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
- A few odd-and-ends (NUMS, PQ, ...)

Basics...

- Secret-key, public-key and message digest (hash) algorithms
- Encryption, decryption and integrity protection
- Key managment 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 imporant 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 (pesudo) “random” bits

Fashion items we'll mostly ignore

- Identity-Based Encryption (IBE/IBC) is a new(ish) variety of public key encryption
- Partly or FullyHomomorphic Encryption (FHE) algorithms are elegant but still impractical
- Blockchain to do <stuff that doesn't need it>

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)

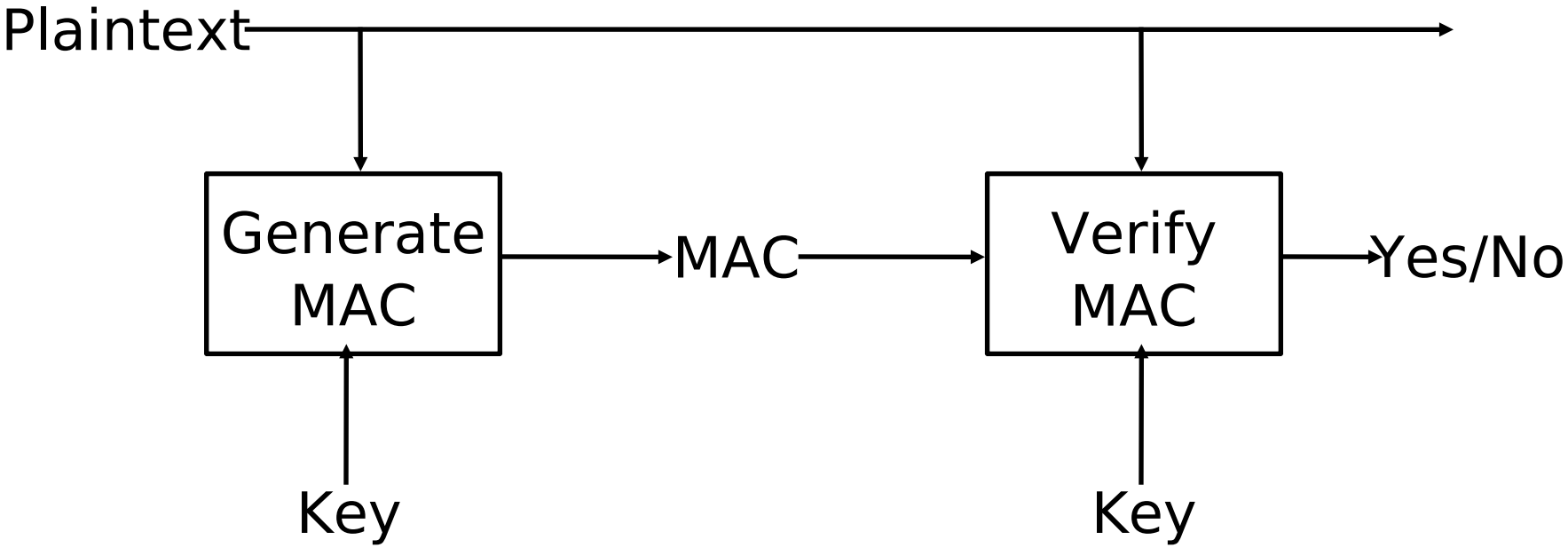
To Publish or Not to Publish

- 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

Main 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)

I'm Alice →

Pick Random R
Encrypt R using K
(getting C)

← If you're Alice, decrypt C

R →

Ancient (pre-1990's) Secret Key Algorithms

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

Ancient (pre-1990's) Secret Key Algorithms

- 56-bit DES key was clearly too short,
  - https://en.wikipedia.org/wiki/EFF_DES_cracker

  leading to...
- 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

1990’s Secret Key Algorithm

- RC4(**VERY** NOT RECOMMENDED)
  - 128-bit key stream cipher
  - once widely used in TLS (up to ~2014)
  - “Ron’s cipher #4”

Secret Key Algorithms

- 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
  - Implemented in h/w on many platforms (e.g. AES-NI instructions on Intel processors)
  - **AES IS RECOMMENDED**

Secret Key Algorithms

- ChaCha20
  - DJB productions
  - 256-bit key stream cipher
  - Considered a good replacement for RC4 – faster than AES on processors without the AES-NI instructions
  - **Recommended** if not using AES
    - today, could change
    - Note that what's recommended is really chacha20-poly1305 but we'll get to AEAD's later

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 XOR A) 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

How to think about “other” cryptographic algorithms

- Many cryptographic algorithms exist for any given purpose
- It is very hard to invent secure algorithms
- For each purpose, we’d like to have a recommended algorithm for current usage and a backup, both deployed, so we can switch
- There are very few (good) reason to want to invent new algorithms, except as a backup for current, or if the new algorithm has some radically new feature, or if you are a cryptographer
- Just having recommended+backup gets hard though as we consider the cross-product of alg/platform/purpose

Reasons for new algs...

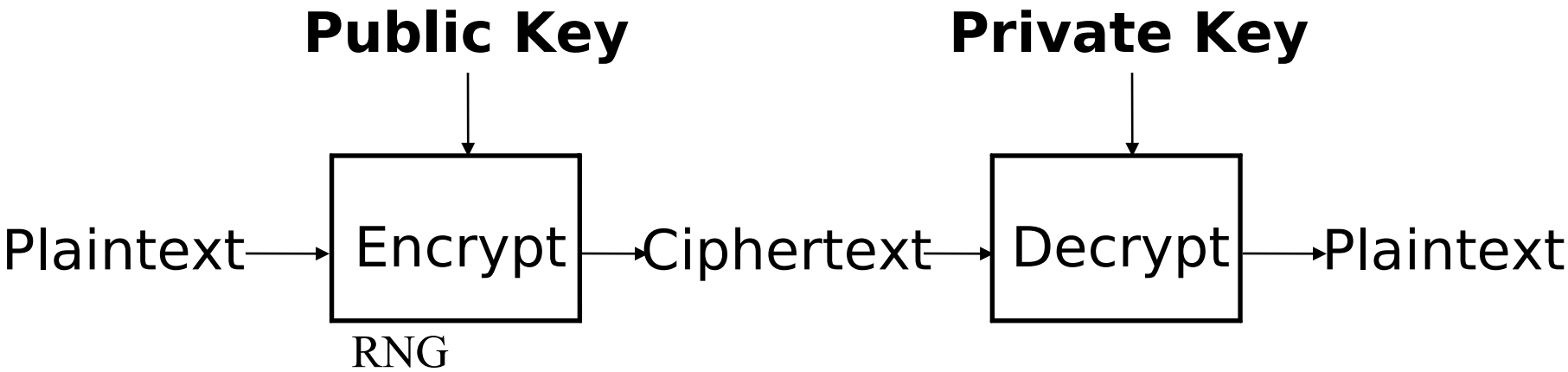
- We would like a “spare” for each cryptographic function that is already deployed so we can turn it on if/when today’s preferred option goes bad
  - Remember: attacks only ever get worse
- Some countries impose national requirements that national ciphers MUST be supported/used
  - Bad plan, but not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

Public Key Cryptography

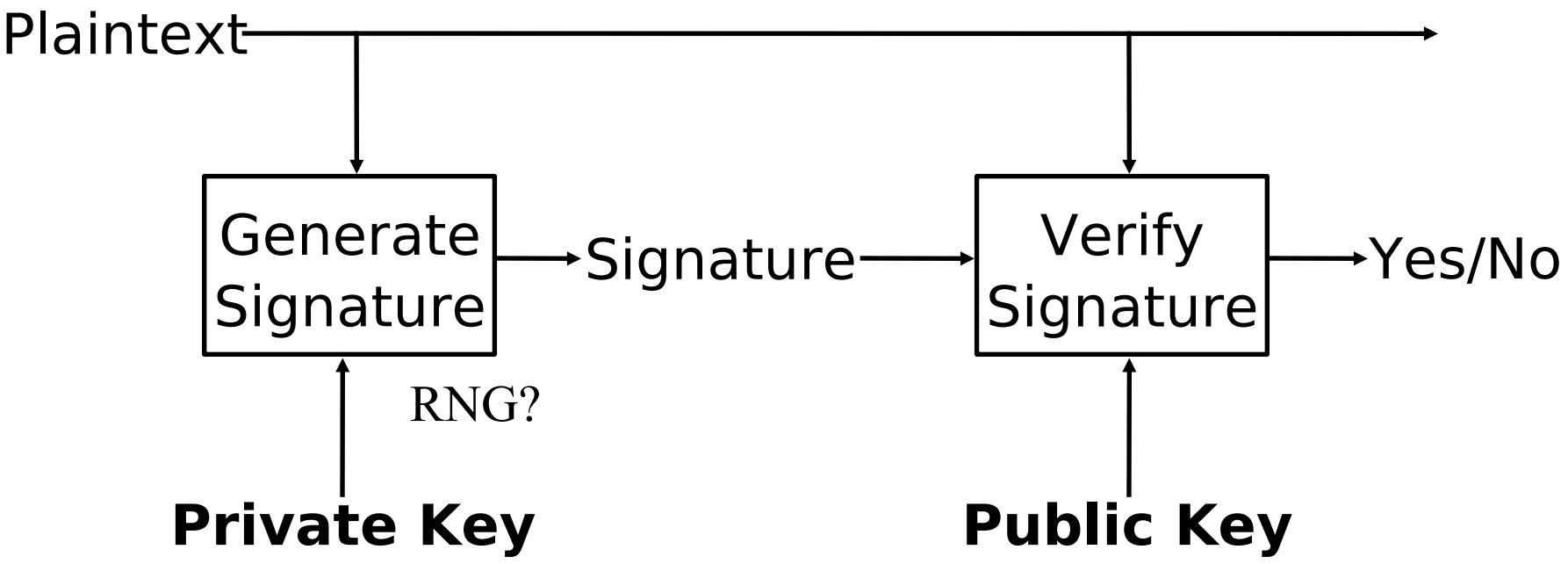
- Two keys per user: a private key and a public key. The keys reverse each other’s effects.
- Encrypt a message for Alice using her public key
- Decryption requires her private key
- Generating Digital Signatures requires the private key
- Verifying them requires the public key

- 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)

Bob (knows A's public key)

I'm Alice →

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
  - MD2, MD4 and MD5 are broken
  - SHA-1 has significant, demonstrated weaknesses
- **SHA-256 currently the best choice**
  - SHA-256 has the same internal structure as SHA-1, which made people worried...
- ... so NIST had a competition for a new digest
  - The winner (aka SHA-3) was Keccak
  - But not very interesting!
  - During the competition, confidence in the strength of SHA-256 grew as people studied it more

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
- To enable repeated experiments with a PRNG, we do need ways to re-use a seed and generate the same sequence of pseudo-random numbers

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) just 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)

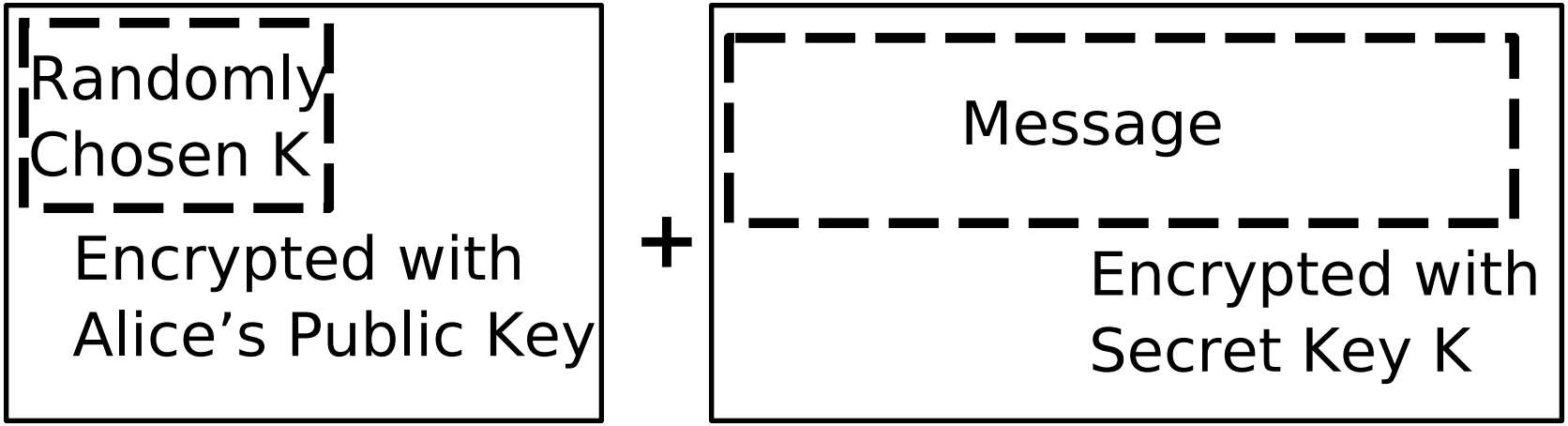
Combining Cryptographic Functions for Performance

- 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

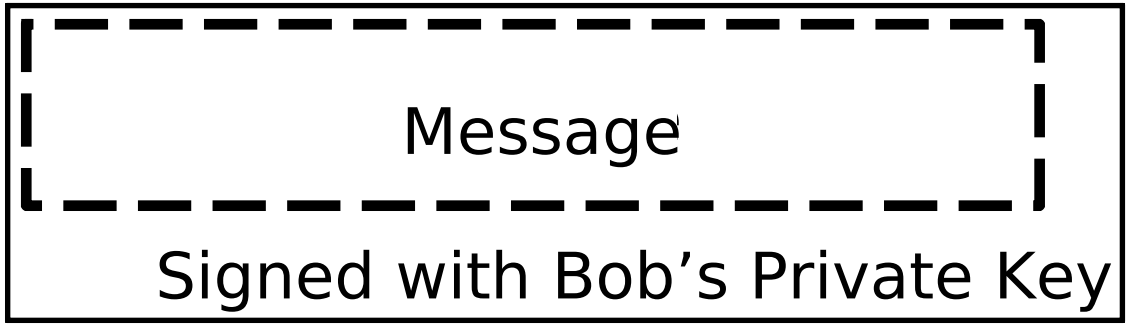
Instead of:

Use:



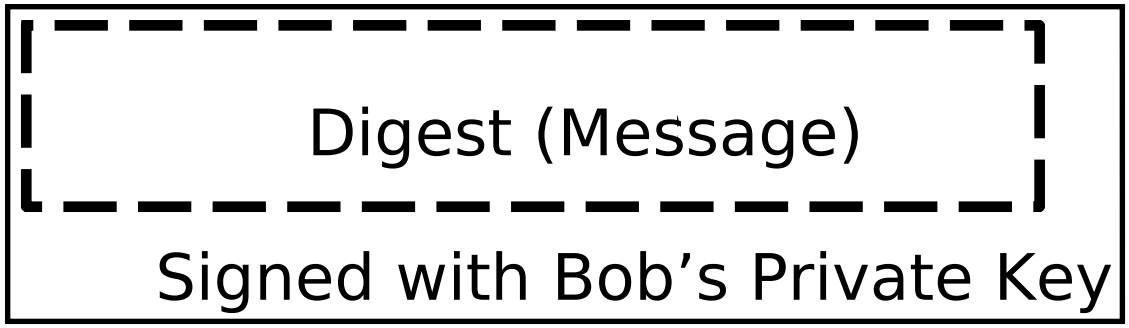
Hybrid Signatures

Instead of:

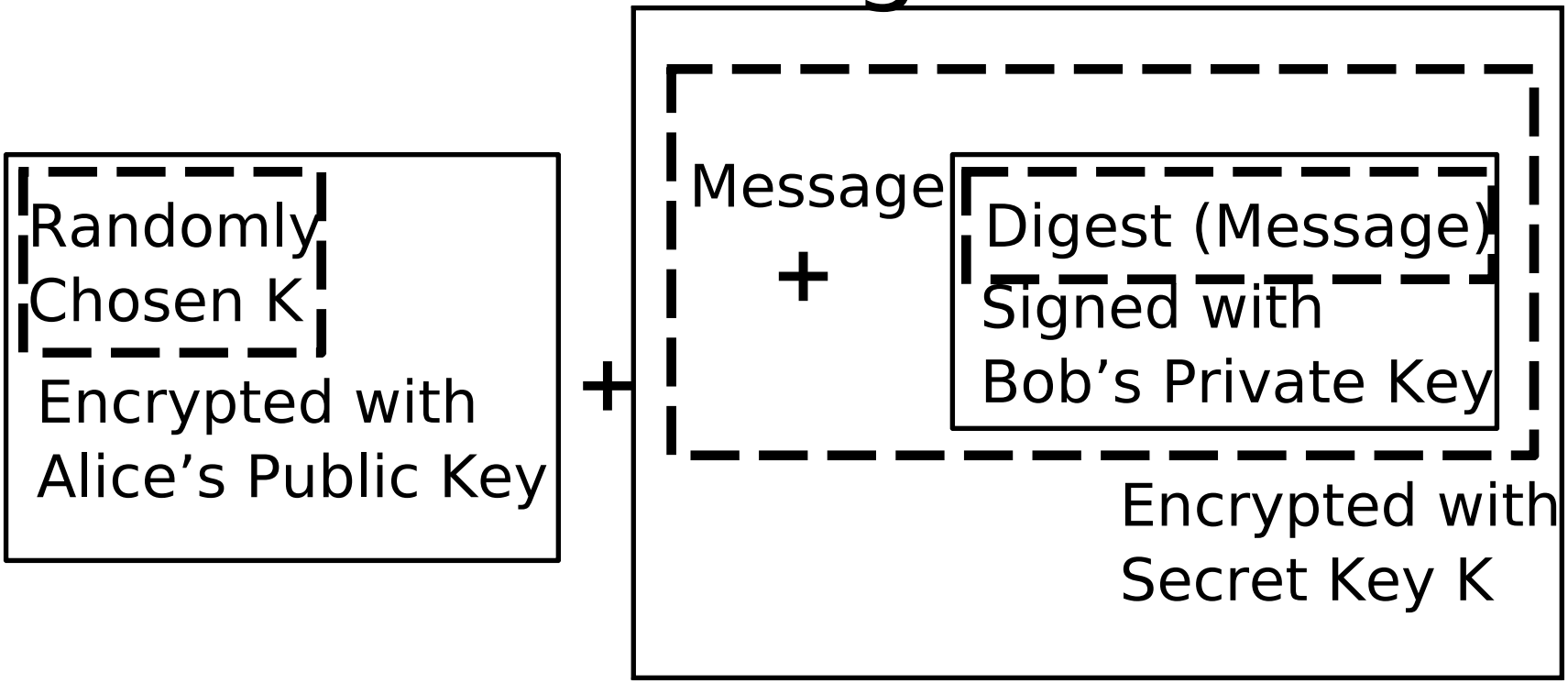


Use:

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 dictionary attack
- Most people will not pick passwords strong enough to withstand such an attack

Sample Broken Protocol

Alice (knows pwd)	Workstation	Server (knows h(pwd))

“Alice”, pwd →

Compute h(pwd)

I’m Alice →

← R (a challenge)

$\{R\}_{h(pwd)}$ →

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 long-term secret key
  - KDC knows them all
  - KDC assigns a session key to any pair who need to talk (this is how Kerberos works)

Key Distribution - Secret Keys

Alice KDC Bob

A wants to talk to B →

Randomly choose K_{ab}

← $\{\text{“B”}, K_{ab}\}_{Ka}$

→ $\{\text{“A”}, K_{ab}\}_{Kb}$

→ $\{\text{Message}\}_{Kab}$

Key Distribution - Public Keys

- Certification Authority (CA) signs “Certificates”
- Certificate = a signed message saying “I, the CA, vouch that <this number> 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 maybe a little easier because user keys need not change
  - But... new cert needed, and typically we rotate user keys with every new cert so not that big an advantage

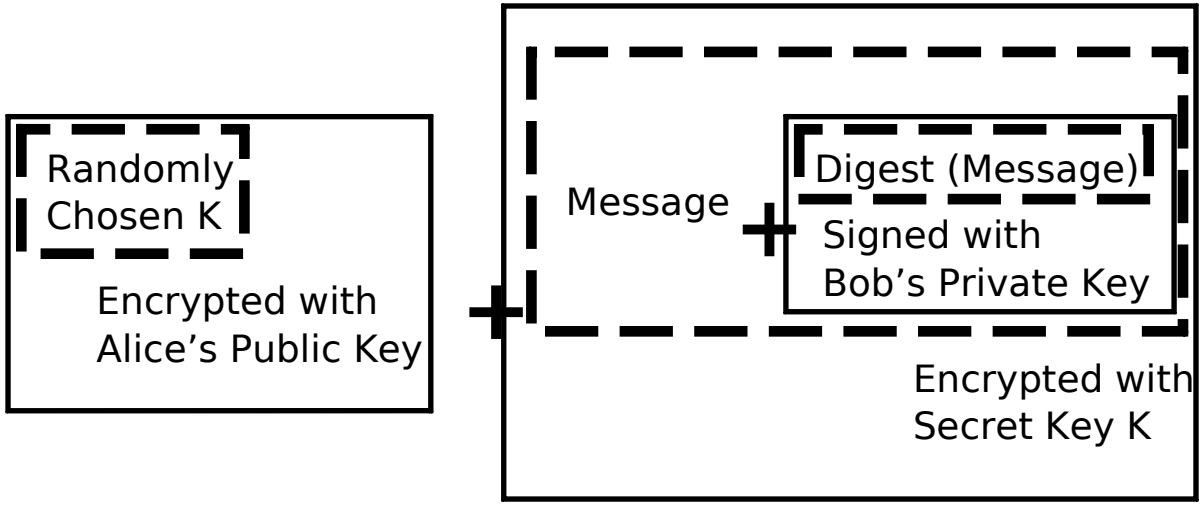
KDC vs CA Tradeoffs

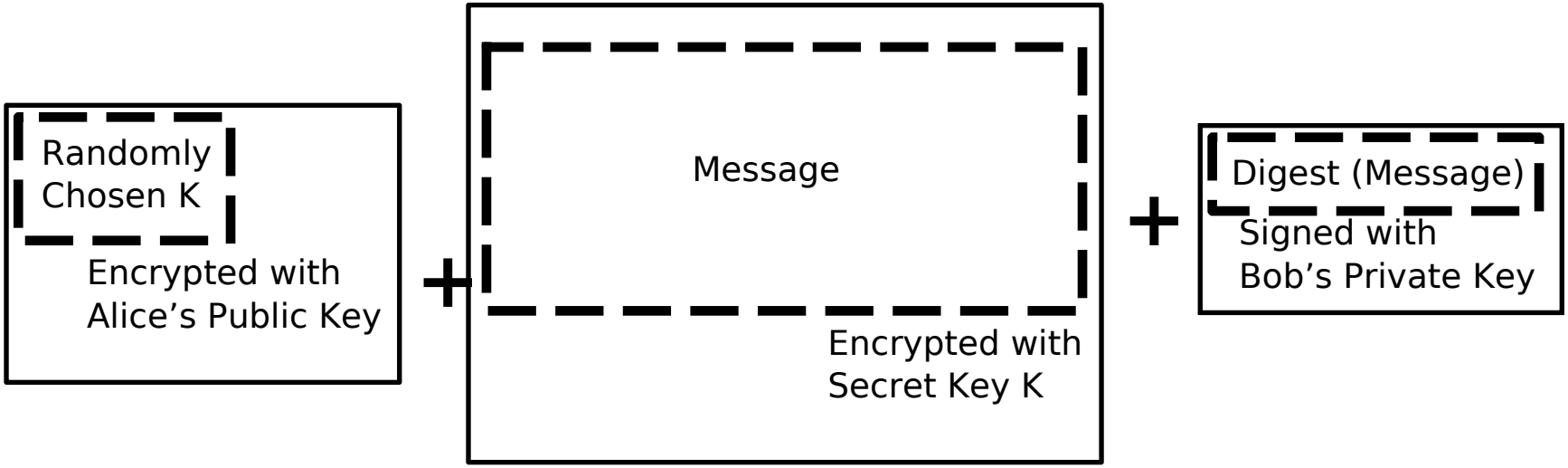
- KDC must be on-line and have good performance at all times
- CA need only be used to create certificates for new users
  - It can be powered down and locked up, avoiding network based attacks, if small scale
- CA’s work better interrealm, because you don’t need connectivity to remote CA’s

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...





Next while...

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

V. Quick Review of crypto

- secret key: 2 operations, inverses
- public key: sometimes one operation, 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
- Reminder: getting the terms secret, public, private key mixed up is a bad plan for exam purposes – make sure you understand that

Authentication with Secret Key

both know secret K

Alice ——— Bob

I'm Alice →

← R

compare:

$\{R\}_K$ →

---

or:

I'm Alice, $\{timestamp\}_K$ →

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
  - 1st 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!!!

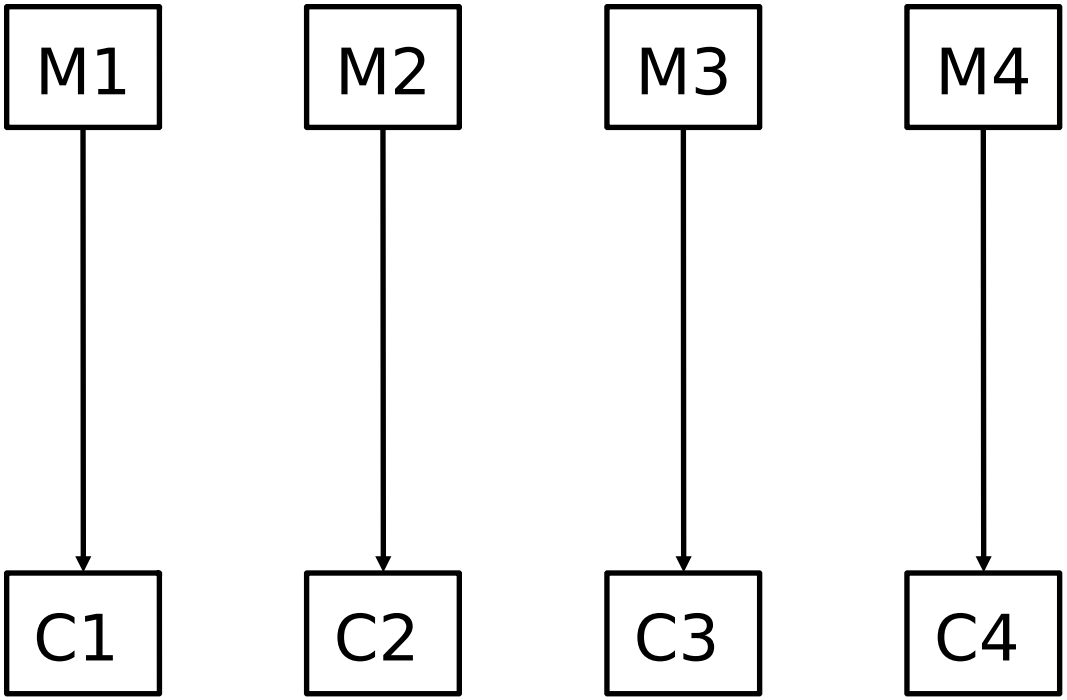
Block size considerations

- 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)
- This can approach being a real issue for some algorithms and applications, e.g. current VPNs with long-lived sessions, so do consider that there is a maximum number of blocks to encrypt for any given key
  - A draft specification analysing AEAD usage limits: https://datatracker.ietf.org/doc/draft-irtf-cfrg-aead-limits/

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!

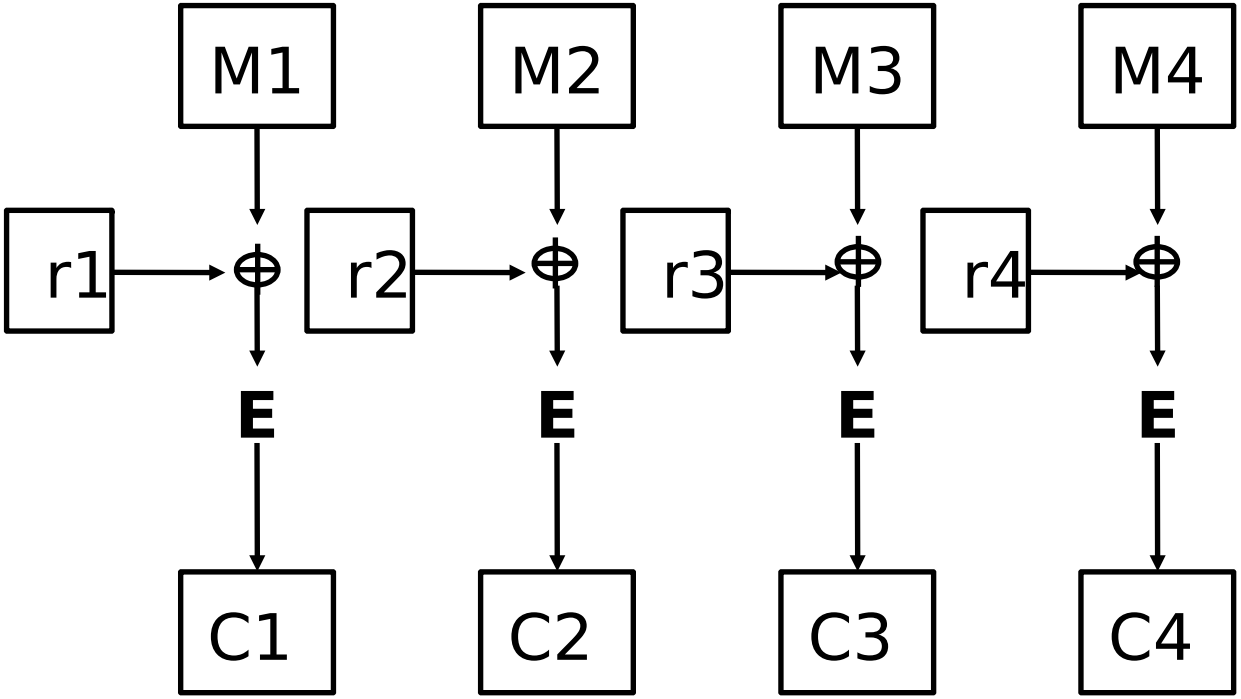
ECB



Problems with ECB

- If ci=cj, then you know pi=pj
- Can reorder blocks
- Can rearrange blocks to affect plaintext

Consider this



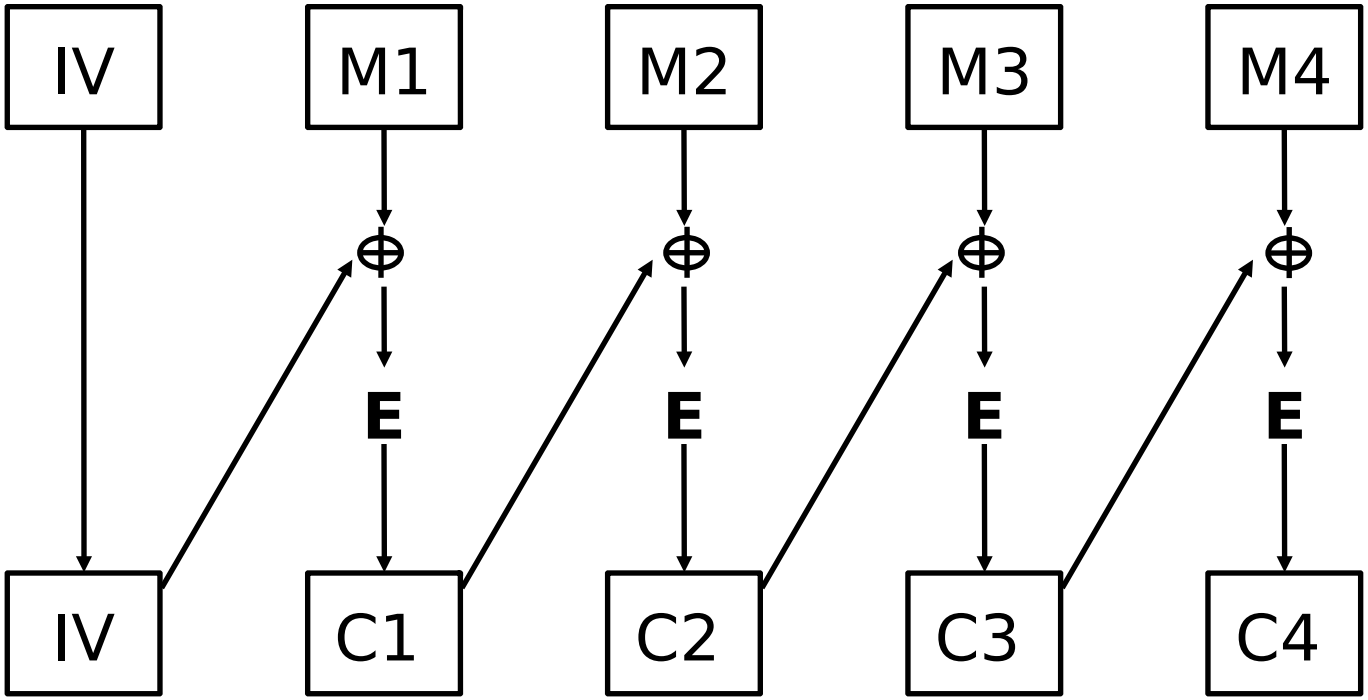
transmit r1, c1, r2, c2, r3, c3, r4, c4

$\oplus$ == Bit-wise XOR (exclusive OR)

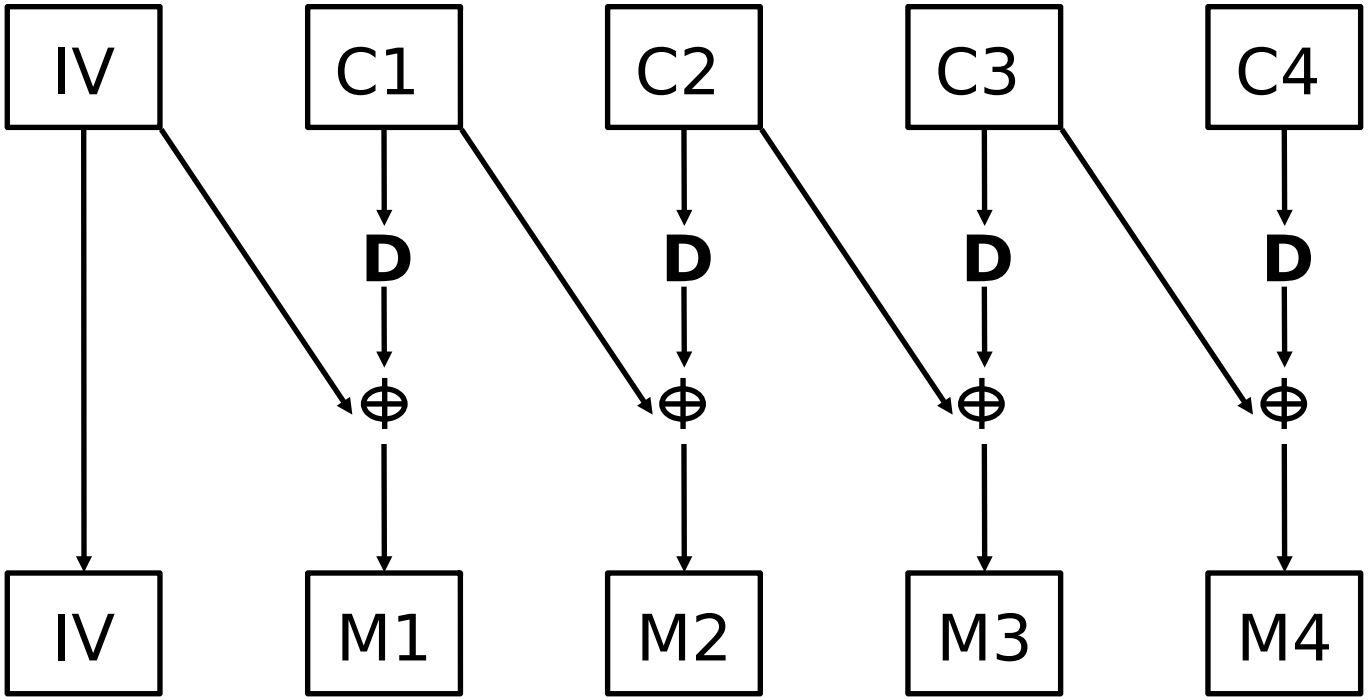
Problems with previous slide

- Need to send twice as much data
- Can still rearrange blocks
- If two ciphertext blocks equal, know XOR of two plaintext blocks = 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



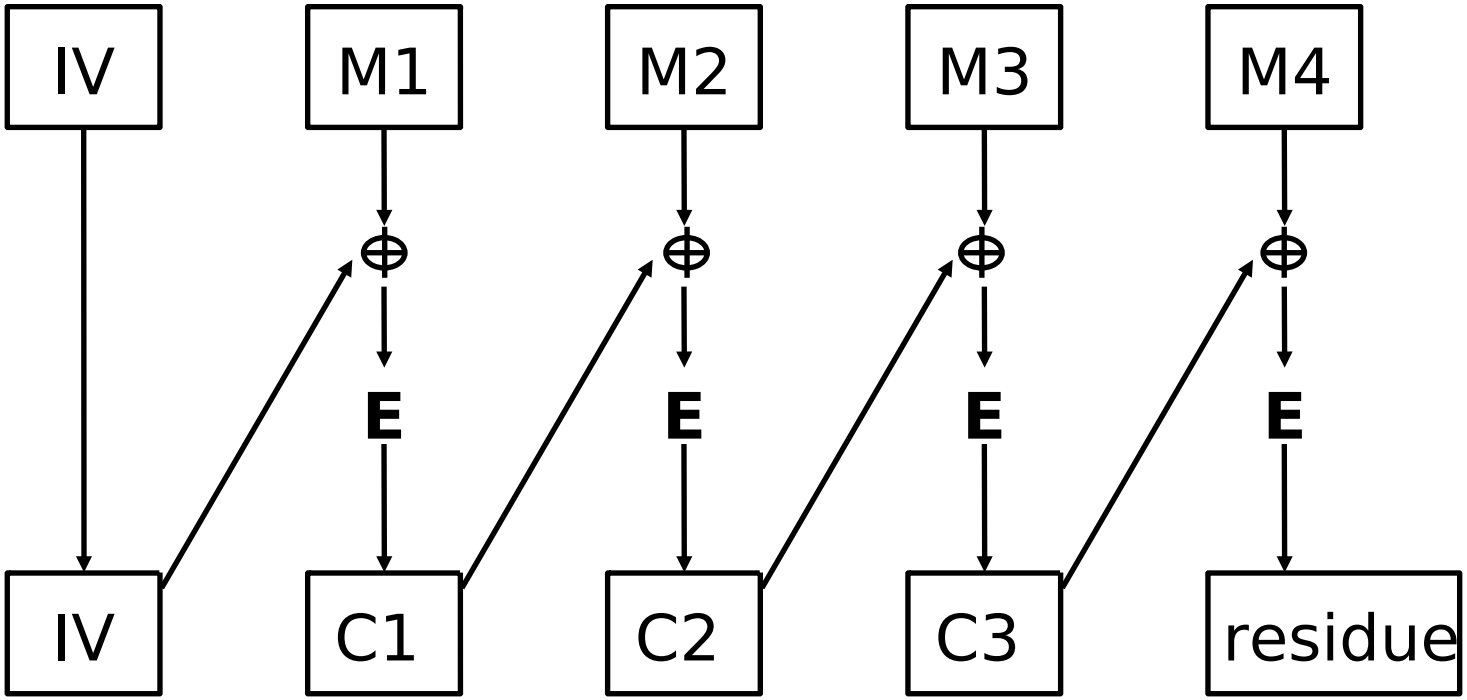
CBC Issues

- What happens if ci gets lost? Garbled? How much data gets lost?
- How can attacker that sees and can modify the ciphertext, and knows (some of) the plaintext, modify the recovered plaintext in a predictable way?

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 possible to generate an arbitrary message with a particular residue if you know the secret key

1) Create message, but leave one block (anywhere) blank.
2) Use any key K and any IV.
3) Start from beginning, doing ordinary CBC residue until get to block left blank.
4) Start from end, doing ordinary CBC decryption (since residue is constrained, you can work backwards from that and the plaintext blocks).
5) 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 from a block cipher

- Output Feed Back (OFB) stream generated:
  - Initialisation Vector (IV, transmitted in clear)
  - pad_1=e(IV, key)
  - pad_2=e(pad_1, key)
  - pad_i=e(pad_{i-1},key)
- Encrypt/Decrypt using bitwise XOR with as many pad bits as needed
- Pad values can be generated in advance
- Can encrypt arbitrary number of bits (vs block cipher)
- What if ciphertext garbled or lost?
- If know plaintext, can easily modify stream

Counter Modes

- c_i=f(key, IV, 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 go 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" where the "tag" authenticates both ciphertext and additional authenticated data (AAD)
- Decryption takes ciphertext (incl. tag) and returns "error" or plaintext, you never get partial decryption (in theory)
- RFC 5116 defines an abstract interface for AEADs; typical tags are 16 octets, typical IV/nonces are 12 octets – so some expansion
- Most new work these days makes use of AEAD modes – Also: older modes like CBC have been shown to be vulnerable in the face of various implementation errors

OpenSSL AEAD Example

- I've implemented the "Hybrid Public Key Encryption" (HPKE, RFC9180**-to-be**) scheme for encrypting "to" a public key:
  - Spec: https://tools.ietf.org/html/draft-irtf-cfrg-hpke
  - My code: https://github.com/sftcd/happykey
  - PR for upstream: https://github.com/openssl/openssl/pull/17172
- 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 call
  - Note to self: **pull up that code** to show init/update/final pattern
  - https://github.com/sftcd/happykey/blob/master/hpke.c#L557
- Not making any claim of "goodness" (NaCL/python/golang equivalents could well be better), but it's a smallish standalone bit of code that seems to do the job

What AEAD modes exist?

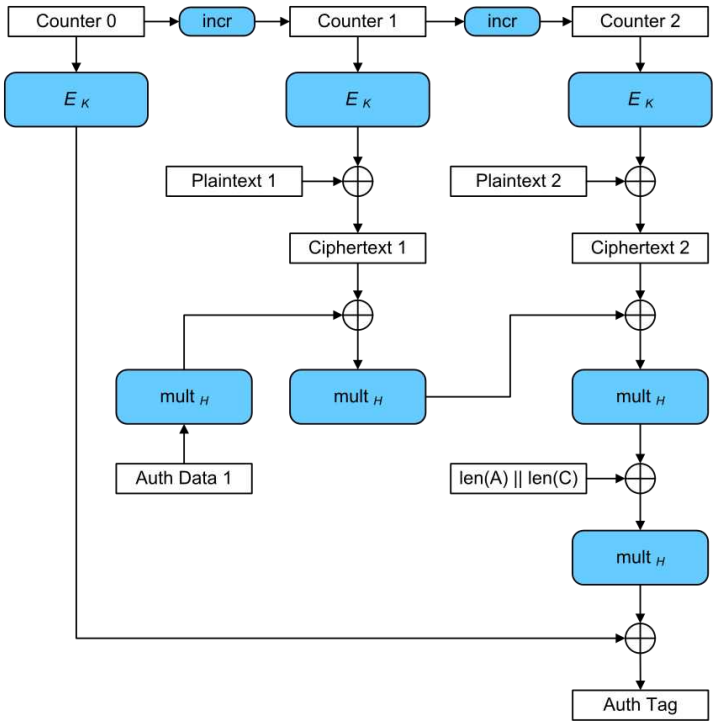
- Loads. Far too many: OCB, CCM, GCM, ChaCha20-poly1305, SIV, ...
- Which are good/bad? All of ‘em... Pretty much, though some are much more used than others.

  https://www.fi.muni.cz/~xsvenda/docs/AE_comparison_ipics04.pdf

  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 that have badly affected adoption even though the inventor tried hard to license liberally
  - https://en.wikipedia.org/wiki/OCB_mode#Patents
- AES-GCM, AES-CCM and ChaCha20-poly1305 are patent-“clean” as far as I know – though there are likely patents on some speed-up implementation 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**
- That’s an issue for the code calling the cryptographic API so is something you may well need to know

AEAD Summary

- **Don’t re-use nonces!**
- AES-GCM, AES-CCM and ChaCha20-poly1305 are good modes
- AES-GCM is a good default 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 AES support
- New AEAD modes will continue to be developed – if you can, don’t use them for a few years after they’re credible (same as all crypto!)

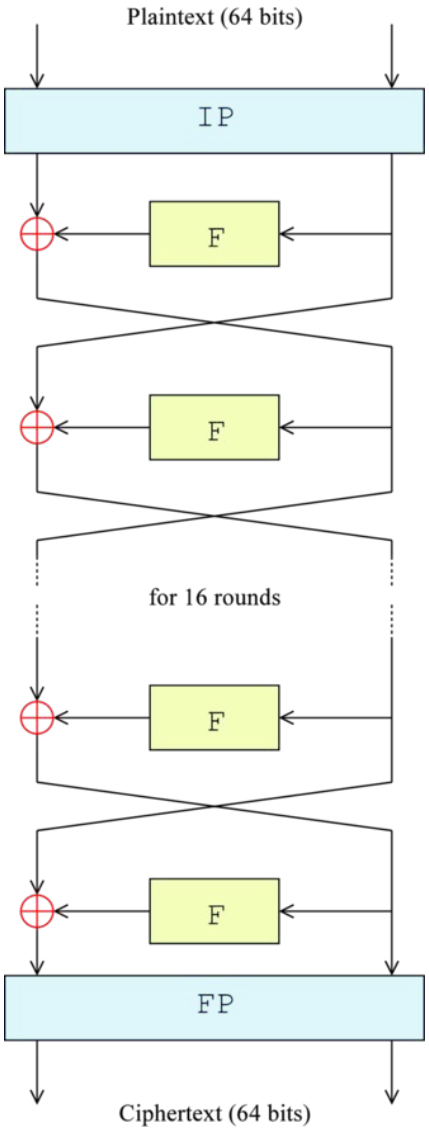
Algorithm Internals

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

- If you want to understand better, then re-implement yourself – doing that for one credible encryption algorithm is a good thing to do at least once in your career – you will throw away that code but end up understanding better

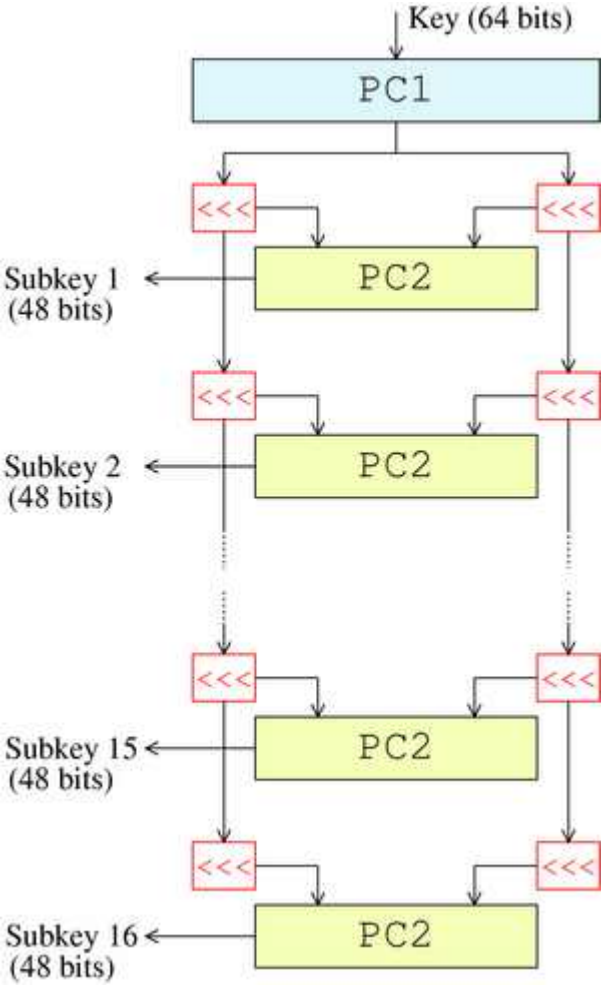
DES

- 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

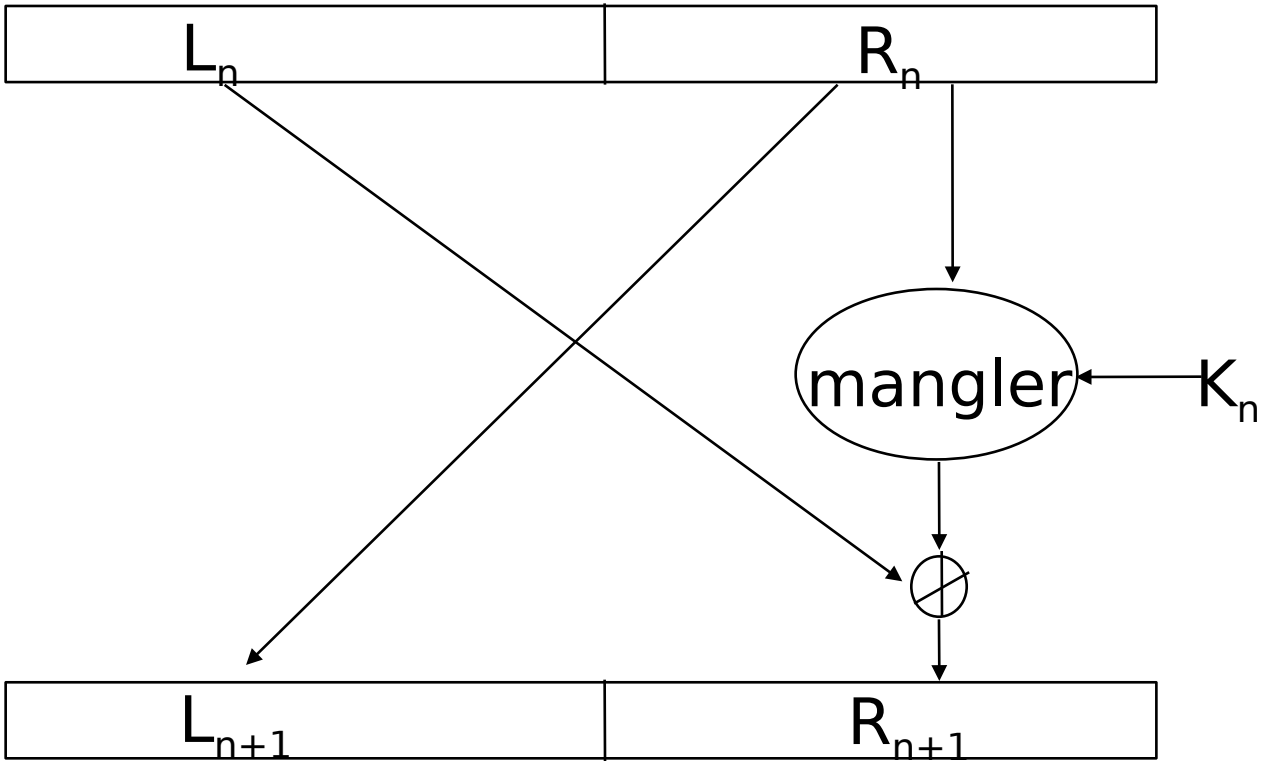


DES Key Schedule

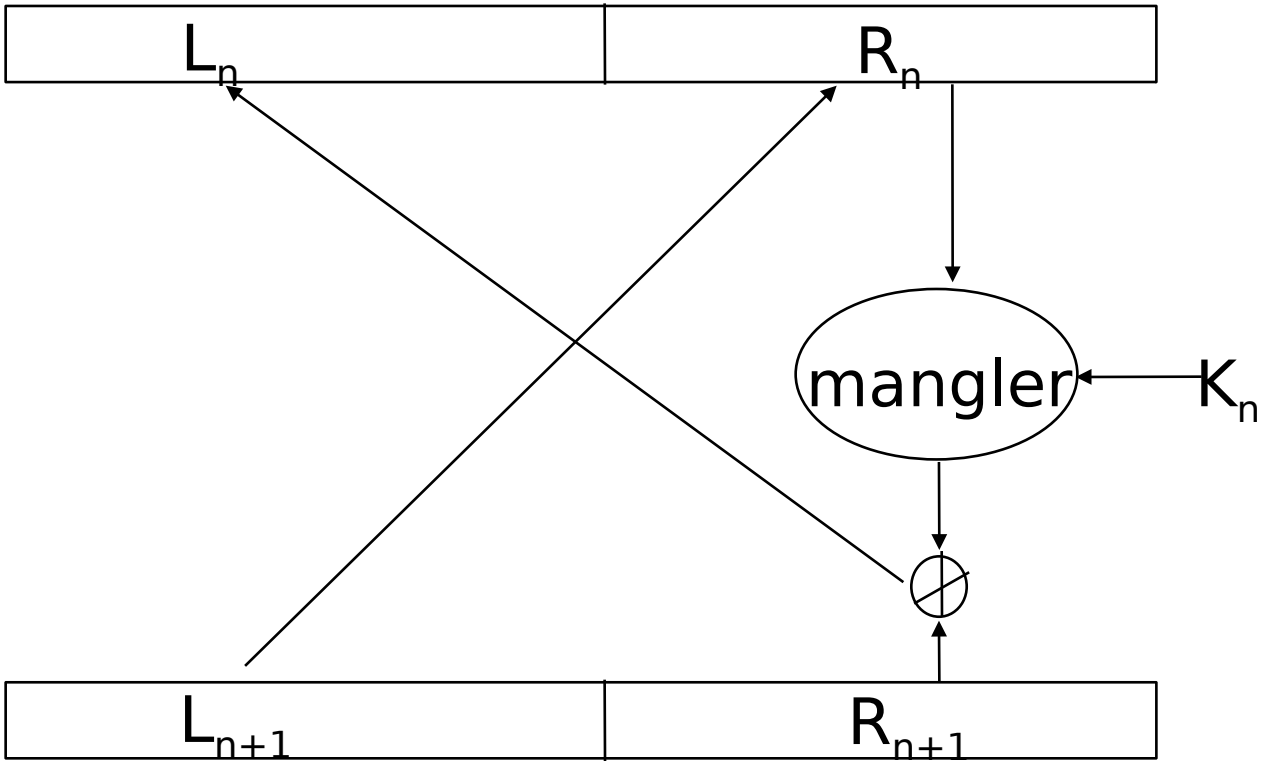
- Key Schedule: 56 key bits -> 48 key bits used at each “round”
- Each time you use a new key, you have to calculate all 16 subkeys
- Key schedule calculation overhead could be significant, e.g. if you changed key for every packet



Feistel Cipher Encryption



Feistel Cipher Decryption



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)

- 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

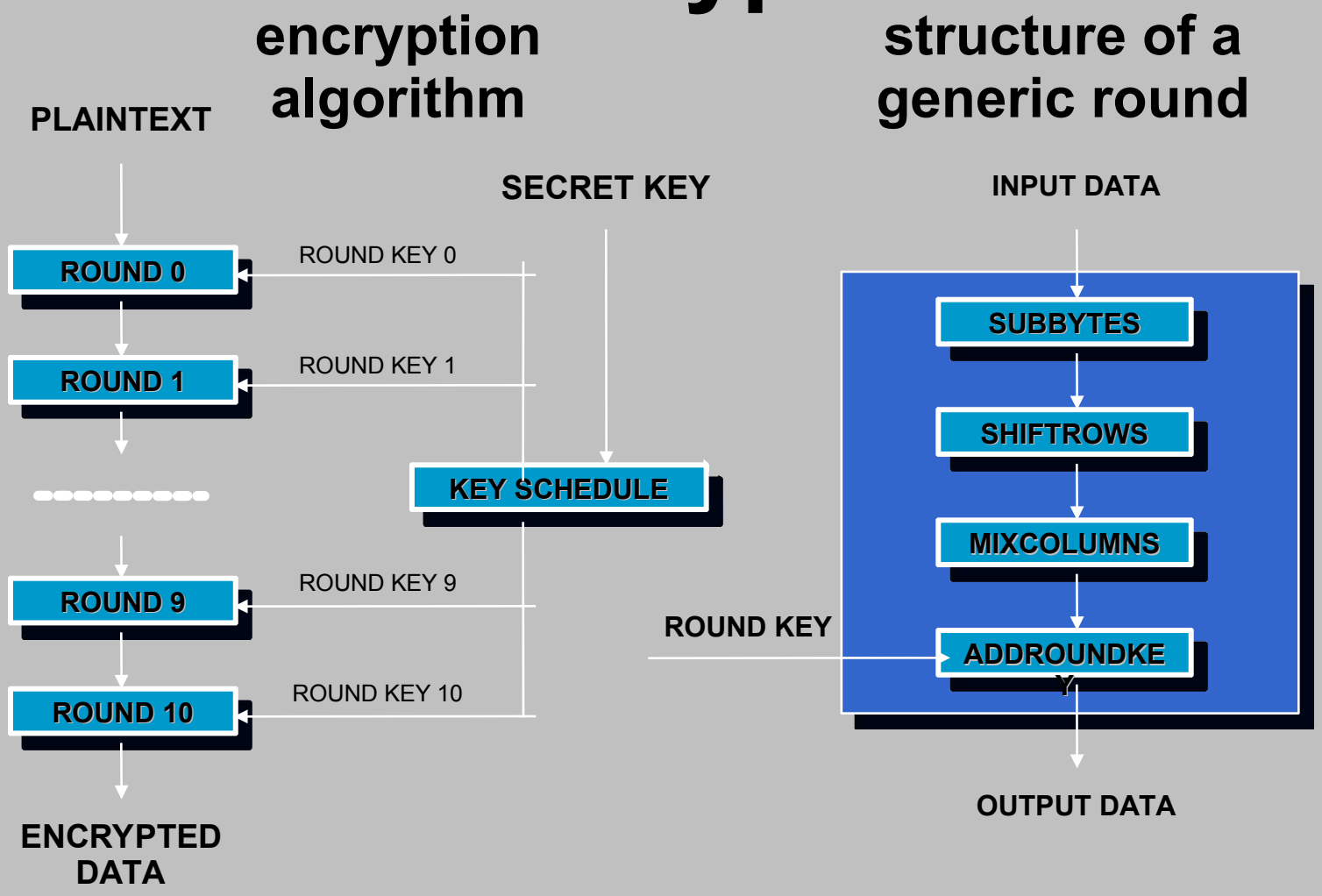
Why EDE instead of EEE?

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

AES Algorithm Description - General

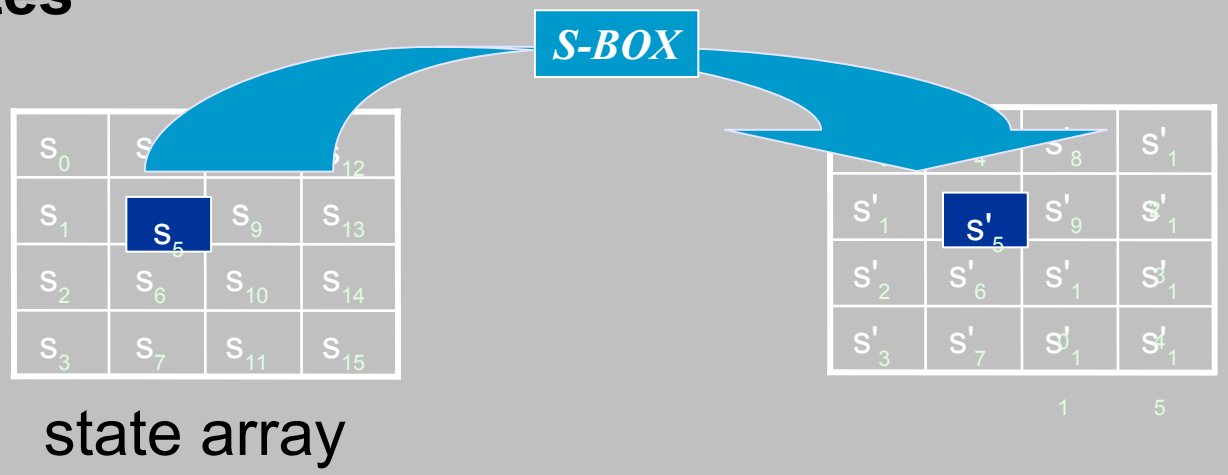
- Rijndael is the selected (NIST competition) algorithm for AES (Advanced Encryption Standard).
- It is a block cipher
- It reads an entire block of data, processes it in rounds and then outputs the encrypted (or decrypted) data.
- Each round is a sequence of four inner transformations.
- The AES standard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

Algorithm Description – Encrypt.



Algorithm Description – Encrypt.

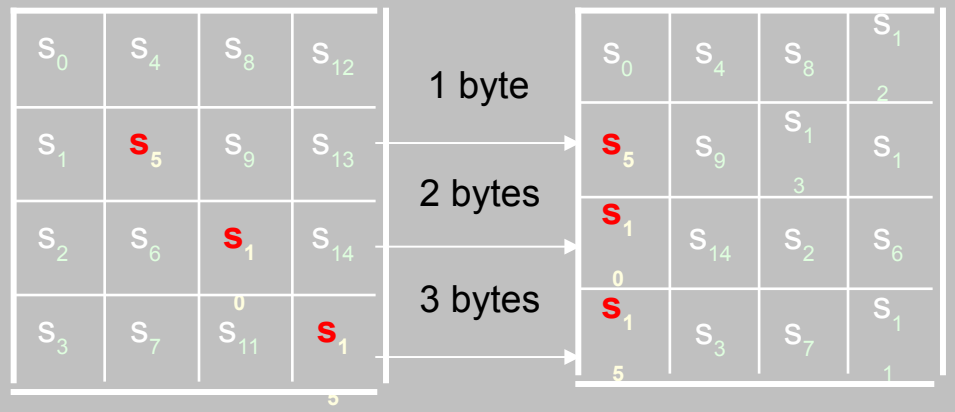
SubBytes



state array

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

=

coeff.s matrix

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

$\otimes$

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

=

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}

$\oplus$

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 or AES-256 if at all possible esp. If your CPU has AES hardware instructions (many do)
  - If not, ChaCha20 is good

Rolling your own...

- Snake-oil from commercial enterprises is occasionally (but persistently) seen
- Usually they claim to have developed a revolutionary new (maybe secret) algorithm
- How might you spot such snake-oil?
- 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.

Hash Functions

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., by bad actor); Tripwire does this
  - https://github.com/Tripwire/tripwire-open-source
  - Why wouldn’t CRC work?
- With secret, can do anything a secret key algorithm can do (authenticate, encrypt, integrity-protect)
  - Not necessarily a good idea though

Authentication with Hash

both know secret K

Alice ——— Bob

I'm Alice →

← R

compare:

hash(R,K) →

---

or:

I'm Alice, f(K,timestamp)? →

Creating Stream Cipher from Hash

- Create pad. First send IV in clear
  - pad_1=hash(K,IV)
  - pad_2=hash(K,pad_1)
  - pad_i=hash(I,pad_{i-1})
- Note, with IV, Alice can precompute pad, but Bob (without IV) can't
- Still: rolling your own like this isn't recommended

Hash function properties

Pre-image resistance: Given a hash value h it should be difficult to find any message m such that h = hash(m). This concept is related to that of one-way function.

Second pre-image resistance: Given an input m1 it should be difficult to find different input m2 such that hash(m1) = hash(m2).

Collision resistance: It should be difficult to find two different messages m1 and m2 such that hash(m1) = hash(m2). 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.

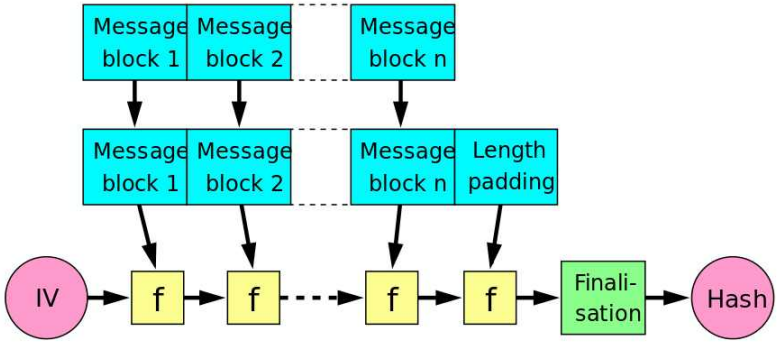
Speed.

SHA-1 Shambles

- 1995: SHA-1 proposed; RFC 3164 (2001) describes alg for implementers
- 2004/2005: Collision example shown in 2004 by Wang et al, paper in 2005
    - https://en.wikipedia.org/wiki/Xiaoyun_Wang
- 2011: Deprecated as NIST standard; RFC 6194 “don’t use that”
- 2017: “Practical” collision demonstrated; browsers deprecate SHA-1 certs
- 2020: “SHA-1 is a Shambles - First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust” Gaëtan Leurent and Thomas Peyrin https://eprint.iacr.org/2020/014
    - 11k US$ for a collision, 45k US$ for a chosen-prefix collision
        - given P, P’ find M, M’ s.t. H(P||M)==H(P’||M’)
    - 900 GPUs, 2 months elapsed time
    - GnuPGP keyring demonstration (thanks gpg, for allowing images;-)
- git still(?) dependent on SHA-1, but “Walk, don’t run, to the exit”
- HMAC-SHA1 still cryptographically ok, not being affected by collisions, but do you want that sha1.c file in your code?

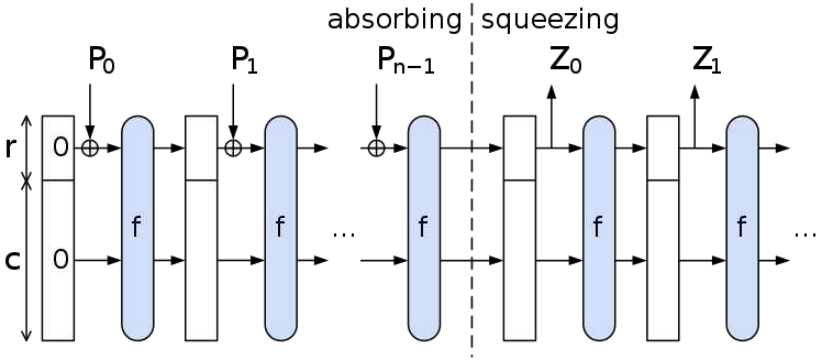
SHA-2

- Also 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

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 the same as 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

MACs with hashes

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

Possible problem

- If do hash(key | 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 hash(key | message) you could continue
- HMAC (RFC 2104) proven not to have this problem but HMAC is a bit more work
- HMAC-H(K,text) = H(K XOR opad, H(K XOR ipad, 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 digest(K,x) even if sees arbitrary (y,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
- Uusally 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

Even Moar Cryptography

Hash and public key Algorithms.

Public key algorithms

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=1 mod 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 and are 1024-bits long, how long is:
    - (X * Y) mod P?
    - (X ^ Y) mod P?
- Does calculating (X ^ Y) mod 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: $x^{phi(n)} \bmod n = 1$
- So $x^{k.phi(n)} \bmod n = 1$ and $x^{k.phi(n)+1} \bmod n = x$
- If we can find d*e = 1 mod phi(n), they'd be "exponentiative inverses"
- Because: $C=M^e$
- $C^d = M^{e*d}$

  $= M^{(1 + k*phi(n))}$

  $= M * M^{(k*phi(n))}$

  $= M$

Computing phi(n)

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

If n=p*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

$$d*e = 1 \bmod phi(n)$$

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 factor n because you constructed n
- After computing d from e, you can forget the factors of n (though in practice we 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
- There are probabilistic tests and deterministic ones (some that can produce proofs)
- State of the art is to use deterministic tests but we'll only look at the basic probabilistic method as it's easier to grok

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)

$$x^{p-1} \bmod p = 1 \text{ if } p \text{ prime}$$

- So to test if p is a prime, pick x < p and raise x to the power of p-1. If the answer is not 1, p is definitely not prime
- But can the result be 1 even if p not prime? Yes, but probably not...
  - Worst case: 1/4 false positives
  - Actually about 10^{-4} for candidate prime sizes of interest
- Iterate with different x's (maybe 10 or 20) until satisfied

Doing exponentiation

- Can’t multiply something by itself a gazillion google times!
- Solution: Repeated squaring
- Result: a 1024 bit exponent requires between 1024 and 2048 multiplications (depending on the number of 1’s in the number) (instead of a trillion trillion trillion trillion google multiplications)
  - Spoiler alert: the “between” above can lead to attacks!

RSA Signatures

- RSA encryption: $C=M^e$; decryption: $M=C^d$
- For RSA Signing we want to apply private key (d,n) to message, and allow public key (e,n) to be usable to verify signature
- We just do the obvious and use a “decrypt” operation on (a hash of) the message to sign and then an “encrypt” operation to verify
- Signature: $S = H(M)^d$
- Verification: Given M, S check if $H(M)==S^e$

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 mod each of n_1 , n_2 , n_3
    - By CRT, can compute m^3 mod $n_1 * n_2 * n_3$
    - Then just take cube root (m 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} mod q

Other 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

Diffie-Hellman Key Agreement

- 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 mod p, what is X?
  - That's the discrete log problem
- With the best known mathematical techniques, this seems to be somewhat harder than factoring a number of the same length as p
- Subtlety: we haven't proven that this cryptographic key derivation algorithm is as hard to break as the underlying problem

Diffie-Hellman

Alice | agree on g,p | Bob

choose random A | choose random B

$T_A = g^A \bmod p$

$\longrightarrow$

$T_B = g^B \bmod p$

$\longleftarrow$

compute T_B^A | compute T_A^B

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

Man in the Middle

Alice Trudy Bob

$T_A = g^A \bmod p$ $\rightarrow$ $T_T = g^T \bmod p$ $\rightarrow$

$T_T = g^T \bmod p$ $\leftarrow$ $T_B = g^B \bmod p$ $\leftarrow$

agree on $g^{AT} \bmod p$ $\leftrightarrow$ agree on $g^{TB} \bmod p$ $\leftrightarrow$

$\{data\}g^{AT} \bmod p$ $\rightarrow$ $\{data\}g^{TB} \bmod p$ $\rightarrow$

$\{data\}g^{AT} \bmod p$ $\leftarrow$ $\{data\}g^{TB} \bmod p$ $\leftarrow$

Signed Diffie-Hellman (Avoiding Man in the Middle)

Alice | Bob

choose random A | choose random B

$[T_A = g^A \bmod p]$ signed with Alice's Private Key $\longrightarrow$

$[T_B = g^B \bmod p]$ signed with Bob's Private Key $\longleftarrow$

verify Bob's signature | verify Alice's signature

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

If you have keys, why do D-H?

- “Perfect Forward Secrecy” (PFS): Prevents attacker from decrypting a conversation even if they break into both parties after it ends
  - The “perfect” in this term isn’t great – just calling it “forward secrecy” is more accurate in reality (due to caching and tickets)
- RSA Key Transport does not have the forward secrecy property:
  - A chooses secret key S, encrypts S with B’s RSA public key and sends ciphertext to B
- Today we strongly encourage forward secrecy for Internet security protocols and applications

Cookie Mechanism, Some Denial of Service Protection

Alice | Bob

I'm Alice →

choose cookie C

← C

C, [$T_A = g^A$ mod p] signed with Alice's Private Key →

verify C

← [$T_B = g^B$ mod p] signed with Bob's Private Key

verify Bob's signature

verify Alice's signature

agree on g^{AB} mod p

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 cookie= {IP address}S

Diffie-Hellman for Encryption

Alice

Bob

choose g, p

choose random B

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

choose random A

compute $T_A = g^A \bmod p$

compute T_B^A

encrypt message using $g^{AB} \bmod p$

send T_A, encrypted msg →

compute T_A^B

decrypt message using $g^{AB} \bmod p$

Flavours of D-H

- Above is called ephemeral-static D-H
  - Alice's value (“A”) is ephemeral, and Bob's (“B”) is static/published
- Static-static D-H is where both use static D-H values
  - See the security considerations of RFC6278
- Static D-H values can enable wiretapping by whomever knows a static private D-H value
  - “Centrally” generated private D-H values enable decryption, so static-static D-H would enable decryption of outbound email messages (and hence was once liked by some .gov type organisations)

Strong Password Protocols

...patents can be bad too

... even if the crypto is quite cute

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)

https://www.ietf.org/proceedings/63/slides/sacred-3/sld1.htm

https://en.wikipedia.org/wiki/Password-authenticated_key_agreement

https://www.jablon.org/passwordlinks.html

Basic Problem

- Using a (possibly hashed) password as a cryptographic 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

Alice | agree on g,p | Bob

choose random A

choose random B

$T_A = g^A \bmod p$

$\longrightarrow$

$T_B = g^B \bmod p$

$\longleftarrow$

compute T_B^A

compute T_A^B

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

EKE (designed for mutual authentication)

Alice ← Share W=h(pwd), g, p → Bob

Pick A

"Alice", $\{g^A \bmod p\}W$ →

Pick B

Decrypt $\{g^A \bmod p\}W$

Calculate $K=g^{AB} \bmod p$

Choose challenge C1

← $\{g^B \bmod p\}W$, $\{C1\}K$

Choose challenge C2

$\{C1,C2\}K$ →

← $\{C2\}K$

SPEKE

Alice ⟷ Share W, p ⟷ Bob

Pick A

"Alice", W^A mod p →

Pick B

Calculate $K=W^{AB}$ mod p

Choose challenge C1

← W^B mod p, {C1}K

Choose challenge C2

{C1,C2}K →

← {C2}K

PDM (Password Derived Moduli)

Alice ⟷ Share p ⟷ Bob

Pick A

Alice → Bob: "Alice", 2^A mod p

Pick B

Calculate $K=2^{AB}$ mod p

Choose challenge C1

Bob → Alice: 2^B mod p, {C1}K

Choose challenge C2

Alice → Bob: {C1,C2}K

Bob → Alice: {C2}K

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 have been 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 PAKEs when doing new device registration, e.g. into a public key infrastructure or symmetric key based system

Elliptic Curve Cryptography

- 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 (Fp or $F2^m$)
- 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 had 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
- ECDH has displaced a lot of RSA key transport (but not TLS server authentication) and most integer D-H

ECC – 3

- Doubts about NIST Curves (P256) were expressed some years ago, but not much among those who understand stuff
  - Non-deterministic signature schemes are worst aspect really
- Today we mostly dislike NIST curves because of performance and it being easier to write buggy code for ‘em
- Curve25519 and Ed25519 are 4x quicker and less likely to result in buggy code
- 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

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)

recently some criticism of deterministic signatures due to fault injection attacks, along with suggestions to add "noise" in a way that won't expose private key,

one unfinished suggested way forward https://datatracker.ietf.org/doc/html/draft-mattsson-cfrg-det-sigs-with-noise

All (signature scheme) bets might be off for adversary-controlled small, commercial-grade devices

- (A) general idea: public key algorithms depend on particular arithmetic; if one can selectively flip bits then outputs from RSA/D-H/ECC functions can be revealing, esp with multiple invocations
- Barenghi et al, “Low Voltage Fault Attacks to AES and RSA on General Purpose Processors” - nicely explains how undervoltage can lead to single bit flips in memory reads and how that can lead to leaking a private key
    - https://eprint.iacr.org/2010/130.pdf
- Not-that-different attack on ECDSA
    - https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1091.528&rep=rep1&type=pdf
- Power analysis attack on EdDSA
    - https://eprint.iacr.org/2017/985.pdf

Odds and Ends: 1 or 2 slides each

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

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

Identity Based Cryptography

- 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 (KG) 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 crypography
- Discrete log again, and D-H but with two groups for which there's a bilinear mapping
  - https://en.wikipedia.org/wiki/Pairing-based_cryptography
- 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

Idea:

- Geometry!
  - Knowing 2 points => know the line
  - 3 points => know the quadrtic
  - k points => know the x^k polynomial
- Distribute N points of an x^k polynomial
- Can be used in real life!

https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing

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, IBC, Dual-EC PRNG

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

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