 2048 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 = ***Sign***(**x**) = $\mathbf{x}^{\mathbf{d}}$ mod **N**
To verify: ***Verif***(S) = $S^{\mathbf{e}}$ mod **N** Check ***Verif***(S) $\overset{?}{=}$ M

Why RSA works

“Completeness” theorem:

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

Proof:

Verif(***Sign***(**x**))

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

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

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

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

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

$= 1^{\mathbf{a}}\mathbf{x} \bmod \mathbf{pq}$ (by Euler’s theorem, $\mathbf{x}^{(\mathbf{p}-1)(\mathbf{q}-1)} \bmod \mathbf{pq} = 1$)

$= \mathbf{x}$

Is RSA secure?

<u>Best known</u> way to compute **d** from **e** is factoring **N** into **p** and **q**.

<u>Best known</u> 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:

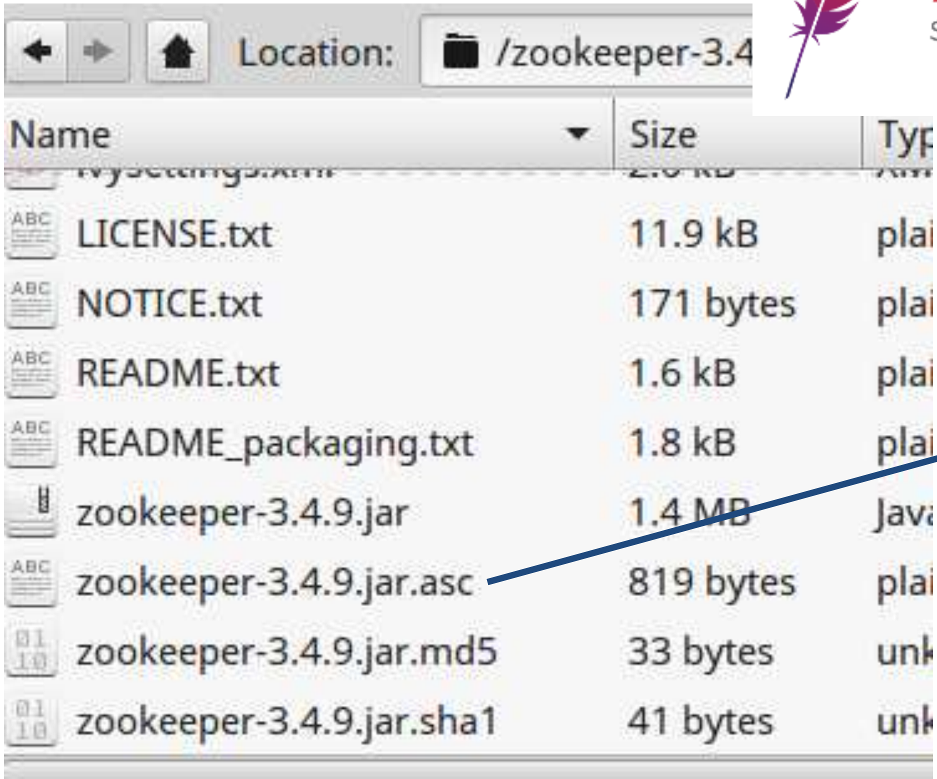
```
$ openssl genrsa -out private.pem 1024
$ openssl rsa -pubout -in private.pem > public.pem
```

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
```

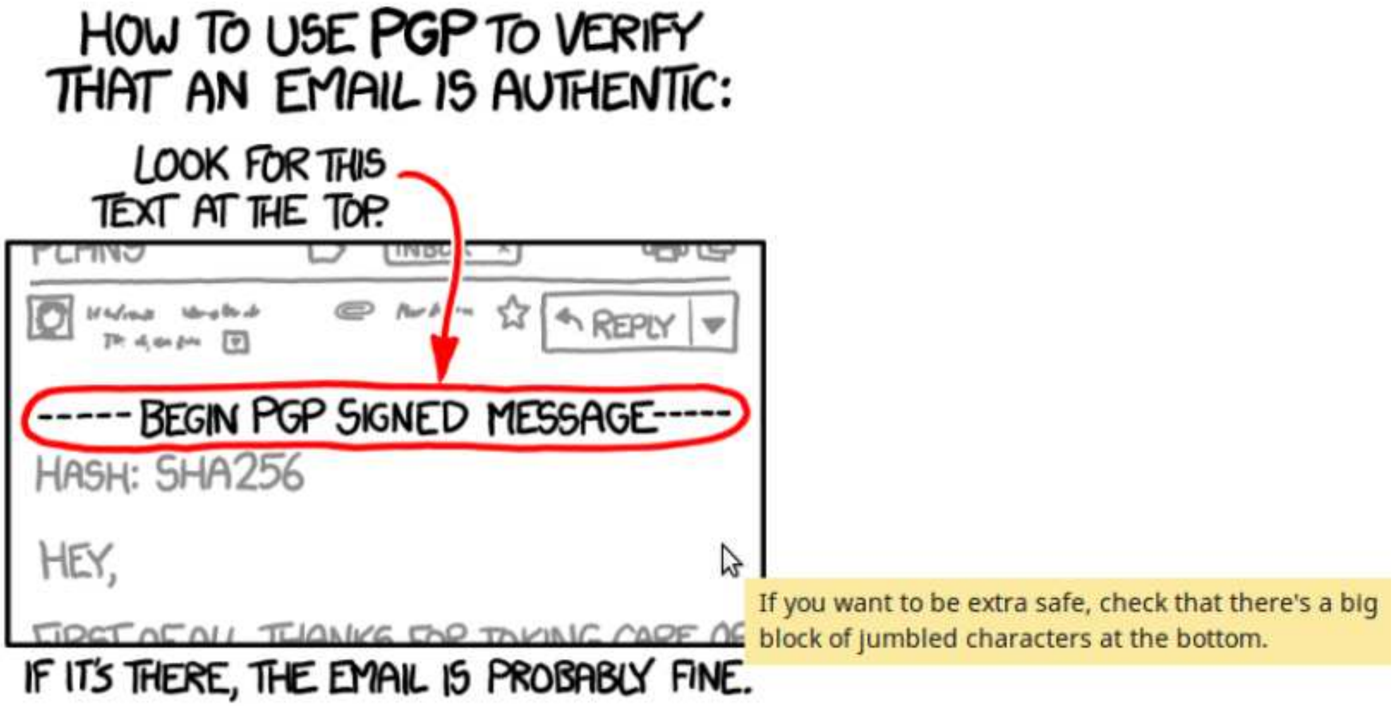


Location: /zookeeper-3.4

Name	Size	Typ
LICENSE.txt	11.9 kB	plai
NOTICE.txt	171 bytes	plai
README.txt	1.6 kB	plai
README_packaging.txt	1.8 kB	plai
zookeeper-3.4.9.jar	1.4 MB	Java
zookeeper-3.4.9.jar.asc	819 bytes	plai
zookeeper-3.4.9.jar.md5	33 bytes	unk
zookeeper-3.4.9.jar.sha1	41 bytes	unk

Public key digital signatures on hashes of code releases

“Pretty Good Privacy”
- alternate command line tool



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

https://xkcd.com/1181/

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: $E(\mathbf{x}) = \mathbf{x}^{\mathbf{e}} \bmod \mathbf{N}$

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

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** := hash(**m**), use RSA to sign the hash

Should always use crypto libraries to get details right

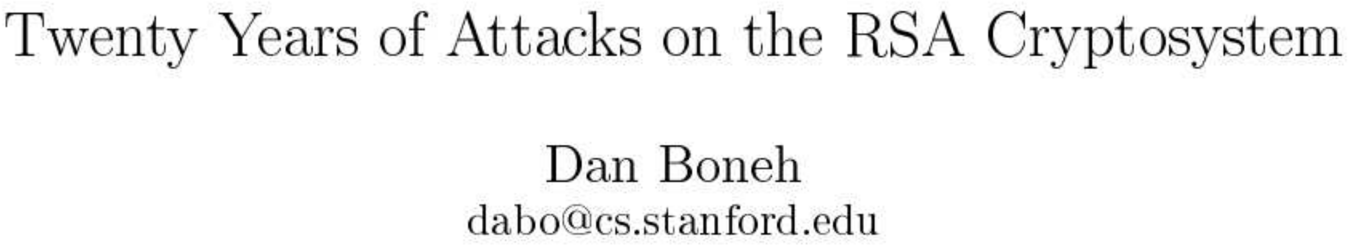
The reality is more complicated

Can’t just compute m^e mod N (what if we know $m < N^{1/e}$?)

Need to pad the message

- Some schemes are good (PSS, OAEP)
- Some schemes are bad (PKCS#1v1.5)
- Different for signatures and encryption

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

$$N_1 = pq_1$$
$$N_2 = pq_2$$

Given two products with a common factor, easy to compute $GCD(N_1, N_2) = p$ 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

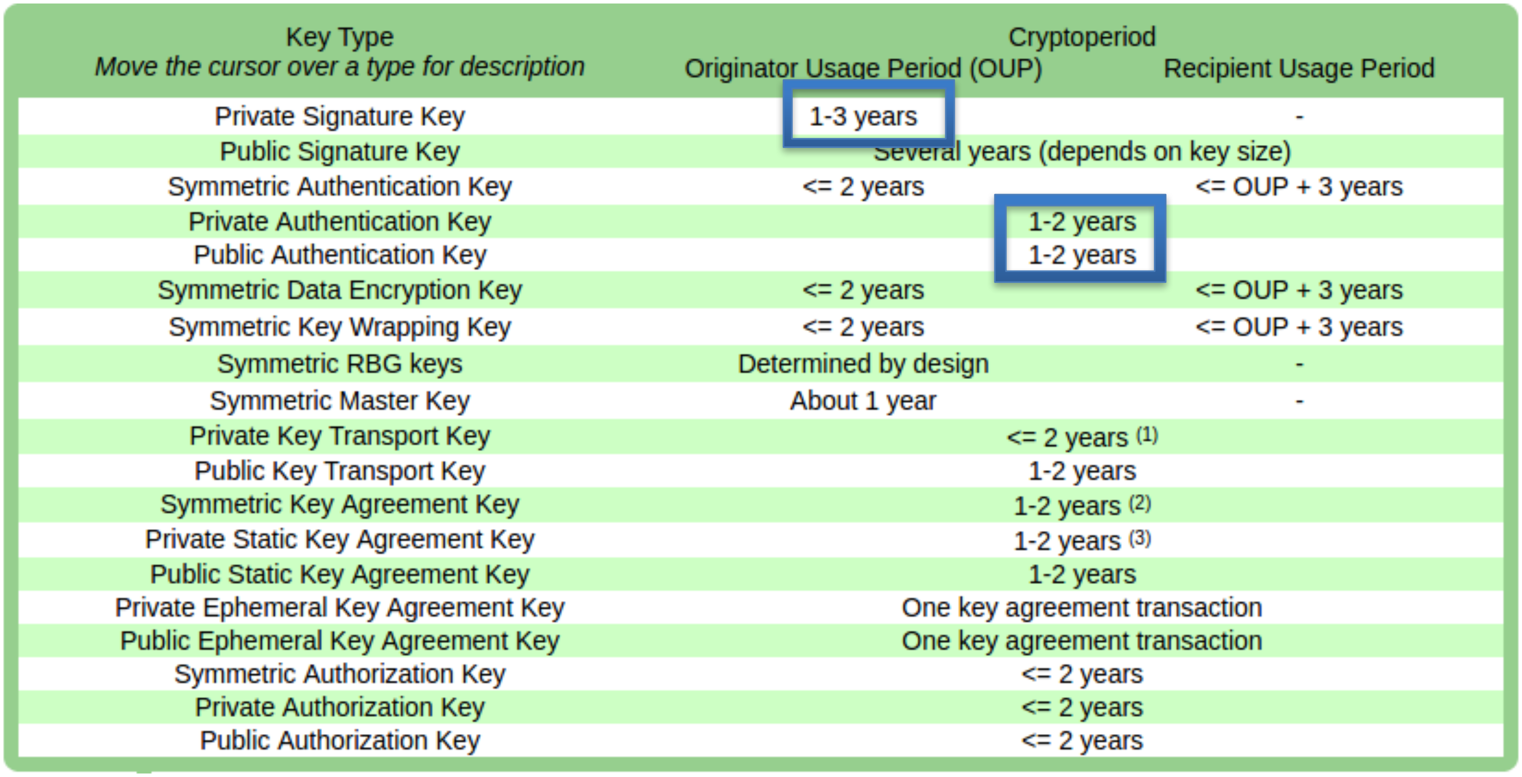
128 bits usually safe for ciphers/PRGs

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

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

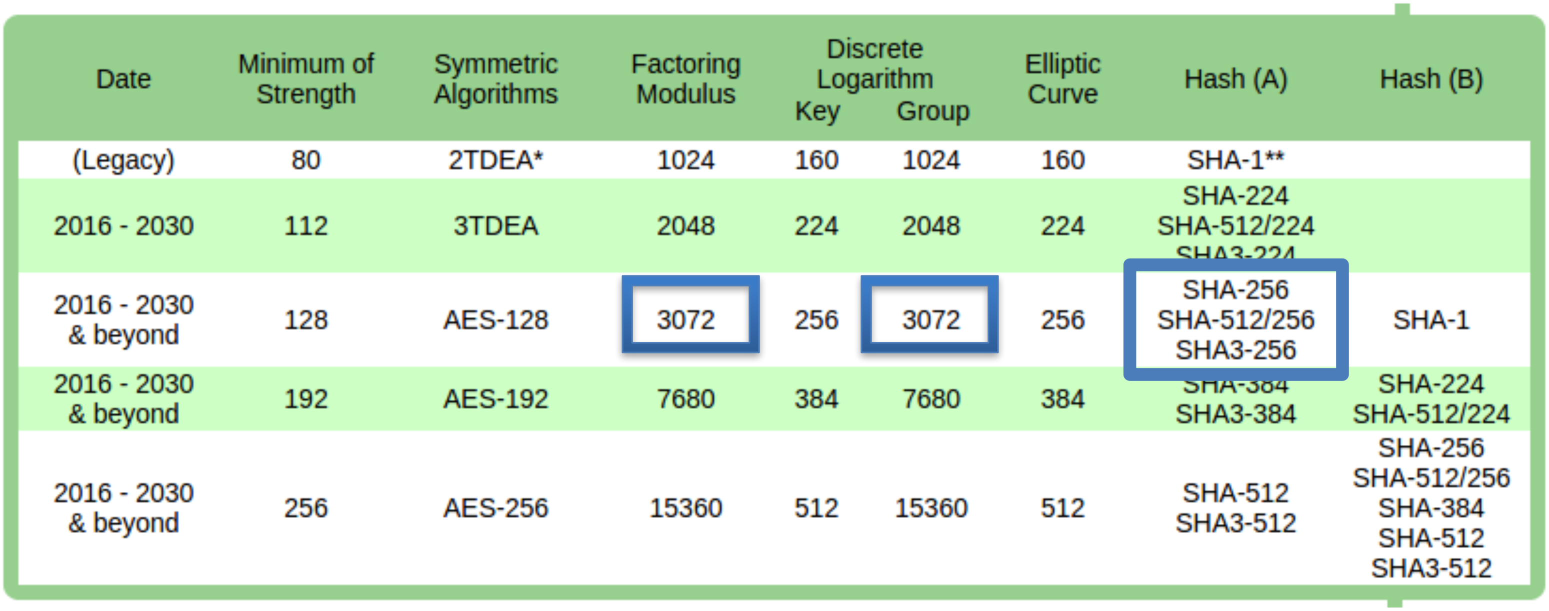
Attack: Generate random values, look for coincidence.
Requires $O(2^{|k|/2})$ time, $O(2^{|k|/2})$ space.
For 128-bit output, takes 2^{64} steps: doable!

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



Key Type *Move the cursor over a type for description*	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	<= OUP + 3 years
Private Authentication Key	1-2 years	
Public Authentication Key	1-2 years	
Symmetric Data Encryption Key	<= 2 years	<= OUP + 3 years
Symmetric Key Wrapping Key	<= 2 years	<= 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	

https://www.keylength.com/en/4/



Date	Minimum of Strength	Symmetric Algorithms	Factoring Modulus	Discrete Logarithm		Elliptic Curve	Hash (A)	Hash (B)
				Key	Group			
(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

https://www.keylength.com/en/4/

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

Quantum Computers:

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

So Far:

Message Integrity

Confidentiality

Key Exchange

Public Key Crypto

Next:

HTTPS and TLS: Secure channels for the web