

# Properties of Cryptographic Algorithms

What do protocol and application designers need to know about the internals of cryptographic algorithms?

# Overview

- Goal: Learn enough about crypto to know what to use, when and how...
- Non-goals: Cryptanalysis, New algorithms, ...
- Logic: Protocol and application developers need to know what crypto can do for them, and how to screw up less

# Contents

- Basic concepts
- Modes of operation
- Algorithm specifics
- Implementation issues

# Basics...

- Secret-key, public-key and message digest (hash) algorithms
- Encryption, decryption and integrity protection
- Key management primitives
- (Pseudo) Random number generators

# Important Terms

- Secret Key Cryptography (e.g. AES)
- Public Key Cryptography (e.g. RSA, Elliptic Curve Cryptography)
- Message Digests (e.g. SHA256)
- Diffie-Hellman (D-H) is an important key agreement primitive
  - Integer and Elliptic Curve D-H (ECDH) variants
- Key Derivation Functions (KDF) allow one to derive a new key from existing keys (securely)
- (P)RNGs generate (pseudo) “random” bits

# Fashion Items

- Identity-Based Encryption (IBE/IBC) is a new(ish) variety of public key encryption
- Partly or Fully Homomorphic Encryption (FHE) algorithms are elegant but still impractical
- Quantum Computer Resistant schemes are another topic of current interest
- Blockchain used to do <stuff that doesn't need proof-of-work>

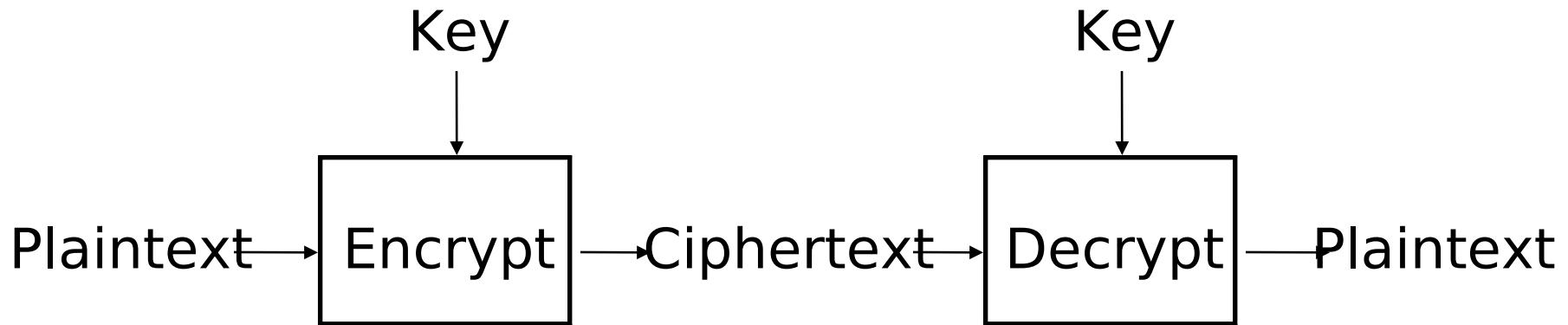
# Secret Key Cryptography

- Originally a way to keep secret data private
  - Encode a message using a secret “key”
    - A long and colorful history
- Today, it has many uses
  - Confidentiality, Authentication, Data Integrity

# What is Encryption?

- We agree on a secret way to transform data, then later...
- Use that transform on data we want to pass over an unsafe communications channel
- Instead of coming up with new transforms, design a common algorithm customized with a “key”

# Secret Key Encryption for Privacy



# “Random” Looking

- Each output value from pretty much any cryptographic function should have about 50% “1” bits
- Changing one bit of input should change about 50% of the output bits (and it ought be unpredictable which)
- Outputs should be uncorrelated, regardless of how closely related the inputs
- Any subset of the output bits should be equally random
- ... but verifying randomness is hard.

# How Secure is Encryption?

- An attacker who knows the algorithm we're using could try all possible keys (brute-force attack)
- Security of cryptography depends on the limited computational power of the attacker
- Even a fairly small key should represent a formidable challenge to the attacker (brute-force isn't possible with 128 bits)
- Algorithms have weaknesses that are independent of key size (strength != length)

# How do we know how good an algorithm is?

- A problem of mathematics: it is very hard to prove a problem is hard
- It's never impossible to break a cryptographic algorithm - we want it to be as hard as trying all keys
- Fundamental Tenet of Cryptography: *If lots of smart people have failed to solve a problem then it probably won't be solved (soon, we hope)*

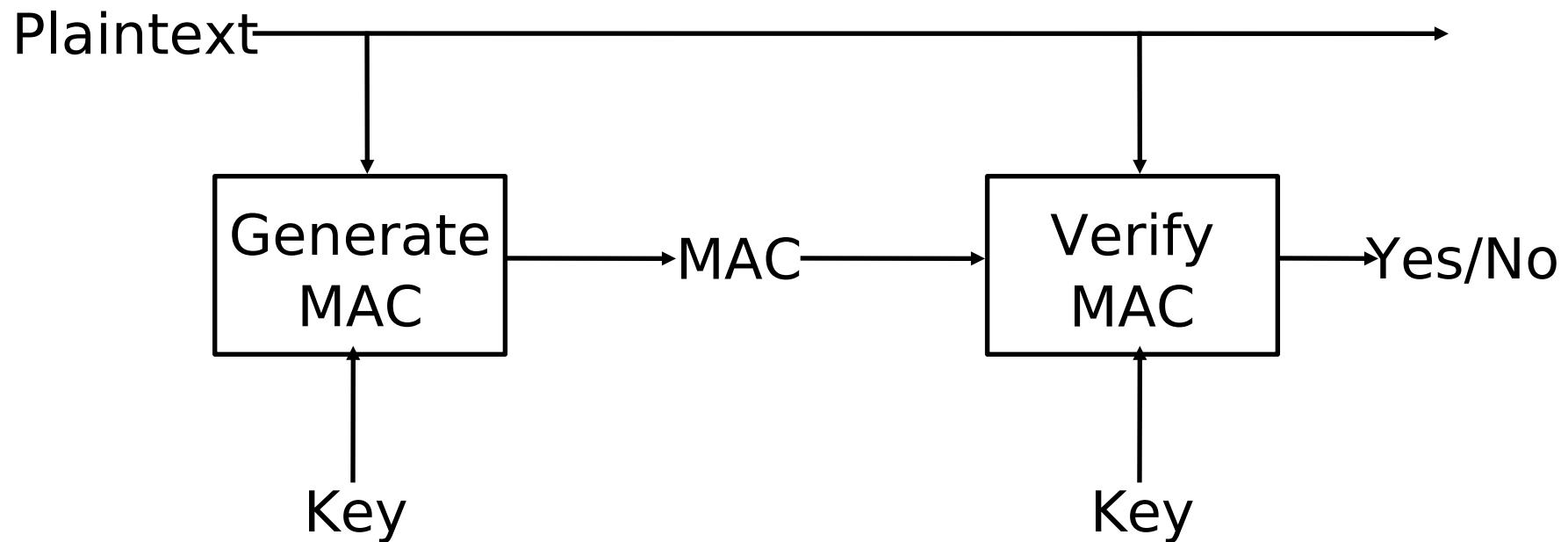
# To Publish or Not to Publish

- If the good guys break your algorithm, you'll hear about it
- If you publish your algorithm, the good guys provide free consulting by trying to crack it
- The bad guys will learn your algorithm anyway
- Today, most commercial algorithms are published; most military algorithms are not

# Uses of Cryptography

- Confidentiality for data in transit: Transmitting secret data over an insecure channel
- Confidentiality for data at-rest: Storing secret data on an insecure medium
- Data Integrity: Message integrity checksum/authentication code (MIC/MAC)
- Authentication: “challenge” the other party to encrypt or decrypt a random number

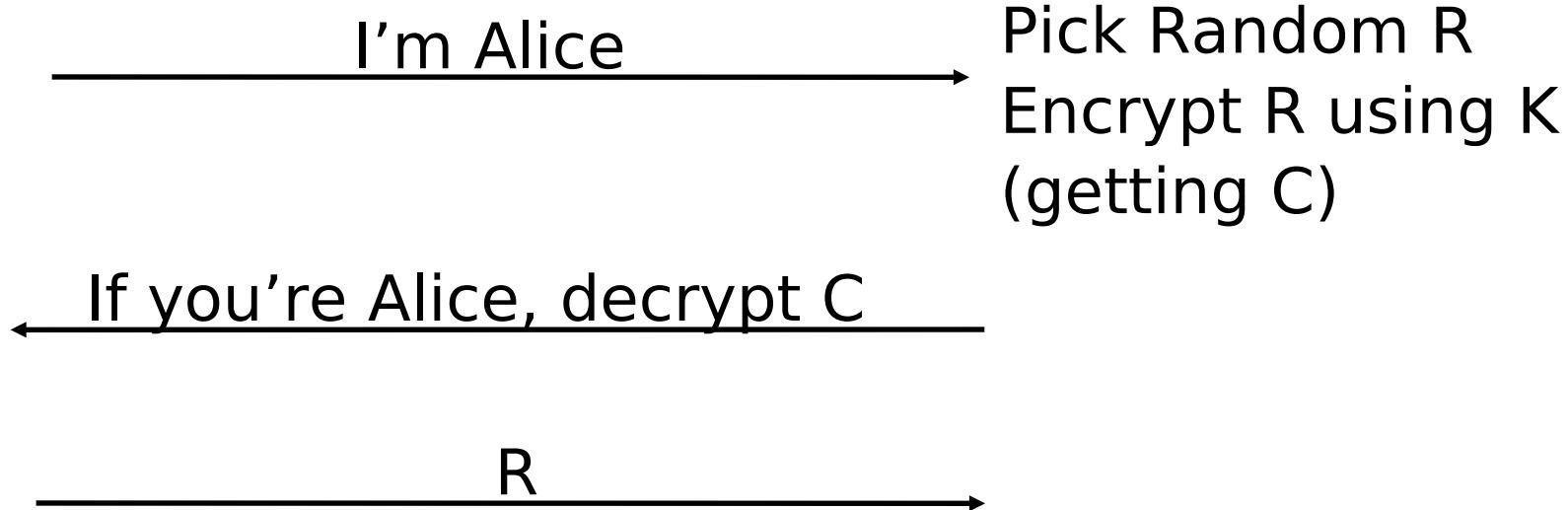
# Secret Key Integrity Protection



# Challenge / Response Authentication

Alice (knows K)

Bob (knows K)



# Ancient (pre-1990's) Secret Key Algorithms

- DES (Data Encryption Standard)
  - 56 bit key (+ 8 parity bits)  
controversial!
  - Input and output are 64 bit blocks
  - slow in software, based on  
(sometimes gratuitous) bit diddling
- IDEA (International Data Encryption Algorithm)
  - 128 bit key
  - Input and output are 64 bit blocks
  - designed to be efficient in software
  - IPR

# Secret Key Algorithms

- Triple DES (NOT RECOMMENDED)
  - Apply DES three times (EDE) using K1, K2, K3 where K1 may equal K3
  - Input and output 64 bit blocks
  - Key is 112 or 168 bits
- Advanced Encryption Standard (AES)
  - 2001 US standard to replace DES.
  - Public Design and Selection Process  
Winner=Rijndael
  - Key Sizes 128,192,256. Block size 128.
  - **AES IS RECOMMENDED**

# Secret Key Algorithms

- RC4(**VERY NOT RECOMMENDED**)
  - 128-bit key stream cipher
  - once widely used in TLS
  - “Ron’s cipher #4”
  - Broken
- ChaCha20
  - DJB productions
  - 256-bit key stream cipher
  - Considered an ok replacement for RC4 – faster than AES on processors without the AES-NI instruction
  - Recommended if not using AES
    - today, could change

# XOR (Exclusive-OR)

- Bitwise operation with two inputs where the output bit is 1 if exactly one of the two input bits is one
- $(B \text{ XOR } A) \text{ XOR } A = B$
- If A is a “one time pad”, this is very efficient and very secure, but **NEVER** re-use anything
- Common stream-cipher encryption schemes (e.g. ChaCha20 described in RFC7539) calculate a pseudo-random stream from a short key
- Actually, RFC7539 is a very good read as it describes a modern cipher in the manner needed for modern implementation

# More Secret Key Algorithms (some busted)

- Magma, Russian cipher
  - Part of GOST family
- SM4 Chinese symmetric cipher
  - SM2, SM3 other cipher stuff
- A2/A5 – GSM algorithms
- Camellia, Aria, SEED, ...
- Simon/Speck
- ... there'll always be more

# Other Secret Key Algorithms

- Many other secret key algorithms exist
- It may appear easy to invent a new variant on one
- It is very hard to invent secure algorithms
- There are few good reasons to invent new ones
  - And there is no reason for any of us here to want to invent new algorithms!!

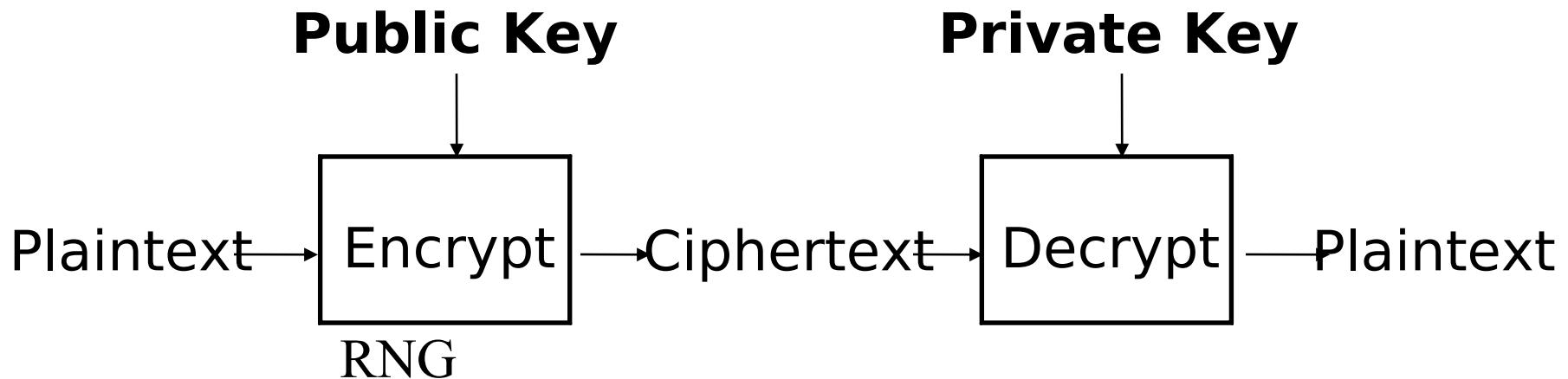
# Reasons for new algs...

- We would like a “spare” for each cryptographic function that is already deployed so we can turn it on if/when today’s preferred option goes bad
  - Remember: attacks only ever get worse
- Some countries impose national requirements that national ciphers MUST be supported/used
  - Bad plan, but not always avoidable

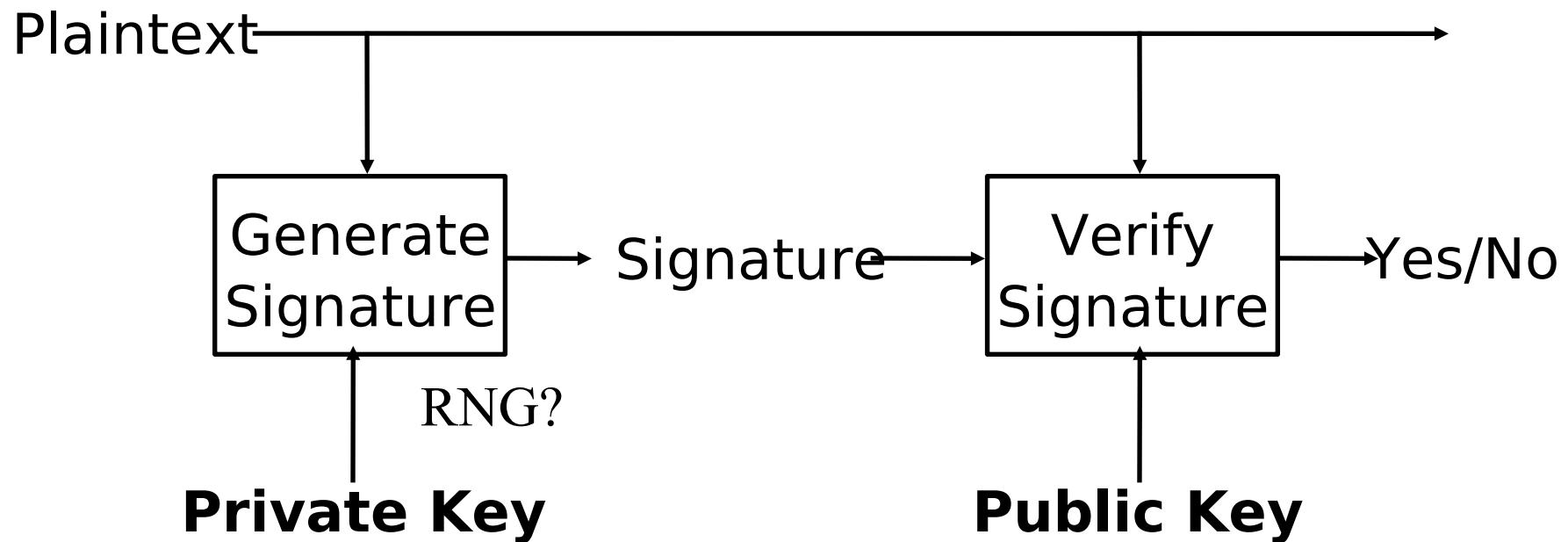
# Public Key Cryptography

- Two keys per user: a private key and a public key. The keys reverse each other's effects.
  - Encrypt a message for Alice using her public key
  - Decryption requires her private key
  - Generating Digital Signatures requires the private key
  - Verifying them requires the public key
- 
- Note: Getting the role of public/private keys backwards is counterproductive, exam-wise:-)

# Public Key Encryption for Confidentiality



# Public Key Integrity Protection



# Public Key Authentication

Alice (knows A's  
private key)

I'm Alice

Bob (knows A's  
public key)

Pick Random R  
Encrypt R using  
A's public key  
(getting C)

If you're Alice, decrypt C

Decrypt C

R

# Message Digest Functions

- Also known as cryptographic hashes
- Non-reversible function
- Takes an arbitrary size message and mangles it into a fixed size digest
- It should be impossible to find two messages with the same MD, or come up with a message with a given MD
- Useful as a shorthand for a longer thing

# Message Digest Functions



# Message Digest Functions

- MD2, MD4, and MD5 used to be most popular.
  - All produce 128 bit digests
  - MD4, MD2 and MD5 are broken
  - SHA-1 has significant, demonstrated weaknesses
- **SHA-256 currently the best choice**
- NIST competition for new digests:
  - SHA-3 is Keccak
  - But not very interesting!
- Message digests are not difficult to design, but most are not secure
  - Sound familiar?

# Random number generators

- True random number generator (TRNG) based on some physical source (e.g. noise diodes)
- Pseudo random number generator (PRNG) based on some algorithm, usually with a seed
  - All PRNGs will eventually repeat
  - Not all PRNGs are good
  - The seed needs to be good, \$PID/time\_t is not enough!
- At least one PRNG was deliberately engineered to be secretly bad <http://dualec.org>

# Using PRNGs

- Do use the system PRNG unless you know that's bad - On linuxes that'd be either /dev/random or /dev/urandom – be careful if you care about blocking/non-blocking calls, maybe install haveged
- It's always ok to add more randomness if you have an interface that allows that, e.g. packet checksums rx'd from n/w, but it's not ok to fully re-start your PRNG with such – be similarly careful with forking and virutalisation
- It's a good idea to have different streams for public random stuff (e.g. nonces, message-IDs) and secret random stuff (keys etc.) in case you're PRNG is borked like DUAL-EC
- Be careful using APIs (e.g. C's rand() is not good) – check out the API you're looking at before using, and don't believe just one posting that says it's ok
- General advice: think about what you want to do, and then, before doing it, check out to see if anyone else has done that and what folks thought about it (e.g. via stackexchange etc, but again don't belive just one source)

# Combining Cryptographic Functions for Performance

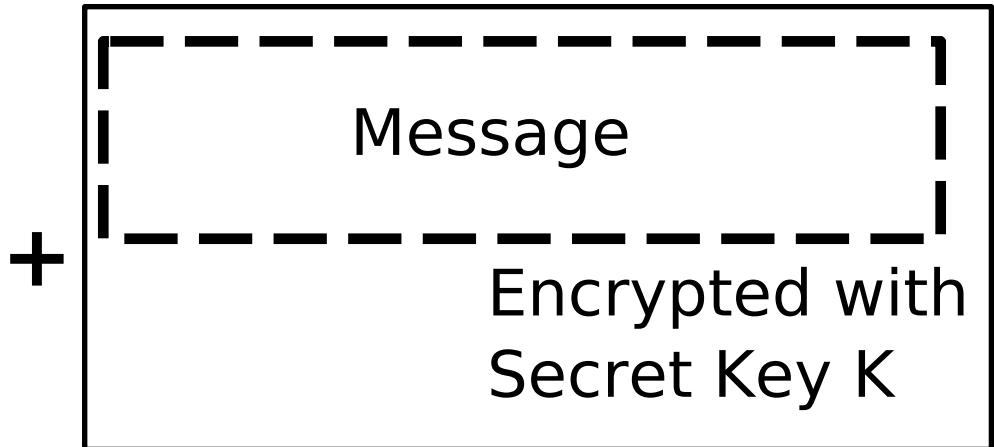
- Public key cryptography is slow compared to secret key cryptography and hashes
- Public key cryptography is often more convenient & secure in setting up keys
- Algorithms can be combined to get the advantages of both

# Hybrid Encryption

Instead of:

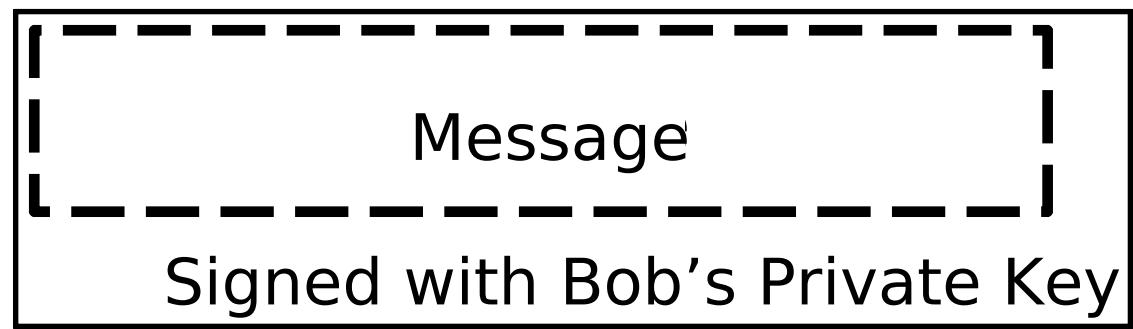


Use:



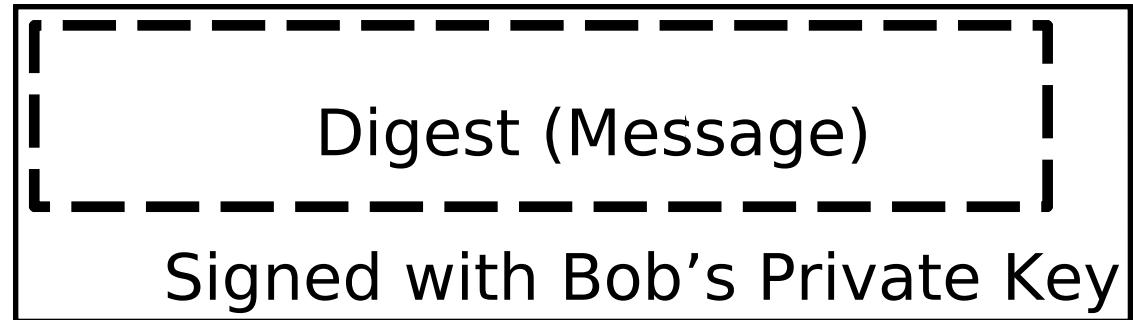
# Hybrid Signatures

Instead of:

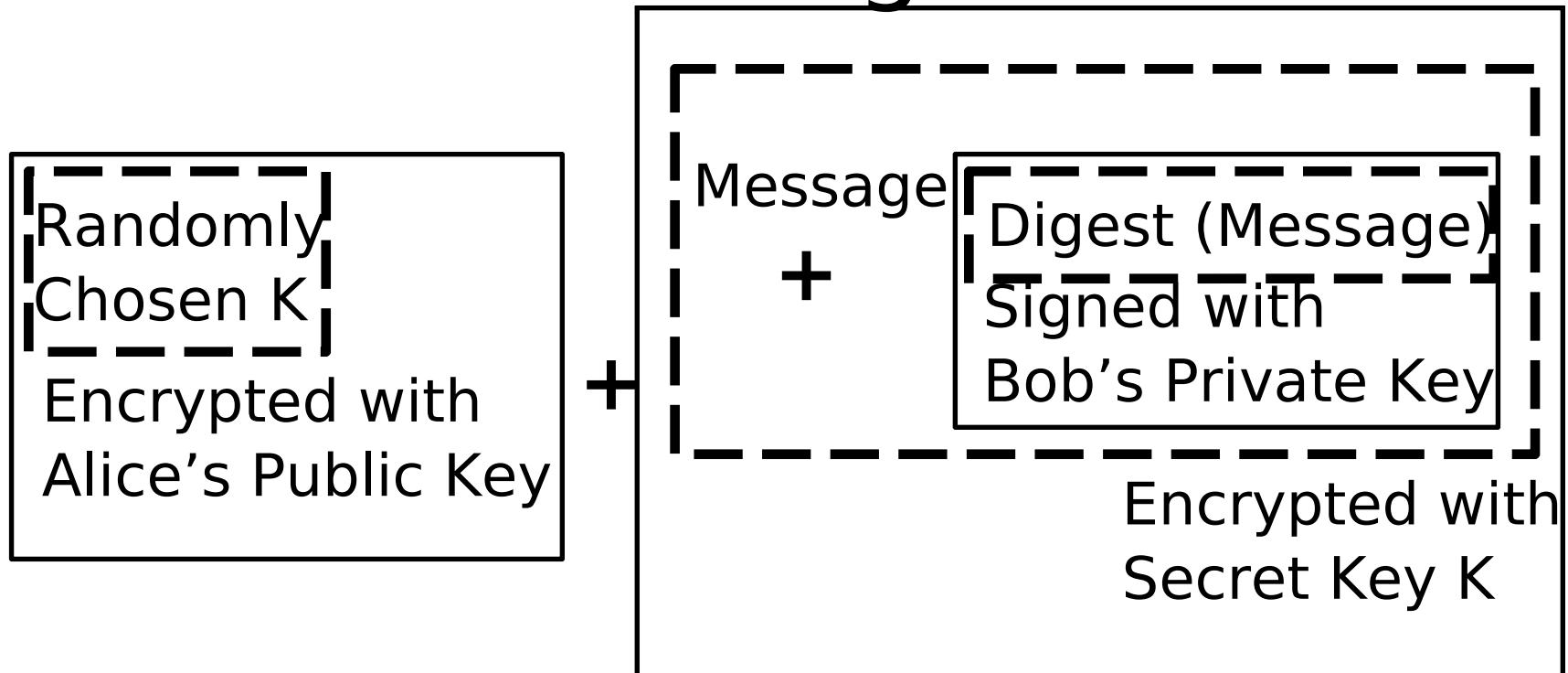


Use:

Message +



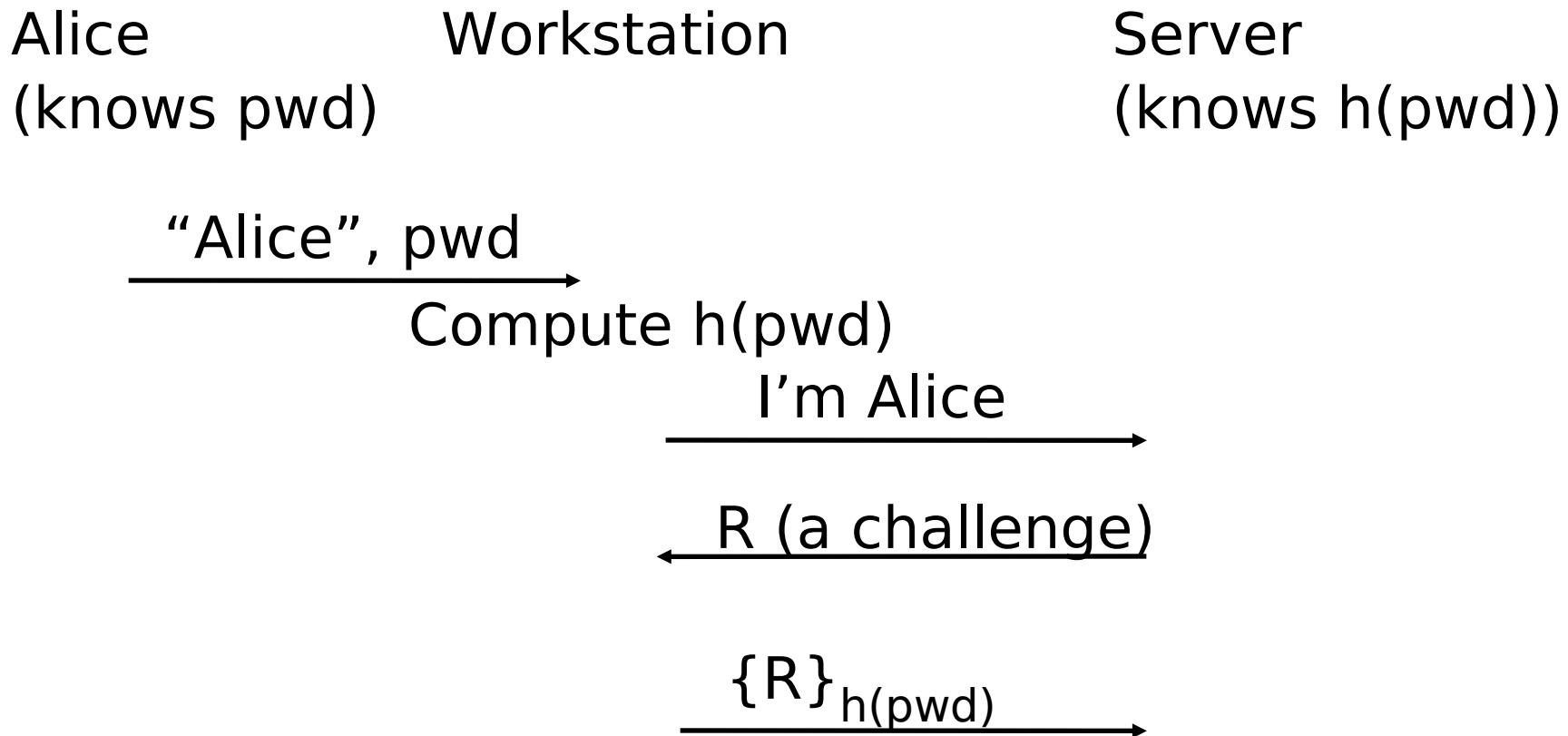
# Signed and Encrypted Message



# Passwords as Secret Keys

- A password can be converted to a secret key and used in a cryptographic exchange
- An eavesdropper can often learn sufficient information to do an off-line attack
- Most people will not pick passwords good enough to withstand such an attack
- More later...

# Sample Protocol



# Key Distribution - Secret Keys

- What if there are millions of users and thousands of servers?
- Could configure  $n^2$  keys
- Better is to use a Key Distribution Center (KDC)
  - Everyone has one key
  - The KDC knows them all
  - The KDC assigns a key to any pair who need to talk

# Key Distribution - Secret Keys

Alice

KDC

Bob

A wants to talk to B

Randomly choose  $K_{ab}$

$\{“B”, K_{ab}\}_{Ka}$        $\{“A”, K_{ab}\}_{Kb}$

$\{Message\}_{Kab}$

# Key Distribution - Public Keys

- Certification Authority (CA) signs “Certificates”
- Certificate = a signed message saying “I, the CA, vouch that 489024729 is Radia’s public key”
- If everyone has a certificate, a private key, and the CA’s public key, they can authenticate

# KDC vs CA Tradeoffs

- Stealing the KDC database allows impersonation of all users and decryption of all previously recorded conversations
- Stealing the CA Private keys allows forging of certificates and hence impersonation of all users, but not decryption of recordings
- Recovering from a CA compromise is easier because user keys need not change

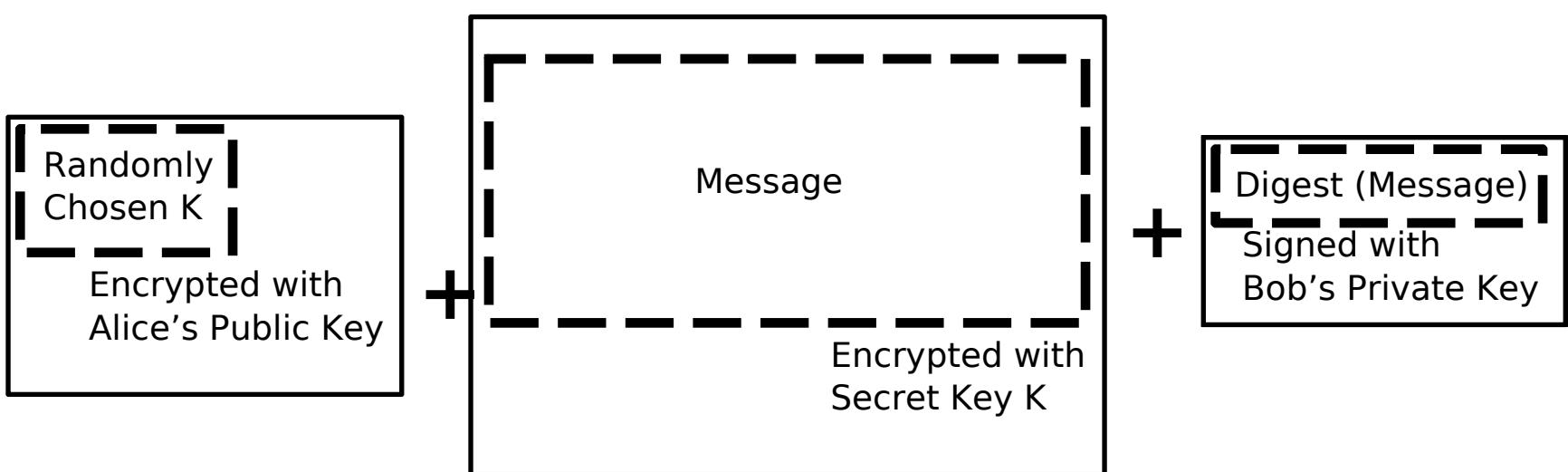
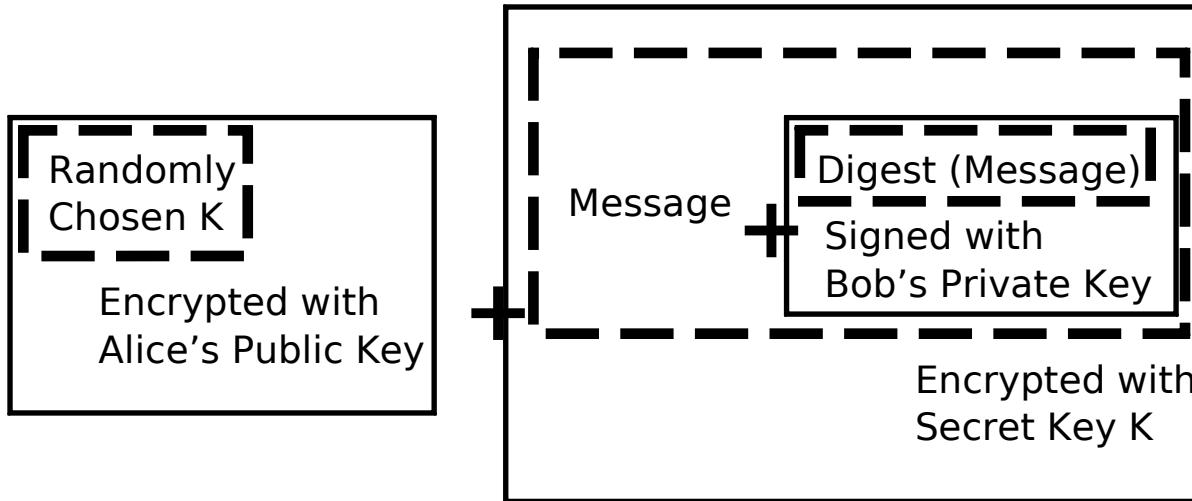
# KDC vs CA Tradeoffs

- KDC must be on-line and have good performance at all times
- CA need only be used to create certificates for new users
  - It can be powered down and locked up, avoiding network based attacks
- CA's work better interrealm, because you don't need connectivity to remote CA's

# KDC vs CA Tradeoffs

- Public Key cryptography is slower and (used to) require expensive licenses
- The “revocation problem” levels the playing field somewhat

# Spot the difference...



# More Cryptography

# Next hour...

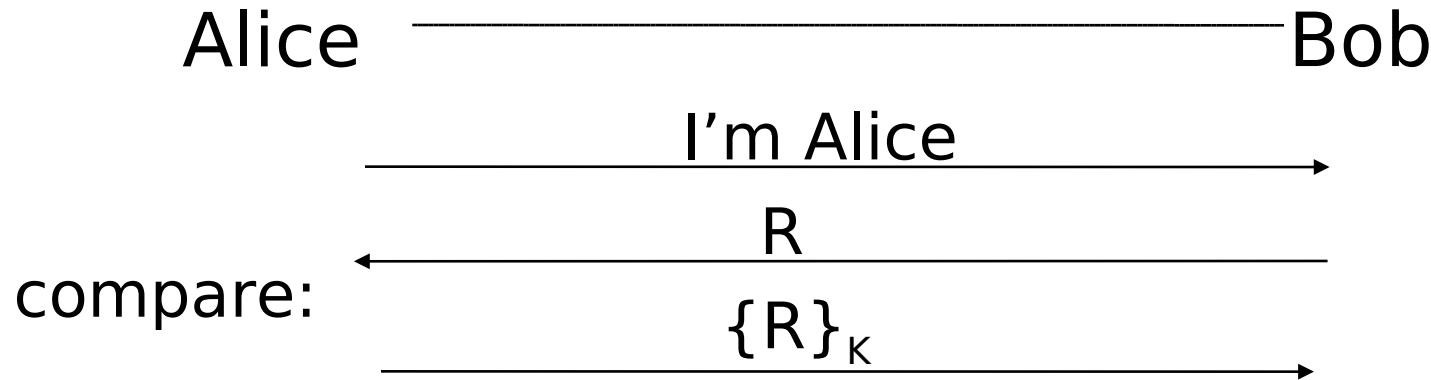
- Review basic crypto
- Modes of operation of algorithms
- Feistel ciphers
- AES
- Snakeoil

# Review of crypto from last (few) lecture(s)

- secret key: 2 operations, inverses
- public key: one op, 2 keys (public, private), which are inverses
- operations
  - encrypt with secret/public, decrypt with secret/private
  - authenticate with secret/private, verify with secret/public
  - compute integrity with secret/private, verify with secret/public

# Authentication with Secret Key

both know secret K



or:

I'm Alice,  $\{\text{timestamp}\}_K$

# Secret Key algorithms

- Stream ciphers (e.g., RC4, ChaCha20)
  - takes key and generates a stream of pseudorandom bits, XOR'd into data
- Block ciphers (e.g., 3DES, AES)
  - takes key and fixed size input block to generate fixed size output block

# Types of attacks

- ciphertext only: can brute-force attack if recognizable plaintext
- 1<sup>st</sup> 1000 bytes of RC4 are a worry today
- sometimes a system allows other attacks:
  - known plaintext
  - chosen plaintext
- Shannon proved XOR with one-time pad unbreakable (no information with brute force attack)
  - BUT BUT BUT, unbreakable here is entirely unrealistic!!!

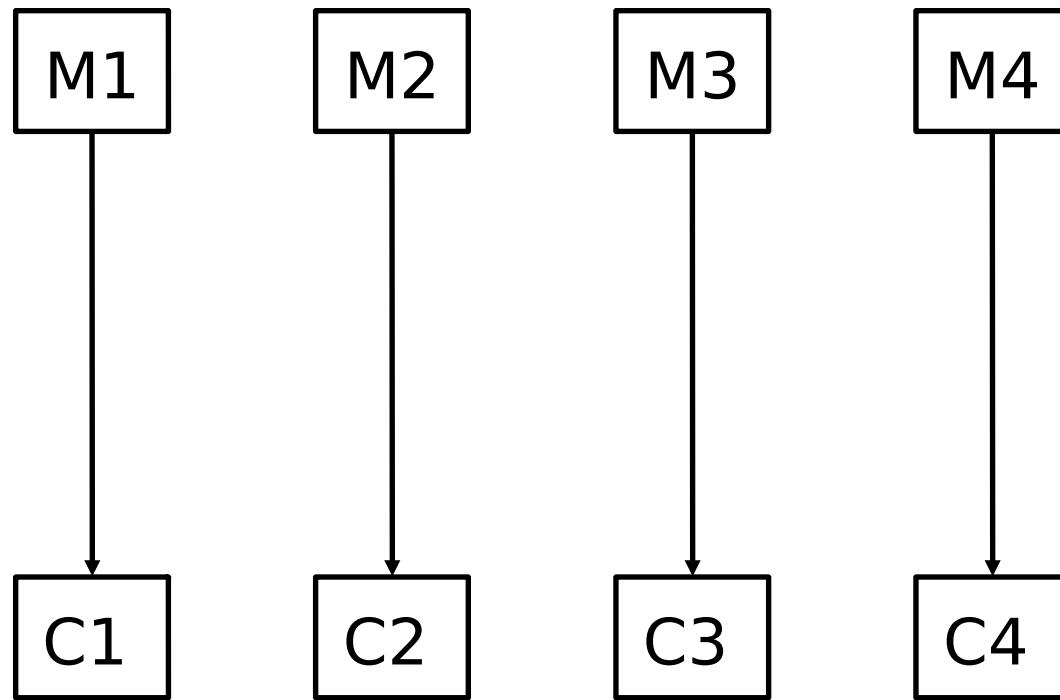
# Block size considerations

- If small block size, could build a table
- If see same ciphertext block, get hints about plaintext
- to avoid probably seeing repeated ciphertext blocks, should change key in number of blocks  
 $2^{\text{half the block size}}$  (birthday problem)

# Encrypting Large Messages

- Basic block ciphers encrypt a small fixed size block
- Obvious solution for large messages is to encrypt a block at a time.
  - This is called Electronic Code Book (ECB)
- Repeated plaintext blocks yield repeated ciphertext blocks
- Other modes “chain” to avoid this (CBC, CFB, OFB)
- Encryption does not guarantee integrity!

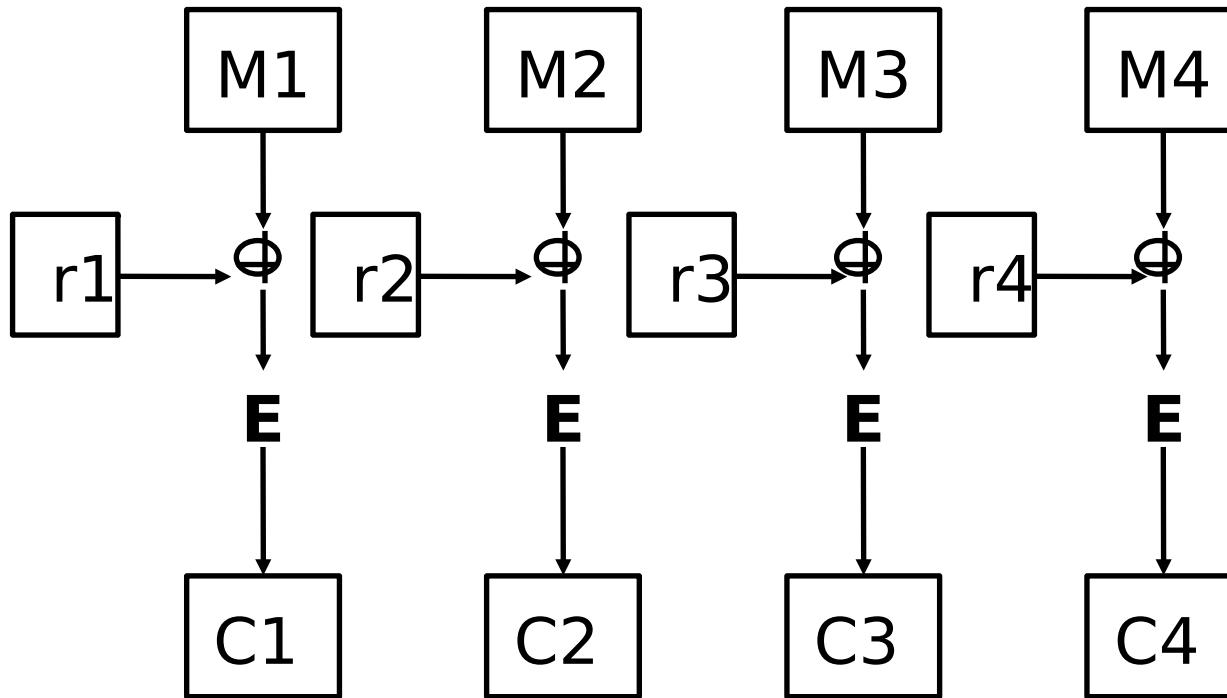
# ECB



# Problems with ECB

- If  $c_i = c_j$ , then you know  $p_i = p_j$
- Can reorder blocks
- Can rearrange blocks to affect plaintext

# Consider this

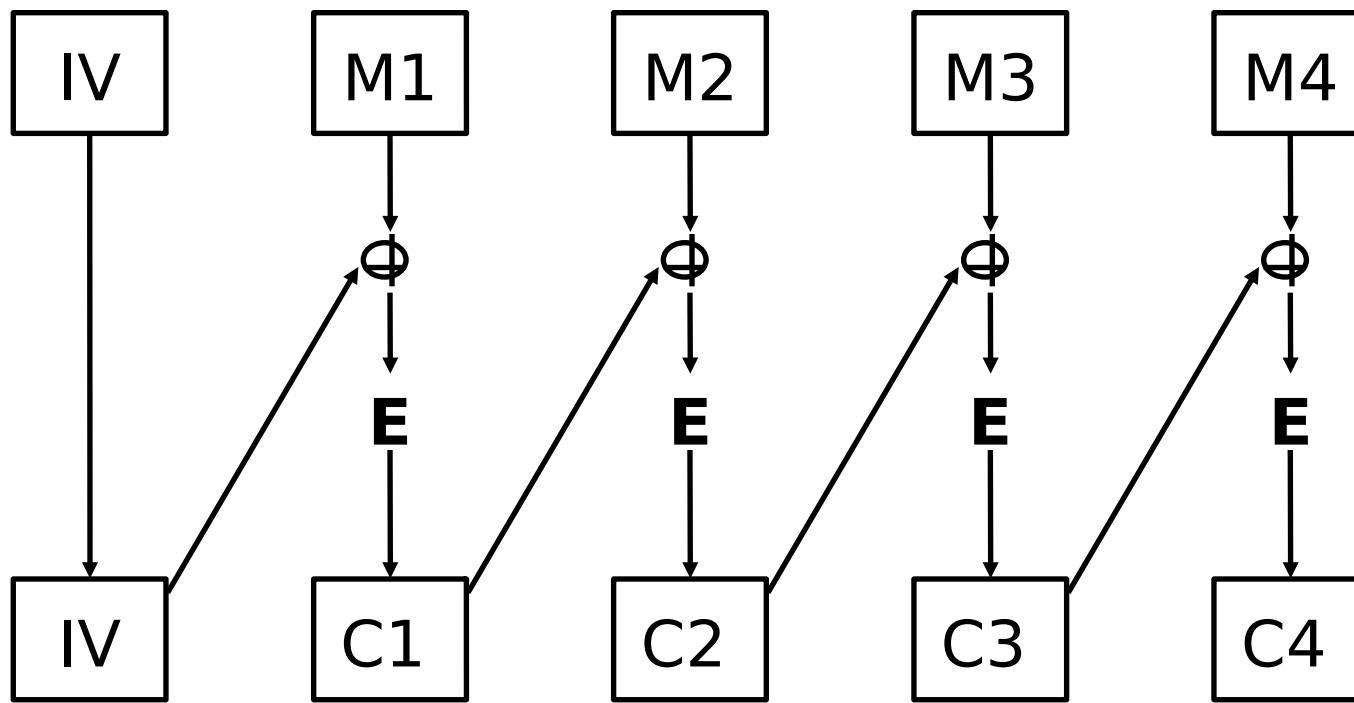


transmit r1, c1, r2, c2, r3, c3, r4, c4

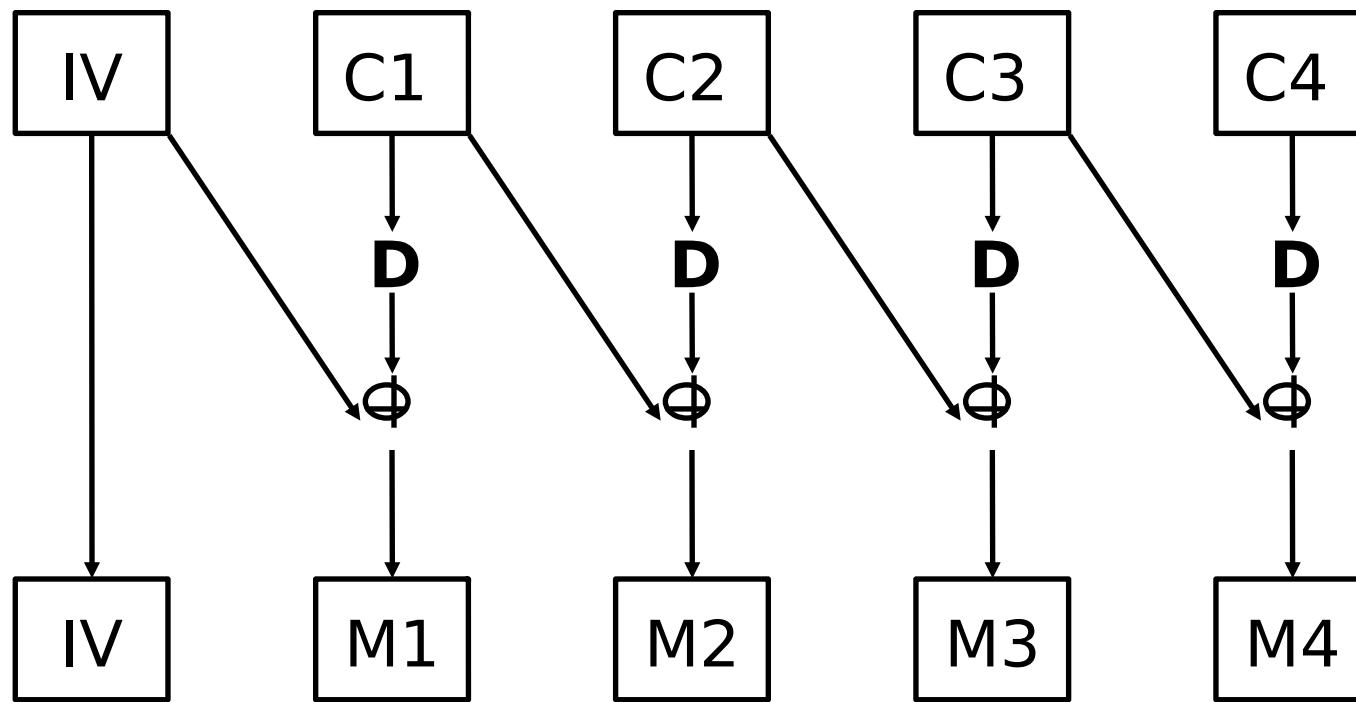
# Problems with previous slide

- Need to send twice as much data
- Can still rearrange blocks
- If two ciphertext blocks equal, know  
 $\text{XOR}$  of two plaintext blocks =  $\text{XOR}$  of  
the corresponding two random  
numbers
- CBC generates its own “random  
numbers” by using previous ciphertext  
block, plus one additional block (the  
“IV”, initialization vector)

# CBC (Cipher Block Chaining)



# CBC Decryption



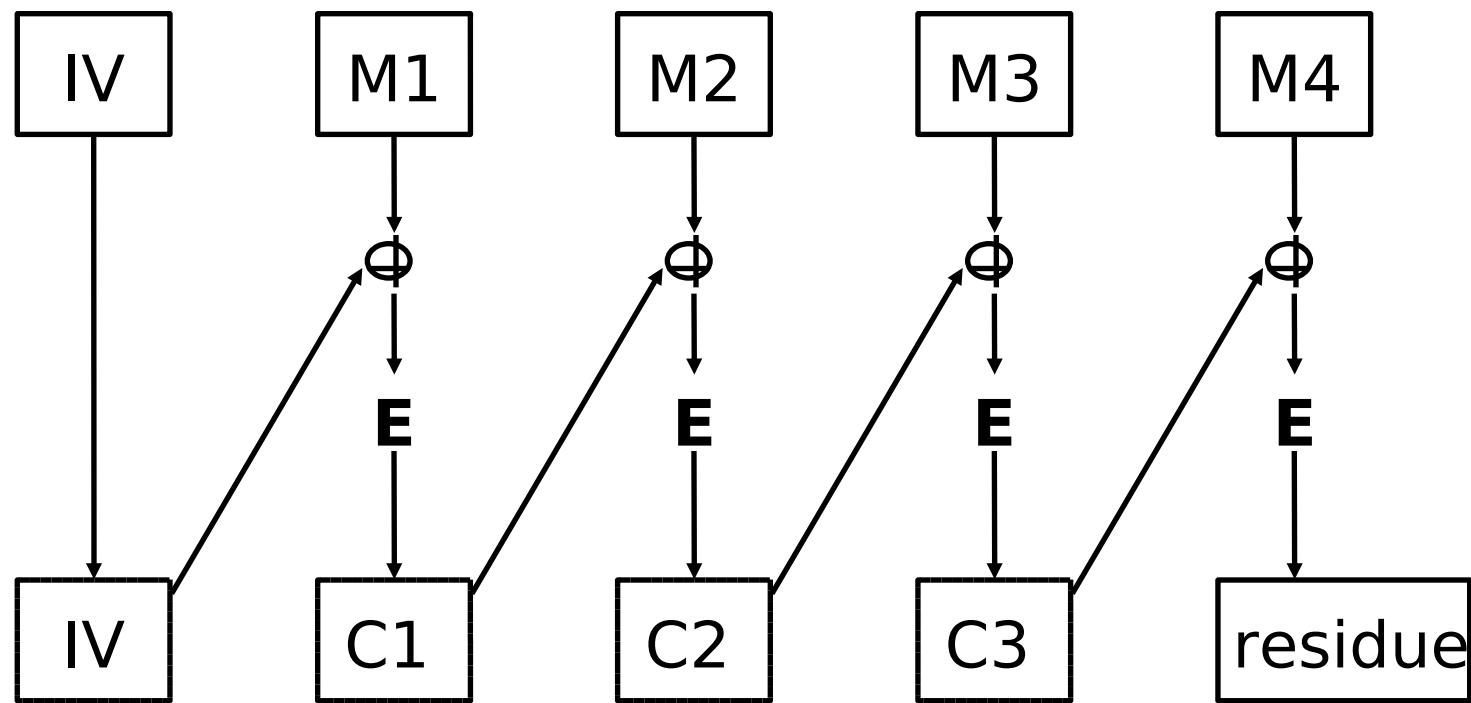
# CBC

- What happens if  $c_i$  gets lost? Garbled? How much data gets lost?
- How can attacker that sees and can modify the ciphertext, and knows the plaintext, modify the plaintext in a predictable way? What other effects will it have?

# Integrity Check (MAC) with Secret Key

- “CBC Residue”, uses key K and generates integrity check on message M
- Do CBC encryption on M using key K, throw away all but last block. Last block is “residue”, and is used as integrity check
- Used in banking
- Has property that if you don’t know the key you can’t generate (or verify) the MAC, or modify the message without (probably) changing the MAC

# CBC Residue



# CBC Residue

- Note that it is easy to generate an arbitrary message with a particular residue

Create message, but leave one block (anywhere) blank. Use any key K and any IV. Start from beginning, doing ordinary CBC residue until get to block left blank. Start from end, doing ordinary CBC decryption (since residue is constrained, you can work backwards from that and the plaintext blocks). Finally you will find two quantities that must be XOR'd together to yield the value that must be in the blank block.

# Creating a stream cipher

- Output Feed Back (OFB) stream generated:
  - IV (transmitted in the clear)
  - $\text{pad}_1 = e(\text{IV}, \text{key})$
  - $\text{pad}_2 = e(\text{pad}_1, \text{key})$
  - $\text{pad}_i = e(\text{pad}_{i-1}, \text{key})$
- Can be generated in advance
- Can encrypt arbitrary # of bits (vs block cipher)
- What if ciphertext garbled or lost?
- If know plaintext, can easily modify stream

# Counter Mode

- $c_i = f(\text{key}, \text{IV}, \text{block number}, p_i)$
- Can decrypt an arbitrary block (useful for, e.g., random access file encryption)

# Authenticated Encryption (with Additional Data)

- Authenticated encryption (AE) with additional data (**AEAD**) is a general scheme for combining confidentiality and data-integrity as a single primitive
- Motivation: many applications/protocols involve cleartext headers that goes with ciphertext and we'd like both to be protected in a single operation with a single key (context) as input
- General idea is you encrypt data, and provide the additional data (e.g. headers), result is ciphertext that includes a “tag”
- Decryption takes ciphertext (incl. tag) and returns “error” or plaintext, you never get partial decryption
- RFC 5116 defines an abstract interface for AEADs
- Also – older modes like CBC have been shown to be vulnerable to various implementation and other vulnerabilities

# OpenSSL AEAD Example

- I recently implemented the (draft) “Hybrid Public Key Encryption” scheme for encrypting “to” a public key:
  - Spec: <https://tools.ietf.org/html/draft-irtf-cfrg-hpke>
  - Code: <https://github.com/sftcd/happykey>
- Spec combines (Elliptic curve) Diffie-Hellman key agreement with AEAD encryption, with various (far too many) options
- `hpke_aead_enc()` function in `hpke.c` has OpenSSL AEAD calls
- Note to self: pull up that code to show init/update/final pattern
- Not making any claim of “goodness” (NaCL/python/golang equivalents could be better), but it’s a smallish standalone bit of code that seems to do the job

# What AEAD modes exist?

- Loads. Far too many: OCB, CCM, GCM, ChaCha20-poly1305, SIV, ...
- Which are good/bad? All of 'em... Pretty much, though some are much more used than others.

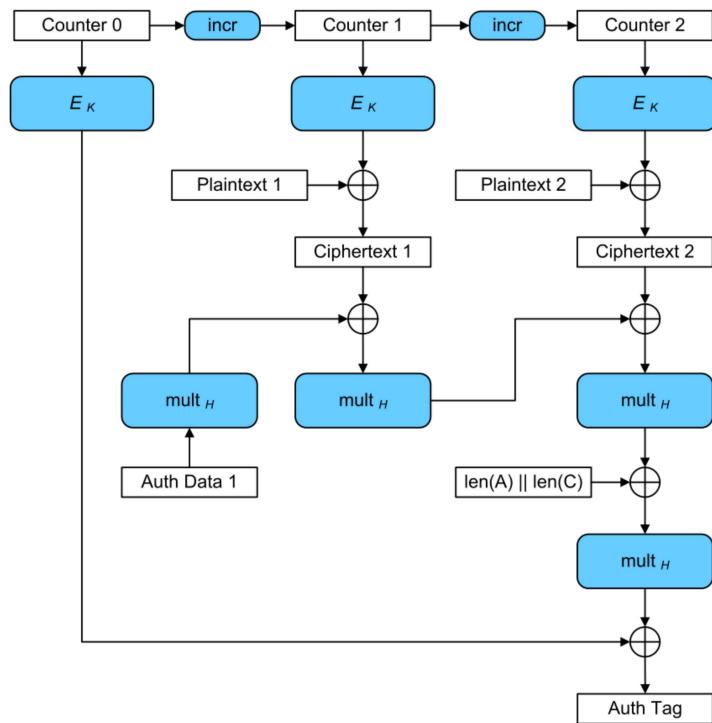
[https://www.fi.muni.cz/~xsvenda/docs/AE\\_comparison\\_ipics04.pdf](https://www.fi.muni.cz/~xsvenda/docs/AE_comparison_ipics04.pdf)

isn't a bad comparison of a bunch of 'em

- So what do I need to know about 'em?
  - Complexity, Performance aspects, Patents, Brittleness

# Complexity

- Most AEADs are a bit harder to understand and hence their caveats are harder to explain and remember
- The diagram shows AES-GCM  
GCM = Galois Counter Mode
- Implementing AEADs is a bit more difficult but in any case you **SHOULD NOT** be coding this kind of thing – use a good library!!
- So internal-complexity is basically hidden from our POV



# Performance Aspects

- AES-GCM allows for super-parallel and pipelined implementation, e.g. as would be needed in a high-speed router with many 10GB ports
- AES-CCM (CCM = Counter with CBC-MAC) requires you know the lengths ahead of time, but is popular for smaller processors, e.g. as needed in WiFi and smaller client devices like Zigbee
- ChaCha20-poly1305 – fast if you don't have hardware support (many CPUs do have AES h/w support e.g. Intel AES-NI)
- All AEAD modes add more overhead – nonces and tags which can be a pain if you have lots of small packets and are constrained by bandwidth or packet sizes, but mostly the overhead is ok

# Patents

- Lots of cryptographers like OCB (Offset Code Book) but there are patents which has killed adoption
- Even though the inventor has tried hard to license liberally
- AES-GCM, AES-CCM and ChaCha20-poly1305 are IPR “clean” as far as I know – though there are likely patents on some speed-up tricks, e.g. with highly parallel implementations of AES-GCM as used in high-end routers

# Brittleness

- AES-GCM has a “gotcha” - if you **ever** re-use the same key and nonce you’re screwed!
- That’s because AES-GCM ends up behaving like a stream cipher
- Any AEAD mode can have nonce-reuse brittleness
  - Some are worse than others, but none are good
- So don’t re-use nonces – **ever**

# AEAD Summary

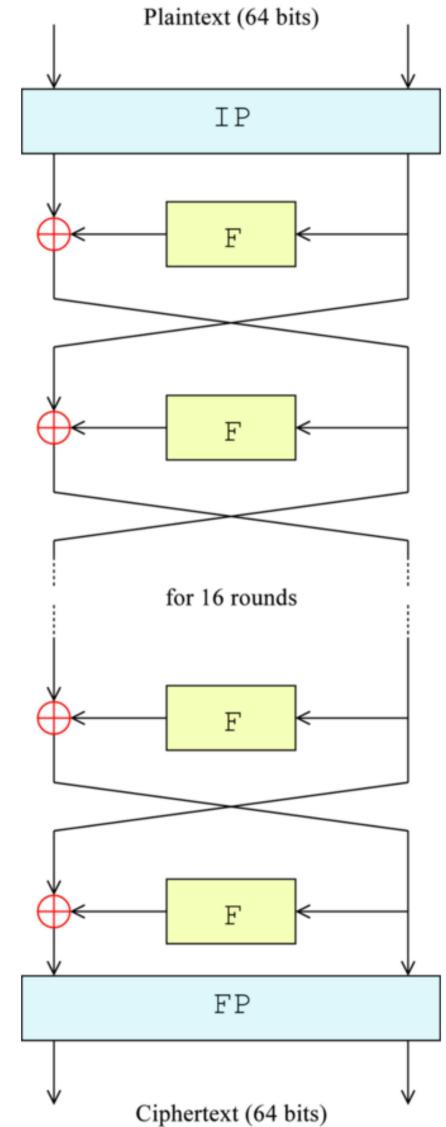
- Don't re-use nonces!
- AES-GCM, AES-CCM and ChaCha20-poly1305 are good modes
- AES-GCM is good for more capable devices/less-challenged situations
- AES-CCM is widely supported on less-capable devices
- ChaCha20-poly1305 is good if you don't have h/w support
- New AEAD modes will continue to be developed – don't use them for a few years after they're credible (same as all crypto!)

# Algorithm Internals

We'll look at some algorithm internals to get a feel for what's happening under the hood...

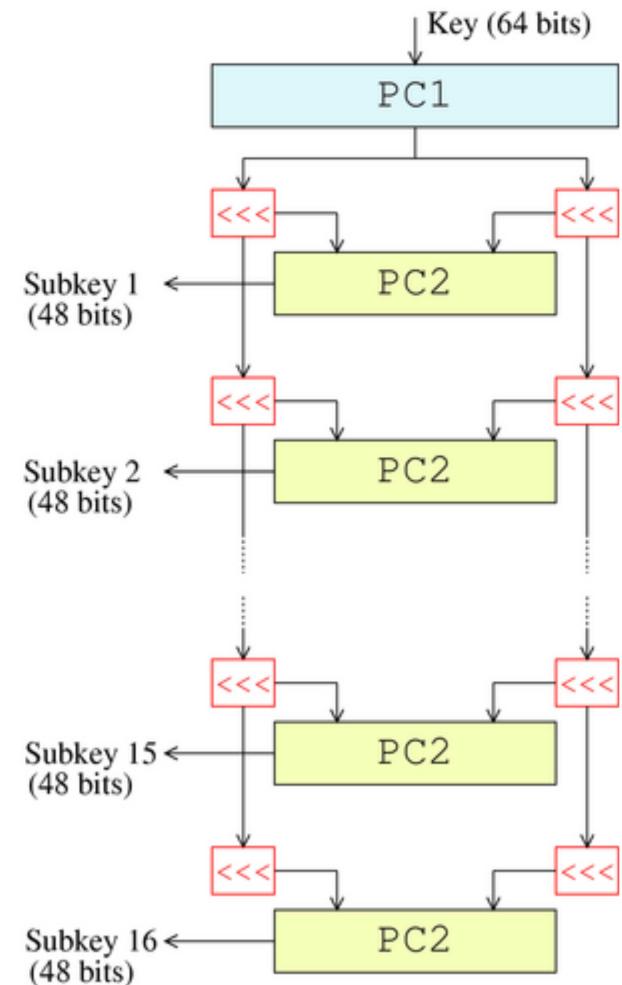
# DES

- 1970's era block-cipher...
- 56-bit key, 64-bit block
- 16 “rounds”
- Initial Permutation (IP) and final permutation (FP)
- “Feistel” function run in each round with 32 bits of (was-plaintext) input and 48 key bits
- Encryption -> go “down” the ladder
- Decryption -> go “up” the ladder

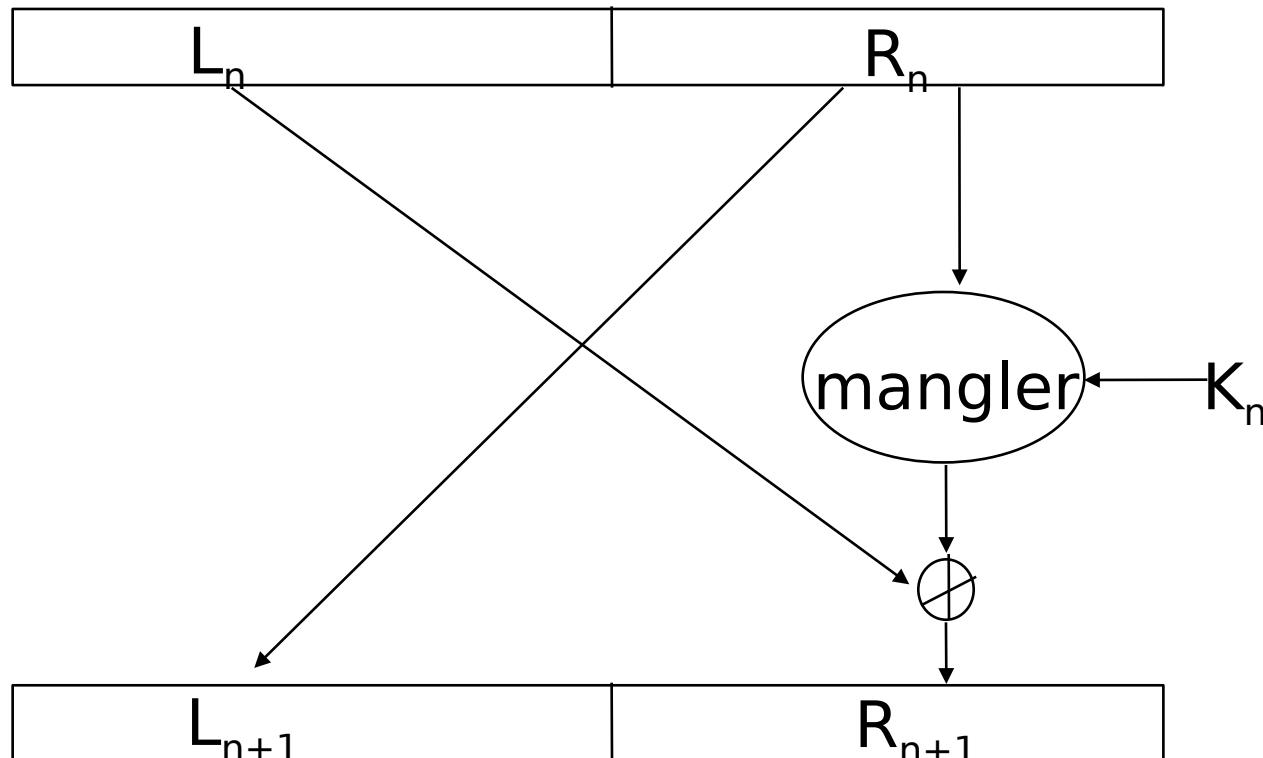


# DES Key Schedule

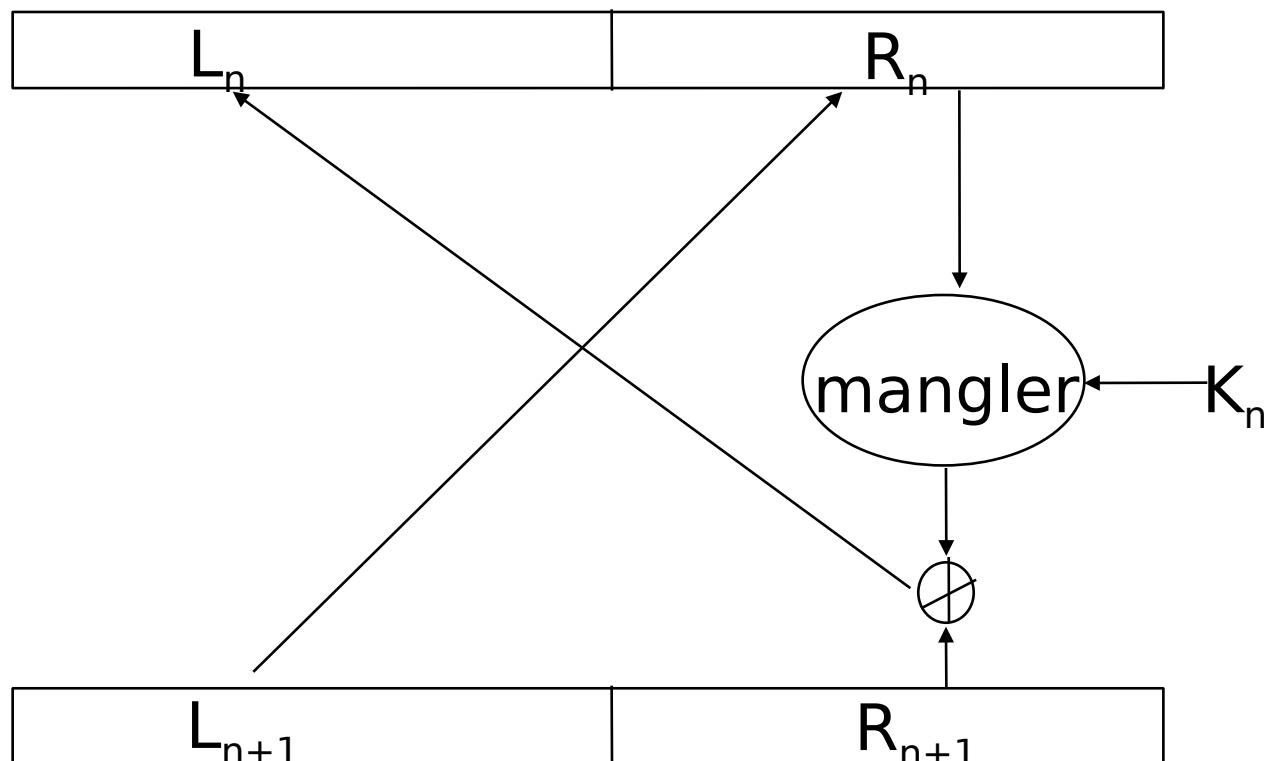
- Key Schedule: 56 key bits  $\rightarrow$  48 key bits used at each “round”
- Each time you use a new key, you have to calculate all 16 subkeys
- Key schedule calculation overhead could be significant, e.g. if you changed key for every packet



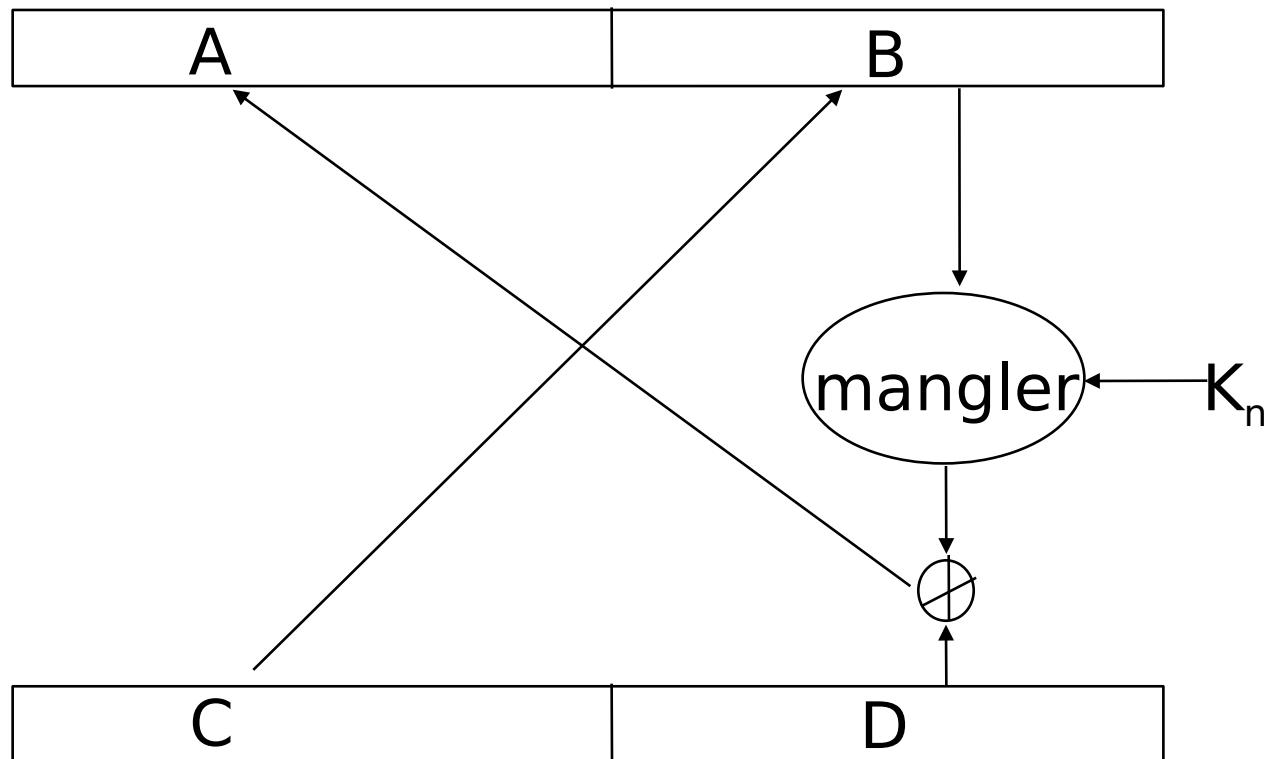
# Feistel Cipher Encryption



# Feistel Cipher Decryption



# Feistel Cipher Decryption



# Why Feistel

- So Mangler function doesn't need to be reversible
- DES is Feistel
- AES, IDEA are not. All functions are reversible.

# Triple DES (3DES)

- Purpose: expand key size from 56 bits (almost enough until recently) to >80 bits of “work”
- Why not double DES?
  - encrypt with  $K_1$  twice. How much more work (over DES) for good guys? Bad guys?
  - encrypt with  $K_1$  then  $K_2$ . What is time/memory for bad guys? Good guys?

# Triple DES (3DES)

- Defined as doing EDE with K1, K2, K3 (sometimes with K3==K1)
  - reason: because of “meet-in-the-middle” attack, 3DES is considered to only have time-strength equal to 112 bit key, not 168.
  - also, 112 bits was considered enough when 3DES was reasonable

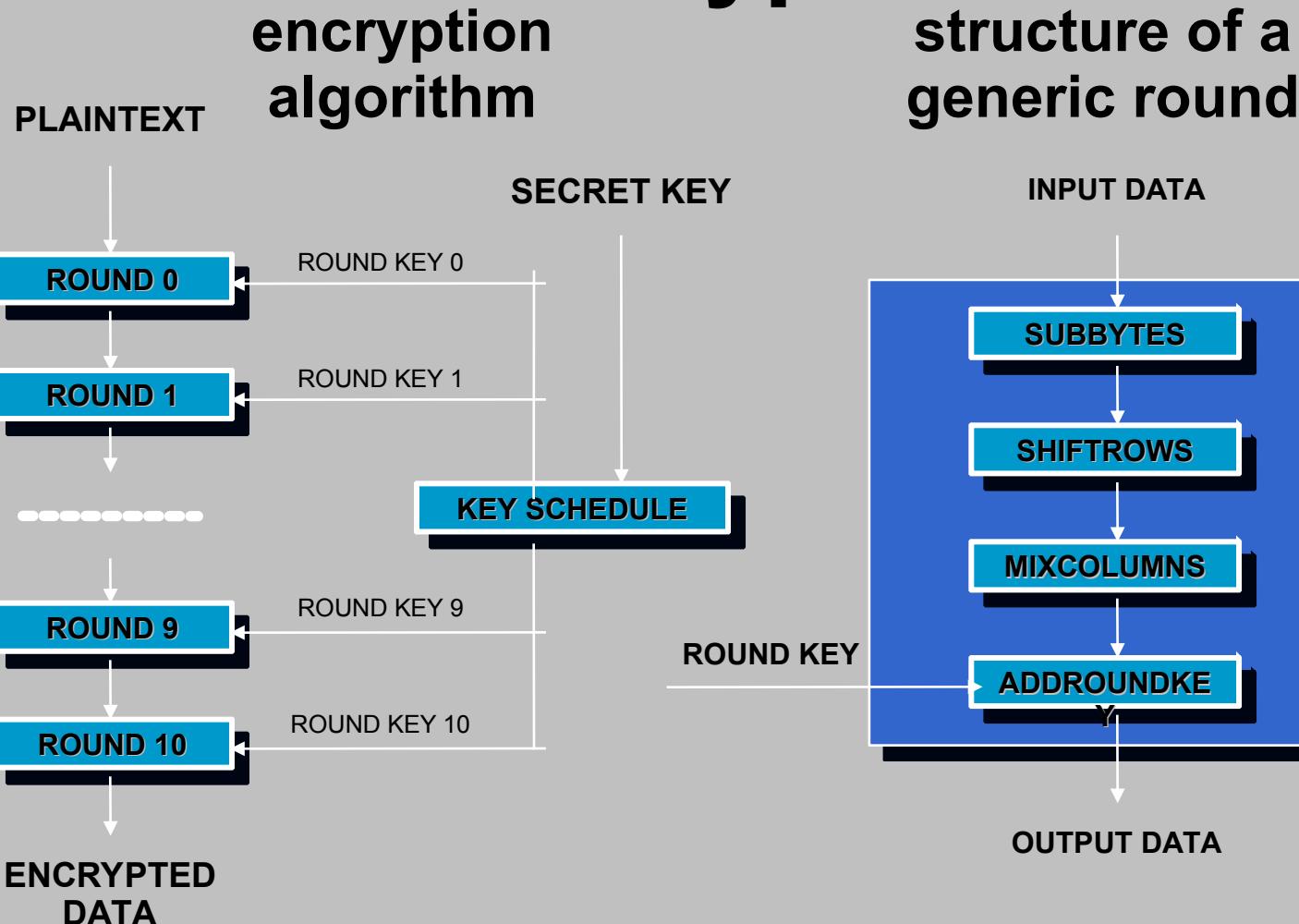
# Why EDE instead of EEE?

- Initial and final permutations would cancel each other out with EEE (minor advantage to EDE)
- EDE compatible with single DES if  $K1=K2=K3$ .

# AES Algorithm Description - General

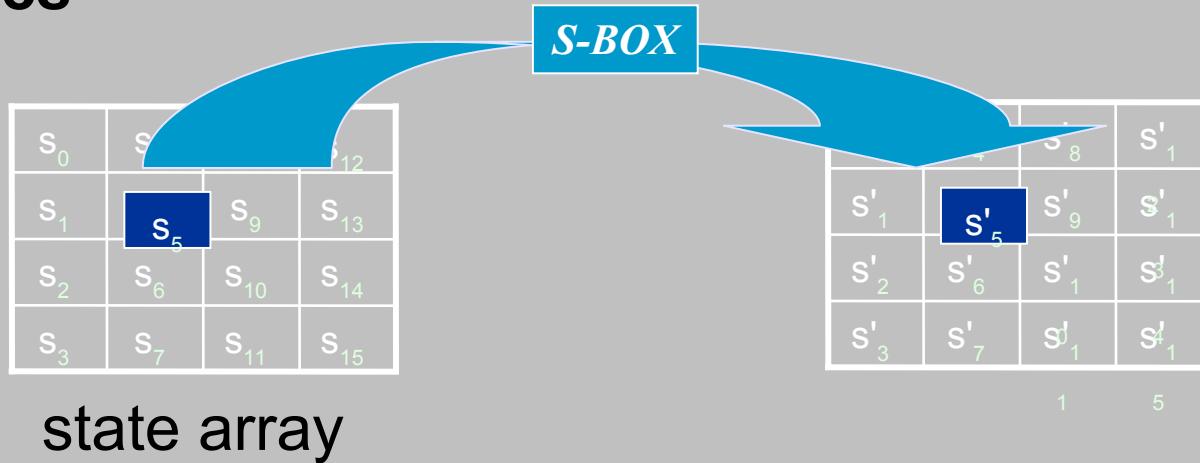
- Rijndael is the selected (NIST competition) algorithm for AES (Advanced Encryption Standard).
- It is a block cipher algorithm, operating on blocks of data.
- It needs a secret key, which is another block of data.
- Performs encryption and the inverse operation, decryption (using the same secret key).
- It reads an entire block of data, processes it in rounds and then outputs the encrypted (or decrypted) data.
- Each round is a sequence of four inner transformations.
- The AES standard specifies 128-bit data blocks and 128-bit, 192-bit or 256-bit secret keys.

# Algorithm Description – Encrypt.



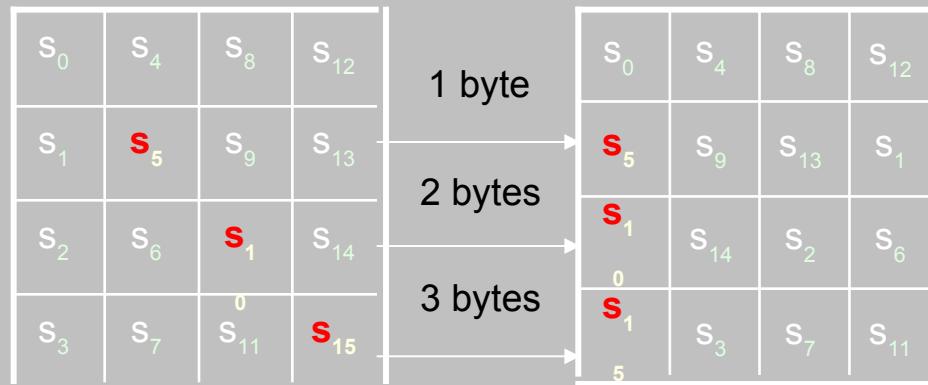
# Algorithm Description – Encrypt.

## SubBytes



## ShiftRows

state array



# Algorithm Description – Encrypt.

**MixColumns**

$s'_0$	$s'_4$	$s'_8$	$s'_1$
$s'_1$	$s'_5$	$s'_9$	$s'_1$
$s'_2$	$s'_6$	$s'_1$	$s'_1$
$s'_3$	$s'_7$	$s'_1$	$s'_1$

1            5

$$= \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \otimes$$

state array

$s_0$	$s_4$	$s_8$	$s_{12}$
$s_1$	$s_5$	$s_9$	$s_{13}$
$s_2$	$s_6$	$s_{10}$	$s_{14}$
$s_3$	$s_7$	$s_{11}$	$s_{15}$

**AddRoundKey**

$s'_0$	$s'_4$	$s'_8$	$s'_1$
$s'_1$	$s'_5$	$s'_9$	$s'_1$
$s'_2$	$s'_6$	$s'_1$	$s'_1$
$s'_3$	$s'_7$	$s'_1$	$s'_1$

1            5

$$= \begin{array}{l} \text{state array} \\ \oplus \end{array}$$

round key

$k_0$	$k_4$	$k_8$	$k_{12}$
$k_1$	$k_5$	$k_9$	$k_{13}$
$k_2$	$k_6$	$k_{10}$	$k_{14}$
$k_3$	$k_7$	$k_{11}$	$k_{15}$

# AES is the Secret Key Algorithm De-Jure

- Cute AES implementation as a spreadsheet
  - <https://www.nayuki.io/page/aes-cipher-internals-in-excel>
  - Intended for debugging, but gives a flavour of what happens internally when you change input bits
- Use AES-128 if at all possible esp. If your CPU has AES hardware instructions (many do)
  - If not, ChaCha20 is good
- AES-256 is generally overkill (though some dispute that)
  - Why?

# Rolling your own...

- Snakeoil from commercial enterprises is fairly common in this area
  - Usually they claim to have developed a revolutionary new algorithm
- Why should you not try to develop your own encryption algorithm?
- How should you debunk such claims?

# Even More Cryptography

*Hash and public key Algorithms.*

# Next hour...

- Hash and public-key algorithms

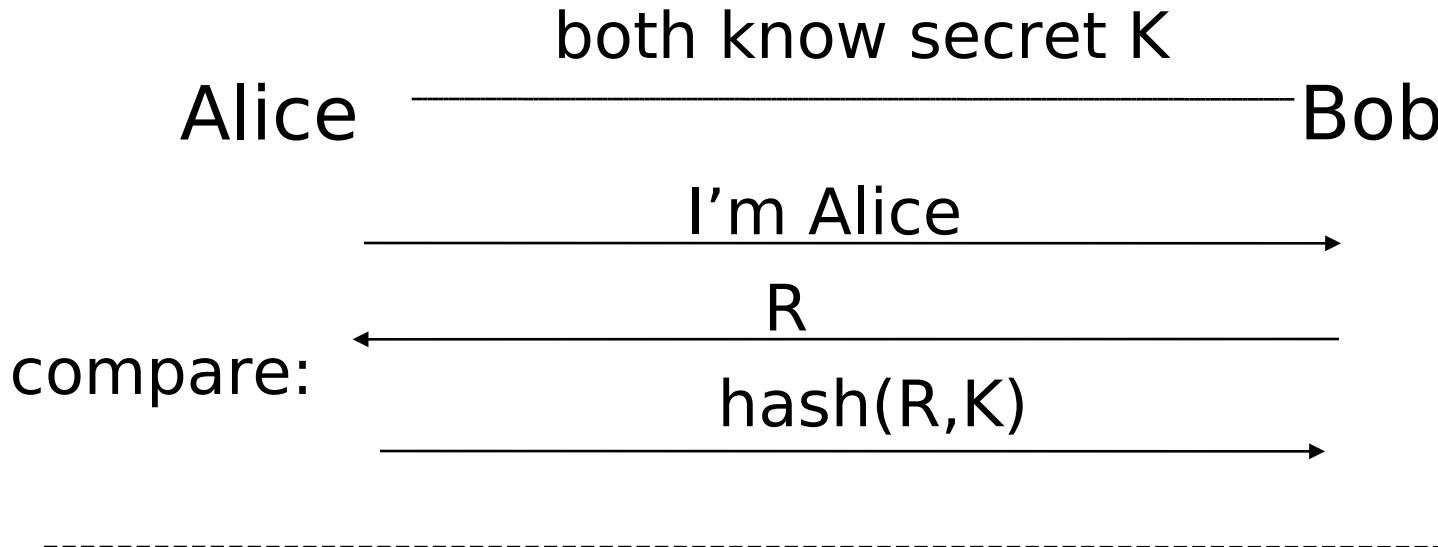
# Hash/Message Digest

- takes arbitrary sized input, generates fixed size output
- cryptographic hash/message digest
  - one-way (computationally infeasible to find input for a particular hash value)
  - collision-resistant (can't find two inputs that yield same hash)
  - output should look “random”

# Uses of Hashes

- Sign hash (digest) instead of message
- Store digests of files, to look for changes (e.g., viruses). (Tripwire does this) Why wouldn't CRC work?
- With secret, can do anything a secret key algorithm can do (authenticate, encrypt, integrity-protect)
  - Not all necessarily a good idea though

# Authentication with Hash



or:

I'm Alice, f(K,timestamp)?

# Hash function properties

**Pre-image resistance:** Given a hash value  $h$  it should be difficult to find any message  $m$  such that  $h = \text{hash}(m)$ . This concept is related to that of one-way function. Functions that lack this property are vulnerable to preimage attacks.

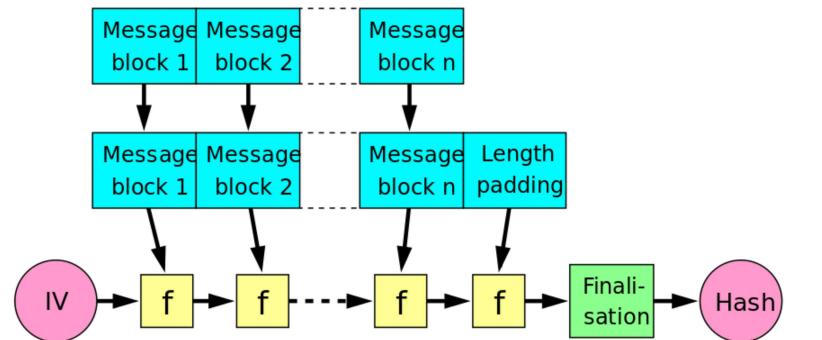
**Second pre-image resistance:** Given an input  $m_1$  it should be difficult to find different input  $m_2$  such that  $\text{hash}(m_1) = \text{hash}(m_2)$ . Functions that lack this property are vulnerable to second-preimage attacks.

**Collision resistance:** It should be difficult to find two different messages  $m_1$  and  $m_2$  such that  $\text{hash}(m_1) = \text{hash}(m_2)$ . Such a pair is called a cryptographic hash collision. This property is sometimes referred to as strong collision resistance. It requires a hash value at least twice as long as that required for preimage-resistance; otherwise collisions may be found by a birthday attack.

**Speed.**

# SHA-2

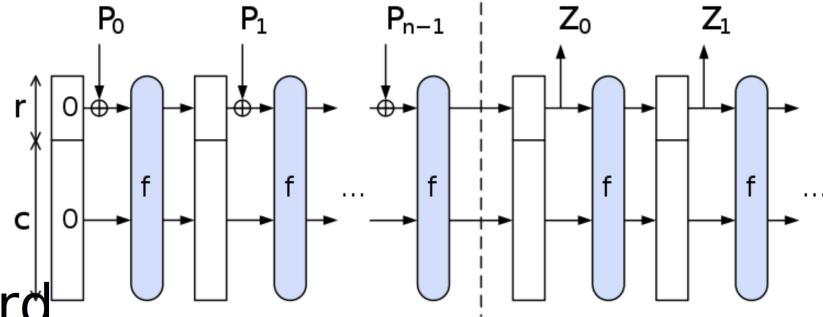
- Designed by NSA
- Merkle-Damgård design, same as MD5, SHA-1
- SHA-2 family output lengths 224, 256, 384 or 512
  - Spec and code: RFC 6234
  - SHA-256: 64 rounds, 256 bit output, compression function  $f$  is yet more complex bit fiddling
  - `Init()`; `update()`; `final()`; APIs



[https://en.wikipedia.org/wiki/File:Merkle-Damgard\\_hash\\_big.svg](https://en.wikipedia.org/wiki/File:Merkle-Damgard_hash_big.svg)

# SHA-3/Keccak

<https://en.wikipedia.org/wiki/File:SpongeConstruction.svg>



- Designed by Keccak team for NIST competition (like AES)
- <https://keccak.team/index.html>
- Sponge design, not Merkle-Damgård
- Block transformation function 'f' uses xor, not and and operations
- Many variations (sigh): SHA-3-224, SHA-3-256, SHA-3-384, and SHA-3-512, SHAKE128, SHAKE256
- Slower than SHA-256 (or SHA-2-256;-) in s/w
- Not in widespread use; guessing: SHA-3-512 will get used

Loadsa slides: <https://csrc.nist.gov/csrc/media/projects/hash-functions/documents/keccak-slides-at-nist.pdf>

# Salt

- Protects a database of hashed passwords
- Salt is non-secret, different for each user
- Store  $\text{hash}(\text{pwd}, \text{salt})$
- Users with same pwd have different hashes
- Prevents intruder from computing hash of a dictionary, and comparing against all users
- Password hashing is not the same as hashing
  - Argon2!

# Properties of password hashing function ‘f’?

Do we want ‘f’ to be speedy?

Do we want ‘f’ to use as little memory as possible?

Do we want ‘f’ to be easy to parallelise?

‘f’ is often called a password hashing algorithm but is really a verifier, the password hashing alg produces pv given p and s (and sometimes u)

It’s ok to not be terminologically pure:-)

See <https://password-hashing.net/> for much more on a recent competition related to that kind of function

# MACs with hashes

- Combine message with key and digest that
- Collision resistance isn't important here.  
(why?)

# Possible problem

- If do  $\text{hash}(\text{key} \mid \text{message})$  and use entire result as MAC, many hash algorithms have an understandable issue
  - Hash continuation for speed
  - Most hash algorithms can continue from where they left off
  - So even if you didn't know the key, if you knew  $\text{hash}(\text{key} \mid \text{message})$  you could continue
- HMAC (RFC 2104) proven not to have this problem but HMAC is a bit more work
- $\text{HMAC-H(K, text)} = \text{H}(\text{K XOR opad}, \text{H}(\text{K XOR ipad}, \text{text}))$ 
  - Where 'H' is e.g. SHA-1 or SHA-256

# If others, why HMAC

- HMAC comes with a **proof**
  - assuming underlying function is
    - collision resistant
    - if attacker doesn't know  $K$ , cannot compute proper  $\text{digest}(K,x)$  even if sees arbitrary  $(y,\text{digest}(K,y))$
- Others likely just as secure, but no “proof”

# Cryptographic proofs

- Most cryptographers prefer to base their work on results that have mathematic proofs
- Usually those end up depending on some hopefully reasonable underlying assumption, e.g. that discrete logarithms are hard.
- Until recently the set of assertions for which we had proofs was fairly limited, mostly about internals of algorithms and not covering how those were used
  - That's starting to improve (see TLS1.3 materials later)
  - Bottom line: basing your work on algorithms with associated proofs is usually better, though is no guarantee of security nor of correctness as implementation flaws are far more common than algorithm design flaws

# How Public Key Algorithms Work

- We want an algorithm with the following properties:
  - two different numbers:  $e$  and  $d$
  - $e$  and  $d$  are inverses; using one reverses the effect of the other
  - you shouldn't be able to compute  $d$  from  $e$
  - if must be efficient to find a matching pair of keys
  - it must be efficient to encrypt and decrypt

# Example (Insecure) Public Key Algorithm

- Multiplication modulo p (where p is a prime)
- For example, let  $p=127$
- Choose e and d so that  $e*d \equiv 1 \pmod{127}$ 
  - e.g.  $e=53$  and  $d=12$
- To encrypt a number, multiply by 53 mod 127
- To decrypt a number, multiply by 12 mod 127
- Decryption must restore the initial value!

# Why Isn't This Secure?

- The number 127 is too small. You could compute  $d$  from  $e$  by trying all possible values
- Modular division is possible - the inverse can be computed quickly even when  $p$  is large (Euclid's algorithm...patent long expired)

# Multi-precision arithmetic

- Asymmetric crypto depends on multi-precision arithmetic, so... A very, very, very brief primer...
- How long is the product of 2 32-bit numbers?
- How long is the product of 2 1024-bit numbers?
- If  $X, Y$  and  $P$  are 1024-bits long, how long is:
  - $(X * Y) \bmod P$ ?
  - $(X ^ Y) \bmod P$ ?
- Does calculating  $(X ^ Y) \bmod P$  require  $Y$  multiplications?

# A Summary of RSA

- Named after its inventors: Rivest, Shamir, and Adelman
- Uses modular exponentiation
- Choose a modulus  $n$  and a public exponent  $e$
- The modulus  $n$  is chosen as the product of two primes:  $p,q$
- We depend on factoring  $n$  being “hard”
- Public key encryption is:  
$$\text{ciphertext} = \text{plaintext}^e \bmod n$$
- Public key decryption is:  
$$\text{plaintext} = \text{ciphertext}^d \bmod n$$

# So how RSA works...

- Define  $\phi(n)$  to be the # of integers  $< n$  and relatively prime to  $n$
- Euler proved:  $x^{\phi(n)} \text{ mod } n = 1$
- So  $x^{k\phi(n)} \text{ mod } n = 1$  and  $x^{k\phi(n)+1} \text{ mod } n = x$
- If we can find  $d \cdot e = 1 \text{ mod } \phi(n)$ , they'd be “exponentiative inverses”
- Because:
  - $C = M^e$
  - $C^d = M^{e \cdot d} = M^{(e \cdot d \text{ mod } \phi(n))} = M$

# Computing $\phi(n)$

If  $p$  is a prime,  $\phi(p) = p-1$

If  $n=p \cdot q$  ( $p, q$  primes),  $\phi(n)=(p-1)(q-1)$  (remove multiples of  $p$  and multiples of  $q$ )

Given  $e$ ,  $\phi(n)$ , Euclid's algorithm (aka long division) allows us to compute  $d$  such that

$$d \cdot e = 1 \pmod{\phi(n)}$$

So, we can find  $d$  from  $e$  if we know  $\phi(n)$

We need to know how to factor  $n$  in order to know  $\phi(n)$

# Why do we think RSA is secure?

- If you can find  $d$  from  $e$ , why can't someone else?
- Factoring large numbers is hard (as far as we know)
- Finding  $d$  from  $e$  is easy if you can factor  $n$ , but it's hard if you can't
- Pick two large primes and multiply them together to get  $n$ . You can now factor  $n$  because you constructed  $n$
- After computing  $d$  from  $e$ , you can forget the factors of  $n$  (though in practice you don't)

# How to Find Large Primes

- If factoring is hard, how do you find large primes?
- It turns out you can test a number for primality easily even though factoring is hard!
- Pick random large numbers and test them until you find a prime one

# How do you test for primality?

- Fermat's theorem (note: Fermat was born 100 years earlier than Euler..it's a special case of Euler's theorem)  
 $x^{p-1} \text{ mod } p = 1$  if  $p$  prime
- So to test if  $n$  is a prime, pick  $x < p$  and raise  $x$  to the power of  $p-1$ . If the answer is not 1,  $n$  is definitely not prime
- But can the result be 1 even if  $n$  not prime? Yes, but probably not
- Worst case:  $1/4$  false positives, actually about  $10^{-4}$
- Iterate with different  $x$ 's (may 10 or 20) until satisfied

# Doing exponentiation

- Can't multiply something by itself a gazillion google times!
- Solution: Repeated squaring
- Result: a 512 bit exponent requires between 512 and 1024 multiplications (depending on the number of 1's in the number) (instead of a trillion trillion trillion google multiplications)

# Optimizing Public Key ops

- Turns out RSA secure even if  $e$  in  $(e,n)$  is small (like 3 or  $2^{16}+1$ )
- Have to be somewhat careful if  $e=3$ 
  - if  $m$  is smaller than cube root of  $n$
  - if send same message to 3 people, all using  $e=3$ 
    - know public keys  $(3,n_1)$ ,  $(3,n_2)$ ,  $(3,n_3)$
    - know  $m^3 \bmod n_1$ ,  $n_2$ ,  $n_3$
    - By CRT, can compute  $m^3 \bmod n_1 * n_2 * n_3$
    - Then just take cube root ( $n$  is smaller than each of the moduli so its cube will be less than  $n_1 * n_2 * n_3$ )

# More problems with $e=3$

- 3 must be relatively prime to  $\phi(n)$
- How do we ensure this?
- Why are these problems not an issue with  $e=2^{16}+1$ ?

# Optimizing Private Key ops

- Use Chinese Remainder Theorem (CRT) and do arithmetic mod p and mod q, then combine
- precompute what d is mod p and mod q
- precompute  $p^{-1} \bmod q$

# Other arcane RSA threats (Standards avoid these)

- If you just encrypt a guessable plaintext, eavesdropper can verify a guess
- Trivial to forge a signature if you don't care what you're signing
- Smooth numbers (sign msg combo of other messages already signed)
- If pad on right with random data, someone can choose padding such that msg will be perfect cube

# Smooth threat

- Suppose you see signature on  $m_1$  and on  $m_2$ . What's the sig on  $m_1 * m_2$ ;  $m_1/m_2$ ;  $m_1^{-1}$ ?
- Suppose you can factor the m's, and see lots of signatures (like tens of thousands). Various combinations are likely to give you signatures on various primes, and then you can sign anything which has just those primes as factors

# An Intuition for Diffie-Hellman

- Allows two individuals to agree on a secret key, even though they can only communicate in public
- Alice chooses a private number and from that calculates a public number
- Bob does the same
- Each can use the other's public number and their own private number to compute the same secret
- An eavesdropper can't reproduce it

# Why is D-H Secure?

- We assume the following is hard:
  - Given  $g$ ,  $p$ , and  $g^x \text{ mod } p$ , what is  $X$ ?
  - That's the discrete log problem
- With the best known mathematical techniques, this is somewhat harder than factoring a composite of the same magnitude as  $p$
- Subtlety: they haven't proven that the algorithms are as hard to break as the underlying problem

# Diffie-Hellman

Alice

agree on  $g, p$

Bob

choose random A

choose random B

$$T_A = g^A \bmod p$$



$$T_B = g^B \bmod p$$

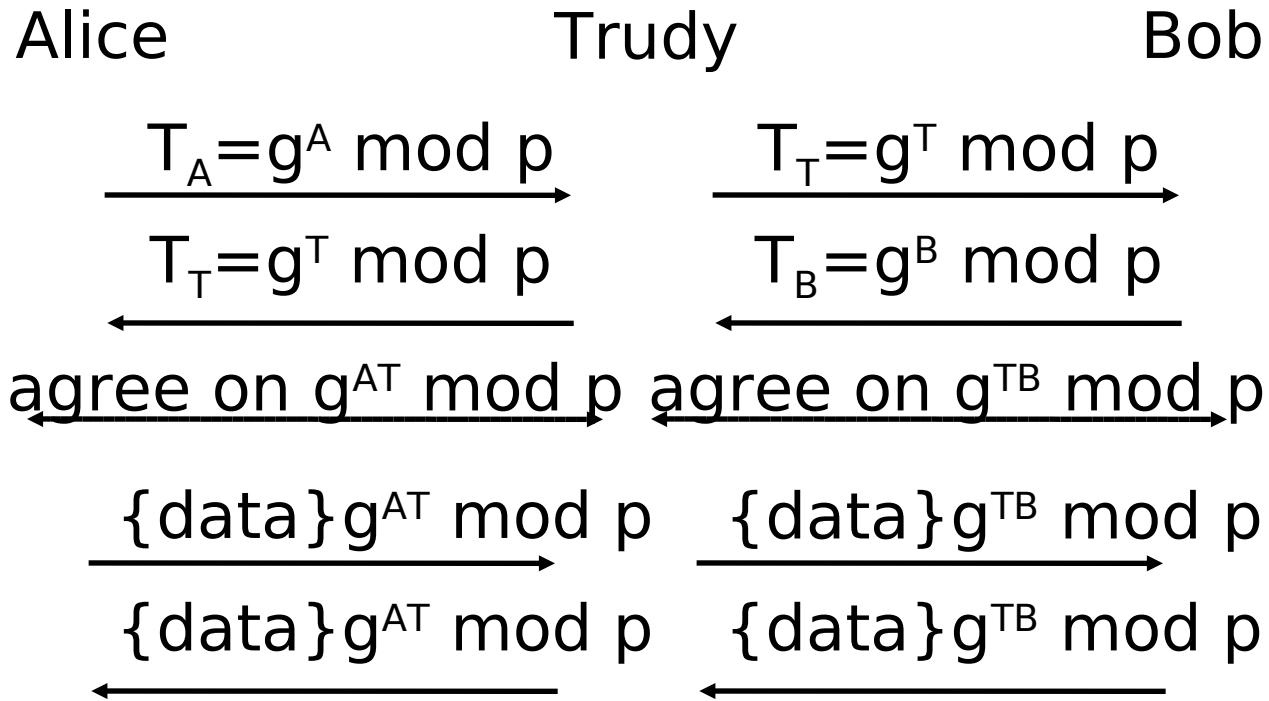


compute  $T_B^A$

compute  $T_A^B$

agree on  $g^{AB} \bmod p$

# Man in the Middle



# Signed Diffie-Hellman (Avoiding Man in the Middle)

Alice

Bob

choose random A

choose random B

$[T_A = g^A \text{ mod } p]$  signed with Alice's Private Key

$[T_B = g^B \text{ mod } p]$  signed with Bob's Private Key

verify Bob's signature

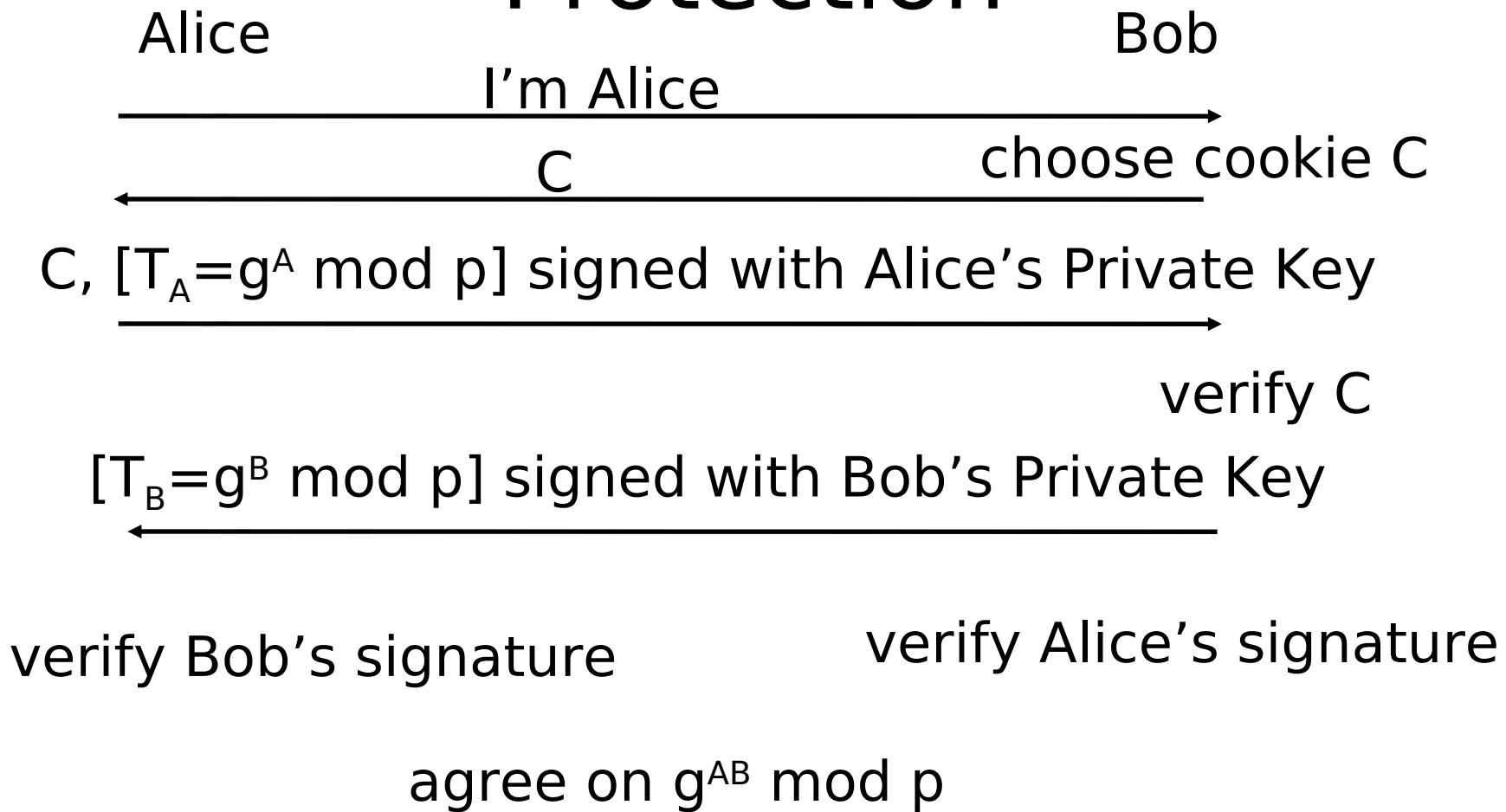
verify Alice's signature

agree on  $g^{AB} \text{ mod } p$

# If you have keys, why do D-H?

- “Perfect Forward Secrecy” (PFS)
  - Prevents me from decrypting a conversation even if I break into both parties after it ends (or if private key is escrowed)
- Ex. non-PFS:
  - A chooses key S, encrypts it with B’s RSA public key and sends it to B
- Today we strongly encourage PFS in Internet security protocols (see later wrt snowdonia)

# Cookie Mechanism: Some Denial of Service Protection



# Stateless Cookies

- It would be nice if Bob not only can avoid doing computation, but can avoid using any state until he knows the other side is sending from a reasonable IP address
- A “stateless” cookie is one Bob can verify without maintaining per connection state
- For instance, Bob can have a secret  $S$ , and  $\text{cookie} = \{\text{IP address}\}S$

# Diffie-Hellman for Encryption

Alice

Bob

choose random A

compute  $T_A = g^A \text{ mod } p$

compute  $T_B^A$

encrypt message using  $g^{AB} \text{ mod } p$

choose  $g, p$

choose random B

publish  $g, p, T_B = g^B \text{ mod } p$

send  $T_A$ , encrypted msg

compute  $T_A^B$

decrypt message using  $g^{AB} \text{ mod } p$

# Flavours of D-H

- Above is called ephemeral-static D-H
  - Alice's value ("A") is ephemeral, and Bob's ("B") is static/published
- What would be the consequences of a static-static scheme?
  - Hint: think about groups

# Strong Password Protocols

*...patents can be bad too*

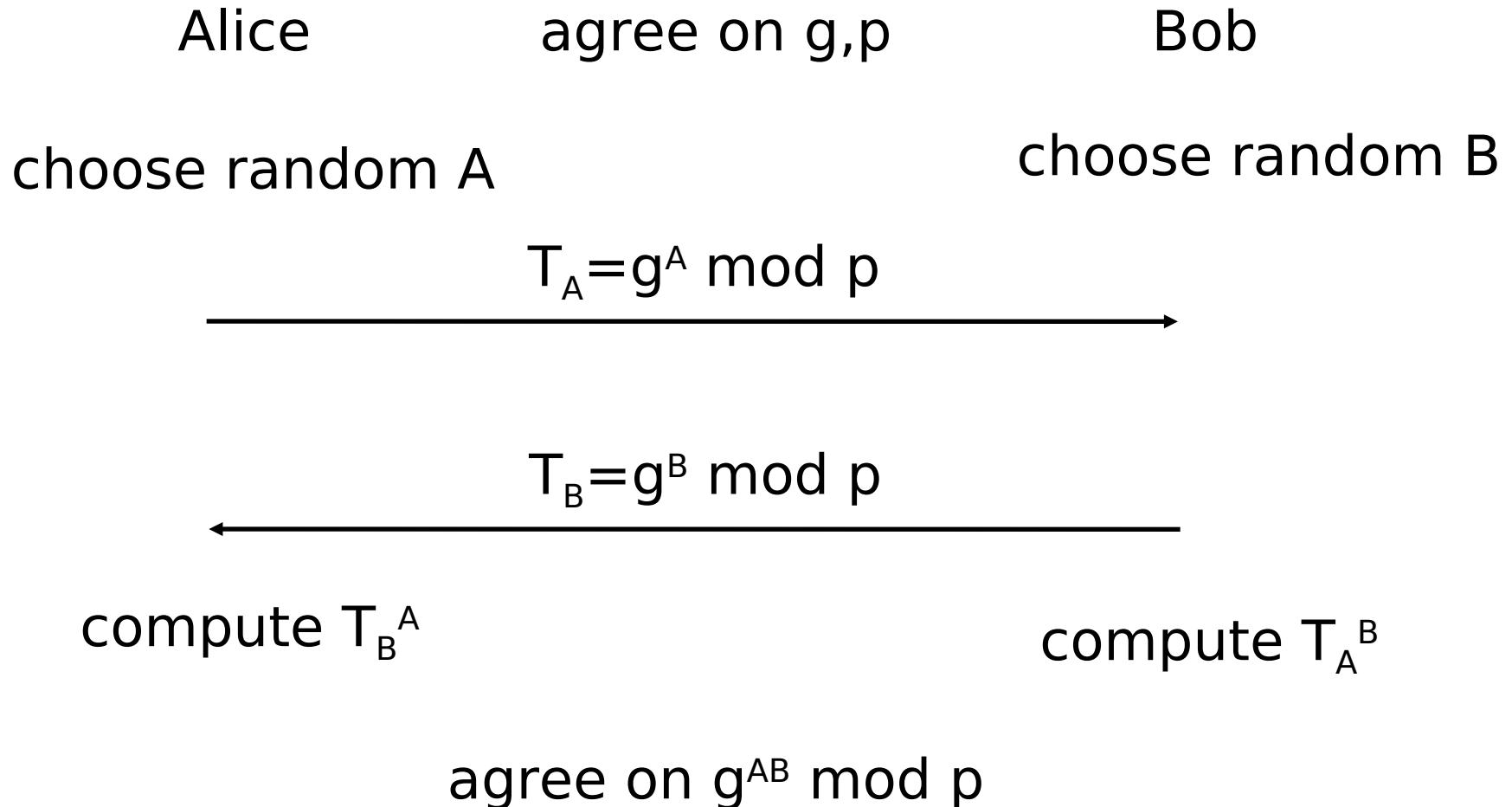
# This hour

- Strong password protocols
  - Sometimes called “Password Authenticated Key Exchange” (PAKE) protocols
  - EKE, SPEKE,....
- System considerations
- ACK: Some stolen slides! (From Radia Perlman’s presentation at the Summer ’05 IETF)
  - <http://www3.ietf.org/proceedings/05aug/slides/sacred-3/sld1.htm>

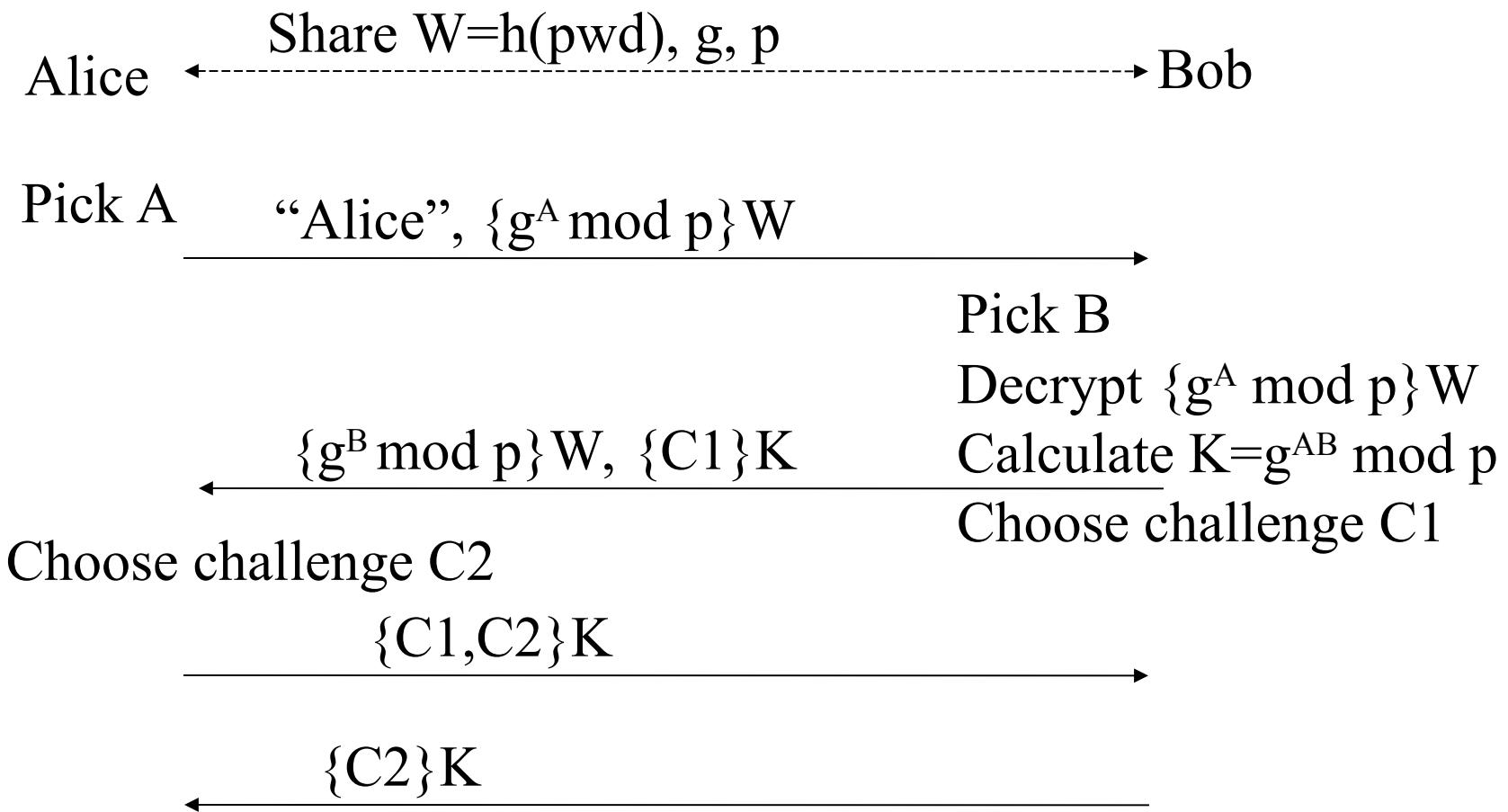
# Basic Problem

- Using a password as a key in any protocol is vulnerable to a dictionary attack so long as there is any structure detectable in the data protected with the password-as-key
- Unavoidable?
  - Think D-H?

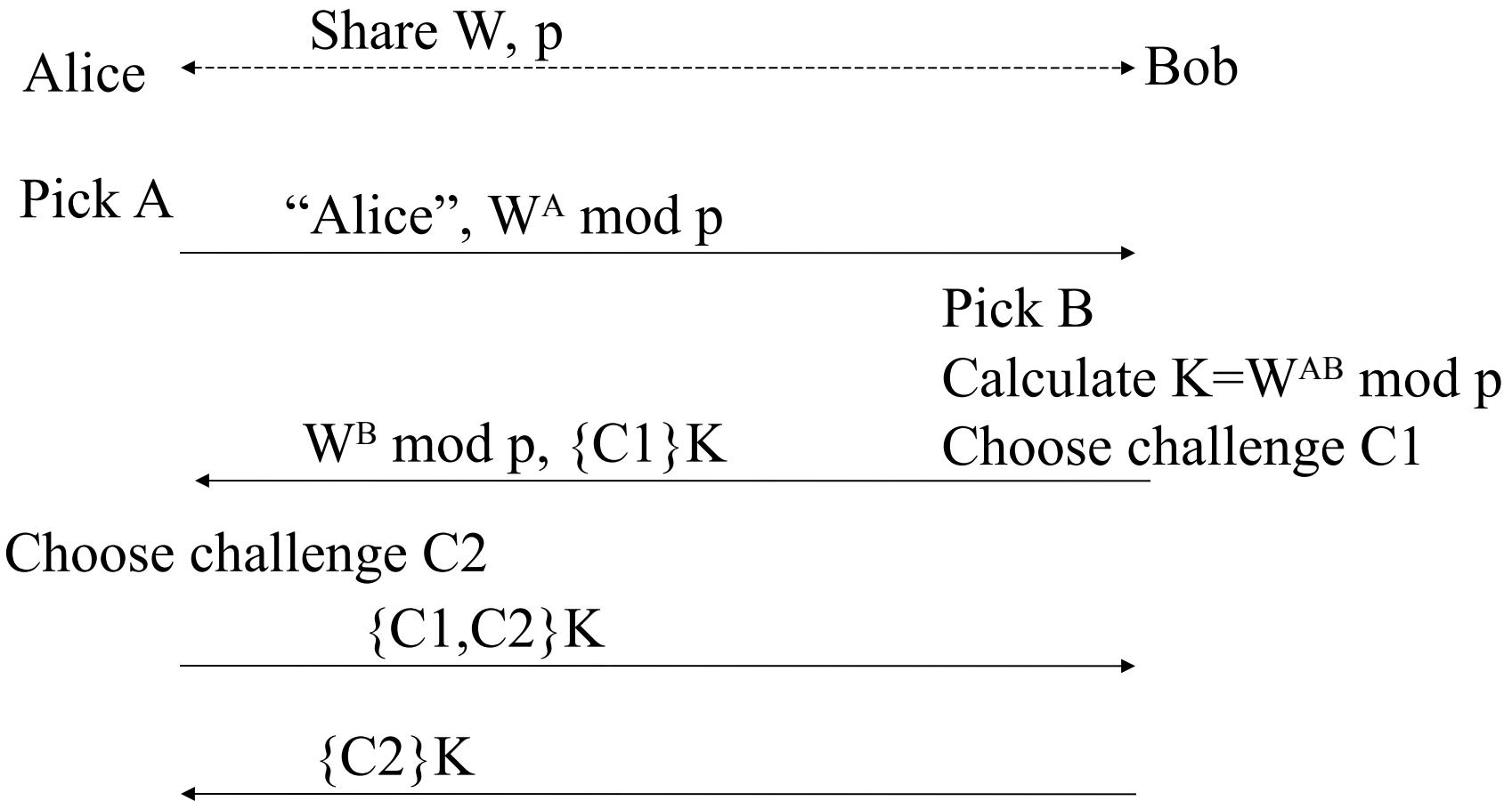
# Diffie-Hellman



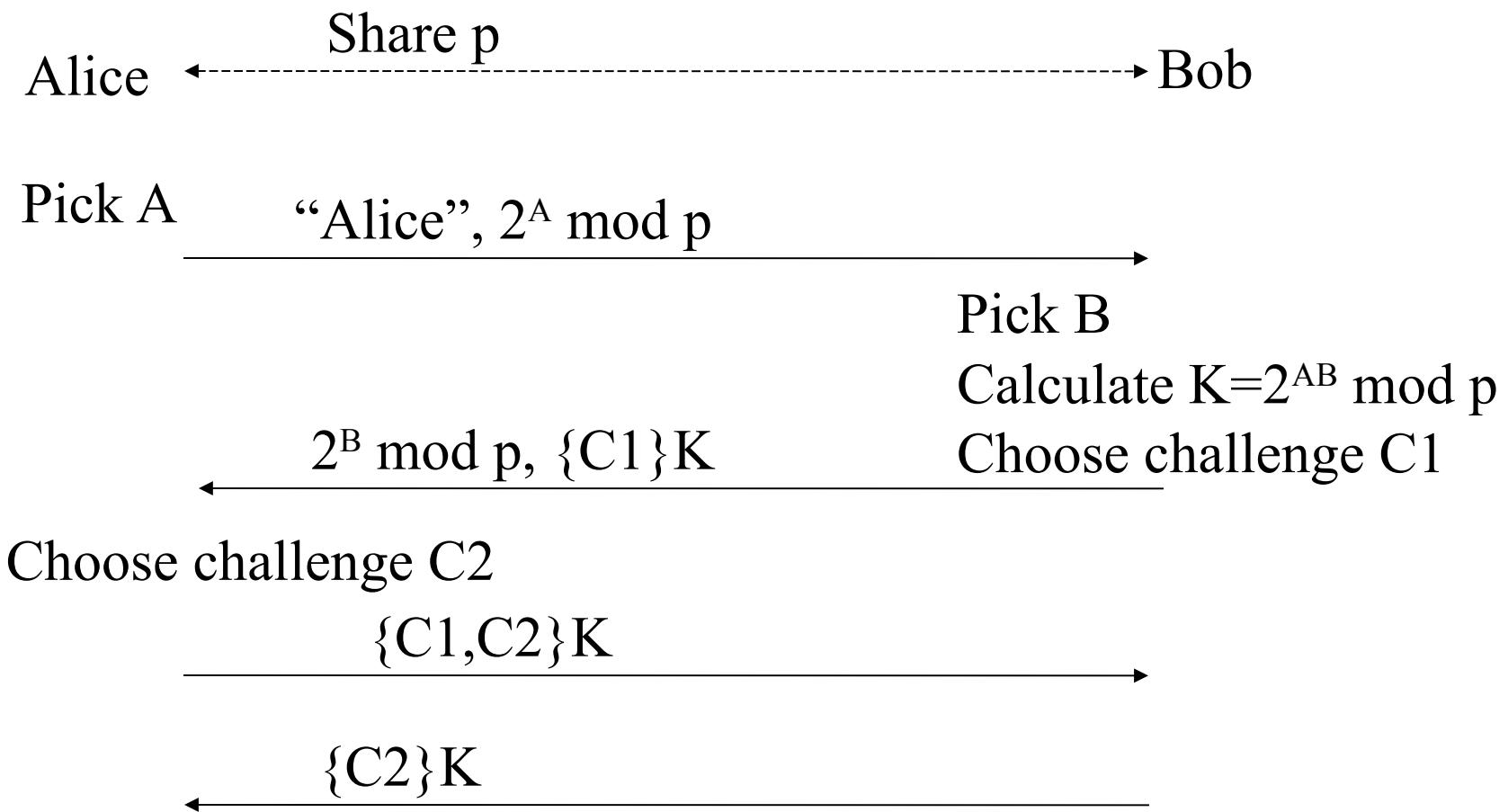
# EKE (designed for mutual authentication)



# SPEKE



# PDM (Password Derived Moduli)



# System Considerations

- Each of the protocols requires some shared information in addition to the password (e.g. EKE requires  $g$  &  $p$ )
- In all cases, this is not human memorable information
  - So some configuration or hard-coded values are required
- People also cannot play the protocols themselves
  - So some s/w is needed

# Possible Solutions

- Client side:
  - User uses standard PC
    - Install client code and configuration
  - Users uses token or smart card
    - Token/smart card contains configuration and code
  - Issues?
- Server side
  - Database and s/w
- What about account management?

# Strong Password Protocol Conclusions

- These protocols would probably be more widely used were it not for the IPR situation
  - EKE patent (held by Lucent) U.S. Patent 5,241,599 expired in 2011
  - Other IPR does exist in this space
  - PAKEs still considered damaged goods by some though
- Some others think there's a useful role here for new device registration, e.g. into a public key infrastructure or symmetric key based system

# ECC

- Nice blog from Nick Sullivan  
<https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/>
- Elliptic curve, e.g:  $y^2 = x^3 + ax + b$ 
  - $x, y$  can be Real or members of finite groups ( $F_p$  or  $F_{2^m}$ )
- Points on curve form a group
  - Group operation: addition  $P+P=2P$  etc.
- Hard problem: EC discrete log problem
  - Given  $P, Q$  on curve find  $k$  s.t.  $P=kQ$

# ECC - 2

- Certicom IPR on *lots* of implementation related things (e.g. point compression)
  - RFC 6090 based only on *old* references
  - People finally seem ok with IPR now
- Keys/signatures are smaller
  - AES 128 ~ RSA 3072 ~ ECC 256
- Run times
  - Public/private operations same
  - In the “middle” compared to RSA enc/dec
- Starting to displace RSA and a lot of integer D-H

# ECC – 3

- Recently ECC deployment is increasing a lot
- Doubts about NIST Curves (P256) are a part of that but not so much among those who understand stuff
  - Non-deterministic signature schemes are worst aspect really
- Mostly this is down to performance
- Curve25519 and Ed25519 are 4x quicker
- Curve448 and Ed448 are “turn it up to 11” settings
- References:
  - RFC 7638 (basic curve definitions)
  - RFC 8032 (EdDSA)

# ECDSA Signing

Sony’s “oops” in 2010:

[https://events.ccc.de/congress/2010/Fahrplan/  
attachments/  
1780\\_27c3\\_console\\_hacking\\_2010.pdf](https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf)

- local copy in “materials”

GOTO slide 122

- explains ECDSA signing and issues with non-deterministic signatures
- deterministic signatures: RFC 6979

Better mitigation: Use Ed25519 (RFC 8032)

# Key Rotation

- N-squared issue mentioned already – a KDC or CA or equivalent is needed
- Symmetric algorithms all have a cryptographic limit to the number of times you should use one key
  - A very large number, but very fast comms channels exist
- Deployments however are vulnerable, so keys can leak out, e.g. on discarded hard disks, via ex-employees, backups, ...
- Some (dim) vendors still ship products with hard-coded keys, even today
- So you always want to rotate keys periodically, for **every** kind of key used in a system
- And you need to build this in from the start, or the chances are very high that it won't happen (so don't depend on any manual interaction to rotate keys!)
- This can become visible to the application layer, e.g. if you have some redundancy and use session-tickets; that's a PITA, but you MUST handle it

# Odds and Ends: 2 slides each

- Identity Based Cryptography (IBC/IBE)
- Secret Sharing
- NUMS

# IBC

- Idea: Just use someone's name as the (public) key to encrypt something to them
- Can work if there's a trusted (by the recipient) key generator where the sender has (a high integrity copy of) the KG's public value
- Almost all schemes require that KG can decrypt all traffic (mandatory key escrow)
- Functionally like RSA group-keys
- Truth-in-advertising: you absolutely need more than just identities!

# IBC - 2

- IPR again: Voltage
- Interesting math behind this: pairing based cryptography
  - <http://www.larc.usp.br/~pbarreto/pblounge.html>
  - Discrete log again, and D-H but with two groups for which there's a bilinear mapping
- Not really mature enough (IMO) and encumbered up the wazoo so more of academic interest for now

# Secret Sharing

- What if I have a secret to share out among N entities so that any k of them can reconstruct the secret
- Useful for ways to distribute things amongst partially trusted entities
  - Long term secrets
  - High availability solutions
  - P2P schemes

# Shamir Secret Sharing

<http://szabo.best.vwh.net/secret.html>

- Idea:
  - Geometry!
    - Knowing 2 points => know the line
    - 3 points => know the quadratic
    - $k$  points => know the  $x^k$  polynomial
  - Distribute  $N$  points of an  $x^k$  polynomial

# Nothing Up My Sleeve

In many crypto schemes if you can mess about with crypto parameters, then you can break many things

Examples: bad D-H parameters, IBE, Dual-EC

The absence of a detailed explanation for how the NIST curves (p256 etc) were generated has generated controversy

One approach is to reduce the wriggle-room available to designers is to pick (few) known constants (e.g. Pi, SQRT(2)) and derive “free” parameters from those

Surprisingly even that generated a big argument

Not a bad design pattern though

# (Post-)Quantum Stuff

I'll use Kenny Paterson's slides from a recent IETF meeting:

<https://datatracker.ietf.org/meeting/99/materials/slides-99-saag-post-quantum-cryptography>

- URL fix for that:  
<https://www.ncsc.gov.uk/whitepaper/quantum-key-distribution>
- NIST competition round 2 has 26 candidates, announced Jan 2019  
<https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions>

# (Post-)Quantum Stuff

Important note about hash-based signature schemes:

- Yes, they are quantum resistant, so worth exploring, esp for things like software-signing where we won't need too many signatures and sizes aren't a big deal
- BUT they need a different cryptographic API as those private keys have a fixed, limited number of uses allowed, (maybe  $2^{10}$  or  $2^{20}$ ) and if you go over that limit, you lose all security (signatures could be forged)
- Also, if you EVER re-use a private key you're hosed, so you need to be very very careful e.g. about reboots and state maintenance
- So you can't just replace Ed25519 with XMSS or LMS without changes to applications/systems
- XMSS: RFC 8391
- LMS: <https://tools.ietf.org/html/draft-mcgrew-hash-sigs-15>

# Embed PQ exchanges with classic DH

- Likely TLS, IPsec and other “classic” key exchanges will be extended to allow mixing in of PQ key exchanges **in addition** to the classic DH exchange
- This may involve betting on which algorithms you think may “win” the NIST competition (or just picking those you like)
- One VPN operator I chatted with recently is interested in embedding 3 different PQ key exchanges into wireguard (an IPsec alternative) as a hedge against future quantum attacks
- Efficiency and scaling issues are tricky and not worked out yet really

# Crypto Overall Summary

- Don't invent stuff
- Algs de-jour: sha-256, aes-128, rsa-2048 or longer for long-term apps
- RSA key transport being displaced by ECDH esp. P256 and curve25519
- RSA signatures still rule the roost for authentication
- Code/libraries exist: use them (properly!)
- Key management is required and hardly ever easy
- Don't invent stuff