Software Security

Avoiding bugs in cryptographic mechanisms

Panayiotis Kotzanikolaou
{ pkotzani at unipi dot gr }

Dimitrios Glynos
{ daglyn at unipi dot gr }

Department of Informatics
University of Piraeus

Part I

Terminology

Terminology

- plaintext: the input data to an encryption process.
- ciphertext: the data output by an encryption process.
- preimage: the input to a hash function.
- hash: the output of a hash function.
- cipher: a reversible transformation (usually for encryption purposes) requiring a key, where only a key-holder can reverse the transformation and the transform bears no resemblance to the input (Example: AES).
- nonce: a random quantity used in transformations to produce different results from the same input.

Terminology (cont'd)

- symmetric key cryptography: the same secret key is used for the encryption of the plaintext and decryption of the ciphertext (Example: AES).
- asymmetric key cryptography (or public key cryptography): encryption occurs with a public key and decryption with a private key (Example: RSA).
- block cipher: the transformation works on a *block* of data (with *padding* if required) and where next blocks are generated according to a *mode* of operation (e.g. AES in CBC mode with PKCS#5 padding).
- stream cipher: a transformation based on a symmetric key scheme that outputs bits that correspond to the input bits, encrypting bit by bit (Example: ChaCha20).

Terminology (cont'd)

- cryptographic hash function: a one-way transformation that makes it difficult to deduce the input that led to the transform and where the transform has a very low probability for collisions (Example: SHA512).
- HMAC: a keyed transform involving a cryptographic hash function that mixes[1] the secret key with the input in a way that only the key holder will be able to produce a correct transform (Example: HMAC-SHA512).

[1]Due to the fact that the mixing occurs with more than one application of the cryptographic hash function, it is safe to say that the HMAC is stronger than the cryptographic hash function it applies (i.e. HMAC-SHA1 is not necessarily broken due to SHA1 being broken).

Terminology (cont'd)

- key derivation function: a transformation that perturbates a series of bits (usually through a cryptographic hash function applied multiple times) into a form that carries the desired properties for a cryptographic key (Example: PBKDF2).
- digital signature: a scheme where the cryptographic hash of a document is encrypted with the private key of the author and the document reader may later verify using the public key of the author (Example: DSA).

Part II

Good practices

Applied Cryptography: A useful tool

Applied Cryptography is commonly used as a tool for

- Securely *identifying actors* to systems (authentication)
- Securely *transferring data* over untrusted networks
- Generating *random numbers* in the absence of a true random number generator
- Protecting *data at rest*
- Protecting *privacy* attributes in online transactions
- Protecting *anonymity* in sensitive transactions (e.g. voting)
- *Keeping peers honest* on a multi-stakeholder system (e.g. blockchain)
- Providing *Attestation* capabilities
- Providing *Confidential Computing* capabilities

"Not invented here" (NIH) syndrome

NEVER 'invent' your own cryptographic algorithm
(unless you are a cryptographer).

- Use well-known and tested cryptographic algorithms and protocols.
- Use well-known, tried, tested and preferably open-source cryptographic libraries for the programming language you use (e.g. bouncycastle).
- Don't reinvent the wheel, it is very dangerous!.

On randomness

There are two major uses of random numbers

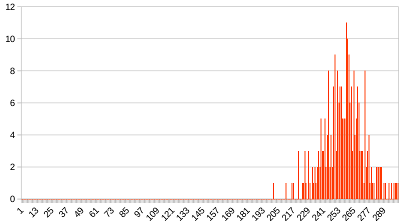
1. **Scientific simulations and multi-player games**, where we need **the new random states to have a fair distribution** or exercise all states equally in a system (within a period), and where we may need to replay the generation of random numbers to audit our process.
    - Example - glibc random(3), a Linear Congruential Generator RNG[2]
2. **Cryptography**, where we need to generate **unpredictable series of bytes** for key/nonce use.
    - Example - Linux kernel getrandom(2) system call, generating random numbers from multiple sources incl. hardware (e.g. interrupts)
    - These are sometimes called *Cryptographically Secure Random Number Generators (CSRNG)*

Do not use #1 where you need #2, or vice versa. You will end up with unfair results and/or weak cryptography!

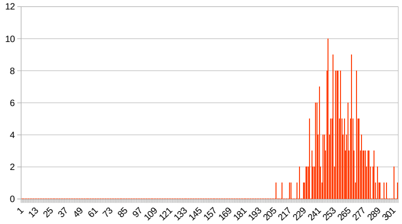
[2]Random Number Generator

random(3) PRNG vs getrandom(2) CSRNG

Comparing byte values of 65536 single-byte samples.



random(3) value frequency diagram



getrandom(2) value freq. diagram

Notice the values surrounding the optimal (fair) value of 256 occurrences of the same value in random(3). If we wish to achieve **fairness** we must be looking for a pointy and narrow bell curve around this value.

random(3) PRNG vs getrandom(2) CSRNG

Comparing byte values of 65536 single-byte samples.

```
$ ent random-output
Entropy = 7.996972 bits per byte.

Optimum compression would reduce the size
of this 65536 byte file by 0 percent.

Chi square distribution for 65536 samples is 273.56,
and randomly would exceed this value 20.27 percent
of the times.

Arithmetic mean value of data bytes is 127.5532
(127.5 = random).

Monte Carlo value for Pi is 3.147408899
(error 0.19 percent).

Serial correlation coefficient is 0.004471
(totally uncorrelated = 0.0).
```

```
$ ent getrandom-output
Entropy = 7.996949 bits per byte.

Optimum compression would reduce the size
of this 65536 byte file by 0 percent.

Chi square distribution for 65536 samples is 276.80,
and randomly would exceed this value 16.64 percent
of the times.

Arithmetic mean value of data bytes is 127.5391
(127.5 = random).

Monte Carlo value for Pi is 3.115546603
(error 0.83 percent).

Serial correlation coefficient is 0.002647
(totally uncorrelated = 0.0).
```

Notice the low correlation coefficient between values in getrandom(2) making new values more **unpredictable**.

Sources of random numbers

- **True Random Number Generators (TRNG)** - they collect random quantities from physical phenomenons (e.g. number of electrons passing through a gate), and we may need to remove *bias* in their measurements.
- **Pseudo-Random Number Generators (PRNG)** - they compute random quantities using a discrete set of algorithms.
- CSRNGs for keys, session IDs, nonces, unique identifiers etc. are more and more relying on TRNGs, however where these are not available they rely on PRNGs (using key-derivation techniques such as secure hashing and HMAC, or ciphers).
- At least one security agency was found to have actively undermined the security of a standard CSPRNG[3] (the Dual EC-DRBG case).

[3]Cryptographically Secure Pseudo Random Number Generator

On entropy

- **Entropy** is a measure of *chaotic behaviour* (read: unpredictability) in information.
- Information having a great amount of entropy is not subject to patterns (i.e. cannot be compressed easily) and cannot be easily guessed.
- A password is low entropy information (i.e. ASCII characters coming from a very narrow set of byte values, and words have predictable patterns).
- A secret key must be based on high entropy data so that it is non-guessable.
- If we consider that the bits of the key of an encryption algorithm come from a high entropy source, then we sometimes refer to the key length (or key strength) as *key entropy*, as it is easier to guess a 2-bit key in comparison to a 160-bit one.

Use appropriate entropy

- Cryptographic algorithms may be secure in theory and well implemented.
- However security relies also on the entropy of the parameters / keys used.
- Consider which key sizes / parameter sizes are appropriate for the security level of your application.
  - NIST Transitioning the Use of Cryptographic Algorithms and Key Lengths
  - ENISA Algorithm Key Size and Parameters Report
  - ENISA Study on Cryptographic Protocols

Randomness issues

- What does randomness affect?
  - Session ID's and API keys
  - Temporary passwords (set after password reset)
  - Nonces
  - Public / Private key pairs, generated Secret keys, Session Keys
- The case of Debian (CVE-2008-0166)
  - Analysis of the Debian OpenSSL predictable PRNG vulnerability
  - Someone patched an OpenSSL PRNG because "a static analysis tool complained about an unitialized buffer", forcing the PRNG to generate values with low entropy. Affected Debian SSH keys got blacklisted.
- Widespread weak keys in network devices
  - Research by Univ. of Michigan and UC San Diego found 9.6% of SSH hosts are sharing public keys, possibly due to poor entropy sources.
- G. Argyros and A. Kiayias found that many PHP applications issued a guessable reset password due to the use of a Mersenne Twister PRNG.

On selecting a PRNG

- Make sure to use a Cryptographically Secure PRNG (with input from a TRNG if possible) for
  - Session ID's and API keys
  - Temporary passwords (set after password reset)
  - Nonces
  - Public / Private key pairs, generating Secret keys, Session Keys
- Most modern operating systems expose a hardware-influenced CSPRNG (e.g. Linux getrandom(2) based on ChaCha20)
- This is then provided to applications by the standard library of the programming language (e.g. java.security.SecureRandom in Java)
- And is then used in frameworks (e.g. java.util.UUID.randomUUID() uses java.security.SecureRandom under the hood)

Verifying the authenticity of data

- Systems may need to verify if data they are processing (e.g. firmware, JWT token etc.) have been generated by a trusted source.
  - In asymmetric cryptography, we can digitally sign the data with a private key at data generation time, and verify the signature with the corresponding public key at data reception time.
  - When no PKI[4] exists, and the generator of the data is also the receiver of the data (e.g. when a server wishes to pass form data to the browser) we can alternatively apply an HMAC with a secret key, to verify whether the data were returned in their authentic form.

[4] Public Key Infrastructure

Troubles with signatures

- Sometimes the HMAC secret is the default one, is a simple pass-phrase, or is hard-coded in the public source code of an online service.
- Due to known issues with RSA signatures and required large key lengths (e.g. 4096) we nowadays opt for elliptic curve cryptography signatures (ECDSA, EDDSA etc.).
- ECDSA nonce reuse attack was used to break the signatures of PlayStation 3
- ECDSA implementation was broken in Java (verified invalid signatures due to a multiplication by zero).
- EDDSA is vulnerable to private key leakage if it exposes a signing oracle that accepts arbitrary public keys (in EDDSA the author's public key is also used during the signature).

Troubles with signatures

- One of the main concerns with signatures are **replay** attacks.
  - The Gnossis Safe wallet smart contract transaction signature did not include the chain[5] on which it was created (in the signed payload), so it was used on Optimism (another chain) to steal 20 million OP tokens.
- Many signed payloads (e.g. JWT) include a *nonce*, the *expiry time* and *audience* for the server to detect replayed, outdated or badly-addressed payloads.
- On embedded systems, if the firmware version value is not protected in hardware (e.g. through RPMB[6]), it may be possible to push older (and possibly vulnerable) firmware versions[7].

[5]read: blockchain
[6]Replay Protected Memory Block
[7]Firmware Downgrade Attack

Storing Secrets

- To protect arbitrary secrets at storage time we need to **encrypt** them.
- Symmetric encryption is good at encrypting large quantities of data (so we prefer it to asymmetric encryption).
- We need to take special care of how we handle the encryption key.
    - One option is to **not persist** the key, but generate it each time through a key derivation function (e.g. PBKDF2) applied to a user supplied password.
    - Another option is to persist the key in **wrapped form** (see AES-KW key wrapping algorithm) where a Key Encryption Key is used for the wrapping.
        - The Key Encryption Key may be derived from user input, or managed solely by hardware (TPM).
    - Another option is to request the key dynamically from a **key vault** service (see Hashicorp Vault, Amazon KMS etc.).

Storing Secrets

- Protect filesystem blocks at kernel level
  - A key resident in kernel memory is used to encrypt filesystem blocks with block-optimized encryption (e.g. AES-XTS)
- Protect arbitrary data at application level
  - AES-GCM or AES-GCM-SIV are good candidates, as they guaranty both confidentiality and integrity (more on this later)
- Protect stored passwords (e.g. PCI-DSS requirements[4])
  - Use password hashed the right way (secure hash function, salt).
  - Examples of good password hashing functions: Argon2, Bcrypt, PBKDF2.
- Protect stored keys in web applications servers.
  - Keys stored in separate encrypted file, are decrypted via passphrase in config file. Used for symmetric decryption of database data.
- Storing secrets in memory
  - Memory lock pages so that they don't end up in swap!

Secure wipe of sensitive data

- In memory: overwrite actual buffer data
- Immutable Strings problem (Java/.NET): Strings are never allowed to change at runtime.
    - Updating a String (e.g. concatenating new characters) allocates new memory on the heap with the updated version of the string.
    - The old copy remains until de-allocation.
    - Passing a string into functions in other layers of code can increase the exposure by allocating an additional copy of that string in the heap.
- On-disk (overwrite with random bytes)

Protecting the integrity - Message Authentication Codes

HMAC: Keyed version of a hash function, used for integrity protection. Let m be the message, K be the session secret key and *hash* be a secure hash function. Then, (depending on the implementation):

$$HMAC_K(m) = \begin{cases} hash(m|K), & \text{or} \\ hash(K|m|), & \text{or} \\ hash(K|m|K) \end{cases}$$

Three known options of using encryption and integrity protection

1. Encrypt-and-MAC
2. MAC-then-encrypt
3. Encrypt-then-MAC (why this is better)

...Protecting the integrity: (1) Encrypt-and-MAC

Let E be the encryption function. The ciphertext is generated by encrypting the plaintext and then appending a MAC of the plaintext

- $E_K(m) \rightarrow c$
- $HMAC_K(m) \rightarrow h$
- Send c, h
- SSH works approximately this way
- An attacker can attack the encryption function and the hash function independently and in parallel

...Protecting the integrity: (2) MAC-then-encrypt

First produce the HMAC and then encrypt everything

- $HMAC_K(m) \rightarrow h$
- $E_K(m, h) \rightarrow c$
- Send c
- SSL (until TLS 1.0) worked this way.
- An attacker can "fool" the receiver to decrypt random messages
- Vulnerable to *padding oracle* attacks[8,9].

[8] https://research.nccgroup.com/2021/02/17/cryptopals-exploiting-cbc-padding-oracles/

[9] https://github.com/mpgn/Padding-oracle-attack

Padding Preliminaries

Block ciphers require $|m| = n \times \text{BlockSize}$

- Padding is used to ensure proper message length
- The plaintext can be padded in a number of different ways
- PKCS#5 and PKCS#7 padding: adds between 1 to n whole bytes (n is the block length). The value of each byte is equal to the number of bytes added
- ANSI X9.23 padding: all padding bytes are zero except the last one, which is equal to the number of bytes added

Padding oracle attacks

Problem:

- Padding depends (on some way) on the plaintext
- Padding depends on the actual length of the plaintext

Attack: Since padding is checked during decryption, the attacker flips some bits in ciphertext to modify padding

- The attacker can then examine changes in behavior and time
- This may lead to full disclosure of plaintext!

Known practical attacks based on padding oracles include BEAST and Lucky 13 (vulnerable until TLS 1.1).

...Protecting the integrity: (3) Encrypt-then-MAC

First produce the ciphertext and then generate a MAC of the ciphertext

- $E_K(m) \rightarrow c$
- $HMAC_K(c) \rightarrow h$
- Send c, h
- IPSec works this way
- This mode is the only mode that is *provably secure*
- An attacker *cannot* fool a receiver to decrypt a random message (why?)

Protecting the blocks - Block encryption

Block ciphers can operate in various modes

- ECB: electronic codebook mode
- CBC: cipher-block chaining mode
- PCBC: propagate ciper-block chaining
- CFB: cipher feedback mode
- OFB: output feedback mode
- CTR: Counter mode
- XTS mode for disk encryption
- CCM: CTR+CBC-MAC (integrity and confidentiality)
- OCB: Offset codebook mode (integrity and confidentiality)
- GCM: Galois/Counter Mode (integrity and confidentiality)

Electronic Code Booke (ECB) mode

Let P_i denote the i-th block of the plaintext and C_i denote the corresponding ciphertext block.
Encryption: $C_i = E_K(P_i)$
Decryption: $P_i = D_K(C_i)$

- In ECB each block is encrypted independently of the others
- Problem: two identical plaintext blocks will produce the same ciphertext
- 1-1 mapping betwee the plaintext and the corresponding ciphertext block for any particular key!
- Completely insecure when encrypting more than one block with a key

Cipher Block Chaining (CBC) mode

- Improves ECB by making the encryption of each block dependent on the ciphertext of the previous block
- Since each block of ciphertext depends on all the previous plaintext blocks, it prevents parallelization of the encryption process (decryption can still be parallerized)
- Any error can propagate to the subsequent block
- CBC is vulnerable to bit flipping attacks
- Padding is still necessary in CBC

...CBC mode

Encryption:

$$C_i = \begin{cases} E_K(P_i \bigoplus C_{i-1}), & i \geq 1 \\ \text{IV}, & i = 0 \end{cases}$$

Decryption:

$$P_i = \begin{cases} D_K(C_i) \bigoplus C_{i-1}, & i \geq 1 \\ \text{IV}, & i = 0 \end{cases}$$

- Each plaintext block is XORed with the previous ciphertext block before it is encrypted
- Thus, each ciphertext block depends on all previous plaintext blocks
- To make each message unique, a unique initialization vector (IV) must be used in the first block
- MANY developers forget to set a unique IV...
- Possible solutions: the developer friendly AES-GCM-SIV

PCBC mode

Encryption:

$$C_i = \begin{cases} E_K(P_i \bigoplus P_{i-1} \bigoplus C_{i-1}), & i \geq 1 \\ P_0 \bigoplus \mathsf{IV}, & i = 0 \end{cases}$$

Decryption:

$$P_i = \begin{cases} D_K(C_i) \bigoplus P_{i-1} \bigoplus C_{i-1}, & i \geq 1 \\ C_0 \bigoplus \mathsf{IV}, & i = 0 \end{cases}$$

- Close to CBC mode
- It produces self-synchronizing stream cipher

Cipher Feedback (CFB) mode

Encryption:

$$C_i = \begin{cases} E_K(C_{i-1}) \bigoplus P_i, & i \geq 1 \\ \mathsf{IV}, & i = 0 \end{cases}$$

Decryption:

$$P_i = \begin{cases} D_K(C_i) \bigoplus C_{i-1}, & i \geq 1 \\ \mathsf{IV}, & i = 0 \end{cases}$$

- Similar to synchronized CBC but it is self-synchronizing. If one block is lost, it does not affect decryption of the others
- Encryption and decryption functions are identical
- It does not require padding of the plaintext data
- The initialization vector (IV) must still be used

Output Feedback (OFB) mode

Encryption: $C_j = P_j \bigoplus O_j$
Decryption: $P_j = C_j \bigoplus O_j$
where, $O_j = E_K(I_j)$ and $I_j = O_{j-1}$ and $I_0 = IV$.

- Similar to synchronized CBC but it is self-synchronizing. It turns a block cipher into a synchronous stream cipher
- Based on the key and an IV, it genrates keystream bits (in blocks) which are then XORed with the plaintext
- Encryption and decryption functions are identical
- It does not require padding of the plaintext data
- Problem: OFB (and CTR) behave like stream ciphers and are more vulnerable to IV attacks
- Reusing an IV leads to key bitstream reuse, breaking security!

Counter (CTR) mode

Encryption: $C_j = P_j \bigoplus O_j$
Decryption: $P_j = C_j \bigoplus O_j$
where, $O_j = E_K(IV||counter_j)$.

- Similar to OFB.
- It genrates the keystream bits (next keystream block) by encrypting successive values of $IV|counter$
- Encryption and decryption functions are identical
- Problem: OFB and CTR behave like stream ciphers and are more vulnerable to IV attacks
- Reusing an IV leads to key bitstream reuse, breaking security!

XTS mode

- IEEE has approved XTS-AES mode for the protection of information stored on block storage devices.
- The key material consists of a data encryption key (used by the AES cipher) as well as an independent ”tweak key”, used to incorporate the logical position of the data block into the encryption.
- The XTS-AES addresses threats such as copy-and-paste attack.

CCM mode

- Provides both confidentiality and authentication.
- Combines Counter mode encryption with CBC-MAC (message authentication code) authentication.
- Applied in authenticate-then-encrypt mode, thus the same key can be used for both operations.
- It requires two cipher operations on each block.
- It is less efficient than OCB (but OCB is pattent-protected).

OCB mode

- Provides both confidentiality and authentication.
- Based on Integrity Aware Parallelizable Mode (IAPM).
- It integrates a MAC into the block cipher process.
- Only one block cipher operation is needed, but OCB is pattented.
- Variations OCB1(2001) and OCB3(2011) are considered secure.
- An existential forgery attack against OCB2 was published in 2018[10]

[10] https://eprint.iacr.org/2018/1040

GCM mode

- Provides both confidentiality and authentication
- Combines Counter mode encryption with Galois-MAC (message authentication code) authentication
- It requires one cipher operation + one Galois field 128-bit multiplication on each block
- Easiliy parallelized operations
- Proven secure in the Concrete Security Model
- Used by Amazon for all cloud storage by default.
- AES-GCM-SIV no longer requires a random IV from developers.

AEAD ciphers

- Authenticated Encryption with Associated Data
- AES GCM supports AEAD
- AEAD allows appending to the ciphertext a plaintext, but integrity protected, tag
- You can use this tag as a developer to understand if the software is trying to decrypt the data *in the wrong context*

Key exchange

- Diffie Hellman (Elliptic Curve DH)
- PAKE

Transport cryptography

- How SSL/TLS works (recall the network security course)
- BEAST / CRIME / BREACH / Lucky 13
- heart-bleed attack
- Weak key sizes, ciphers and hashing algorithms (supported by bad web server config - google for "SSL labs server test")
- Issues of ECB, CBC modes

Anonymity issues

Tricks for protecting the user's anonymity.

- Example: to check if a site is known to be of malicious nature, browsers' hash the URL the user visits and make a lookup on the online db of vulnerable sites according to this hash.
- Not very anonymized is it?

Part III

Cryptographic Security models

Provable security

- AdHoc protocols should not be used
- A security protocol be considered adequately secure, only if it comes with formal security proof
- There exist many security models around

The standard security model

- The adversary is usually a polynomial bounded ppt (probabilistic polynomial time) Turing machine
- Security relies on known hard problems
- A (new) crypto protocol must be shown by reduction, to be (at least) as hard as a known hard problem
- The capabilities of the adversary must be well defined (what he knows and what he can do)
- The adversary is modeled through a security experiment, during which the adversary has access to one or more oracles

Other security models: The Universal Composibility model

- The goal of the UC model is to formally analyze the security of protocols composed of other protocols
- It assumes an Ideal World and a Real Word function
- The goal is to deduce the capabilities of the ideal world function to the real world one.

Formal verification tools

- SAW is such a tool that:
  - can check mathematical properties of a function at the machine level (using LLVM)
  - allows one to write a *hypothesis* with a cryptographic Domain Specific Language (DSL) called Cryptol
  - allows one to verify that a hypothesis holds true for a function in question using a solver (such as Z3)
- Has been used for the verification of many cryptographic libraries including Amazon's s2n TLS library.

Part IV

Modern Challenges for Cryptography

Side Channel Attacks

- On devices with physical access
  - Measure the delay in processing and deduce a secret key (timing attack)
  - Measure the power consumption and deduce the secret key (differential power analysis)
  - Measure the electromagnetic emissions and deduce the secret key (EM side channel)
- On the cloud or over the network (e.g. through Javascript in the victim browser)
  - TLS timing attacks (Lucky 13 TLS protocol issue, OpenSSL issues etc.)
  - Measure microarchitectural side-channels to leak system secrets (cache side channel, Meltdown, Spectre, etc.)
- There is a trick with HTTP/2 that allows for 100ns-level delay measurements (regardless of the network jitter) and could thus be abused to perform timing attacks to remote web applications...

An example Timing issue

```
int compare_secret(char *secret1, char *secret2,
                   size_t secret_len)
{
     size_t i;
     for (i=0; i<secret_len; i++) {
         if (secret1[i] != secret2[i]) {
            return 0; // false
         }
     }
     return 1; // true
}
```

- What's the problem here?
- Early function return when a byte is different; we can infer the byte sequence with *secret_len* $\times 2^8$ tries (which is $<< 2^{\textit{secret\_len} \times 8}$).

Side Channel Attacks

- To avoid timing attacks, use **constant-time algorithms** (or random delay workarounds) for key comparison[11], encryption etc.
- Microarchitectural attacks require chip redesign, chip microfirmware updates or OS workarounds (e.g. see Linux retpoline measure).
- Differential power analysis attacks and EM side channel attacks may be impacted by algorithm re-design and hardware re-design efforts.
- We expect to find many more types of side channels in the future, so it is good practice:
  - to avoid exposing secret key material on shared computing environments (shared cloud, browser JavaScript runtimes etc.) as much as possible.
  - consider any keys placed on embedded devices to be exposed (thus generate device specific keys whose access can be revoked).

[11]for example see OpenSSL's CRYPTO_memcmp(3)

Fault Injection Attacks

- For devices where the attacker has physical access
  - input clock glitching attack (skip instruction)
  - temperature (heating) attack (modification of data in memory or stored in CPU)
  - input voltage glitching attack (skip instruction, set register to zero etc.)
  - EM pulse attack (skip instruction, change value etc.)
  - laser attack (skip instruction, change value etc.)
- For remote systems
  - Rowhammer allows Javascript payloads (or VM guest code) to flip nearby bits in physical memory that is based on DRAM chips. This brings down any form of isolation on the system.
- Fault injection can be used to aid the extraction of secret keys or to bypass cryptographic checks.

Fault Injection Attacks

- For general fault injection there's a number of hardware measures that aim to make fault experiments less likely to succeed.
- In the software domain there are similar measures, such as checking for a condition twice (see DOUBLECHECK pattern).
- For rowhammer, a number of DRAM chips have introduced proactive protections (see TRR).
- It is best to consider that **a fault injection will eventually be possible** on a captured device and thus that the cryptographic code will not always run as expected.

Post Quantum Cryptography

- Most of the asymmetric cryptography we use today is based on the *integer factorization problem*, the *discrete logarithm problem* or the *elliptic-curve discrete logarithm problem*.
- All of these difficult mathematical puzzles can be solved easily on a sufficiently powerful *Quantum Computer* (using Shor's algorithm or other approaches).
- Symmetric cryptography may be slightly sped up through Grover's algorithm in Quantum Computers.
- Governments and the industry fear of a *"harvest now, decrypt later"* threat.
- Post Quantum Cryptography[12] deals with the introduction of new quantum-safe algorithms for cryptographic use.

[12] follow PQCrypto conference for all the latest news on Post-Quantum Cryptography.

Post Quantum Cryptography

As of 2024

- Many of NIST's PQC finalists were characterized as not safe.
- The current best practice is to deploy hybrid approaches relying on both a PQC[13] and a legacy cryptographic algorithm for asymmetric cryptography.
  - See X25519Kyber768 used for key agreement in Google Chrome.
  - There are A LOT of projects going on in the industry right now to adopt such hybrid approaches.
- AES-256 or similar strength algorithms should be used for symmetric encryption.
- SHA-384 or greater strength algorithms should be used for cryptographic hashing.

[13] see liboqs library by the Linux Foundation

As developers what should we do?

- Side Channel Attacks, Fault Injection Attacks and Post Quantum Cryptography are **open problems**.
- Deduce a suitable threat model for your application
    - *What is at stake and how well resourced are your likely opponents?*
- Don't use shared hosting and watch out for timing attacks, if you manage sensitive data (e.g. secret key to sign JWT tokens).
- Plan for key compromise by legal owners of sold devices.
- Plan for a transition to a hybrid PQC approach for key agreement, key transport and digital signatures.

Part V

Conclusions

Important Concepts

- PRNG vs CSRNG
- Entropy
- Signatures and replays
- Key Management and Key Wrapping
- Padding Oracles and Encrypt-then-MAC
- Encryption for filesystems with AES-XTS
- Integrity protected encryption with AES-GCM (or AES-GCM-SIV)
- AEAD
- Security Models and Formal Proofs
- Side Channel and Fault Injection attacks
- Post Quantum Cryptography

Further Reading Material

- *Jonathan Katz and Yehuda Lindell: Introduction to Modern Cryptography. CRC Press (2011)*
- `https://www.enisa.europa.eu/publications/algorithms-key-size-and-parameters-report-2014`
- `https://www.enisa.europa.eu/publications/study-on-cryptographic-protocols`
- `https://www.pcisecuritystandards.org/security_standards/documents.php?association=PA-DSS`
- Serious Cryptography

Questions?