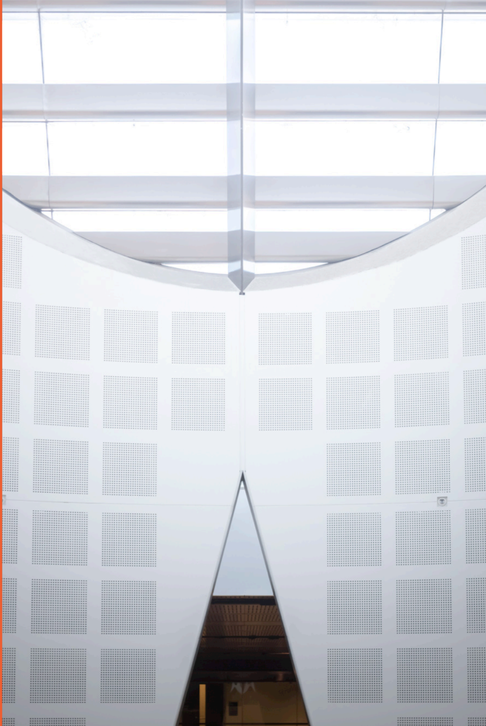
Empirical Security Analysis

Cryptography in the real world

Presented by
Ralph Holz
School of Information Technologies

THE UNIVERSITY OF SYDNEY



Cryptography

An indispensable tool

- Secure protocols and systems use cryptography for:
  - Data confidentiality (encryption)
  - Data integrity (signatures, message authentication codes)
  - Authentication & data origin verification (signatures, message authentication codes)
  - Non-repudiation (signatures)
- Fundamental understanding of the **principles** of cryptography is required for many careers in IT.
- In this course, we focus more on the **factors** that cause cryptography to fail
- Cryptography is rarely broken, it is usually bypassed

Cryptography

A double-edged sword—complex and many subtleties

- Inconsiderate application of cryptography: likely insecure system
  - Worse, an insecure system that seems secure to its developers and users
  - The attacker doesn't know it is supposed to be secure and will take it apart
- Implementing cryptography requires tremendous experience and an intimate knowledge of the peculiarities of hardware and programming languages
  - Do **not** cook (implement) your own crypto and use it for real
  - **Recognised** way to become **proficient** is to practice, find a mentor, discuss with other implementers $\rightarrow$ long career path with no shortcuts!

Our agenda

- Functional overview of cryptography
  - Cryptographic primitives
- Cryptography as it is used in the real world
  - What can go wrong
  - Good and bad choices
- Poor entropy and unwise settings
  - Breaking RSA and DSA on a large scale
  - Breaking Diffie-Hellman exchanges

Cryptographic primitives

We discuss the following primitives:

- Randomness
- Hash functions
- Two forms of cryptography:
  - Symmetric (shared-key cryptography)
  - Asymmetric (public-key cryptography)

Part I

Randomness

Why randomness?

Randomness is important in many security scenarios

Intuitive: the more random a value it is, the less predictable it is.

- Think of Transaction Numbers (TANs) to confirm a transaction
- Think of session IDs in Web cookies!
  - Must be 'unguessable'
  - Used in all important web sites: Weibo, Facebook, Google, Twitter, …
- Think of short-links in cloud services
  - `http://pastebin.com/74KXCaEZ`
- Passwords must exhibit 'unpredictability'
- **Many cryptographic algorithms are insecure without random numbers as input**
  - *E.g.*, key generation, certain signatures, …

True randomness

Very few things are truly random

- Rolling the dice:

True randomness

Very few things are truly random

- Rolling the dice: no—just too hard to predict for, *e.g.*, players in a board game

True randomness

Very few things are truly random

- Rolling the dice: no—just too hard to predict for, *e.g.*, players in a board game
- Decay of an atom:

True randomness

Very few things are truly random

- Rolling the dice: no—just too hard to predict for, *e.g.*, players in a board game
- Decay of an atom: yes, according to quantum mechanics

True randomness

Very few things are truly random

- Rolling the dice: no—just too hard to predict for, *e.g.*, players in a board game
- Decay of an atom: yes, according to quantum mechanics

Information-theoretic modelling

- Assume a **transmitter** that outputs bits (0s and 1s, *w.l.o.g.*)
- Completely random: every new output value is equally likely (independent of any other variable)

Computational infeasibility

- Computer systems rarely have a source of true randomness.
- Fortunately, we usually do not need **true** randomness
- It is enough if our random values are **sufficiently hard** to predict
  - 'Threat model' of cryptography: not predictable in **computationally feasible** way
- **Computationally infeasible**: time and storage requirements exceed the capabilities of today's technology (by far)
  - This is the domain of **computational complexity**
  - If our **machine model** changes (e.g., quantum computers?), the meaning of computationally infeasible also changes

CSPRNG

Cryptographically Secure Pseudo-Random Number Generator

- A **PRNG** outputs a **deterministic** sequence of numbers
  - Takes a seed value as start value
  - Output sequence depends on the seed
- A PRNG is cryptographically secure, *i.e.*, a **CSPRNG**, if:
  - It is computationally infeasible to brute-force the seed value (try all possible values) to correctly predict the output
  - It is computationally infeasible to distinguish the output sequence from true randomness
- Only option for the attacker is, in other words, to know the seed
  - We will see that this is one of the most neglected attack vectors, and the downfall of some systems

CSPRNGs on your computer

Modern operating systems have CSPRNGs

- Crucial property is **seed value—**must be as random as possible
- Small entropy pool is enough: *e.g.*, a few hundred bits from …
  - Storage seek times
  - Time between hardware interrupts
  - Input devices (mice, keyboards)
  - Clock time
  - Hardware entropy generator (if available)
- Note: some softare packages create their own entropy pool
- Plenty of interesting studies of randomness in different OSes

Entropy in Linux

Example: Linux

- Fill entropy pool at boot, add new entropy at runtime
- Exposes to userland via two block devices:
  - `/dev/random`
  - `/dev/urandom`
  - Applications can read directly from these
- The two devices use different entropy pools
- This is actually a curiosity among UNIX-like systems

Entropy in Linux

Difference between the interfaces

- Their output comes from the same CSPRNG (!)
- Only difference:
  - `/dev/random` tries to estimate entropy in its pool
  - If that is not enough, it blocks on a read
  - `/dev/urandom` does no estimate and does not block
- **After seeding,** the **quality** of the random numbers is **the same**
- There is only one crucial thing you must know:
- **Before proper seeding,** `/dev/random` will block on your read. `/dev/urandom` will not.
- Hence, a good Linux distribution **ensures a good seed at startup.**

Entropy in Linux

Difference between the interfaces

- Their output comes from the same CSPRNG (!)
- Only difference:
  - `/dev/random` tries to estimate entropy in its pool
  - If that is not enough, it blocks on a read
  - `/dev/urandom` does no estimate and does not block
- **After seeding**, the **quality** of the random numbers is **the same**
- There is only one crucial thing you must know:
- **Before proper seeding**, `/dev/random` will block on your read. `/dev/urandom` will not.
- Hence, a good Linux distribution **ensures a good seed at startup.**
  - What about embedded systems? (in a minute)

More entropy in Linux

- Linux design is less developer-friendly
  - Tempts developers to use `/dev/random`—at the cost of hanging applications
  - Use of `/dev/urandom` **can** be dangerous directly after startup if your Linux distribution has not seeded properly
- Contrast with, *E.g.*, FreeBSD: CSPRNG blocks **until seeded**, then never again—always safe to use
- Since Linux 3.17: `getrandom()` system call
  - Blocks until entropy high enough, and then never again
  - Not a block device, however!

More CSPRNG

Some software packages come with their own CSPRNG

- **Not** as good as kernel CSPRNG
  - Kernel has raw access to devices; userland does not
  - Kernel can make sure CSPRNG state is not leaked between processes
- Famous example: `openssl`
  - Supports very large number of old systems
  - Some without CSPRNG
- When you inspect some application's code, and they do their own CSPRNG
  - If all platforms on which the application must run have a CSPRNG: fix it

Part II

Hash functions and MACs

Cryptographic hash functions

Hash function

A function that takes an input of variable length and produces a fixed-length output. Also called: *digest*. Example: CRC-32.

Cryptographic hash function

A hash function that fulfills three criteria:

- Pre-image resistance
- Second pre-image resistance
- Collision resistance

Examples:

- MD5 (128 bit output, considered broken, phased out)
- SHA1 (160 bit output, under pressure, in phase-out)
- SHA2, SHA3, RIPEMD160 (all considered strong)

Resistance properties

Due to the fixed-length output, *any* hash function H has collisions, *i.e.*, $H(a) = H(b), a \neq b$. We want to make finding them **hard**:

(First) pre-image resistance

Given a randomly chosen y from H's range of output values, it is computationally infeasible to find an x such that $H(x) = y$.

Second pre-image resistance

Given a randomly chosen x, it is computationally infeasible to find any $x', x' \neq x$ such that $H(x) = H(x')$.

Collision resistance

It is computationally infeasible to find any pair $(x, x'), x \neq x'$ such that $H(x) = H(x')$.

Why do we need this? Enter authenticity.

Message Authentication Codes

- Scenario: Alice and Bob want to send each other messages, and they want to make sure a given message m really came from the respective other (authenticity).
- Construct a Message Authentication code for this purpose:
  - Find a clever function T to get a fixed-length tag t, such that no attacker can feasibly find a t' for any $m' \neq m$.
  - In other words, the attacker cannot find a second, different message **and** predict the correct tag to prove authenticity.
- Send m together with t.

Message Authentication Codes

- Scenario: Alice and Bob want to send each other messages, and they want to make sure a given message m really came from the respective other (authenticity).
- Construct a Message Authentication code for this purpose:
  - Find a clever function T to get a fixed-length tag t, such that no attacker can feasibly find a t' for any $m' \neq m$.
  - In other words, the attacker cannot find a second, different message **and** predict the correct tag to prove authenticity.
- Send m together with t.

What must this function T be like?

Message Authentication Codes

Requirements for T

- MAC function T must have resistance properties of a cryptographic hash function.
- Must also be computationally infeasible to predict a correct tag t' for a message $m' \neq m$ even when allowed to know any other combination of (m, t).

Construction

- A cryptographic hash function takes care of the first requirement. Mixing in a secret s will address the second requirement **if** the resistance properties hold.

HMAC

Based on double-hashing, including a shared secret k.

Construction

$$\mathrm{HMAC}(k, m) = H((k \oplus \mathrm{opad}) \parallel H((k \oplus \mathrm{ipad}) \parallel m)$$

- H can be any cryptographic hash function: SHA2, SHA1, MD5, …
- `ipad` and `opad` are constant bit strings
- HMAC construction bolsters the security of H considerably
  - Although HMAC-MD5 is not yet broken, recommendation is to use HMAC-SHA1 or higher

Cipher-based constructions

Can use ciphers (*e.g.*, AES) construct MACs that have the same security properties. $\rightarrow$ Need to understand symmetric crypto first.

Part III

Symmetric Cryptography

Symmetric Cryptography

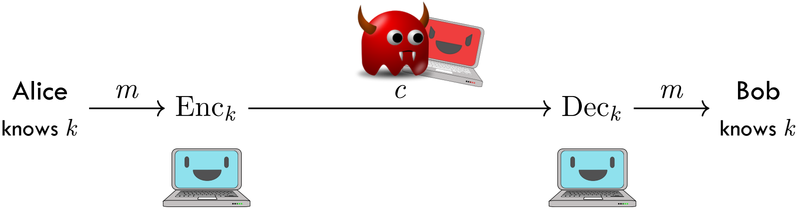
Characteristics

- Alice and Bob share a **secret** key k that only they know
  - Used to encrypt **and** decrypt
- Symmetric crypto allows
  - Encryption/decryption
  - Constructing Message Authentication Codes

Terminology

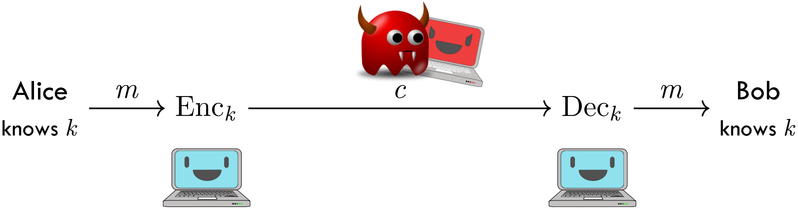
- Plaintext m—the message itself
- Ciphertext c—the encrypted plaintext
- Encryption: $c = \mathrm{Enc}_k(m)$; decryption: $m = \mathrm{Dec}_k(c)$
- Cipher: the combination of $\mathrm{Enc}_k(m)$ and $m = \mathrm{Dec}_k(c)$

Example



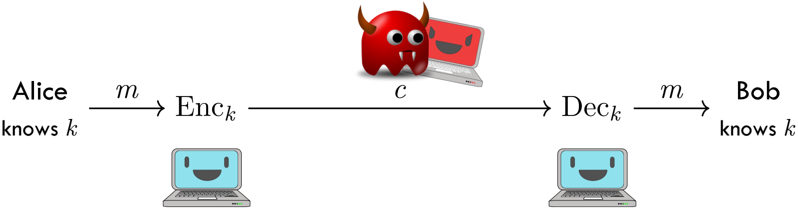
– Let's see an example. But advance warning!

Example



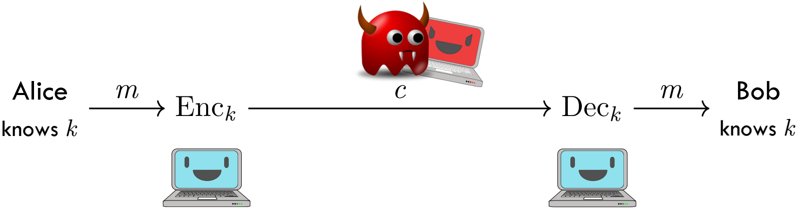
– We use AES-ECB, which is not a sensible way to encrypt!

Example



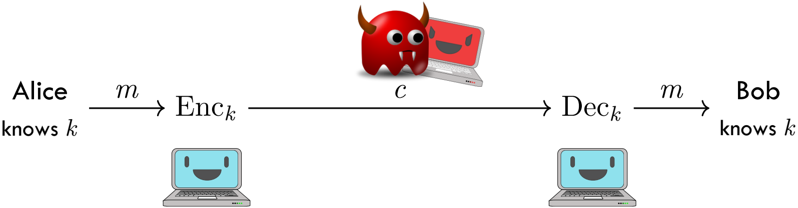
– But it’s good to demonstrate the principle.

Example



- $m =$ `Toy example - not a good way to encrypt.`
- $k =$ 95 eb 50 0c 31 07 46 6f 88 8a f7 0b dd fb d7 64
- $c =$ e9 e0 11 d3 f6 9f 72 b7 fc 64 73 df 82 b0 25 0d fb db c5 46 02 36 a0 70 49 29 46 d6 7b 2f 61 01 5f 5a 8c e9 d9 cf e0 11 9e db dd 5f 29 11 6d fc
- $\mathrm{Enc} =$ AES-128-ECB

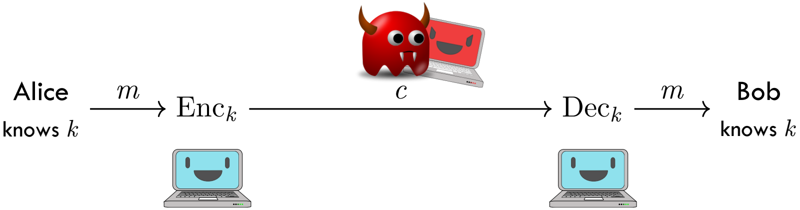
Example



Which security goals can we fulfill?

- Confidentiality?

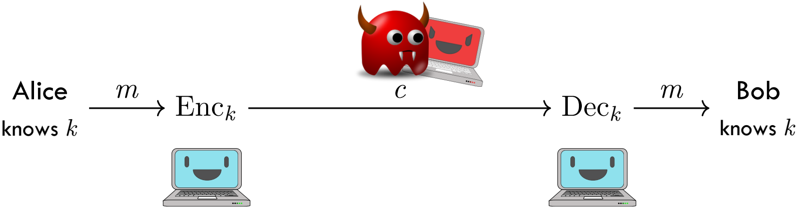
Example



Which security goals can we fulfill?

- Confidentiality? **Yes.**

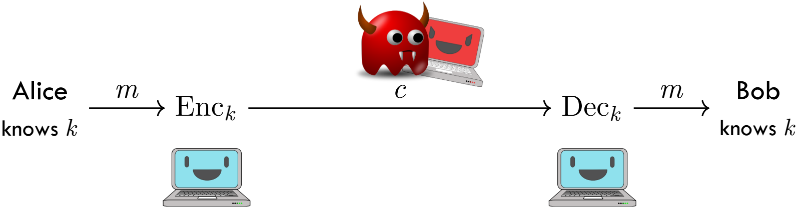
Example



Which security goals can we fulfill?

- Confidentiality? **Yes.**
- Integrity?

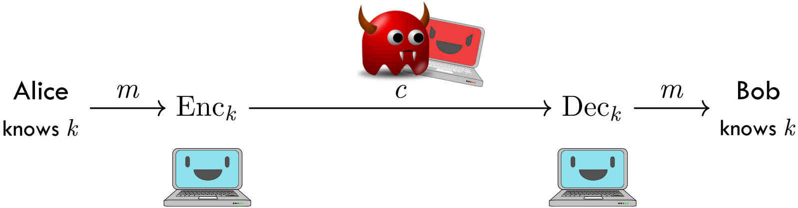
Example



Which security goals can we fulfill?

- Confidentiality? **Yes.**
- Integrity? **No!** An attacker could alter c.

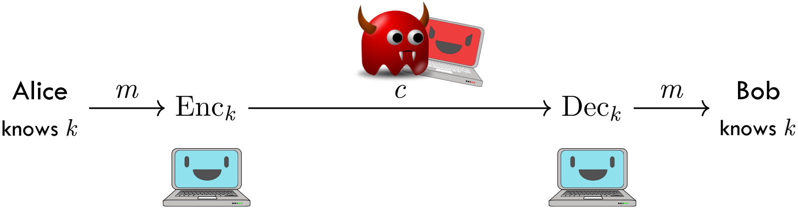
Example



Which security goals can we fulfill?

- Confidentiality? **Yes.**
- Integrity? **No!** An attacker could alter c.
- Authenticity?

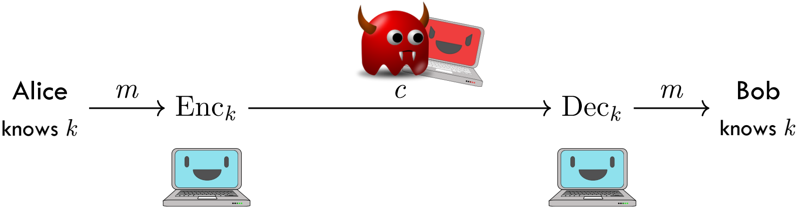
Example



Which security goals can we fulfill?

- Confidentiality? **Yes.**
- Integrity? **No!** An attacker could alter c.
- Authenticity? **No!**

Example



Which security goals can we fulfill?

- Confidentiality? **Yes.**
- Integrity? **No!** An attacker could alter c.
- Authenticity? **No!** *'But if Bob can decrypt it to something sensible and only Alice has the key, then it must be from her, right?'* **Wrong!**

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).
- $k =$ `95 EB 50 0C 31 07 46 6F 88 8A F7 0B DD FB D7 64`

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).
- $k =$ `95 EB 50 0C 31 07 46 6F 88 8A F7 0B DD FB D7 64`
- $c =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E3 B5 5D 75 F2 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).
- $k =$ `95 EB 50 0C 31 07 46 6F 88 8A F7 0B DD FB D7 64`
- $c =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E3 B5 5D 75 F2 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`
- Brute-force bytes starting from position 17 (`0x00`, `0x01`, ...)

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).
- $k =$ `95 EB 50 0C 31 07 46 6F 88 8A F7 0B DD FB D7 64`
- $c =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E3 B5 5D 75 F2 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`
- Brute-force bytes starting from position 17 (`0x00`, `0x01`, ...)
  - Interesting hit after 640 tries!

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).
- $k =$ `95 EB 50 0C 31 07 46 6F 88 8A F7 0B DD FB D7 64`
- $c =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E3 B5 5D 75 F2 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`
- Brute-force bytes starting from position 17 (`0x00`, `0x01`, ...)
  - Interesting hit after 640 tries!
- $c' =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E0 00 00 28 02 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).
- $k =$ `95 EB 50 0C 31 07 46 6F 88 8A F7 0B DD FB D7 64`
- $c =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E3 B5 5D 75 F2 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`
- Brute-force bytes starting from position 17 (`0x00`, `0x01`, …)
  - Interesting hit after 640 tries!
- $c' =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E0 00 00 28 02 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`
- Decrypts to: `Please send AUD 73` + whitespace + garbage

No authenticity

Assume 'protocol': messages end with a number, followed by whitespace

- $m =$ `Please send AUD 1␣` + arbitrary bytes (padding).
- $k =$ `95 EB 50 0C 31 07 46 6F 88 8A F7 0B DD FB D7 64`
- $c =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E3 B5 5D 75 F2 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`
- Brute-force bytes starting from position 17 (`0x00`, `0x01`, ...)
  - Interesting hit after 640 tries!
- $c' =$ `87 8B 69 74 BC 7C A5 9E D5 DA FA 15 04 9C 24 95 E0 00 00 28 02 64 D7 ED 2F 2C 21 A9 EE F8 E0 2B`
- Decrypts to: `Please send AUD 73` + whitespace + garbage

Lesson for all symmetric encryption: plausibility does not imply authenticity, in particular not in automated systems.

Construction of symmetric ciphers

Goal: obfuscate relationships between input and output

- **Confusion:** Each bit of the ciphertext depends on as many bits of the key as possible.
- **Diffusion:** Changing 1 bit of the message changes as many bits of the ciphertext as possible.
- Symmetric ciphers **scramble** the input to achieve excellent confusion and diffusion.
- Many ciphers belong to families/groups of approaches:
  - Substitution-permutation networks (AES)
  - Feistel networks (Twofish, RC6, 3DES)
  - Stream ciphers of various kinds
- Minimum recommended key length is currently 128 bit

Substitution-permutation network

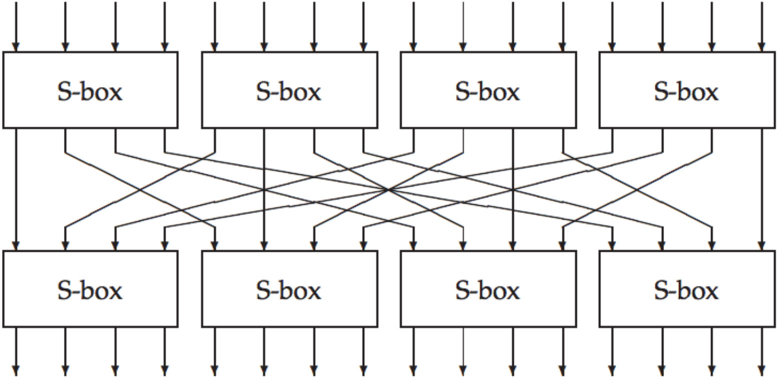


Figure 5.10: A simple 16-bit SP-network block cipher

Figure: Anderson, 2008.

Feistel network

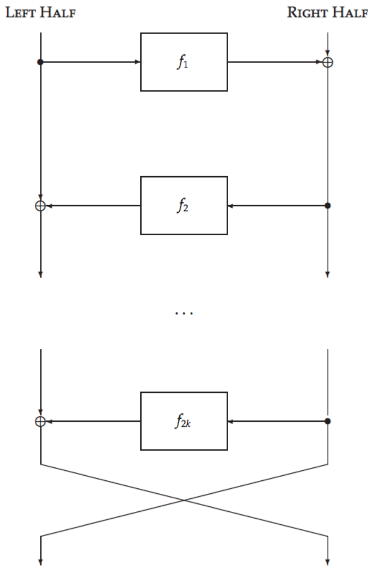


Figure 5.12: The Feistel cipher structure

- f_i may use S-boxes and P-boxes
- Figure from Anderson, 2008.

Two forms of symmetric ciphers

- Stream ciphers
  - Continous operation
  - Often very fast
  - Examples: A5/1 (GSM), RC4, ChaCha20
- Block ciphers
  - Split messages in blocks
  - Often slower, but can be computed in parallel

Stream ciphers

Idea: use key to generate a key stream

- Apply $\mathrm{Enc}_k(s)$ to obtain a **stream** $b_0, b_1, b_2, \ldots$ of bits
  - This key stream is essentially a CSPRNG
  - The sequence of output bits is unpredictable (within computational limits)
  - Security depends on seed!
- Apply a combination function c on plaintext bits $(p_0, p_1, p_2, \ldots)$ and $(b_0, b_1, b_2, \ldots)$—common choice $c = \oplus$ (XOR)
- Ciphertext: $c_0 = c(b_0, p_0); c_1 = c(b_1, p_1); c_2 = c(b_2, p_2), \ldots$

Common stream ciphers

- ChaCha20 and Salsa20
  - Family of related ciphers by DJ Bernstein
  - Current de-facto choice for stream ciphers in TLS and SSH
  - Basis of CSPRNG in OpenBSD and Linux `/dev/urandom`
- RC4
  - Used to be de-facto standard for TLS/SSL
  - Always known to have bias in first few hundred bytes—but for long believed to be workable by avoiding initial bytes
  - Feasible breakage for use in TLS in 2013 (2^{32})
  - **Avoid.**

Block ciphers

Let k be a fixed-length key.

Idea: split message in blocks of equal length

- Split the plaintext p in **blocks** of length $\frac{|p|}{|k|}$
  - apply padding if necessary
- A **block mode** defines how to apply the encryption and decryption functions $\mathrm{Enc}_k(m)$, $\mathrm{Dec}_k(m)$

Block ciphers

Variety of ciphers with different key lengths and block sizes

- Advanced Encryption Standard (AES)
  - Very well researched; product of a competition
  - Supported block sizes/key lengths: 128, 192, 256 bit
- 3DES
  - In phase-out on the Web; in use by financial industry
- Less common: Camellia, RC6, Twofish, …

Block modes

- Ciphers must handle messages of arbitrary length
- Solution: split messages in block and process them according to a **cipher block mode.**
- These modes of operation can introduce new security problems if not designed and **used** properly.
- **Classic block modes** only encrypt data
  - E.g., Electronic Codebook (ECB), Cipher Block Chaining (CBC), Counter (CTR), …
- Modern modes provide **authenticated encryption (AE, AEAD)**
  - Combine encryption and integrity protection
  - E.g., Galois Counter Mode (GCM), Counter-with-CBC-MAC (CCM) mode, …

Electronic Code Book Mode – ECB

- Block-wise: $c_i = \mathrm{Enc}_k(m_i)$

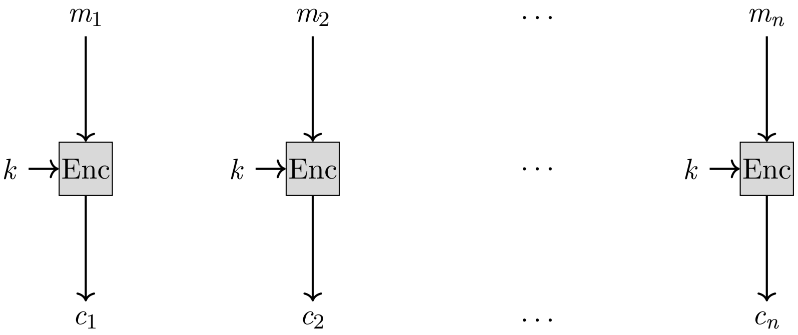


Figure: Courtesy TUM.

Why not to use ECB

Figure: Before ECB encryption.

Why not to use ECB

It is hard to give a clear-cut use case for ECB:

- Really useful mostly in teaching
- What about: errors in ECB blocks are limited to those blocks only?
  - There are other modes that have similar properties
  - Not necessarily a good thing
- What about: you can compute blocks in parallel?
  - There are other modes with similiar properties
- The above advantages are of little relevance in the vast majority of systems
- For all purposes you are likely to encounter: **do not use ECB**

CBC mode

CBC Encrypt: $c_i = \mathrm{Enc}_k(c_{i-1} \oplus m_i)$

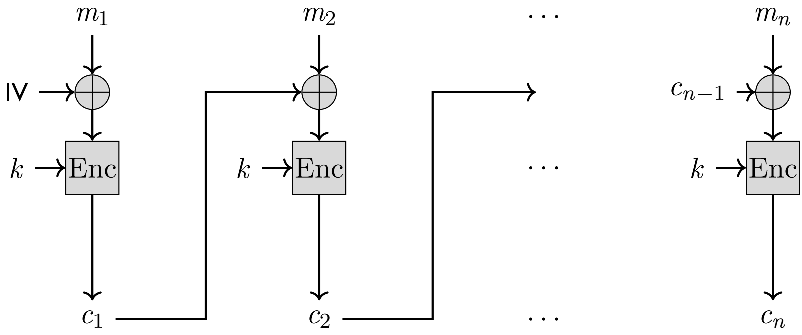


Figure: Courtesy TUM.

CBC Decrypt

$$m_i = \mathrm{Dec}_k(c_i) \oplus c_{i-1}$$

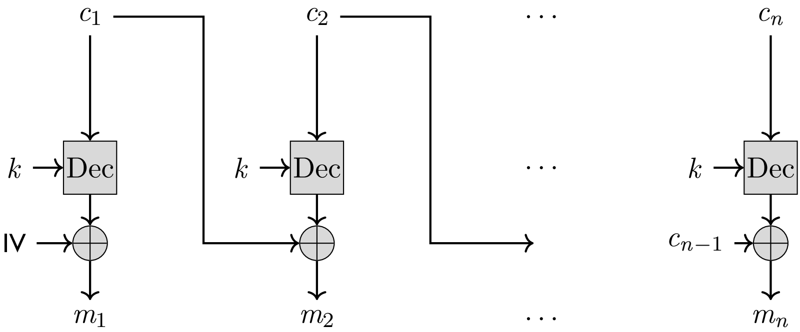


Figure: Courtesy TUM.

Example using CBC

Figure: After CBC encryption.

Construction principles

Chaining with $\oplus$:

The chaining ensures that identical plaintext blocks are encrypted to different ciphertexts.

Initialisation vector (IV):

Completely identical messages (*i.e.*, block-wise identity) are still encrypted to different ciphertexts.

Important consequences:

- You **must** choose a fresh IV for every new message.
- Does the IV have to be secret?

Construction principles

Chaining with $\oplus$:

The chaining ensures that identical plaintext blocks are encrypted to different ciphertexts.

Initialisation vector (IV):

Completely identical messages (*i.e.*, block-wise identity) are still encrypted to different ciphertexts.

Important consequences:

- You **must** choose a fresh IV for every new message.
- Does the IV have to be secret?
  - No! Why not?

These principles are used in other block modes, too.

Problems with CBC

- CBC mode not *per se* insecure.
- But error propagation can be nasty trap when using in protocols:
  - Changing one bit in ciphertext block scrambles complete plaintext block
  - Also inverts **corresponding** bit in **following** plaintext
- Opens venue for **timing-based oracle attacks**:
  - Receivers are faster or slower, depending on correctness of blocks
  - BEAST attack against HTTPS (2^{13} sessions required)
  - Lucky13 against confidentiality in TLS (2^{23} sessions)
- Too close for comfort; sparked move to other modes and stream ciphers
  - Block modes: CTR, GCM
  - RC4 broken soon after, now ChaCha20

Counter Mode – CTR

With $ctr_i = \mathrm{IV} \parallel i$, i.e., concatenation of IV and counter:
$c_i = \mathrm{Enc}_k(ctr_i) \oplus m_i$

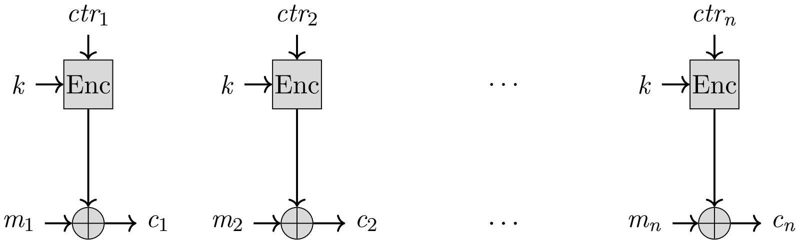
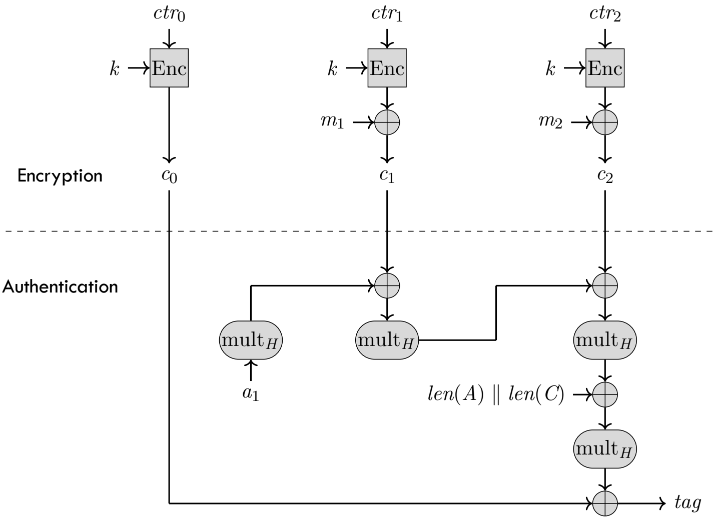


Figure: Courtesy TUM.

IV may be public but must be fresh (new and different) for every new communication! CTR is very similar to stream ciphers.

Galois Counter Mode – GCM



Mult: multiplication in $\mathrm{GF}(2^{128})$

A: additional data to authenticate

Problems with counter modes

- All counter modes absolutely require a new nonce/IV for every message that is encrypted under the same key, or immediate breakage
  - Trap for developers
  - Especially when encrypting large chunks of data!
  - How to create secure nonce
- AES-GCM is particularly brittle:
  - Very hard to implement while avoiding timing attacks
  - Nonce reuse is fatal

Some current research tries to identify counter modes with resilience to nonce misuse—but no concrete, established proposals yet.

agl on nonce reuse

> So, if you generate a random key and use it to encrypt a single message, it's ok to set the nonce to zero. If you generate a random key and encrypt a series of messages you must ensure that the nonce never repeats. A counter is one way to do this, but if you need to store that counter on disk then stop: the chances of you screwing up and reusing an nonce value are way too high in designs like that.
>
> It would be nice if reusing an nonce just meant that the same plaintext would result in the same ciphertext. That's the least bad thing that an AEAD could do in that situation. However the reality is significantly worse: common AEADs tend to lose confidentiality of messages with a repeated nonce and authenticity tends to collaspe completely for all messages. (I.e. it's very bad.) We like these common AEADs because they're fast, but you *must* have a solid story about nonce uniqueness. AEADs like AES-GCM and ChaCha20-Poly1305 fail in this fashion.

Figure: `https://www.imperialviolet.org/2015/05/16/aeads.html`

agl on encrypting large data

If you look at AEAD APIs you'll generally notice that they take the entire plaintext or ciphertext at once. In other words, they aren't "streaming" APIs. This is not a mistake, rather it's the streaming APIs that are generally a mistake.

I've complained about this in the past, so I'll be brief here. In short, old standards (e.g. PGP) will encrypt plaintexts of any length and then put an authenticator at the end. The likely outcome of such a design is that some implementations will stream out unauthenticated plaintext and only notice any problem when they get to the end of the ciphertext and try to check the authenticator. But by that time the damage has been done—it doesn't take much searching to find people suggesting piping the output of `gpg` to `tar` or even a shell.

Figure: `https://www.imperialviolet.org/2015/05/16/aeads.html`

Study: Nonce reuse in the wild

H. Boeck *et al.*: Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. WOOT 2013.

- Internet-wide scans for AES-GCM implementations in HTTPS (ca. 48M IPs)
- Straight nonce reuse rare: 184 hosts
  - VISA, German stock exchange in FRA
  - Load balancer with poor implementation to blame
- Another 70k seem to use random nonces (not counters)
  - Only problematic if they send large data
  - IBM and network equipment producers affected, and again load balancers

What does this leave us with?

- There is no cryptocalypse.
  - Just careful consideration required
  - Attacks against crypto require huge amount of sophistication
- Use AES-GCM in an implementation that gets nonces right
- Better: use current-secure ChaCha20 stream cipher (can do AEAD)

Part IV

Public-key cryptography

Public-key cryptography

- Symmetric cryptography is based on **shared keys** and **scrambling** of messages to achieve **confusion** and **diffusion**.

Public-key cryptography

- Symmetric cryptography is based on **shared keys** and **scrambling** of messages to achieve **confusion** and **diffusion.**
- Public-key cryptography is a change of paradigm in two respects:
  - Each participant has a **public key** (publicly distributed) and a **private key** (secret)
  - Anyone may use receiver's public key to **encrypt** a message
  - Only the receiver (owner of the private key) can **decrypt** it
  - Cryptographic operations based on mathematical problems with certain properties
  - For this reason, public-key cryptography is also known as asymmetric cryptography
- A number of mathematical problems are believed to have the desired properties

Operations

Two fundamental operations possible with asymmetric cryptography:

Encryption

- Bob uses Alice's public key PK to send her an encrypted message $c = \mathrm{Enc}_{PK}(m)$.
- Alice uses her private key SK to decrypt it: $m = \mathrm{Dec}_{SK}(c)$

Signatures (using cryptographic hash functions)

- Bob computes $h(m)$ and uses his **private key** to encrypt $h(m)$: $s = \mathrm{Enc}_{SK_B}(h(m)) =: \mathrm{Sig}_B(m)$. Bob sends c, s.
- Alice decrypts c to obtain m, then computes $h(m)$.
- Alice decrypts the signature: $\mathrm{Dec}_{SK_A}(\mathrm{Enc}_{SK_B}(h(m))) = h(m)$.
- Alice compares the two hash values—if they are the same, the signature is correct.

Suitable mathematical problems

Trap-door property

- Recall the resistance properties of cryptographic hash functions
- The mathematical problems we are looking for are similar functions:
  - Computationally fast to compute the function value $f(x)$
  - Computationally infeasible to compute the inverse function f^{-1} **unless** we are in possession of a certain piece of information that allows to speed up the computation
  - Hence the name 'trap-door function'
  - It is unknown if trap-door functions exist.
- There are **candidates**, and our public-key cryptography systems are built on them, *e.g.*, RSA, Diffie-Hellman, Elliptic Curve, …

Candidate problems with trap doors

The following are informal descriptions for intuition only:

Discrete logarithms in modular arithmetics

It is computationally infeasible to compute the discrete logarithm modulo p for certain p.

Discrete logarithm over elliptic curves

It is computationally infeasible to compute the discrete logarithm of an element on an elliptic curve, a special algebraic structure.

RSA problem

It is computationally infeasible to compute the e-th root of an integer modulo n, for certain n.

Candidate problems with trap doors

The following are informal descriptions for intuition only:

Discrete logarithms in modular arithmetics

It is computationally infeasible to compute the discrete logarithm modulo p for certain p.

Discrete logarithm over elliptic curves

It is computationally infeasible to compute the discrete logarithm of an element on an elliptic curve, a special algebraic structure.

RSA problem

It is computationally infeasible to compute the e-th root of an integer modulo n, for certain n. (If factorisation modulo n is feasible, RSA is also feasible. It is unknown if the inverse is true.)

Candidate problems with trap doors

The following are informal descriptions for intuition only:

Discrete logarithms in modular arithmetics

It is computationally infeasible to compute the discrete logarithm modulo p for certain p.

Discrete logarithm over elliptic curves

It is computationally infeasible to compute the discrete logarithm of an element on an elliptic curve, a special algebraic structure.

RSA problem

It is computationally infeasible to compute the e-th root of an integer modulo n, for certain n.

Unless a trap door information is available.

Diffie-Hellman

- First *public* crypto algorithm that implemented public-key (trap-door) principle
- Became the basis for:
  - Diffie-Hellman Key Exchange
  - ElGamal encryption/decryption
  - Digital Signature Standard (DSS)
- Diffie-Hellman is based on modular arithmetics
  - It can be redefined on *elliptic curves*
- We limit ourselves to the principles here

Diffie-Hellman Key Exchange: principle

(Details: Let p be a prime, g is a primitive root modulo p, *i.e.*, it generates $1, 2, \ldots, p-1 \bmod p$.)

Alice

Bob

Diffie-Hellman Key Exchange: principle

(Details: Let p be a prime, g is a primitive root modulo p, *i.e.*, it generates $1, 2, \ldots, p-1 \bmod p$.)

Alice	Bob

- Choose random value $a < p$
- Compute $X = g^a \bmod p$

Diffie-Hellman Key Exchange: principle

(Details: Let p be a prime, g is a primitive root modulo p, *i.e.*, it generates $1, 2, \ldots, p-1 \bmod p$.)

Alice	Bob
– Choose random value $a < p$	– Choose random value $b < p$
– Compute $X = g^a \bmod p$	– Compute $Y = g^b \bmod p$

Diffie-Hellman Key Exchange: principle

(Details: Let p be a prime, g is a primitive root modulo p, *i.e.*, it generates $1, 2, \ldots, p-1 \bmod p$.)

Alice

- Choose random value $a < p$
- Compute $X = g^a \bmod p$
- Send X

Bob

- Choose random value $b < p$
- Compute $Y = g^b \bmod p$
- Send Y

Diffie-Hellman Key Exchange: principle

(Details: Let p be a prime, g is a primitive root modulo p, *i.e.*, it generates $1, 2, \ldots, p-1 \bmod p$.)

Alice

- Choose random value $a < p$
- Compute $X = g^a \bmod p$
- Send X
- Compute $k = Y^a \bmod p$

Bob

- Choose random value $b < p$
- Compute $Y = g^b \bmod p$
- Send Y
- Compute $k = X^a \bmod p$

Diffie-Hellman Key Exchange: principle

(Details: Let p be a prime, g is a primitive root modulo p, *i.e.*, it generates $1, 2, \ldots, p-1 \bmod p$.)

Alice

- Choose random value $a < p$
- Compute $X = g^a \bmod p$
- Send X
- Compute $k = Y^a \bmod p$

Bob

- Choose random value $b < p$
- Compute $Y = g^b \bmod p$
- Send Y
- Compute $k = X^a \bmod p$

- $Y^a \bmod p = (g^b)^a = g^{ab} = (g^a)^b = X^b \bmod p$: both obtain k.
- Based on Discrete Logarithm: it is computationally infeasible to compute logarithms over $\bmod p$ if p is very large.

Active MitM on DH

- DH only secure in **absence of attacker who can tamper** with communication
- If an active attacker must be countered, the DHE exchange must use **signatures** on the DH values (later)

Principles of RSA (Rivest, Shamir, Adleman)

Based on infeasibility of computing roots in modular arithmetics.

Key generation

- Choose large primes p, q, compute $n = pq$ and $\phi(n) = (p-1)(q-1)$
- Choose $e, 1 < e < \phi(n)$ such that $\gcd(e, \phi(n)) = 1$
- With Extended Euclidean Algorithm, compute d with $ed \equiv 1 \bmod \phi(n)$
- Public key is (n, e); private key is d (trap door)

Key generation in practice

Generation of primes uses **randomness**: create random (large) number, test primality.

Principles of RSA

Encryption

$$c = m^e \bmod n$$

Decryption

$$m = c^d \bmod n$$

Signatures

Encrypting a hash value with private key as before.

Encryption/decryption requires mapping your message to the input space.

Problems of 'textbook' RSA

Malleability

It is possible to make predictable changes to ciphertexts. *E.g.*, if an attacker obtains c, he can compute $c' \equiv c \cdot 2^e \bmod b$, which decrypts to $2m$. (Note: ElGamal also suffers from this problem.)

No 'semantic security' in RSA

Because of the lack of a random element in the algorithm, an attacker can forward-compute (likely) messages and see if they match a given ciphertext.

Solutions

– Non-malleability and semantic security can be added with appropriate padding. The security of RSA depends on **armouring**, *i.e.*, **secure padding**.

– Unfortunately, bad padding introduces **oracle attacks**.

Padding and side-channel attacks

Padding in PKCS 1.5 standard

- Does not prescribe specific values for padding—gives rise to Bleichenbacher attack, dating to 1998
- In RSA: padding + message converted to integer, then encrypted
- Receiver **must not** indicate if decrypted padding was correct
  - Else, attacker has an oracle to try: send random strings until one produces a valid padding

PKCS-OAEP

- Defines a provably secure padding
- Much more specific on padding bytes, makes it computationally infeasible to find a random string that produces a valid padding.

Hybrid encryption

Actual use case for public-key cryptography

- Public-key cryptography is **slow—**long keys!
- Also has small input space to map messages onto
- Solution:
  - Generate random symmetric key k (ideally: Diffie-Hellman)
  - Encrypt actual message as $c_m = \mathrm{Enc}_k(m)$
  - Encrypt k as $c_k = \mathrm{Enc}_{PK}(k)$
  - Send (c_k, c_m)

All cryptographic protocols we are going to describe use hybrid encryption in some form.

Part V

Cryptography in deployment

Definition: effective key length, security margin

- **Effective** key length: function of total key length and suceptibility to best 'practical' attacks
  - Some attacks are of no practical relevance: they can be avoided by proper use of the cipher
  - AES-128 is estimated at 126 bits 'of security' for attacks of practical relevance; 3DES at 80 (key length 128 bit)
- **Cryptographically broken**: an attack exists that is better than brute-forcing the key
  - Not the same as **practical** or **feasible**
  - E.g., AES is considered secure
- **Security margin**: an **estimate** how much better the best-known attack must become in order to achieve **practical** breakage
  - A **rough** estimate, based on experience and previous effort

Weak spots in cryptography

Many things to consider

- Security of cipher (cf. RC4, DES) and appropriate key length
- Correct use for security goal (encryption, authentication, …)
- Correct choice of cipher mode (cf. ECB vs. CBC vs. GCM)
- Need for entropy (both ciphers and cipher modes)
- Armouring (padding) in asymmetric crypto
- Avoiding side-channels (CPU, oracles, …) in implementations

Weak spots in cryptography

Many things to consider

- Security of cipher (cf. RC4, DES) and appropriate key length
- Correct use for security goal (encryption, authentication, …)
- Correct choice of cipher mode (cf. ECB vs. CBC vs. GCM)
- Need for entropy (both ciphers and cipher modes)
- Armouring (padding) in asymmetric crypto
- Avoiding side-channels (CPU, oracles, …) in implementations

And this is just to get the building block 'crypto' right. Further issues exist when using the building block in conjunction with others to, *e.g.*, construct protocols.

Typical attacks

- **Rare, but bad:** cryptanalysis with effective keylength reduced such that is within reach of current computational power
  - *E.g.*, MD5 (1995) and RC4 (2013)
  - But even here: HMAC-MD5 not affected (yet?)
- **Often exploitable:** side-channel attacks
  - Attack where applying the cryptography leads to observable reaction that gives away some information
  - Oracle attacks: meddling with message in transmission leads to reaction in receiver—*e.g.*, a reply with error message
  - Timing attacks—measure message delay, or CPU time
  - Emission attacks: heat, fan, noise
- **Often fatal:** wrong implementation of cryptographic algorithm or poor entropy
  - *E.g.*, predictable or reused IV

Fascinating example

Researchers crack the world's toughest encryption by listening to the tiny sounds made by your computer's CPU

By Sebastian Anthony on December 18, 2013 at 2:27 pm | 94 Comments

15.0K shares

Security researchers have successfully broken one of the most secure encryption algorithms, 4096-bit RSA, by listening — yes, with a *microphone* — to a computer as it

Remaining agenda

- We are now going to look at cases where cryptography fails in practice
- In particular, three research papers that uncovered evidence of cryptographic failure and determined the impact by using *empirical measurement*
    - Debian's OpenSSL bug
    - Factorable RSA, vulnerable DSS
    - Vulnerable Diffie-Hellman

Empirical analyses

What impact would weaknesses have *in practice*?

- What is the quality of the deployed cryptography, in actual implementations?
- In which protocols, services, applications, and devices is weak cryptography used?
- How fast are implementations updated after a weakness becomes known?

Answers can be obtained by active scans and passive monitoring

- Active scans—large-scale samples or Internet-wide
- Passive monitoring local ISP traffic
- Long-term observations to determine update trends

Debian RNG vulnerability in OpenSSL library

Debian Security Advisory

DSA-1571-1 openssl -- predictable random number generator

Date Reported:
13 May 2008

Affected Packages:
openssl

Vulnerable:
Yes

Security database references:
In Mitre's CVE dictionary: CVE-2008-0166.

More information:

Luciano Bello discovered that the random number generator in Debian's openssl package is predictable. This is caused by an incorrect Debian-specific change to the openssl package (CVE-2008-0166). As a result, cryptographic key material may be guessable.

Figure: Bug fix introduces vulnerability.

What had happened?

```
The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif
```

- Valgrind codechecker was complaining about reading from uninitialised memory
  - Unitialised buffer has very little entropy—removing is no problem?
  - Code pieces **seem** to be a read/hash operation on the same buffer

What had happened?

```
The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif
```

- Change in #467 is OK!
  - Called by application when it needs random bytes
  - Buffer content before it is filled is added to entropy pool
  - Fine to remove

What had happened?

```
The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif
```

- Change in #247 is different
  - Called by `RAND_add` to add bytes to entropy pool—in some places, this is called on unitialised buffer
  - **Here**, it adds the buffer holding **most** of the entropy to the pool
  - Result: a PRNG with super-low entropy seeded

What had happened?

```
The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif
```

- Entropy is used in key generation
  - This code would always produce the same public/private key pairs (≈ 200k)
- Bug occurred in 2006
  - Found and fixed in 2008

Vulnerability significance

What types of systems are affected?

- OpenSSL's primary use is as a full SSL/TLS implementation
- PRNG is used in many parts of the protocol:
  - Creation of symmetric key (without Diffie-Hellman)
  - Creation of Diffie-Hellman parameters
  - Public/private key pairs
- However, mostly a server problem:
  - Browsers rarely use OpenSSL
  - Linux/Unix are relatively rare among end-users
  - Servers, however, do often run under Linux/Unix

Vulnerability significance

Client-side—the attacker can:

- Precompute symmetric keys
- Precompute Diffie-Hellman values (yields symmetric key)
- Authenticate as client (rare use case)

Server-side—the attacker can:

- Precompute Diffie-Hellman values (harder than in client case)
- Authenticate as server!

Attack types

- Person-in-the-middle attack—if attacker has good network position or can attack routing, too
- Can decrypt recorded traffic later

Impact (affected deployments)

Paper

When private keys are public—Results from the 2008 Debian OpenSSL vulnerability. S. Yilek *et al.*, 2009

Impact (affected deployments)

Paper

When private keys are public—Results from the 2008 Debian OpenSSL vulnerability. S. Yilek *et al.*, 2009

Methodology

- Precompute possible RSA public/private key pairs
- Scan $\approx$ 50,000 hosts on the HTTPS port
  - Daily, starting 4 days after discovery
  - For six months
- Retrieve certificates
- Compare against precomputed keys

Impact (affected deployments)

Paper

When private keys are public—Results from the 2008 Debian OpenSSL vulnerability. S. Yilek *et al.*, 2009

Key results

– ≈ 750 had a vulnerable public/private key pair (1.5%)
– 30% of hosts still had one 6 months after discovery
– New certificates with vulnerable key were still being issued long after vulnerability was fixed

Aftermath and engineering lessons

- Changes to cryptographic implementations need careful review between ‘upstream’ (here: OpenSSL) and ‘downstream’ developers (here: Debian)
- Weaknesses can remain undetected for months or years
- Loss of entropy is desastrous for RSA key generation
- Update rate slower than expected
  - Hard to determine causes, but not implausible: some devices are hard to update (*e.g.*, embedded with requirement for firmware update)
  - Later studies showed that vulnerable keys were phased out
  - Blacklists with vulnerable keys were distributed

Mining Ps and Qs

- Heninger *et al.*: *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices, USENIX Security*, 2012.

Mining Ps and Qs

- Heninger *et al.*: *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices, USENIX Security*, 2012.
- Tests hypothesis:
  Weak entropy has been known to be a source of cryptographic failure for a very long time, so one would expect that today's systems do not exhibit weaknesses due to poor randomness.
- Based on Internet-wide scans of servers with public keys
  - SSL/TLS and SSH
  - Determine if public keys are secure (strong) or weak
- The findings contradicted the hypothesis and showed a systematic use of **poor entropy** on certain Internet-facing systems

Study summary

Internet-wide scans

- Scanned entire IPv4 space (in $\approx$ 24hrs)
  - 5.8M unique TLS certificates (from 12.8M hosts)
  - 6.2M unique SSH keys (from 10.2M hosts)

Key reuse and key weakness

- 5.5% of TLS hosts and 9.6% of SSH hosts use the same keys as other hosts in a vulnerable manner
- Could compute private keys for 0.5% of TLS public keys and 1.1% in case of SSH
- Investigated **why, under which circumstances, the weaknesses occur**

TLS and SSH in brief

- Protocols carry out similar handshakes between client and server
- TLS most commonly uses server-only authentication:
  - Server authenticates with public key (in certificate) plus challenge-response protocol
  - Carry out **signed** Diffie-Hellman or hybrid scheme
- SSH: two versions, but only SSH2 has serious deployment today
  - Server authenticates with public key in challenge response protocol
  - Always signed **signed** Diffie-Hellman

Scans and post-processing

- 2010: Electronic Frontier Foundation carried out the first Internet-wide scan of TLS
  - Scans took several months
  - Hence some drawbacks in methodology, but wide publicity
- Heninger's study could scan IPv4 much faster by using cloud-based scanners
  - 25 instances for host discovery (`nmap`)—24 hours
  - 1 instance for TLS and SSH handshakes—96 hours
- Data post-processing
  - Parsing certificates
  - Identifying vulnerable device models from certificate fields and TCP/IP fingerprinting
  - **Responsible disclosure**

We defer the details of scanning for later.

Repeated keys

- Key reuse is rampant on the Internet
  - 61% (7.7M) of TLS hosts, 65% of SSH hosts (6.6M) use a non-unique key
  - Common reason: large hosting providers
  - Cannot clearly say if vulnerability or not: hosting setup unknown
- But two common **vulnerabilities**:
  - Manufacturer default keys in devices (RSA: $> 5\%$)
  - Repeated keys due to **insufficient** entropy (RSA: $\approx 0.3\%$)
- Also found: short keys
  - Almost 1% of RSA keys were 512 bit—too short for today

RSA weakened by poor entropy

- You will recall that keys in RSA are created by choosing two large, **random** values p, q and testing for primality to compute $n = pq$.
- What happens if two independent systems happen to choose the same two random numbers?

RSA weakened by poor entropy

- You will recall that keys in RSA are created by choosing two large, **random** values p, q and testing for primality to compute $n = pq$.
- What happens if two independent systems happen to choose the same two random numbers?
  - $p_1 = p_2$ and $q_1 = q_2$: devices have the same key $n = pq$

RSA weakened by poor entropy

- You will recall that keys in RSA are created by choosing two large, **random** values p, q and testing for primality to compute $n = pq$.
- What happens if two independent systems happen to choose the same two random numbers?
  - $p_1 = p_2$ and $q_1 = q_2$: devices have the same key $n = pq$
  - Then they also have the same private key.

RSA weakened by poor entropy

- You will recall that keys in RSA are created by choosing two large, **random** values p, q and testing for primality to compute $n = pq$.
- What happens if two independent systems happen to choose the same two random numbers?
  - $p_1 = p_2$ and $q_1 = q_2$: devices have the same key $n = pq$
  - Then they also have the same private key.
  - And it might be possible to extract it by just buying one such device.

Common repeated keys—RSA (SSH)

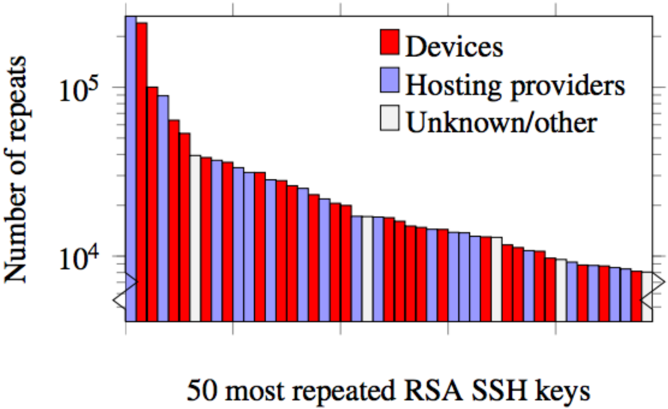


Figure: Heninger *et al.*, 2012.

Common repeated keys—DSS (SSH)

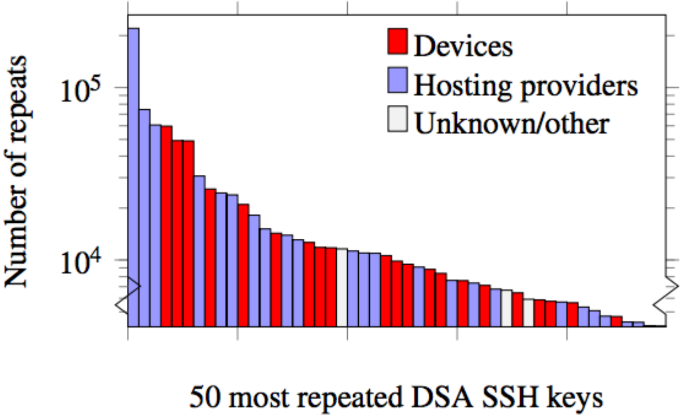


Figure: Heninger *et al.*, 2012.

Returning to RSA

- What happens if only one prime is the same?

Returning to RSA

- What happens if only one prime is the same?
  - $n_1 = pq_1$ and $n_2 = pq_2$, or $n_1 = p_1 q$ and $n_2 = p_2$
    - then $\gcd(n_1, n_2) = p$ or $\gcd(n_1, n_2) = q$

Returning to RSA

- What happens if only one prime is the same?
  - $n_1 = pq_1$ and $n_2 = pq_2$, or $n_1 = p_1 q$ and $n_2 = p_2$
    - then $\gcd(n_1, n_2) = p$ or $\gcd(n_1, n_2) = q$
  - Computing the GCD is **fast**.

Returning to RSA

- What happens if only one prime is the same?
  - $n_1 = pq_1$ and $n_2 = pq_2$, or $n_1 = p_1 q$ and $n_2 = p_2$
    - then $\gcd(n_1, n_2) = p$ or $\gcd(n_1, n_2) = q$
  - Computing the GCD is **fast**.
  - Once we know p, then q_1, q_2 are trivial to compute.

Returning to RSA

- What happens if only one prime is the same?
  - $n_1 = pq_1$ and $n_2 = pq_2$, or $n_1 = p_1 q$ and $n_2 = p_2$
    - then $\gcd(n_1, n_2) = p$ or $\gcd(n_1, n_2) = q$
  - Computing the GCD is **fast**.
  - Once we know p, then q_1, q_2 are trivial to compute.
  - The private key is then easy to compute.

Returning to RSA

- What happens if only one prime is the same?
  - $n_1 = pq_1$ and $n_2 = pq_2$, or $n_1 = p_1 q$ and $n_2 = p_2$
    - then $\gcd(n_1, n_2) = p$ or $\gcd(n_1, n_2) = q$
  - Computing the GCD is **fast**.
  - Once we know p, then q_1, q_2 are trivial to compute.
  - The private key is then easy to compute.
- Problem: 6×10^{13} moduli in the data.
  - Still takes decades to compute pairwise GCDs

Returning to RSA

- What happens if only one prime is the same?
  - $n_1 = pq_1$ and $n_2 = pq_2$, or $n_1 = p_1 q$ and $n_2 = p_2$
    - then $\gcd(n_1, n_2) = p$ or $\gcd(n_1, n_2) = q$
  - Computing the GCD is **fast**.
  - Once we know p, then q_1, q_2 are trivial to compute.
  - The private key is then easy to compute.
- Problem: 6×10^{13} moduli in the data.
  - Still takes decades to compute pairwise GCDs
  - Unless you use a shortcut

Product tree + remainder tree (idea: djb)

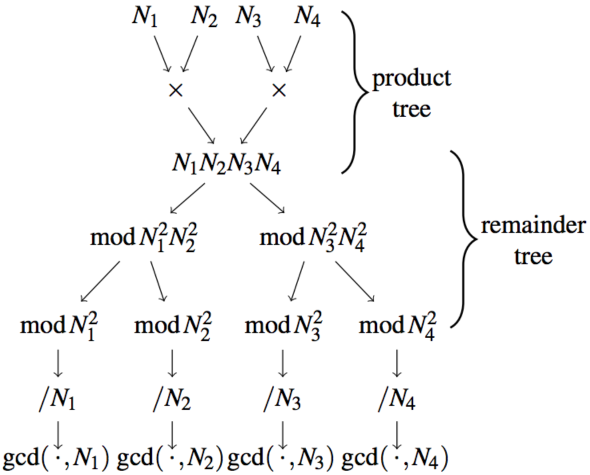
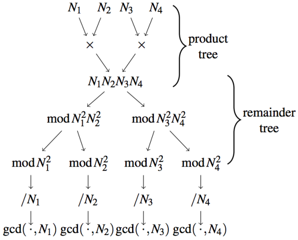


Figure: Heninger *et al.*, 2012.

fastgcd



– Allows to reduce computation from decades to hours (or on AWS: 1hr, USD 5).

Factorable keys

- 11M+ distinct RSA moduli (included EFF data)
  - 2300 distinct prime divisors
  - dividing 16,717 distinct public keys

Factorable keys

- 11M+ distinct RSA moduli (included EFF data)
  - 2300 distinct prime divisors
  - dividing 16,717 distinct public keys
- Allowed to compute 24,000 private keys for TLS
  - Used on 64,000 hosts
- And 1000 RSA SSH host keys on 2500 hosts

Factorable keys

- 11M+ distinct RSA moduli (included EFF data)
  - 2300 distinct prime divisors
  - dividing 16,717 distinct public keys
- Allowed to compute 24,000 private keys for TLS
  - Used on 64,000 hosts
- And 1000 RSA SSH host keys on 2500 hosts
- Who are these devices?

Factorable keys

- 11M+ distinct RSA moduli (included EFF data)
  - 2300 distinct prime divisors
  - dividing 16,717 distinct public keys
- Allowed to compute 24,000 private keys for TLS
  - Used on 64,000 hosts
- And 1000 RSA SSH host keys on 2500 hosts
- Who are these devices?
  - Clustered by divisor

Factorable keys

- 11M+ distinct RSA moduli (included EFF data)
  - 2300 distinct prime divisors
  - dividing 16,717 distinct public keys
- Allowed to compute 24,000 private keys for TLS
  - Used on 64,000 hosts
- And 1000 RSA SSH host keys on 2500 hosts
- Who are these devices?
  - Clustered by divisor
  - Allowed to identify 41 vendors

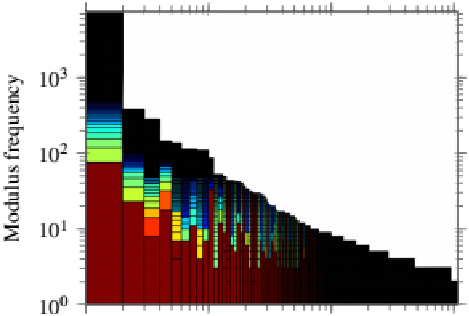
Factorable keys

- 11M+ distinct RSA moduli (included EFF data)
  - 2300 distinct prime divisors
  - dividing 16,717 distinct public keys
- Allowed to compute 24,000 private keys for TLS
  - Used on 64,000 hosts
- And 1000 RSA SSH host keys on 2500 hosts
- Who are these devices?
  - Clustered by divisor
  - Allowed to identify 41 vendors
    - routers, firewalls, headless/embedded devices

Factorable keys

- 11M+ distinct RSA moduli (included EFF data)
  - 2300 distinct prime divisors
  - dividing 16,717 distinct public keys
- Allowed to compute 24,000 private keys for TLS
  - Used on 64,000 hosts
- And 1000 RSA SSH host keys on 2500 hosts
- Who are these devices?
  - Clustered by divisor
  - Allowed to identify 41 vendors
    - routers, firewalls, headless/embedded devices
  - Vast majority: Juniper router
    - 47,000 devices in dataset, 27% vulnerable

Primes allow to identify vendors



(a) Primes generated by Juniper network security devices

Figure: Heninger *et al.*, 2012.

Long-tail distribution typical for many devices; prime factors identify them.

Weaknesses in DSS

- Recall DSS property:
  - Every signature must use new, ephemeral, random z
  - Reuse of z: attacker can compute private key

Weaknesses in DSS

- Recall DSS property:
  - Every signature must use new, ephemeral, random z
  - Reuse of z: attacker can compute private key
- Study found hosts that reuse z:
  - 9M signatures collected (mostly 2 from each host)
  - 4300 signatures reused a previous z
- Allowed to compute private keys for 105,000 hosts (1.6%)
  - Key reuse among hosts!

Analysis for one vendor

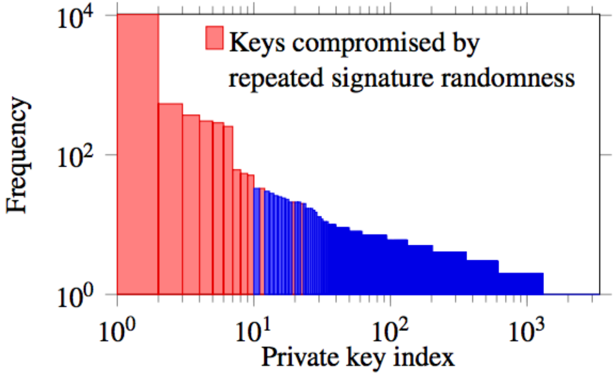


Figure: Heninger *et al.*, 2012.

Correlation between how often a key occurs and how often it can be compromised due to reused z.

Root causes

- The study could determine several causes for the observed bad entropy
  - Linux boot-time entropy hole on some devices
  - Key generation with OpenSSL on some devices
  - Entropy hole at start-up time: SSH server Dropbear affected

Linux boot-time entropy hole

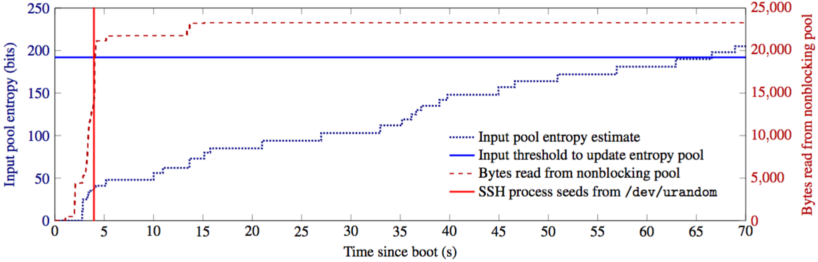
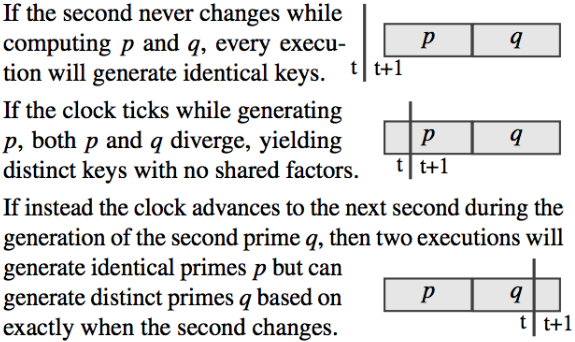


Figure: Heninger *et al.*, 2012.

Weak devices (headless, embedded, etc.) at risk: reading from `/dev/urandom/` too shortly after boot causes predictable seed for CSPRNG.

OpenSSL key generation


If the second never changes while computing p and q, every execution will generate identical keys.

If the clock ticks while generating p, both p and q diverge, yielding distinct keys with no shared factors.

If instead the clock advances to the next second during the generation of the second prime q, then two executions will generate identical primes p but can generate distinct primes q based on exactly when the second changes.

Figure: Heninger *et al.*, 2012.

On *otherwise identically configured* devices, the only entropy source is time.

Conclusions from study

- Lack of entropy is desastrous for RSA and DSS
  - RSA only affected during key generation
  - DSS affected *on every signature*
- `/dev/urandom` is a usability failure
  - There is no way for developers to determine a condition of low entropy safely
  - Suggested fail-safe alternative: FreeBSD-like mode (block until entropy high, then never again)

Engineering lessons

- OS developers:
  - Fail-safe behaviour of `/dev/urandom`; communicate entropy to applications
  - Test CSPRNG on *weak* platforms, too
- Application developers and end-users
  - Generate keys on first use, not at boot-time
  - Regenerate any default key
- Device manufacturers
  - Do not use default keys
  - Seed entropy at factory for embedded or headless devices
  - Test CSPRNG on device before going into production

Imperfect forward security

- Adrian *et al.*: *Imperfect forward security: how Diffie-Hellman fails in practice*, CCS 2015.
- Practical, computational attack on Diffie-Hellman key exchange
  - Attacks 'export-grade' DH by precomputation
  - Determines impact on Internet servers
  - Gives estimate of impact if precomputation for 1024bit DH should be possible
- Shows danger of **legacy cryptography** and **downgrading** attacks

Ingredients for attack (background)

- In the 1990s, US export restrictions prohibited the ‘export’ of cryptography with a certain strength

Ingredients for attack (background)

- In the 1990s, US export restrictions prohibited the ‘export’ of cryptography with a certain strength
- SSL and TLS implementations complied by supporting ‘export grade’ Diffie-Hellman, with primes of no more than 512 bit

Ingredients for attack (background)

- In the 1990s, US export restrictions prohibited the ‘export’ of cryptography with a certain strength
- SSL and TLS implementations complied by supporting ‘export grade’ Diffie-Hellman, with primes of no more than 512 bit
- ‘512 bit cryptography’ is today within reach of the so-called *Number Field Sieve*, the fastest known algorithm to compute discrete logarithms (and for factoring)

Ingredients for attack (background)

- In the 1990s, US export restrictions prohibited the ‘export’ of cryptography with a certain strength
- SSL and TLS implementations complied by supporting ‘export grade’ Diffie-Hellman, with primes of no more than 512 bit
- ‘512 bit cryptography’ is today within reach of the so-called *Number Field Sieve*, the fastest known algorithm to compute discrete logarithms (and for factoring)
- Despite official phase-out from $\approx$ 2000 on, 512 bit primes are still supported by servers
  - Historically not considered a problem as clients will not offer such weak primes, and can reject them

Ingredients for attack (background)

- In the 1990s, US export restrictions prohibited the ‘export’ of cryptography with a certain strength
- SSL and TLS implementations complied by supporting ‘export grade’ Diffie-Hellman, with primes of no more than 512 bit
- ‘512 bit cryptography’ is today within reach of the so-called *Number Field Sieve*, the fastest known algorithm to compute discrete logarithms (and for factoring)
- Despite official phase-out from $\approx$ 2000 on, 512 bit primes are still supported by servers
  - Historically not considered a problem as clients will not offer such weak primes, and can reject them
- But can use a flaw in TLS (not vulnerability!) to downgrade strong DH to export-grade

Number field sieve

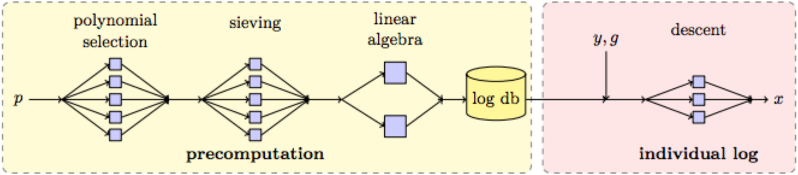


Figure: Adrian, 2015.

First three stages depend only on prime p that determines the group.

Number field sieve

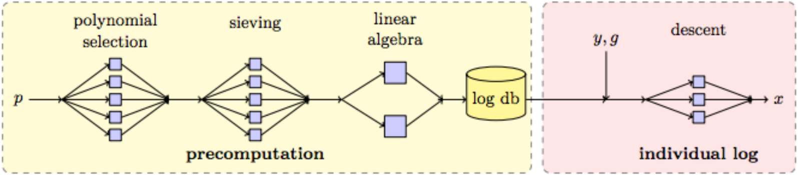


Figure: Adrian, 2015.

First three stages depend only on prime p that determines the group.
It can be done in advance for *known* p.

Number field sieve

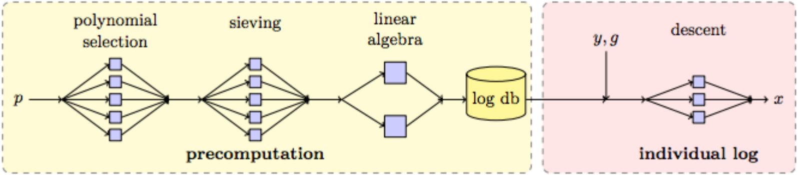


Figure: Adrian, 2015.

First three stages depend only on prime p that determines the group.
It can be done in advance for *known* p.
Only last stage depends on ephemeral DH parameters of the handshake.

Empirical results for 512 bit primes

Source	Popularity	Prime
Apache	82%	9fdb8b8a004544f0045f1737d0ba2e0b 274cdf1a9f588218fb435316a16e3741 71fd19d8d8f37c39bf863fd60e3e3006 80a3030c6e4c3757d08f70e6aa871033
mod_ssl	10%	d4bcd52406f69b35994b88de5db89682 c8157f62d8f33633ee5772f11f05ab22 d6b5145b9f241e5acc31ff090a4bc711 48976f76795094e71e7903529f5a824b
(*others*)	8%	(463 distinct primes)

Figure: Adrian, 2015.

Only very few primes (groups) in use for 512 bit export-grade crypto.

Empirical results for 512 bit primes

Source	Popularity	Prime
Apache	82%	9fdb8b8a004544f0045f1737d0ba2e0b 274cdf1a9f588218fb435316a16e3741 71fd19d8d8f37c39bf863fd60e3e3006 80a3030c6e4c3757d08f70e6aa871033
mod_ssl	10%	d4bcd52406f69b35994b88de5db89682 c8157f62d8f33633ee5772f11f05ab22 d6b5145b9f241e5acc31ff090a4bc711 48976f76795094e71e7903529f5a824b
(*others*)	8%	(463 distinct primes)

Figure: Adrian, 2015.

Only very few primes (groups) in use for 512 bit export-grade crypto.
Precomputation stages for three most common groups : $\approx$ **3 weeks.**

Empirical results for 512 bit primes

Source	Popularity	Prime
Apache	82%	9fdb8b8a004544f0045f1737d0ba2e0b 274cdf1a9f588218fb435316a16e3741 71fd19d8d8f37c39bf863fd60e3e3006 80a3030c6e4c3757d08f70e6aa871033
mod_ssl	10%	d4bcd52406f69b35994b88de5db89682 c8157f62d8f33633ee5772f11f05ab22 d6b5145b9f241e5acc31ff090a4bc711 48976f76795094e71e7903529f5a824b
(*others*)	8%	(463 distinct primes)

Figure: Adrian, 2015.

Only very few primes (groups) in use for 512 bit export-grade crypto.

Precomputation stages for three most common groups : $\approx$ **3 weeks.**

One individual log in one of these groups: takes $\approx$ **70s.**

Exploiting a flaw in TLS

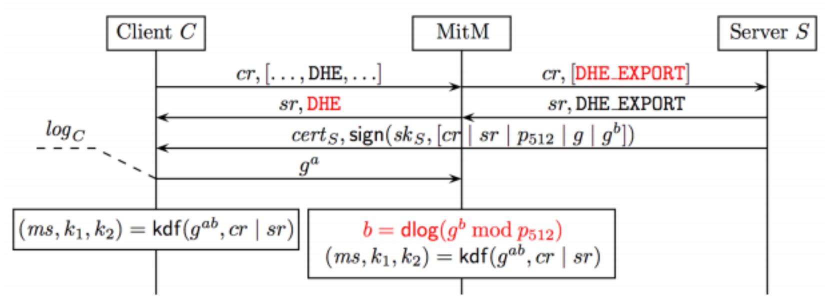


Figure: Adrian, 2015.

Exploiting a flaw in TLS

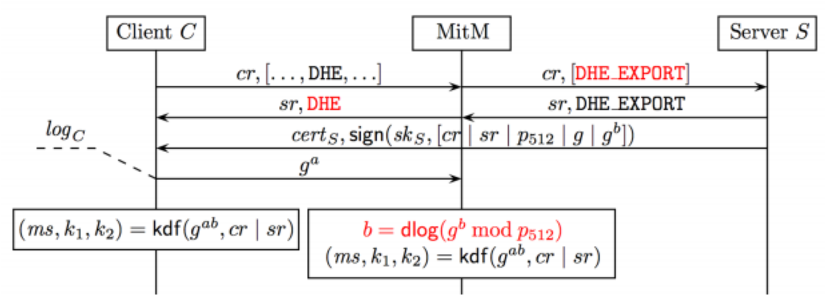


Figure: Adrian, 2015.

The choice of **DHE strength** is **not** included in TLS's signatures.

Exploiting a flaw in TLS

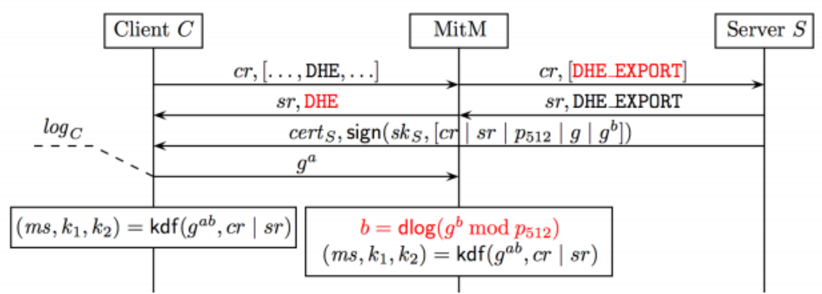


Figure: Adrian, 2015.

The choice of **DHE strength** is **not** included in TLS's signatures.
Attacker can downgrade to export-grade DHE.

Exploiting a flaw in TLS

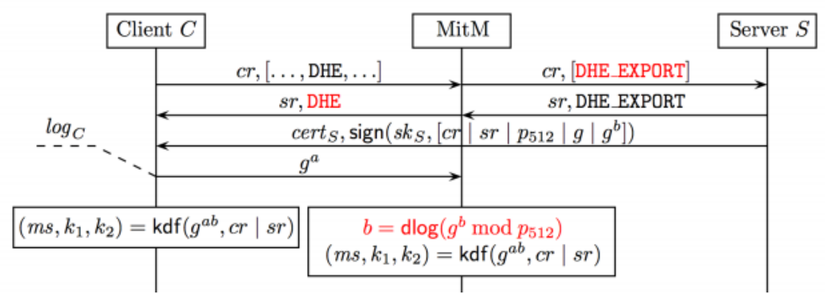


Figure: Adrian, 2015.

The choice of **DHE strength** is **not** included in TLS's signatures.

Attacker can downgrade to export-grade DHE.

Computation of the log is possible within 70s—practically real-time.

Defences

- The immediate defence to the attack is to have all clients reject small primes, independently of what the server signals.
  - In the wake of disclosure, all major browser vendors stopped accepting export-grade cryptography
- Server vendors began to roll out updates that deactivated this cryptography, too.
- IETF recommendations today call for 2048 bit DHE
  - But had to consider old implementations (Java!) that could not easily follow.
- Call for more use of Elliptic Curve cryptography
  - Not without its problems, either

Extrapolating to 1024 bit?

- The study attempts to estimate the computational power needed to carry out the precomputation stage for 'large' primes (1024 bit)
  - Large number of factors involved—the authors put it at 45M core years
  - This might, or might not, just be within reach of nation-state with ample resources
- Much depends on how many primes are commonly used on Internet servers

Estimate of impact

	Vulnerable servers, if the attacker can precompute for ...			
	all 512-bit groups	all 768-bit groups	one 1024-bit group	ten 1024-bit groups
HTTPS Top 1M w/ active downgrade	45,100 (8.4%)	45,100 (8.4%)	205,000 (37.1%)	309,000 (56.1%)
HTTPS Top 1M	118 (0.0%)	407 (0.1%)	98,500 (17.9%)	132,000 (24.0%)
HTTPS Trusted w/ active downgrade	489,000 (3.4%)	556,000 (3.9%)	1,840,000 (12.8%)	3,410,000 (23.8%)
HTTPS Trusted	1,000 (0.0%)	46,700 (0.3%)	939,000 (6.56%)	1,430,000 (10.0%)
IKEv1 IPv4	–	64,700 (2.6%)	1,690,000 (66.1%)	1,690,000 (66.1%)
IKEv2 IPv4	–	66,000 (5.8%)	726,000 (63.9%)	726,000 (63.9%)
SSH IPv4	–	–	3,600,000 (25.7%)	3,600,000 (25.7%)

Figure: Adrian, 2015.

These are rough **estimates. If** 1024 bit are breakable, a sizeable fraction of Internet traffic could be impacted.
One clear message is: the security margin of 1024 bit DHE is much smaller than we would like.

Engineering lessons

- Security-wise, ‘export-grade’ cryptography is a very poor idea that can turn against its creators.
  - Once ‘genie is out of the bottle’, it is hard to put it back in.
- Negotiation of cryptographic parameters can be highly beneficial
  - Does not remove problem entirely: some primes are large, yet weak for other reasons—hard to detect for a client, as attacker could attempt to ‘downgrade’ to such a prime.
- ‘Crypto agility’ is a useful property—the ability to easily replace one primitive with another without major upgrades and rollouts.
- Client-side detection of poor cryptography as offered by the server.

Keeping track of crypto

- Cryptographic developments are not announced centrally
- Good points of call:
  - NIST—`nist.gov`
  - IETF—*e.g.*, Crypto Forum Research Group (CFRG)
  - Websites providing guidance
    - *e.g.*, `bettercrypto.org`
  - Mailing lists
    - `www.metzdowd.com/mailman/listinfo/cryptography`
    - `lists.randombit.net/mailman/listinfo/cryptography`

References (papers and talks)

- S. Yilek *et al.*: When private keys are public—Results from the 2008 Debian OpenSSL vulnerability. Proc. Internet Measurement Conference, 2009.
- N. Heninger *et al.*: Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. Proc. 21st USENIX Security Symposium, Aug 2012.
- D. Adrian *et al.*: Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. Proc. 22nd ACM Conference on Computer and Communications Security (CCS), Oct 2015.
- T. Ramos: The Laws of vulnerabilities. Talk at RSA Conference, 2006.
- AlFardan and Paterson: Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. `http://www.isg.rhul.ac.uk/tls/Lucky13.html`.

References (books)

- W. Stallings: Cryptography and Network Security, 6th ed., 2016.
- J. Katz, Y. Lindell: Introduction to modern Cryptography, 2nd ed., 2014.