Lecture 07 – Public key Crypto

Ryan Cunningham
University of Illinois
ECE 422/CS 461 – Fall 2017

Correction! (I lied)

To encrypt: $\mathbf{c_i} = \mathbf{p_i}$ xor $\mathbf{k_i}$

To decrypt: $\mathbf{p_i} = \mathbf{c_i}$ xor $\mathbf{k_i}$

"one-time" means you should <u>never</u> reuse any part of the pad.

If you do:

Let $\mathbf{k_i}$ be pad bit

Adversary learns (**a** xor $\mathbf{k_i}$) and (**b** xor $\mathbf{k_i}$)

Adversary xors those to get (**a** xor **b**),

which is useful [How?]

- a xor b has statistical properties of plaintext!

04154a1813536d004a021d511a4546001607100600360c16520843120142071d4d250a165e460c0f523e01154c09413941054826
3a454107004515480a4647004c0b0e011000070a0545571f0e5346180a574c1d0d450c4c4f26114506491141020c124f522c1600
15185362084c0016424718010842161 04f1418521a001c58000d492c0a44590e0c1248492f0a49520f11040b441f014c020a0545
4e01594c0041184b410a06080c1d47411c1147100a114e0d0b00491b0c0a5500020d53174f05181c451b45180b4f1f480e124646
07050619160d52051a444606004d410f174f0f0c52041c114c02411b0345371d471b001d444d522a13151b4e2152120100 0c4d4c
08530b4b57682245130917014 3071d020c54111a52190a154e041d44460404054441034f0603450c4e06410702040002140 14356
1611430403444c0d1a1f0945031c4d040044571b0745181952081705470 50c0354174f111c414d681a1c4d16500a054915041159
501b001d45124c070f0846025e451603014e4645740 5041d45440844471b0701000709040b4746000754571d0343050a1000261f
0a05000d064e410c164342311a160008015 6111a0b4d021c1b551e05454441060d4d541f0a1f09001603134e000545141545155 5
0618450e410c0d57001e02124c120e00020a490a4910480b4f124f151f0e03474f3a464574050a52034f1301071505090f000e09
45134906410d04034115161c574356111a45414d2817174d0c014e0610540d481148061b5612131d4d0c0054460b001b00534649
394445410c0b4d0a005200500a1e114e0012450611451917 4e0c164d0a45420510440a481b52164d520b580a1c540b0157414c0c
071641171a4c061b10520800140b450001164d411a071c0d1c001c0e5214020d41180e04000e0c52011a4800090e004815000a00
55170d4b4d6d08041b1d002e1806061c430149050018481606451203520 80a0a5416001a0d1d06540c52041d10000d1d461a0a10
00131a0a570a0109540005080 41e1541474e3b010018070b47421205021d0001000a10451a010b47034c090b45014b4d35010b45
074f071a050601170153411901520149175 4011d11444d0b0b0554041852070b53014d430a1d4a003c0d455337171303064e1c55
164503095742 0d054441194906090655 0f1d14104b440a10490d0a1c4e4f0356111a1c4916411657045953 03481d0c484d160e17
4543091d1a06491d54110d0d5454010a00534d483b1a000a00021c10084454060a1900160a00171a0d0e0e4d2717454908040414
19000a541a4e461e0e0344541f0c53061e1c570c0b0a0012010b4e501b001d451d41546212064f03010b15454d18171745411f12
08000a4141540e04190016010c4e0b110c45334d0813074308005 3541c0d4f16081d4e15174a003d0a0402480d09520748164f17
080516451a1a4d570a0711000f015242000 0561b061541061c0e41231a4501552418040d17160c540e0007470049002f04164d0c
0f4403490a1d170d521e4f0d041445411b140016061249 1a44451f04020a520056121b0a1146494c265341170100161912 5e5911
00440a48001f4c47160a445204125042475746457 70b0a1b4e070d0945064f020c0052124e034f154c0e4d430a4e104f45020d17
101d4e0741000445445848476 11e110b4b4e002e1806061c442813194f0542014546151503001c0011131c4d411a480a55061213
0418450f4e0748120519191f0b444f044106084f1a004d0d480c4e7509060200164e3e110f004953501d09562809001649 0a150c
441e0c451c0a5446080d000a000f00014557121b11484f040d174107080d15450a49031d005941010200030e0f49520a04160315
4e0003491e0659540e04520e1c1e1a4e0516521e59431b45111252151c0c1747431b1d094444060b1345140007534 10f034e000d

BBVKGBNTbQBKAh1RGkVGABYHEAYANgwWUghDEgFCBx1NJQoWXkYMD1I+ARVMCUE5QQVIJjpFQQcAR
RVICkZHAEwLDgEQAAcKBUVXHw5TRhgKV0wdDUUMTE8mEUUGSRFBAgwST1IsFgAVGFNiCEwAFkJHGA
EIQhYQTxQYUhoAHFgADUksCkRZDgwSSEkvCklSDxEEC0QfAUwCCgVFTgFZTABBGEtBCgYIDB1HQRw
RRxAKEU4NCwBJGwwKVQACDVMXTwUYHEUbRRgLTx9IDhJGRgcFBhkWDVIFGkRGBgBNQQ8XTw8MUgQc
EUwCQRsDRTcdRxsAHURNUioTFRtOIVISAQAMTUwIUwtLV2giRRMJFwFDBx0CDFQRGlIZChVOBB1ER
gQEBURBA08GA0UMTgZBBwIEAAIUAUNWFhFDBANETA0aHwlFAxxNBABEVxsHRRgZUggXBUcFDANUF0
8RHEFNaBocTRZQCgVJFQQRWVAbAB1FEkwHDwhGAl5FFgMBTkZFdAUEHUVECERHGwcBAAcJBAtHRgA
HVFcdA0MFChAAJh8KBQANBk5BDBZDQjEaFgAIAVYRGgtNAhwbVR4FRURBBg1NVB8KHwkAFgMTTgAF
RRQVRRVVBhhFDkEMDVcAHgISTBIOAAIKSQpJEEgLTxJPFR8OA0dPOkZFdAUKUgNPEwEHFQUJDwAOC
UUTSQZBDQQDQRUWHFdDVhEaRUFNKBcXTQwBTgYQVA1IEUgGG1YSEx1NDABURgsAGwBTRkk5REVBDA
tNCgBSAFAKHhFOABJFBhFFGRdODBZNCkVCBRBECkgbUhZNUgtYChxUCwFXQUwMBxZBFxpMBhsQUgg
AFAtFAAEWTUEaBxwNHAAcDlIUAg1BGA4EAA4MUgEaSAAJDgBIFQAKAFUXDUtNbQgEGx0ALhgGBhxD
AUkFABhIFgZFEgNSCAoKVBYAGg0dBlQMUgQdEAANHUYaChAAExoKVwoBCVQABQgEHhVBR047AQAYB
wtHQhIFAh0AAQAKEEUaAQtHA0wJC0UBS001AQtFB08HGgUGARcBU0EZAVIBSRdUAR0RRE0LCwVUBB
hSBwtTAU1DCh1KADwNRVM3FxMDBk4cVRZFAwlXQg0FREEZSQYJBlUPHRQQS0QKEEkNChxOTwNWERo
cSRZBFlcEWVMDSB0MSE0WDhdFQwkdGgZJHVQRDQ1UVAEKAFNNSDsAAoAAhwQCERUBgoZABYKABca
DQ4OTScXRUkIBAQUGQAKVBpORh4OA0RUHwxTBh4cVwwLCgASAQtOUBsAHUUdQVRiEgZPAwELFUVNG
BcXRUEfEggACkFBVA4EGQAWAQxOCxEMRTNNCBMHQwgAU1QcDU8WCB1OFRdKAD0KBAJIDQlSB0gWTx
cIBRZFGhpNVwoHEQAPAVJCAABWGwYVQQYcDkEjGkUBVSQYBA0XFgxUDgAHRwBJAC8EFk0MD0QDSQo
dFw1SHk8NBBRFQRsUABYGEkkaREUfBAIKUgBWEhsKEUZJTCZTQRcBABYZEl5ZEQBECkgAH0xHFgpE
UgQSUEJHV0ZFdwsKG04HDQlFBk8CDABSEk4DTxVMDk1DCk4QT0UCDRcQHU4HQQAERURYSEdhHhELS
04ALhgGBhxEKBMZTwVCAUVGFRUDABwAERMcTUEaSApVBhITBBhFD04HSBIFGRkfC0RPBEEGCE8aAE
0NSAxOdQkGAgAWTj4RDwBJU1AdCVYoCQAWSQoVDEQeDEUcClRGCA0ACgAPAAFFVxIbEUhPBA0XQQc
IDRVFCkkDHQBZQQECAAMOD0lSCgQWAxVOAANJHgZZVA4EUg4cHhpOBRZSHllDG0URElIVHAwXR0Mb
HQlERAYLE0UUAAdTQQ8DTgANC1QSTA8PGAlBTR0GAAtBHAFBRCEMEAdIHQYGGFlCT0kbThJBFkweB
EkZDRdUEgtNChwAEQcdAAoGEAAEGgYXVRQGHAwOQToRRUEBDw8JFABMElRXBB4JDlQgAE5XEhtJEV
AVFk8BCwASHgFDAA8IDAMRHUQMAFRSCAwBVBJIDlISAAAOTzkETwQEFwAAB0UcHEUPBR8GFwgWHVR
UBk4DAAZBGgsATUEHClQBRQINFwAPFxwcGgVYRRBPVAdJEw1JGg1JTVcLBQgGAABAWVtIFgpEGA1F
GE4TTxAUEE8cEVYDHVJIFAFUTRQNEUIOF1lIAFYGHVUNA0EOC1o=

Security News

- Krebs reports terrible security in Equifax Argentina
- BlueBorne - Bluetooth vulnerability in smartphones (responsibly disclosed + patch)
- NIST Report on Lightweight Cryptography (not actually new)

NISTIR 8114

Report on Lightweight Cryptography

Kerry A. McKay
Larry Bassham
Meltem Sönmez Turan
Nicky Mouha
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8114

March 2017

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

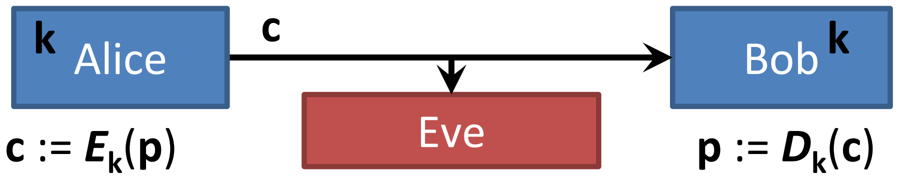
Popular example: **HMAC-SHA256$_k$(m)**

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**

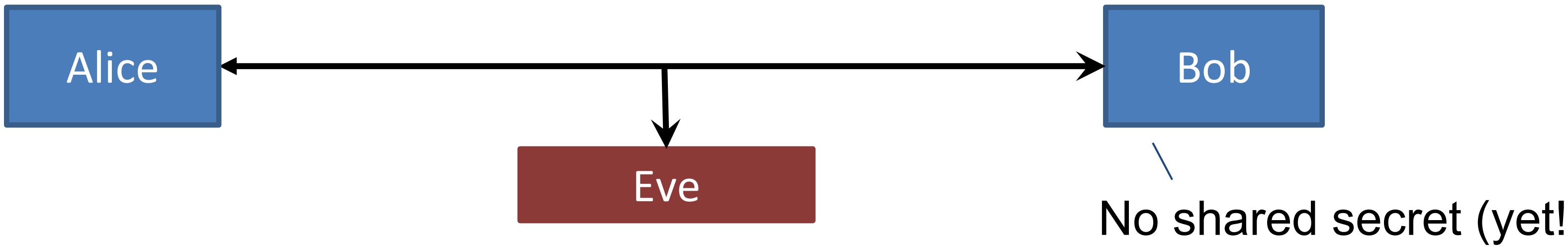
Real-world use: **Stream ciphers** (can't reuse **k**)

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

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

Key Exchange

Issue: How do we get a shared key?



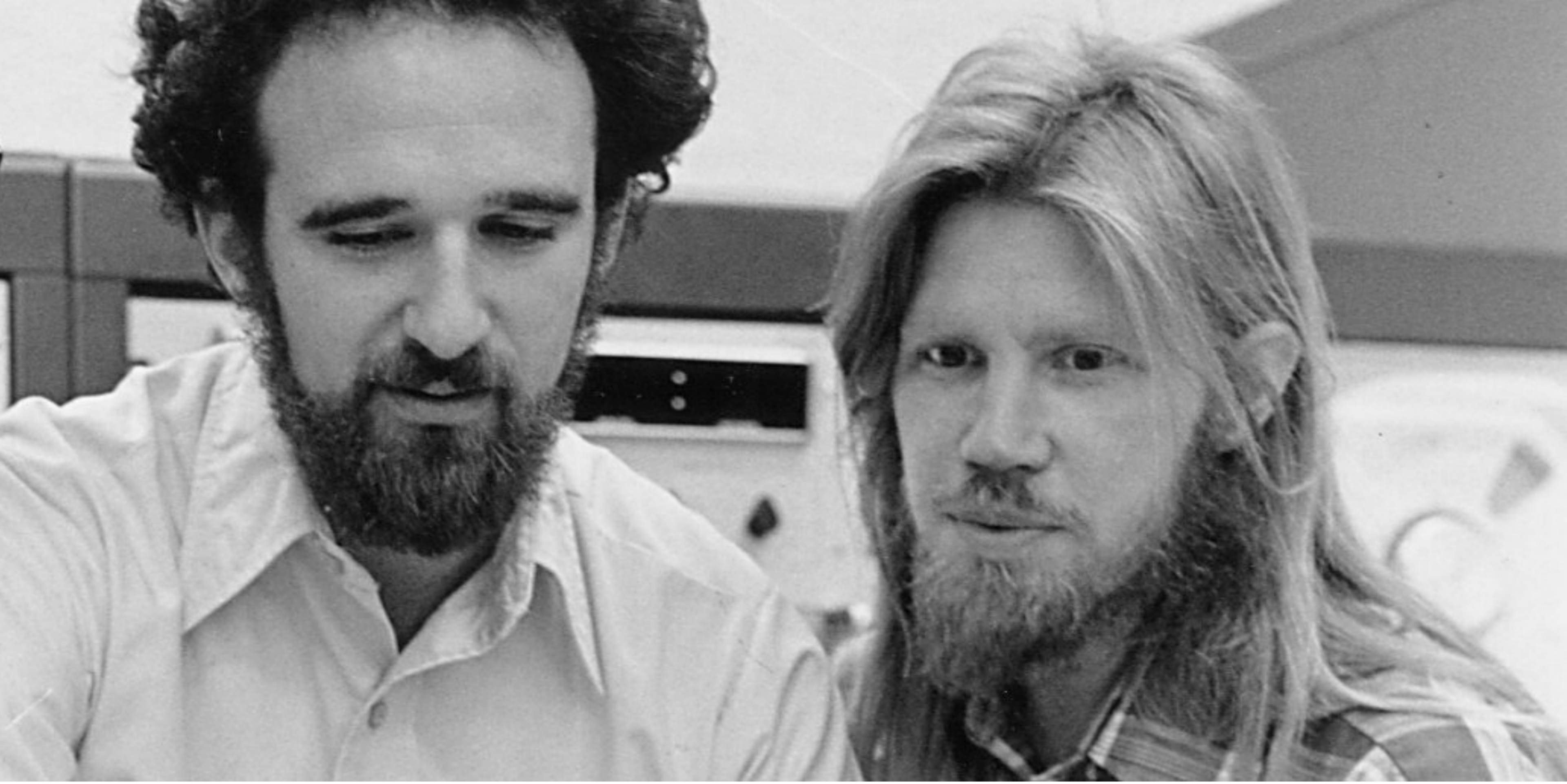
Amazing fact:

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

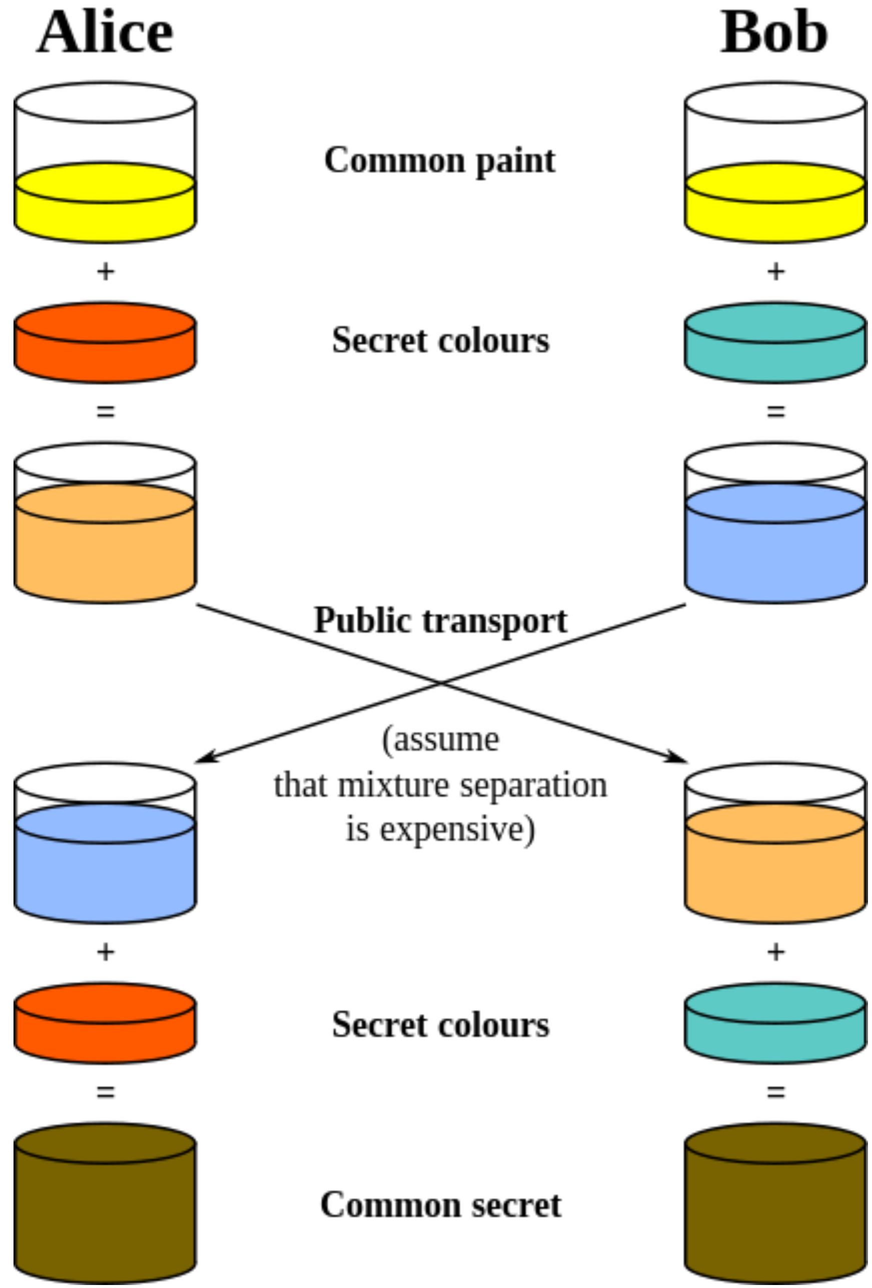
A visual analogy:

“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.



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 <= **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 $\mathbf{x}^{-1}\,\mathbf{x} = \mathbf{e}$
- *(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” $\mathbf{p}$

   $\mathbf{p}$ is a “safe prime” if ($\mathbf{p}$ - 1) / 2 is also prime

2. Pick a random number $\mathbf{g_0}$ in the range 2 to ($\mathbf{p}$ - 1)

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

   This is the “generator” of the group.

- A number $\mathbf{x}$ is in the group if $\mathbf{x}^2 \neq 1$ mod $\mathbf{p}$
- We can compute inverses $\mathbf{x}^{-1}$ s.t. $\mathbf{x}^{-1}\mathbf{x} = 1$ mod $\mathbf{p}$

Problems assumed "hard" in Schnorr groups:

- Discrete logarithm problem

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

- Diffie Hellman problem (computational)

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

- Diffie Hellman problem (decisional)

Flip a bit *c*, generate random exponents **a**,**b**,**r**

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

Guess *c*

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

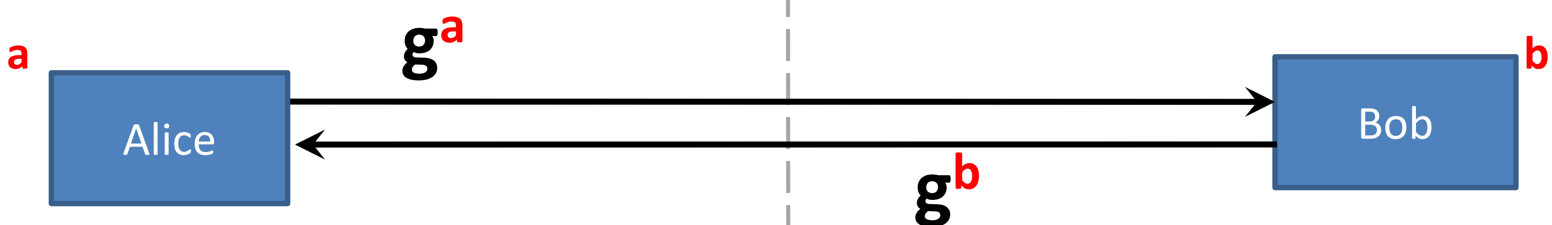
Diffie-Hellman protocol

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

2.

Alice
Generates random secret exponent $\mathbf{a}$.

Bob
Generates random secret value $\mathbf{b}$.



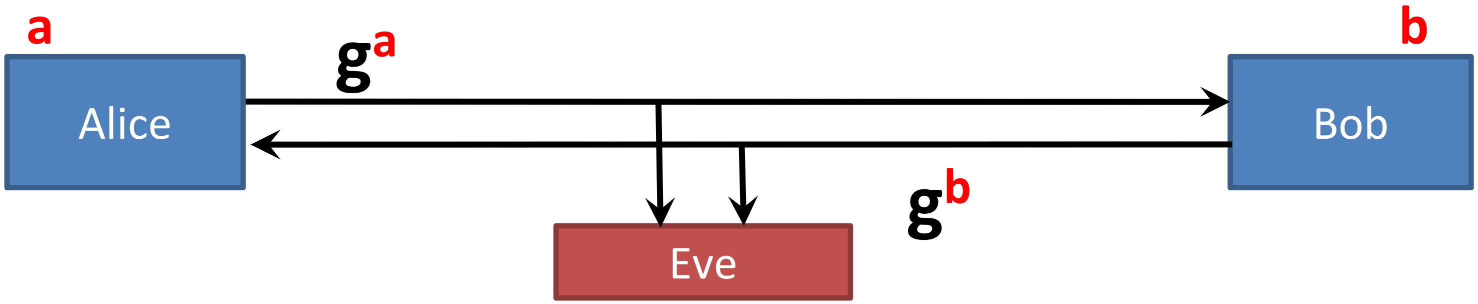
3.

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

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

(Notice that $\mathbf{x} == \mathbf{x}'$)
Can use $\mathbf{k}$:= hash($\mathbf{x}$) 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.

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 (Alice):

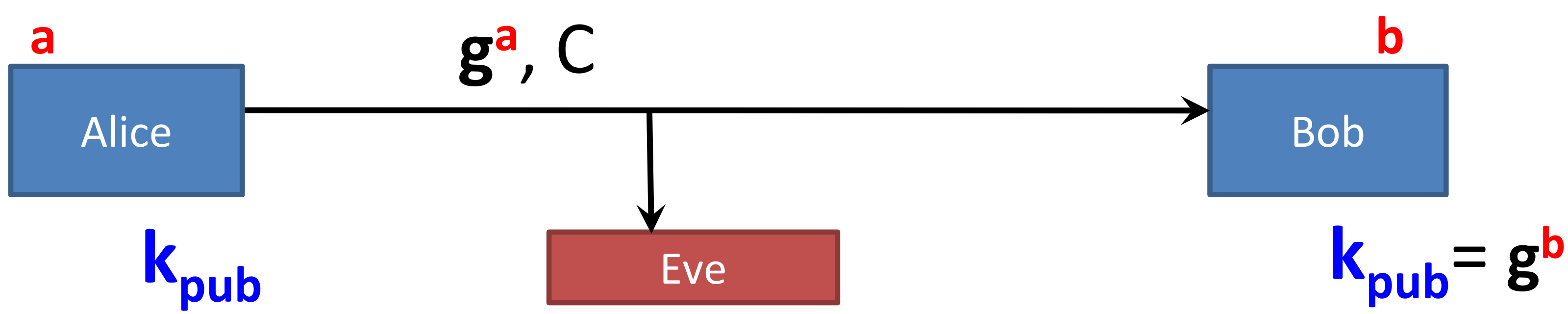
k_{priv}:= b generated randomly, and k_{pub}:= g^b

Encrypt (Bob):

Generate random a, set x := hash($k_{pub}{}^a$), encrypt M using AES with key x. Send (g^a, C)

Decrypt(Alice):

Compute x = hash($(g^a)^b$), decrypt using AES



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 *anyone* 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}

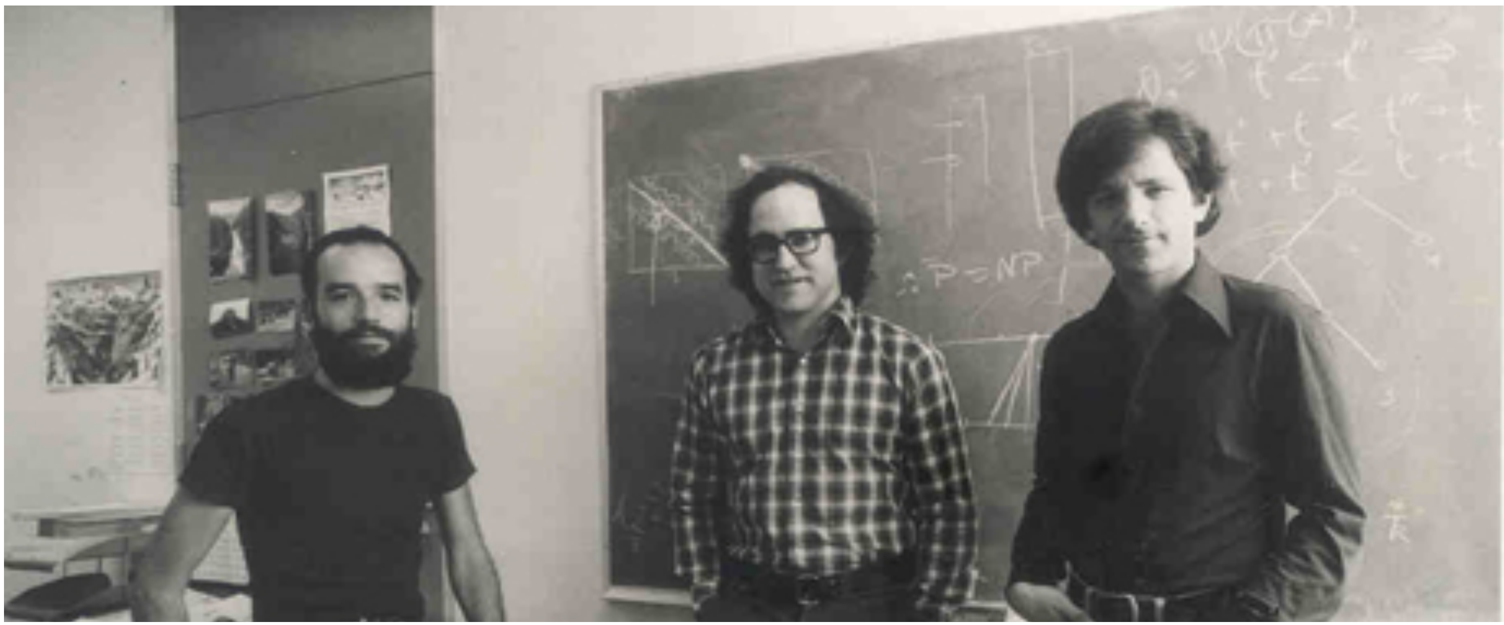
- Bob can sign a message M, resulting in signature S =Sign(k_{priv}, M)

- Anyone who knows k_{pub} can check the signature: 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 $\Phi(N)=(\mathbf{p}-1)(\mathbf{q}-1)$
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) =? M

Why RSA works

"Completeness" theorem:

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

Proof:

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

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

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

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

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

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

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

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

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

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:

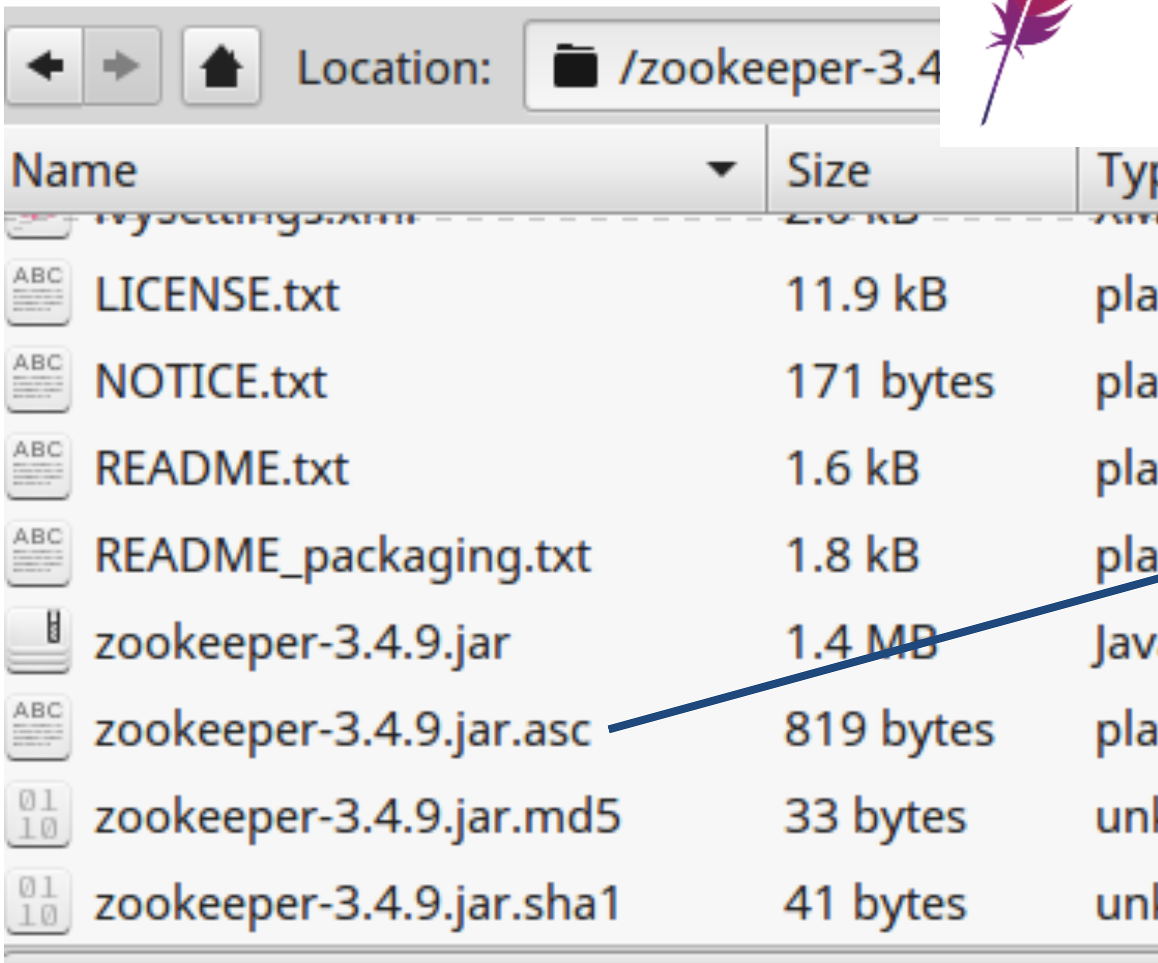
```
$ openssl genrsa -out private.pem 1024
$ openssl rsa -pubout -in private.pem > public.pem
```

To sign a message with RSA:

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

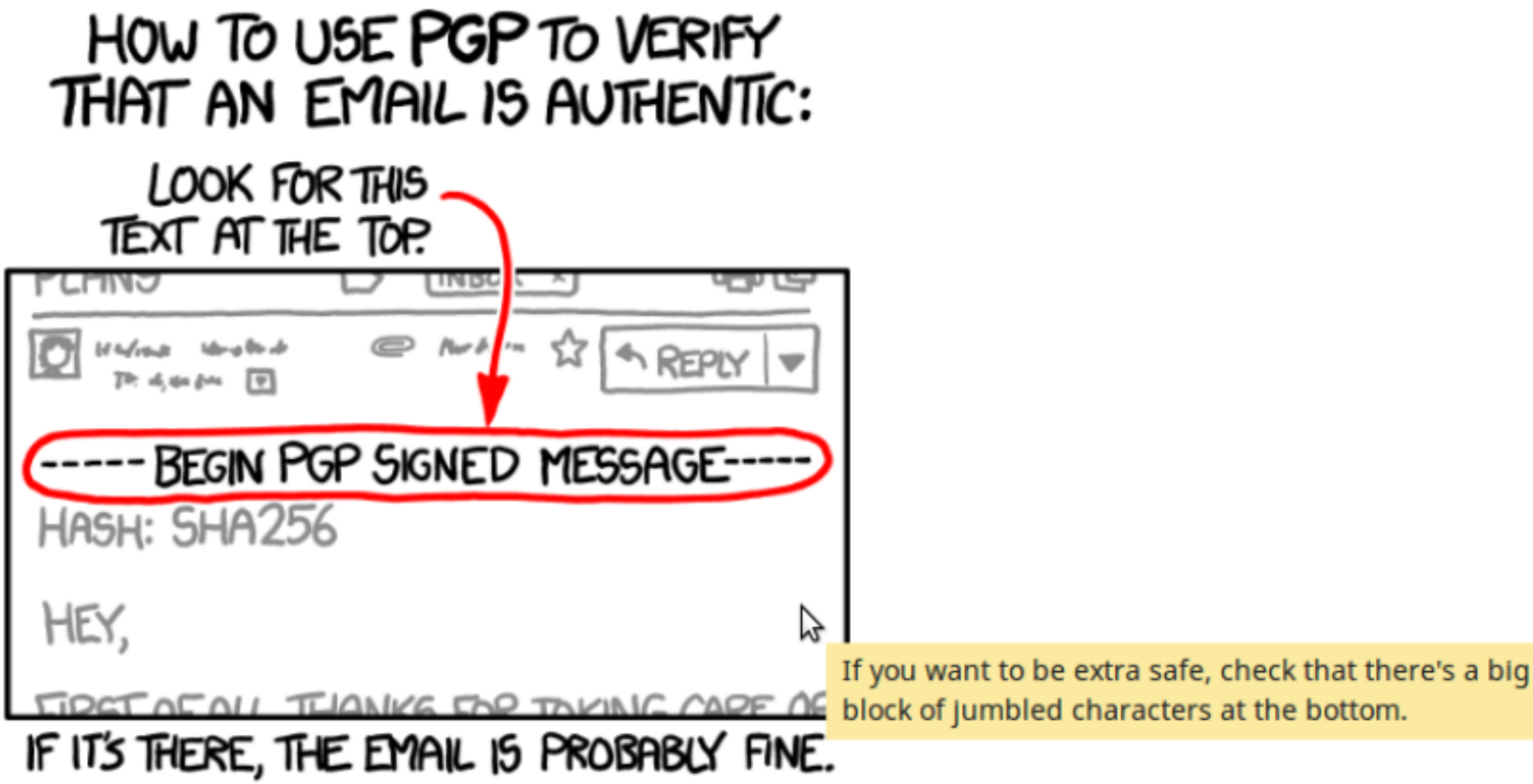
To verify a signed message with RSA:

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



Public key digital signatures on hashes of code releases

“Pretty Good Privacy” - alternate command line tool



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

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 $GCD(N_1, N_2)$ 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)

2. Vulnerability of a key increases:
   a. The more you use it.
   b. The more places you store it.
   c. The longer you have it.

3. Keep your keys far from the attacker.

4. 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 <u>any two messages</u> 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 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

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

So Far:

Message Integrity

Confidentiality

Key Exchange

Public Key Crypto