Notes on

APPLIED CRYPTOGRAPHY

v0.4 — September 15, 2025

Applied Cryptography (ALEPH) Research Unit
Center for Cybersecurity — *Fondazione Bruno Kessler*
Trento, Italy `https://aleph.fbk.eu`

A collection of notes on the subject of applied cryptography.

What this handbook is for. This handbook (re)collects, correlates, and *concisely* describes basic concepts, advanced constructions, security concerns, and pragmatic aspects within the realm of applied cryptography.

What this handbook is not. This handbook is by no means a horizontally complete or vertically exhaustive treatment of all facets of (especially theoretical) cryptography. In particular, mathematical details, formal proofs, and descriptions of the inner functioning of cryptographic algorithms are intentionally minimized (where not omitted entirely) to maintain brevity and focus. For more comprehensive coverage of these aspects, many authoritative resources are readily available elsewhere (see Appendix B).

How to read this handbook. This handbook is structured into notes, each offering a *short, self-contained, and convenient resource* for quick lookup and topical reference. Although a top-down reading of all notes is possible, readers can choose and read any one single note by referring to the table of contents. Each note provides — where necessary or appropriate — suitable pointers to other notes (denoted with "§") or more detailed sources.

Intended audience. This handbook is primarily aimed at anyone who (*i*) requires a practical understanding of cryptography without delving (too much) into its theoretical underpinnings, (*ii*) seeks a reference companion to standard cryptographic texts, or (*iii*) needs a high-level overview of various cryptographic components and their relationships. While prior familiarity with cryptography is helpful, this handbook strives to remain accessible to readers with a general background in computer science or cyber-security.

How to contribute. This is a living handbook curated by the Applied Cryptography (*ALEPH*) research unit of the *Center for Cybersecurity* in *Fondazione Bruno Kessler*. Contributions such as corrections, suggestions, or improved formulations are welcome. Please direct any feedback to `sberlato@fbk.eu` or use the anonymous form `https://forms.gle/2KzxAmv3M1VnwUAVA`. Finally, note that any references to existing cryptographic services, libraries, and hardware should not be interpreted as an endorsement of such services, libraries, and hardware (unless stated otherwise).

If you come across any content in this handbook that lacks proper attribution, we sincerely apologize. We have compiled material over many years, hence some sources may not have been documented accurately. Please let us know, and we will promptly make the necessary corrections.

Table of Contents

NOTES ON NOTATION AND PRELIMINARIES **6**

1 Basics of Cryptography **7**
1.1 Cryptographic Keys . . . 8
1.2 Cryptographic Algorithms . . . 10
1.3 Cryptographic Ciphers . . . 11
1.4 Cryptosystems . . . 14
1.5 Cryptographic Protocols . . . 14

2 Mathematical Underpinnings **16**
2.1 Factorization . . . 16
2.2 Modular Arithmetic . . . 17
2.3 Groups . . . 18
2.4 Fields . . . 19
2.5 Elliptic Curves . . . 21

3 Trapdoor Functions **25**
3.1 Integer Factorization . . . 25
3.2 Composite Residuosity . . . 26
3.3 Discrete Logarithm . . . 27
3.4 Elliptic Curve Cryptography . . . 27
3.4.1 Bilinear Pairings . . . 29

4 Security Properties **32**

NOTES ON HASH FUNCTIONS **36**

5 Hash Functions **37**
5.1 Cryptographic Hash Functions . . . 38
5.1.1 Keyed Cryptographic Hash Functions . . . 39
5.2 Merkle Trees . . . 40

NOTES ON SYMMETRIC CRYPTOGRAPHY **41**

6 Symmetric Cryptography **42**
6.1 Message Authentication Codes . . . 42
6.1.1 Hash-based Message Authentication Codes . . . 43
6.1.2 Cipher Block Chaining Message Authentication Codes . . . 43
6.2 Authenticated Encryption . . . 43
6.2.1 Authenticated Encryption with Associated Data . . . 44
6.3 Format Preserving Encryption . . . 44
6.4 Ciphers for Symmetric Cryptography . . . 45

6.4.1 Advanced Encryption Standard . . . 45
6.4.2 ChaCha20 . . . 45

NOTES ON ASYMMETRIC CRYPTOGRAPHY **47**

7 Asymmetric Cryptography **48**
7.1 Ciphers for Asymmetric Encryption . . . 49
7.1.1 Rivest-Shamir-Adleman . . . 49
7.1.2 Paillier . . . 50
7.1.3 ElGamal . . . 50

8 Digital Signatures **52**
8.1 Digital Signatures and Security Properties . . . 52
8.2 Ciphers for Digital Signatures . . . 53
8.2.1 Elliptic Curve Digital Signature Algorithm . . . 53
8.2.2 Schnorr Signature . . . 54
8.2.3 Edwards-curve Digital Signature Algorithm . . . 54
8.2.4 Boneh-Lynn-Shacham Signature . . . 55

9 Diffie-Hellman **57**
9.1 Triple Diffie-Hellman . . . 59
9.2 Extended Triple Diffie-Hellman . . . 60

10 Key Encapsulation Mechanisms **62**
10.1 Authenticity in Key Encapsulation Mechanisms . . . 63
10.2 Key Encapsulation Mechanisms with Multiple Recipients . . . 64

NOTES ON HYBRID CRYPTOGRAPHY **65**

11 Hybrid Cryptography **66**
11.1 Ciphers for Hybrid Cryptography . . . 66
11.1.1 Integrated Encryption Scheme . . . 67
11.1.2 Data Encapsulation Mechanisms . . . 67

NOTES ON PRAGMATIC ASPECTS **68**

12 Cryptographic Modules **69**
12.1 Hardware Security Modules . . . 69
12.2 Secure Elements . . . 70
12.3 Trusted Platform Modules . . . 71
12.4 Trusted Execution Environments . . . 71
12.4.1 Trusted Execution Environment Implementations . . . 73
12.5 Physical Unclonable Functions . . . 74
12.6 Hardware- vs. Software-Based Cryptographic Modules . . . 75

13 Key Management **77**

13.1 Key Management Lifecycle . . . 77
13.1.1 Pre-Operational Key State . . . 77
13.1.2 Operational Key State . . . 81
13.1.3 Post-Operational Key State . . . 83
13.1.4 Destroyed Key State . . . 84

14 Binding Keys to Parties **85**
14.1 Digital Certificates . . . 85
14.1.1 Validation of Digital Certificates . . . 86
14.2 Public Key Infrastructure . . . 86
14.2.1 Registration Authorities . . . 87
14.2.2 Certificate Authorities . . . 87
14.2.3 Validation Authorities . . . 88
14.3 Web of Trust . . . 88
14.3.1 Pretty Good Privacy . . . 89
14.4 Digital Certificates Revocation . . . 89

15 Standards and Certifications **91**

NOTES ON SECURITY **94**

16 Security in Cryptography **95**
16.1 Threat Modeling . . . 95
16.2 Attack Models . . . 96
16.2.1 Security Goals . . . 98
16.3 Attacks Taxonomy . . . 99

17 Secure Implementation of Algorithms **102**
17.1 Attacks on the Implementation of Cryptographic Algorithms . . . 102
17.1.1 Side-Channel Attacks . . . 102
17.1.2 Fault Injection Attacks . . . 106
17.1.3 Data Remanence Attacks . . . 107
17.2 Security Best Practices . . . 108
17.2.1 Secret-Independent Execution . . . 109
17.2.2 Constant-Time Coding . . . 109
17.2.3 Masking and Randomization . . . 110
17.2.4 Hardware Hardening . . . 111
17.2.5 Memory Zeroization . . . 111
17.2.6 Resistance to Fault Injection . . . 112
17.2.7 Key Limit . . . 112
17.2.8 Testing . . . 112
17.2.9 Bad Practices . . . 115

18 Programming Languages and Software **116**
18.1 Programming Languages for Security . . . 116

18.2 Compilers and Interpreters Options for Security 121
18.3 Trusted Computing Base 121

19 Secure Design of Protocols **123**
19.1 Design of Protocols 123
19.1.1 Best Practices 125
19.1.2 Categories of Attacks 129
19.2 Security of Protocols 131

20 References **140**

A Acronyms **150**

Acronyms **150**

B Further Resources **154**
B.1 Books 154
B.2 Online Courses 155
B.3 Hands-On Challenge Platforms 156
B.4 Newsletters and Podcasts 157
B.5 Useful Websites and Reading Guides 157
B.6 Blogs 158

Notes on

NOTATION AND PRELIMINARIES

last revision: 15 Sep 2025

1

BASICS OF CRYPTOGRAPHY

Cryptography is the study of mathematics-based practices providing security properties (§4) for data — across storage, communication, and computation — in the presence of attackers or malicious users (§§ 17 and 19). Conversely, **cryptanalysis** is the study of practices to breach or weaken cryptography — whether in its mathematical foundations or in how it is built or used — with the ultimate goal of compromising security properties. The combination of cryptography and cryptanalysis is called **cryptology**.

> **Suggested Reading 1.1 — Quick introduction to applied cryptography**
>
> The article "Challenges in Cryptography" [133] provides a clear and concise (4 pages only) overview of the main concepts and challenges in applied cryptography. Those with no prior familiarity with cryptography are recommended to read that article before proceeding.

Two principles relevant to cryptography are the **Kerckhoffs's principle** and **security through obscurity**. Kerckhoffs's principle states that a cryptographic algorithm, cipher, or cryptosystem (see the definitions in §1) should remain secure even if everything about it — except secret and private cryptographic keys (§ 1.1) — is public knowledge. Conversely, security through obscurity states that security should (wholly or partially) depend on the secrecy of functioning, design, and implementation details. Kerckhoffs's principle simplifies the implementation, distribution, and interoperability of cryptography, hence it is considered a best practice, while security through obscurity is discouraged by authoritative organizations like the National Institute of Standards and Technology (NIST) [164].

> **Curiosity 1.1 — How do cryptography and steganography relate to each other?**
>
> Although seemingly related and sharing a similar objective, steganography is different in nature from cryptography. **Steganography** is the practice of concealing information within a piece of data in such a way that the presence of the hidden information is difficult to detect and extract. The main applications of steganography are in covert communication and digital watermarking. Steganography may either consider the use of keys (and thus mostly follow Kerckhoffs's principle) or not (and thus follow security through obscurity).

1.1 Cryptographic Keys

A **cryptographic key** (or simply key) is usually one of the inputs of a cryptographic algorithm (§ 1.2). A key may be either a (carefully chosen or generated) meaningless sequence of bytes — as in symmetric cryptography (§6) — or a tuple of one or more mathematically related elements or structures (e.g., integer numbers, points on elliptic curves, isogenies) — as in asymmetric cryptography (§7).
A key used in symmetric cryptography is called **symmetric key**. Generally, symmetric keys are known by one party (e.g., when protecting data at rest) or, more frequently, by a specific set of two or more parties (e.g., when protecting data in transit). Symmetric keys are also called *secret keys* or *shared keys* — when known by more than one party.

> **⚠ Warning 1.1 — Unwise names for symmetric keys**
>
> Symmetric keys are sometimes called *shared secrets* which, however, may be an ambiguous term as it is used in other contexts as well. For instance, the output of Diffie-Hellman (DH) key agreements is also called shared secret — see §9.

A key used in asymmetric cryptography is called **asymmetric key**. Asymmetric cryptography typically employs two kinds of keys: **public keys** and **private keys**. By definition, a public key can be publicly disclosed and known by anyone within a system. Conversely, a private key is (almost[1]) always known by one party only corresponding to the owner of the key. A private key is always related to exactly one public key, either mathematically (i.e., by design) or by embedding random values used to uniquely derive the corresponding public key. However, *it must be computationally infeasible to derive a private key from the corresponding public key*. Conversely, it is usually possible to derive a public key from the corresponding private key. A private key and a public key are (almost[2]) always exclusively assigned to each other and form a **key pair**. Public and private keys may be generated together or at a different point in time (usually, the private key is generated after the public key).
Depending on its origin or use, keys may be:

- **ephemeral**, that is, freshly generated when needed and immediately disposed of after being used once. The use of ephemeral keys provides forward secrecy (§4) and — to some extent (but not necessarily) — protection against replay attacks (§19). Ephemeral keys are usually asymmetric keys and are often used in the DH key agreement (§9);
- **short-term**, that is, freshly generated when needed and disposed of after being used for a limited number of times or period (e.g., in the order of minutes, hours, or a few days). Short-term keys are usually symmetric keys used for protecting data in transit within communications;
- **long-term**, that is, not ephemeral. Long-term keys are also called **static** and are often — though not necessarily — bonded to parties through either a Public Key Infrastructure (PKI) or the Web of Trust (WoT) (§14 and § 14.3, respectively);

[1] "Almost" because some cryptographic constructions — e.g., Identity-based Encryption (IBE) (§ 19.2.1) and Attribute-based Encryption (ABE) (§ 19.2.4) — private keys are actually generated by a trusted party called **Key Generation Authority (KGA)**. Typically, after having generated a private key and shared it with the owner, the KGA should forget the private key (e.g., delete it from memory).
[2] "Almost" because in some cryptographic constructions — again, like IBE and ABE — it is possible that more private keys are assigned to the same public key.

- **Data Encryption Keys (DEKs)**, that is, keys used specifically to en/decrypt data. DEKs are almost always symmetric keys. Reusing the same DEK to encrypt distinct pieces of data is usually discouraged (so to limit the impact of the loss or leak of the DEK);
- **Key Encryption Keys (KEKs)**, that is, keys used specifically to en/decrypt other keys. KEKs may be both symmetric or asymmetric keys;
- **main keys**,[3] which is an ambiguous term that may refer to:
  - **main (KEK) keys**, that is, KEKs which are never encrypted with other KEKs — although main keys are protected, usually with dedicated software (e.g., keystores) or — better — hardware (§12);
  - **main (generation) keys**, that is, private keys hold by KGAs used to generate other private keys, usually in the context of cryptosystems based on bilinear pairings (§ 3.4.1) such as ABE;
- **weak keys**, that is, keys that make specific cryptographic algorithms behave in some undesirable (and insecure) way. Weak keys usually represent a very small fraction of the possible keys and (obviously) should be avoided.

> **Remark 1.1 — Key wrapping and envelope encryption**
>
> The practice of encrypting keys with KEKs is called **key wrapping**, while the practice of encrypting data with DEKs and then encrypting DEKs with KEK — usually in the context of hybrid cryptography (§11) — is called **envelope encryption**.

Key Generation. The process of generating keys usually consists of a protocol whereby one or more keys are created or derived by one or more interacting parties — please refer to § 13.1.1 for more information on protocols for key generation. At a high level, all keys should be generated:

- to be **suitable for use in the intended purpose and application**. For instance, a symmetric key cannot be used in asymmetric cryptography. Similarly, a key generated for a specific cryptographic algorithm usually cannot (and must definitely not) be used as input for a different cryptographic algorithm;
- with a **proper length** as to render guessing attacks computationally infeasible (§ 16.3);
- starting from a **high entropy** (i.e., high randomness or unpredictability) source — in the sense of being predictable by attackers with negligible probability. Hardware-based random bit generators are usually used as high entropy source.

> **Curiosity 1.2 — Can anything be used as a high entropy source?**
>
> The renowned CloudFlare company relies on the random movement of lava lamps (Figure 1) as a source of high entropy.[a] For more details on entropy sources, we refer the interested reader to [30, 184].

[3] As adopted by other authoritative sources such as the NIST [94], in an effort to address biased language we prefer the use of "main" over "master" (and "slave").

Figure 1: Part of the 100 lava lamps in the atrium of Cloudflare's headquarters (from https://www.cloudflare.com/learning/ssl/lava-lamp-encryption/)

> [a]How do lava lamps help with Internet encryption? | Cloudflare (https://www.cloudflare.com/learning/ssl/lava-lamp-encryption/)

Please refer to §13 for more information on key management.

1.2 Cryptographic Algorithms

A **cryptographic algorithm** (or simply algorithm) is a function — that is, having one or more inputs and returning one or more outputs — used in the context of cryptography. Many algorithms rely on one-way and trapdoor functions (§3). An **encryption algorithm** converts **plaintexts** — that is, intelligible data — into **ciphertexts** — that is, unintelligible data. A ciphertext may also be called *encrypted plaintext*. A plaintext is denoted with m, and the set of all plaintexts with $\mathcal{P}$. Similarly, a ciphertext is denoted with c, and the set of all ciphertexts with $\mathcal{C}$. An encryption algorithm is (practically) always paired with a **decryption algorithm** which converts ciphertexts back into the corresponding plaintexts. A pair of encryption and decryption algorithms has the **correctness** property if decrypting any ciphertext using correct and expected parameters (e.g., keys) returns the corresponding plaintext. Ideally, any pair of encryption and decryption algorithms possesses the correctness property. Finally, there exist also **translation algorithms** which translate ciphertexts — that is, plaintexts encrypted under a specific key — into different ciphertexts — that is, the same plaintexts encrypted under a different key; translation does not imply decryption and can be seen as a particular instance of re-encryption (§ 19.5.1).

1.3 Cryptographic Ciphers

A **cipher** is a set of logically related algorithms — such as a pair of encryption and decryption algorithms — typically denoted with ϕ. Besides correctness, ciphers may possess the following properties:

- **perfect secrecy**: an attacker, given any one ciphertext only, cannot derive additional knowledge about the corresponding plaintext — even when knowing the probability distribution of plaintexts. In other words, a plaintext can be mapped to all ciphertexts with the same probability;
- **semantic security**: an attacker, given a ciphertext, can feasibly derive only negligible additional knowledge about the corresponding plaintext. In other words, a cipher achieves semantic security if an attacker cannot feasibly distinguish between two ciphertexts that correspond to two different plaintexts. Semantic security is related to perfect secrecy, although the former is a strong theoretical notion (difficult to achieve in practice), while the latter is a more practical concept that focuses on the computational feasibility of the attacker's ability to learn information about the plaintext from the ciphertext;
- **(non-)malleability**: the en/decryption of data is malleable if it is possible to transform a ciphertext into another ciphertext which decrypts to a related plaintext (note that malleability differs from translation). More formally, given a ciphertext c corresponding to a plaintext m, it is possible to generate another ciphertext c' which decrypts to $f(m)$ for a known function f — without necessarily knowing or learning m. Although malleability is generally an undesirable property, it is the basis for Homomorphic Encryption (HE) (§ 19.2.7);

Another property relevant to ciphers (but also to, e.g., hash functions as discussed in §5) is the avalanche effect. Briefly, the avalanche effect states that a tiny change in any of the inputs of an algorithm (e.g., a bit flips) should make the output change significantly (e.g., half the output bits flip). The lack of the avalanche effect implies poor randomization, hence vulnerability to cryptanalysis.
A cipher having the perfect secrecy property is usually called **perfect cipher**. A cipher is perfect if a key is not reused to encrypt different plaintexts; otherwise, Shannon's Theorem no longer guarantees perfect secrecy.

> **Remark 1.2 — The Clever Boy Assumption and the Shannon's Theorem**
>
> The so-called **clever boy assumption** states that the choice of keys for a cipher is not influenced by the choice of plaintexts. In simpler words, plaintext and keys are independent — a realistic assumption for (ideally) all scenarios. The clever boy assumption is a prerequisite for the **Shannon's Theorem**: assuming that the set of keys, plaintexts, and ciphertexts of a cipher ϕ are equal and comprise n-bit strings[a] only — formally, $\mathcal{K} = \mathcal{P} = \mathcal{C} = \{0,1\}^n$ — Shannon's Theorem states that ϕ is a perfect cipher if and only if (i) the keys are perfectly random and (ii) there exists one key only mapping a given plaintext to a given ciphertext — formally, $\exists! k \in \mathcal{K} : \phi_k(m) = c \; \forall m \in \mathcal{P}, c \in \mathcal{C}$.
>
> [a] The set of all n-bit strings is denoted as either $\{\mathbf{0},\mathbf{1}\}^{\mathbf{n}}$ or $\{\mathbb{F}_2\}^n$. For instance, $\{\mathbf{0},\mathbf{1}\}^{\mathbf{3}} = \{000, 001, 010, 011, 100, 101, 110, 111\}$

Note that, in a perfect cipher, there are as many keys as plaintexts and that, more importantly, keys have the same length as plaintexts. As a consequence, perfect ciphers cannot be broken even by brute force attacks (§ 16.3). As said before, perfect secrecy is a strong theoretical notion difficult to achieve in practice.

An example of a perfect cipher is the **one-time pad cipher** (also called *Vernam cipher*) where ciphertexts are obtained by `xor`-ing the plaintext and the key. Unfortunately, perfect ciphers are not practical.
Besides keys, ciphers typically consider as input also **Initialization Vectors (IVs)**. Essentially, an IV is a (pseudo)random (§ 19.5.4) sequence of bytes used as additional input to the algorithms of a cipher to achieve semantic security. More in detail, IVs prevent patterns and repetitions in ciphertexts which may be exploited by attackers. IVs are usually required to be random but not secret and, more importantly, must never be reused — especially under the same cryptographic key. A sequence of all-zero bytes is usually not suitable as IV. IVs are also called *starting variables*. Although having slightly different purposes and applications, IVs are essentially equivalent to nonces and these two terms **are often used interchangeably** (a **nonce** is a number used just once in cryptographic protocols usually for preventing replay attacks — see §19).
A cipher comprising algorithms operating on fixed-length blocks of data — one block at a time — is called **block cipher**. Conversely, a cipher comprising algorithms operating on one bit of data at a time is called **stream cipher**.

Block Ciphers. A block cipher is a (keyed pseudo)random permutation family on the mappings between plaintexts and ciphertexts (if a block cipher is a true random permutation, then it is a perfect cipher). Block ciphers are slower but deemed theoretically more secure than stream ciphers. Usually, block ciphers take as input a key which is reused on each block of data. An example of a well-known and widely used block cipher is Advanced Encryption Standard (AES) (§ 6.4).

> **Curiosity 1.3 — Do asymmetric ciphers qualify as block ciphers?**
> Although it is true that asymmetric cryptography (§7) usually operates on fixed-length blocks of data, asymmetric ciphers are not considered block ciphers, as they do not match the standard definition of the term — that is, a (keyed pseudo)random permutation. Moreover, an asymmetric cipher should not be used over a piece of data larger than the size of the cipher's block; in case this happens, the use of hybrid cryptography is preferred (§11).

Block ciphers have a **mode of operation** which specifies how to combine multiple blocks. The most common or known modes of operation are:

- **Electronic Codebook (ECB)**: each block of plaintext is encrypted into a corresponding block of ciphertext independently from other blocks. ECB achieves high performance by allowing for parallel encryption and decryption and provides random read access (i.e., blocks can be decrypted independently). However, ECB is deprecated because it lacks key diffusion — that is, similar blocks of plaintext result in similar blocks of ciphertext — and requires padding whenever the length of the plaintext is not (integral) multiple of the block size. Moreover, ECB does not inherently provide integrity and preserves the structure of the plaintext in the ciphertext (as shown by the famous ECB penguin image — see Figure 2);
- **Cipher Block Chaining (CBC)**: each block of plaintext is `xor`-ed with the previous block of ciphertext before encryption. CBC provides random read access (i.e., blocks can be decrypted independently). However, it does not inherently provide integrity, may need padding, and prescribes sequential encryption — hence, limiting parallel processing and worsening efficiency — but allows parallel decryption;

- **Cipher Feedback (CFB)**: each block of plaintext is `xor`-ed with the previous block of ciphertext to generate the next ciphertext block — in a sense, CFB converts a block cipher into a stream cipher. CFB provides random read access (i.e., blocks can be decrypted independently) and does not require padding, but does not inherently provide integrity and prescribes sequential encryption — hence, limiting parallel processing and worsening efficiency — while still allowing parallel decryption;
- **Output Feedback (OFB)**: the key is used to generate a sequence of bytes called **keystream** and then — similarly to CFB — each block of plaintext is `xor`-ed with the keystream. In a sense, OFB converts a block cipher into a stream cipher. OFB does not require padding but does not provide random read access (i.e., blocks can be decrypted independently), does not inherently provide integrity, and prescribes sequential encryption and decryption — hence, limiting parallel processing and worsening efficiency;
- **Counter (CTR)**: blocks of plaintext are numbered sequentially (usually by increments of 1), and the number is combined (e.g., summed or concatenated) with a (unique) counter — that is, the IV — and then encrypted; the result of this encryption is `xor`-ed with the plaintext of the block to produce the corresponding ciphertext — in a sense, CTR converts a block cipher into a stream cipher. CTR provides random read access (i.e., blocks can be decrypted independently), does not require padding, and achieves high performance by allowing for parallel encryption and decryption, but does not inherently provide integrity. CTR is also called *integer counter mode* and *segmented integer counter*;
- **Galois/Counter Mode (GCM)**: one of the most used in 2025, provides both confidentiality and authenticity. GCM is so called because it combines CTR with the Galois mode for authenticity. Similarly to CTR, in GCM blocks are numbered sequentially (usually by increments of 1), and the number is combined (e.g., summed or concatenated) with a (unique) counter — that is, the IV — and then encrypted; the result of this encryption is `xor`-ed with the plaintext of the block to produce the corresponding ciphertext — in a sense, GCM converts a block cipher into a stream cipher. The blocks of ciphertext are considered coefficients of a polynomial manipulated in the corresponding Galois field (see the concept of fields in § 2.4). Finally, GCM produces a tag providing authenticity. GCM provides random read access (i.e., blocks can be decrypted independently), does not require padding, and achieves high performance by allowing for parallel encryption and decryption;
- other less commonly used modes of operation are Counter with CBC-MAC (CCM), XEX-based Tweaked Codebook with Ciphertext Stealing (XTS), Encrypt-Then-Authenticate with Xor (EAX), Synthetic Initialization Vector (SIV), Offset Codebook Mode (OCB), and Liskov, Rivest, and Wagner (LRW).

> **⚠ Warning 1.2 — Which modes of operation (not) to use**
>
> GCM or CCM are the preferred modes of operation: other common or known modes of operation are not as secure or efficient, and other less commonly used modes of operation may be complex to use or configure and may lack support by available implementations.

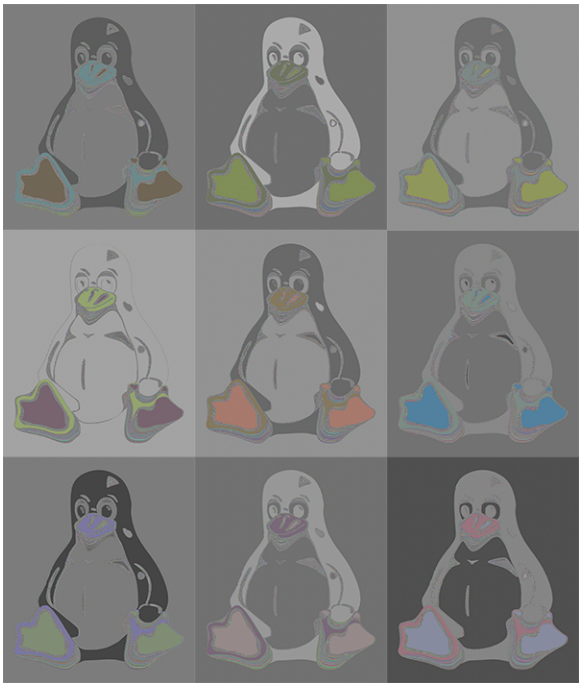

Figure 2: Multiple ECB encryptions with the AES symmetric block cipher (§ 6.4.1) of the Linux penguin mascot with different keys; as noticeable, the underlying image is clearly recognizable, even though ideally encrypted — from `https://words.filippo.io/the-ecb-penguin/`.

Stream Ciphers. Differently from block ciphers, stream ciphers are usually faster but deemed theoretically less secure than block ciphers. Usually, stream ciphers take as input a key that is — often combined with an IV and then — used to generate an (ideally infinite) random keystream. Ultimately, the keystream is used for performing (several) bit-wise operations on the plaintext, hence obtaining the ciphertext (§ 19.5.4 for more details on randomness in cryptography). An example of a well-known and widely used stream cipher is Salsa20/ChaCha20 (§ 6.4).

1.4 Cryptosystems

A **cryptosystem** is an ordered list of ciphers along with the corresponding set of finite possible plaintexts, finite possible ciphertexts, and finite possible keys. There are two main types of cryptosystems: symmetric (§6) and asymmetric (§7).

1.5 Cryptographic Protocols

A **cryptographic protocol** describes an exchange of messages among two or more **parties** — and, possibly, also cryptographic computations local to each party — which involves the use of some cryptographic algorithms; parties involved in a cryptographic protocol are often also called *agents* or *principals*. The definition of a cryptographic protocol includes at least the following:

- **syntax**: messages have well-defined format comprising specific fields;
- **semantics**: each field in the format of a message has a well-defined meaning;

- **temporization** (synchronization): messages have well-defined flows;
- **cryptographic algorithms**: used to provide security properties (§4).

Please refer to §19 for details on the secure design of cryptographic protocols.

2

MATHEMATICAL UNDERPINNINGS

Cryptography is ultimately based on mathematics. Hence, below we provide basic definitions related to mathematical structures typically employed in cryptography to build one-way and trapdoor functions (§3). Being this a handbook on *applied* cryptography, the perspective used to describe the aforementioned structures may sometimes prefer ease of understanding over rigorous formality.

2.1 Factorization

Factorization is the process of breaking down a **number or expression** into a product of its factors. These factors, when multiplied together, will result in the original number or expression. The concept of factorization applies various mathematical objects, including integers and polynomials.[4]

Integer Factorization. Factoring an integer means expressing it as a product (i.e., the result of the multiplication) of two or more smaller positive integers greater than 1 (usually, prime numbers). For instance, the factorization of 12 using only prime numbers is $2 \times 2 \times 3$. An integer which cannot be further factorized is called **prime** number.

Polynomial Factorization. Factoring a polynomial means expressing it as a product (i.e., the result of the multiplication) of two or more smaller — that is, lower-degree — non-constant[5] polynomials **with the same kind of coefficients** (e.g., integers, rationals, reals, or complex). For instance, the factorization of $(x^2 - 4)$ is $(x-2)(x+2)$. A polynomial which cannot be further factorized is called **irreducible**.

[4] A polynomial is a mathematical expression with variables and coefficients, that involves only the operations of addition, subtraction, multiplication and exponentiation to nonnegative integer powers, and has a finite number of terms. For instance, $(x^2 - 4x + 7)$ is a polynomial of degree 2, $(x-2)$ is a polynomial of degree 1, and (1) is a constant polynomial of degree 0 — more information on polynomials can be found at https://mathworld.wolfram.com/Polynomial.html.

[5] In the context of polynomial factorization, we do not consider constant polynomials such as (1), (2), or (3) — they are like the number 1 in integer factorization, that is, irrelevant.

Remark 2.1 — Integer vs. polynomial factorization

Although factoring methods differ, integer and polynomial factorizations are logically equivalent: both suit the definition of writing a number (integer) or expression (polynomial) into a product of its factors. In simpler words, **for a polynomial to be irreducible is equivalent for an integer to be prime**.

2.2 Modular Arithmetic

Modular arithmetic is a system for arithmetic operations where numbers (or expressions) *wrap around* upon reaching a certain fixed number (or expression) called **modulus**. For instance, numbers are reduced by the modulus repeatedly as to reach a number between 0 and the modulus itself. A generic value x under a modulus n is usually written as "$x \pmod{n}$", and it is equal to the remainder of x upon division by n. An operation $x \bullet y$ executed with a modulus n is usually written $(x \bullet y)\%n$ or, more generally, $x \bullet y \pmod{n}$.

Example 2.1 — An example of real-world modular arithmetic

On a 12-hour clock, if it's 9 o'clock and we add 5 hours, we get $9+5 = 14 \equiv 2 \pmod{12}$.[a]

[a] While both symbols represent relationships between expressions, the equal sign (=) generally represents a direct value-for-value equality, whereas the triple bar (≡) means that two expressions are equivalent or congruent in all aspects — not just in value — in a given context (in this case, $14 \equiv 2$ in the context of modulus 12).

Modular arithmetic is useful to both limit sets which otherwise would be infinite. For instance, the set of integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ is infinite, while the set of integers modulo 12 is $\mathbb{Z}_{12} = \{0, 1, \ldots, 10, 11\}$ and counts exactly 12 elements.

Curiosity 2.1 — The elements of $\mathbb{Z}_{12}$ are not actually integers

More formally, the 12 elements in $\mathbb{Z}_{12}$ are so-called **equivalence classes** of integers modulo 12 — not single integers. Informally, the equivalence class of an integer x modulo 12 is a set containing all integers that leave the same remainder upon division by 12; for instance, the integers 2, 14, 26, 38, and 50 are part of the same equivalence class (as they all have remainder 2 upon division by 12). A common notation for representing equivalence classes are square parentheses (i.e., $\mathbb{Z}_{12} = \{[0], [1], \ldots, [10], [11]\}$), where $[x] = \{z \in \mathbb{Z}\ s.t.\ z \equiv x \pmod{12}\}$. Another notation you may see in this context is a horizontal bar on top of an integer; for instance, $\overline{x}$ denotes the residue class of x modulo 12 (a **residue class** is the set of all integers that leave the same remainder upon division by a modulus). Hence, $[x] \equiv \overline{x}$ in modular arithmetic. However, note that bars may denote other concepts in different contexts, so their use may sometimes be ambiguous. Summarizing, the set of integers modulo 12 should be written as $\mathbb{Z}_{12} = \{[0], [1], \ldots, [10], [11]\}$ or as $\mathbb{Z}_{12} = \{\overline{0}, \overline{1}, \ldots, \overline{10}, \overline{11}\}$. Nonetheless, for simplicity, it is common to informally just write $\mathbb{Z}_{12} = \{0, 1, \ldots, 10, 11\}$.

Notably, using modular arithmetic to limit sets makes reversing some operations difficult, e.g., the Discrete Logarithm (DL) problem (§ 3.3) — the underlying intuition is that, every time the result of an operation exceeds the modulus and therefore *wraps around* it, some information is lost. For instance, 2 $\pmod{12}$ may be the result of both $9+5 \pmod{12}$ but also $7+7 \pmod{12}$; while computing an addition modulo 12 is easy, reversing it — i.e., start from 2 and then go back to the addends — is difficult because

more choices may be available.

2.3 Groups

A group is a pair $(G, \bullet)$, where G is a non-empty set while $\bullet$ is a binary operation — that is, a function that takes 2 inputs and returns one output — that satisfies the following properties:

- **closure**: the output of the operation $\bullet$ is still an element of the set G. Formally, for every $x, y \in G$, $\bullet(x, y) \in G$. Note that $\bullet(x, y)$ is often written as $(x \bullet y)$ for simplicity;
- **associativity**: the order in which multiple occurrences of the operation $\bullet$ are applied is irrelevant. Formally, for every $x, y, z \in G$, $(x \bullet y) \bullet z = x \bullet (y \bullet z)$ — in other words, rearranging the parentheses does not change the output;
- **identity element**: there exists an element e in the set G which, when used as input to the operation $\bullet$, always make the operation $\bullet$ return the other input as output. Formally, $\exists e \in G : e \bullet x = x \bullet e = x \forall x \in G$;
- **inverse element**: for each element used as input to the operation $\bullet$, there exists another element that always make the operation $\bullet$ return the identity element e. Formally, $\exists x^{-1} \in G : x \bullet x^{-1} = x^{-1} \bullet x = e \forall x \in G$.

Although a group is formally defined as **the pair** $(G, \bullet)$, it is common to abuse notation by using the symbol G to denote the whole group.

> **Example 2.2 — An example of a group**
>
> The set of integers $\mathbb{Z}$ together with the addition operation $+$ is a group. In fact, any sum between two integers produces an integer as well (closure), parentheses do not matter (associativity), the identity element exists and it is 0, and the inverse element of any integer a is $-a$ (e.g., the inverse of 5 is -5).

> **Remark 2.2 — Abelian groups**
>
> A group $(G, \bullet)$ is called abelian (or commutative) if, besides being a group, the order in which two inputs are provided to the operation $\bullet$ is irrelevant. Formally, $x \bullet y = y \bullet x \forall x, y \in G$.

In cryptography, groups are often used with moduli to limit the number of elements in the groups themselves. The number of elements in a finite group — that is, a group with finitely many elements — is called the group's **order**, and usually (but not necessarily) coincides with the modulus of the group; note that prime numbers are usually chosen as moduli (in which case, we talk about **finite prime group**). For instance, the notation $\mathbb{N}_{13}$ defines the set of natural numbers $\{0, 1, 2, \cdots, 12\}$, where both the modulus and the group order are equal to 13. Conversely, the notation $\mathbb{N}^*_{13}$ (where "*" means to remove the number 0) has modulus 13 but group order 12 — since 0 is not included. Note that there are several notations for defining finite groups, depending on the field (e.g., pure math, number theory, computer science) and the level of formality, as shown in Table 1.

> **Remark 2.3 — Cyclic finite prime groups**
>
> A finite prime group $(G, \bullet)$ of order p is cyclic if, besides being a finite prime group, there exists an

element $g \in G$ that can be used to generate the whole set G by repeatedly applying the operation $\bullet$ to the element g — or, equivalently, so that every element $x \in G$ can be written as $x = g \bullet y$ for some $y \in G$. Formally, $\exists g \in G : \{g \cdot 1, g \cdot 2, \ldots, g \cdot (p-1)\} \equiv G$. The element g may be called **generator**, **base point** (for elliptic curves), **primitive element**, or **primitive root**.

Example 2.3 — An example of an abelian cyclic finite prime group

The group $(\mathbb{Z}_5, +)$ is an abelian cyclic finite prime group with generator 1. In fact, $\mathbb{Z}_5 = \{0,1,2,3,4\} = \{(1+1+1+1+1), (1), (1+1), (1+1+1), (1+1+1+1)\}$ — in fact, $0 \xrightarrow{+1} 1 \xrightarrow{+1} 2 \xrightarrow{+1} 3 \xrightarrow{+1} 4 \xrightarrow{+1} 0$. Also 3 is a generator of $\mathbb{Z}_5 = \{(3+3+3+3+3), (3+3), (3+3+3+3+3), (3), (3+3+3)\}$ — in fact, $0 \xrightarrow{+3} 3 \xrightarrow{+3} 1 \xrightarrow{+3} 4 \xrightarrow{+3} 2 \xrightarrow{+3} 0$. Finally, 0, 2, and 4 are not generators of $\mathbb{Z}_5$.

The vast majority of groups used in cryptography — e.g., for the DL problem (§ 3.3) — are abelian finite prime cyclic groups.

2.4 Fields

A field is a triple $(F, \bullet, \circ)$, where F is a non-empty set while $\bullet$ and $\circ$ are binary operations that satisfy the following properties:

- $(F, \bullet)$ is an abelian group;
- $(F^*, \circ)$ is an abelian group with $F^* = F \setminus 0$ — where 0 is the identity element for the operation $\bullet$.
- **distributivity**: the operation $\circ$ distributes over the operation $\bullet$. Formally, for every $x, y, z \in F$, $x \circ (y \bullet z) = (x \circ y) \bullet (x \circ z)$.

Hence, the operations $\bullet$ and $\circ$ are compatible — in the sense that they can both be applied to elements of the set F. Also, each operation has its own inverse and identity elements. In simpler words, a field $(F, \bullet, \circ)$ is an abelian group which, in addition to the operation $\bullet$, also supports a second operation $\circ$ (over $F^* = F \setminus 0$).

Remark 2.4 — Common names

Often, the operation $\bullet$ is called *addition*, while the operation $\circ$ is called *multiplication*. Consequently, the identity element of $\bullet$ is denoted as 0 and the inverse of an element a as $-a$, while the identity element of $\circ$ is denoted as 1 — with $0 \neq 1$ — and the inverse of an element a as a^{-1}. Also, given a field $(F, \bullet, \circ)$, the group $(F^*, \circ)$ — that is, the group obtained by considering the set $F \setminus 0$ and the multiplication operation — is often called the *multiplicative group of nonzero elements* of F.

Table 1: Common mathematical notations for finite groups

notation	meaning	notes
$\mathbb{Z}_p$	integers modulo p	common in group theory/discrete math
$\mathbb{Z}/p\mathbb{Z}$	quotient group of integers by $p\mathbb{Z}$	most standard in algebra/number theory
$\mathbb{Z}_{/p}$	similar to $\mathbb{Z}/p\mathbb{Z}$	less common
$\mathbb{Z}_p^\times$	multiplicative group of integers mod p	identifies the group $(\mathbb{Z}_p, \times)$
$\mathbb{Z}_p^+$	additive group of integers mod p	identifies the group $(\mathbb{Z}_p, +)$

Most cryptographic algorithms rely on fields with finitely many elements, called **finite fields** or also *Galois Fields*. In fact, finite fields allow for performing polynomial arithmetic reliably (i.e., with properties guaranteeing the presence of identity and inverse elements). The number of elements in a finite field is called the field's **order**. A finite field may be either prime or not prime, as explained below.

Finite Prime Fields. A finite prime field has order p for some prime number p, often denoted as $\mathbb{F}_p = \{0, 1, 2, \ldots, p-1\}$ with addition and multiplication modulo p.

> **Example 2.4 — An example of a finite prime field**
>
> $(\mathbb{F}_7 = \{0,1,2,3,4,5,6\}, +, \times)$ is a finite prime field. Instead, the whole set of integers $\mathbb{Z}$ is not a field as it does not contain many inverse elements for the multiplication operation $\times$, e.g., $\nexists x \in \mathbb{Z} : x \times 5 = 1$; conversely, such an element exists in $\mathbb{F}_7$ and it is 3 (in fact, $3 \times 5 = 15 \equiv 1 \pmod 7$).

Finite Non-Prime Fields. A finite non-prime field has order p^n for some prime number p and integer $n > 1$,[6] often denoted as $\mathbb{F}_{p^n}$. Intuitively, every finite non-prime field $\mathbb{F}_{p^n}$ is somewhat an extension of a smaller prime field $\mathbb{F}_p$ — this is why finite non-prime fields are also called **extension fields**.[7] A finite non-prime field (or, equivalently, an extension field) $\mathbb{F}_{p^n}$ is built starting from a finite prime field $\mathbb{F}_p$ and then defining a polynomial $f(x)$ with the following properties:

- $f(x)$ has degree n;
- $f(x)$ is irreducible over $\mathbb{F}_p$, that is, $f(x)$ cannot be written as a product of two non-constant polynomials — e.g., denoted with $h(x)$ and $g(x)$ — with coefficients in $\mathbb{F}_p$ (§ 2.1). Choosing an irreducible $f(x)$ is crucial, otherwise $\mathbb{F}_{p^n}$ would not be a field — similarly to how choosing a prime p is crucial, otherwise $\mathbb{F}_p$ would not be a field.

  > **Example 2.5 — Example of reducible and non-reducible polynomials in $\mathbb{F}_5$**
  >
  > $f(x) = x^2 + 1$ is reducible over $\mathbb{F}_5$: if we consider $h(x) = (x+2)$ and $g(x) = (x-2)$, then $h(x)g(x) = (x+2)(x-2) = x^2 - 4 \equiv x^2 + 1 \pmod 5 = f(x)$ — remember that $-4 \equiv 1 \pmod 5$. Differently, $f(x) = (x^2 + 2)$ is irreducible over $\mathbb{F}_5$.

Notably, the elements of an extension field $\mathbb{F}_{p^n}$ are not simple integers but polynomials of degree $< n$.

> **Example 2.6 — An example of a finite non-prime field**
>
> Consider the finite prime field $\mathbb{F}_5 = \{0,1,2,3,4\}$, the extension field $\mathbb{F}_{25}$, and the arbitrarily chosen polynomial $f(x) = (x^2 + 2)$ — which has degree 2 and is irreducible over $\mathbb{F}_5$, as said above. To find the 25 elements of $\mathbb{F}_{25}$, we start from the set of all polynomials in x whose coefficients are in $\mathbb{F}_5$ — the set of such polynomials is denoted as $\mathbb{F}_5[x]$ and is equal to $\{c_0 + c_1 x + c_2 x^2 + \ldots + c_d x^d : c_i \in \mathbb{F}_5\}$ (were d denotes the degree of the polynomial, whatever it may be — as long as it is a finite number). Then, we are interested in $\mathbb{F}_5[x]/(x^2+2)$, that is, the set of polynomials $\mathbb{F}_5[x]$ modulus (x^2+2) — similarly to when, starting from the set of integers $\mathbb{Z}$, we define a modulus p in modular arithmetic (§ 2.2).[a] The set $\mathbb{F}_5[x]/(x^2+2)$ is actually isomorphic (i.e., algebraically identical) to $\mathbb{F}_{25}$, so these

[6] The value n needs to be greater than 1, otherwise $\mathbb{F}_{p^1}$ is really just the same as $\mathbb{F}_p$.

[7] More generically, given a base field K, an extension field L is any (larger) field that contains the base field K as a subfield. In the finite-field context of cryptography, an extension field is always a finite non-prime field.

two notations identify the same mathematical structure. Now, note that the polynomial (x^2+2) in $\mathbb{F}_5[x]/(x^2+2)$ is actually equal to the polynomial (0), as (x^2+2) modulus (x^2+2) is obviously (0); hence, $x^2+2=0$ means that $x^2=-2$ and, in $\mathbb{F}_5$, it holds that $-2\equiv 3 \pmod 5$: ultimately, $x^2=3$. This means that, given any polynomial in $\mathbb{F}_5[x]$, we can replace x^2 with 3, so no polynomial in $\mathbb{F}_5[x]/(x^2+2)$ has a degree equal or greater than 2; for instance, (x^2+x+1) in $\mathbb{F}_5[x]/(x^2+2)$ is actually equal to $(x-1)$ or, equivalently, to $(x+4)$. Consequently, every polynomial in $\mathbb{F}_5[x]/(x^2+2)$ has the form $(a+bx)$ for $a,b\in\mathbb{F}_5$. Since we have 5 values for a and 5 values for b, $\mathbb{F}_5[x]/(x^2+2)$ contains exactly $5^2=25$ elements. However, it is better to avoid using x as variable, as x does not correspond to an independent variable anymore since $x^2=3$ in our context. Therefore, it is better to use another symbol, usually α, where α is equal to x but subject to the rule that $\alpha^2=3$. Finally, we can enumerate the elements in $\mathbb{F}_5[x]/(x^2+2)$: when $b=0$ and a goes from 0 to 4, the elements are the polynomials (0), (1), (2), (3), and (4), when $b=1$ and a goes from 0 to 4, the elements are the polynomials (α), $(1+\alpha)$, $(2+\alpha)$, $(3+\alpha)$, and $(4+\alpha)$, …, and when $b=4$ and a goes from 0 to 4, the elements are the polynomials (4α), $(1+4\alpha)$, $(2+4\alpha)$, $(3+4\alpha)$, and $(4+4\alpha)$.

[a] This is why x^2+2 must be irreducible, as it is the modulus of $\mathbb{F}_{25}$.

You may note that there exist multiple extension fields for a finite prime field: after all, it depends on what polynomial is chosen. Still, all of the extension fields of a finite prime field are isomorphic, hence eventually equivalent. Nonetheless, different polynomials may yield different computational costs, better or more secure implementation, or simply be compliant with standards (§15).

2.5 Elliptic Curves

An elliptic curve $E(K)$ is defined as the set of solutions (x,y) to a cubic equation whose most common (or simplest) form is $y^2=x^3+ax+b$ — where any solution (x,y) as well as the coefficients a and b belong to the field K[8] — together with a special "point at infinity" denoted with $\mathcal{O}$.[9] More formally, $E(K)=\{(x,y)\in K\times K : y^2=x^3+ax+b\}\cup\{\mathcal{O}\}$.

Example 2.7 — An example of a simple elliptic curve

Given the finite prime field $\mathbb{F}_{11}$, consider an elliptic curve $E(\mathbb{F}_{11})=y^2=x^3+7$ (hence $a=0$ and $b=7$). When $x=2$, then $y^2=2^3+7=15\equiv 4 \pmod{11}$. Consequently, $y=2$ (that produces $y^2=4$) as well as $y=9$ (that produces $y^2=81\equiv 4 \pmod{11}$) can satisfy the formula, so the points $(2,2)$ and $(2,9)$ are in the elliptic curve $E(\mathbb{F}_{11})$. In total, $E(\mathbb{F}_{11})=\{(2,2),(2,9),(3,1),(3,10),(4,4),(4,7),(5,0),(6,5),(6,6),(7,3),(7,9)\}$ [94].

Curiosity 2.2 — Why are they called *elliptic* curves?

The name "elliptic curve" does not refer to any shortcut or direct connection to ellipses, but rather comes from the fact that these curves were first studied in connection with elliptic integrals — that is, formulas for finding the length of the edge of an ellipse.

Remarkably, $E(K)$ — intended as the set of points of the elliptic curve — is an abelian group, with the point at infinity $\mathcal{O}$ as the identity element and point addition "+" as the operation of the group. However,

[8] K may either be a finite prime field or a finite non-prime field.

[9] The discriminant of the cubic equation must be different from zero to avoid repeated roots; in other words, all solutions (x,y) must be unique.

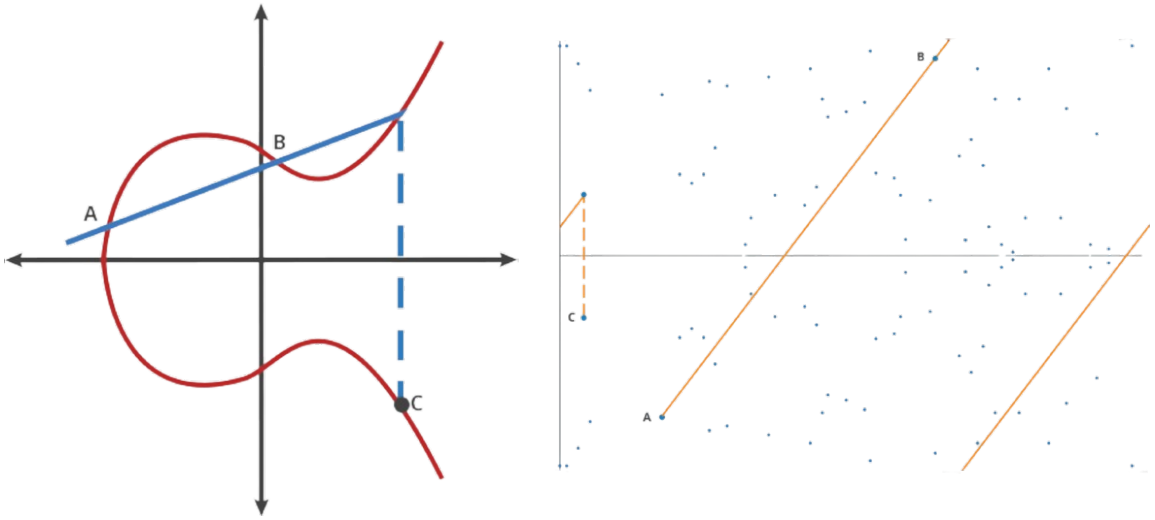

Figure 3: Two visual representations of an elliptic curve (from https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/). On the left, the cubic equation drawn as a curve on a Cartesian graph. On the right, the set of solutions (x, y) to the cubic equation. Each representation also comprises a point addition operation between two points A and B resulting in a third point C

the addition of two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ of an elliptic curve is not trivially equal to $(x_1 + x_2, y_1 + y_2)$. Instead, the addition of two points P and Q is defined as follows (see also Figure 3):

- draw the line connecting P and Q (if $P = Q$, then consider the tangent line as the line connecting P and Q). The line will intersect the cubic equation in a third point denoted with $-R$ (if the line is vertical, use the point at infinity $\mathcal{O}$ as the result of $P + Q$[10]);
- reflect $-R$ across the x-axis to get a new point R which is defined to be $P + Q$;

> **Remark 2.5 — Actual formula to calculate the coordinates of R**
>
> If $P \neq Q$ and $P, Q \neq \mathcal{O}$, given (x_P, y_P) the coordinates of P and (x_Q, y_Q) the coordinates of Q, then the coordinates(x_R, y_R) of R are computed as $x_R = s^2 - x_P - x_Q$ and $y_R = -y_P + s \cdot (x_P - x_R)$, where $s = (y_P - y_Q)/(x_P - x_Q)$

The point addition operation between P and Q when $P = Q$ — that is, summing a point to itself — is called **point doubling** and is denoted with $2P$. Generalizing, nP denotes adding the point P to itself n times. Notably, given two integers n and m, $m(nP) = n(mP)$ — because the point addition operation is associative.

> **Suggested Reading 2.1 — More information on elliptic curves**
>
> The Cloudflare's blog post "A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography" — available at the link https://blog.cloudflare.com/a-relatively-easy-to-

[10] In fact, if the line is vertical, $P = -Q$, hence $P + Q = P - P = \mathcal{O}$ (in elliptic curves, the negative of a point $P(x, y)$ is defined as the point with the same x but the negative y, i.e., $-P(x, -y)$). From this we also get that the point at infinity $\mathcal{O}$ is the identify element for the point addition operation, as $P - P = \mathcal{O}$ implies that $P + \mathcal{O} = P$

understand-primer-on-elliptic-curve-cryptography/ — is an excellent source of more information on elliptic curves.

Remark 2.6 — Different ways to write down elliptic curve equations

The cubic equation $y^2 = x^3 + ax + b$ is also called **Weierstrass form** or *Weierstrass equation*. However, note that elliptic curves may be represented in other forms as well, such as the **twisted Edwards form** ($ax^2 + y^2 = 1 + dx^2y^2$, with $a, d \neq 0$) and the **Montgomery form** ($By^2 = x^3 + Ax^2 + x$).

Popular Elliptic Curves. Intuitively, not all elliptic curves are suitable for cryptographic use; a random curve is most likely insecure in that sense. Therefore, elliptic curves are usually chosen from a list of standard options carefully curated by mathematicians and cryptographers. The most common elliptic curves are:

- **P-256** — also known as NIST P-256, secp256r1 or prime256v1: defined in Weierstrass form over the finite prime field $\mathbb{F}_p$ with modulus $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. This curve is commonly used in Transport Layer Security (TLS) (§ 19.3.1);

  Warning 2.1 — Beware of P-256

  P-256 was created by the NIST with supposedly random coefficients. However, since this claim is not verifiable, some mathematicians and cryptographers suspect the National Security Agency (NSA) to have added a backdoor to this curve (see, e.g., https://words.filippo.io/dispatches/seeds-bounty/).

- **secp256k1**: defined in Weierstrass form over the finite prime field $\mathbb{F}_p$ with modulus $p = 2^{256} - 2^{32} - 977$. This curve is used in blockchains for cryptocurrencies like Bitcoin and Ethereum (§ 19.4.1);
- **Curve25519**: defined in Montgomery form over the finite prime field $\mathbb{F}_p$ with modulus $p = 2^{256} - 19$. This curve is widely used for DH key establishment (§9) in many modern protocols such as TLS, Signal, and WireGuard (§ 19.3). The DH key establishment over Curve25519 is called **X25519**. Also, Curve25519 is free from known patents and is recommended by NIST [65] and in RFC 7748 [122];
- **Ed25519**: defined in twisted Edwards form over the finite prime field $\mathbb{F}_p$ with modulus $p = 2^{256} - 19$. This curve is the bi-rational equivalent twisted Edwards form of Curve25519 (the twisted Edwards form is optimized for operating with digital signatures — see §8);
- **Ed448** — also known as Ed448-Goldilocks: defined in twisted Edwards form over the finite prime field $\mathbb{F}_p$ with modulus $p = 2^{448} - 2^{224} - 1$. This curve is a larger variant of Ed25519;
- **BLS12-381**: defined in (short) Weierstrass form over the finite prime field $\mathbb{F}_p$ with modulus p being a 381-bit prime number. This curve is used for Boneh-Lynn-Shacham (BLS) signatures (§ 8.2) and blockchain applications (§ 19.4.1) — in fact, BLS12-381 is a pairing-friendly curve (§ 3.4.1).

  Curiosity 2.3 — What do numbers in the names of elliptic curves mean?

  Numbers in the name of elliptic curves typically refer to the underlying modulus. For instance, the curves P-256 and secp256k1 have "256" in their name due to the modulus of

their finite prime field (which defines the order of the finite prime field, hence the number of elements in the finite prime field). Similarly, the curves Curve25519 and Ed25519 have "25519" as the underlying modulus is $p = 2^{256} - 19$. Curves with the same (literal part of the) name but different numbers typically denote different sizes of the same curve. For instance, P-256, P-384 (also known as secp384r1), and P-521 (also known as secp521r1) denote the same curve over different moduli; as a rule of thumb, bigger curves offer more cryptographic security at the expense of (computational and space) efficiency.

The choice of the elliptic curve and its form greatly influence the security and efficiency of algorithms.

Suggested Reading 2.2 — More information on the choice of elliptic curves

Bernstein's and Lange's blog post "SafeCurves: choosing safe curves for elliptic-curve cryptography" — available at the link `https://safecurves.cr.yp.to/` — is an excellent source of more information on how to choose elliptic curves (and how their secure implementation is often harder than it seems).

3

TRAPDOOR FUNCTIONS

Many cryptographic algorithms (§ 1.2) rely on one-way and trapdoor functions build upon mathematical structures (§2): a **one-way** function is easy to compute but (computationally) hard to invert — that is, it is prohibitively expensive to obtain the input given the corresponding output — while a trapdoor function is easy to compute but (computationally) hard to invert without knowledge of additional information called **trapdoor** — trapdoor functions are a specific instance of one-way functions. Roughly, hash functions (§5) rely on one-way functions, asymmetric cryptography (§7) relies on trapdoor functions, and symmetric cryptography (§6) usually relies on (keyed pseudo)random permutations or (pseudo)random functions which do not fully match the definition of trapdoor functions.

> **Example 3.1 — Real-world analogies for one-way and trapdoor functions**
>
> A one-way function is like breaking an egg, shredding a sheet of paper, or mixing two colors of paint. Conversely, a trapdoor function is like a padlock, a sealed mailbox, or an hotel room door.

At the time of writing (2025), no function was mathematically proven to be a one-way or trapdoor function. Still, there exist several candidates for trapdoor functions; we briefly describe the most relevant trapdoor functions below. Our goal is not to explain these trapdoor functions exhaustively, but rather to provide the reader with the basic knowledge necessary to connect these trapdoor functions to the algorithms described in later notes and to gain at least an elementary understanding of the subject.

3.1 Integer Factorization

The **Integer Factorization (IF) problem** is a trapdoor function where the "easy part" is computing multiplications between prime numbers while the "hard part" is finding which prime numbers were multiplied to obtain a given number (i.e., factoring — see § 2.1). More formally, given a large composite number n — where n is the product of (usually) two prime numbers p and q, i.e., $n = p \times q$ — is it (believed to be) hard to find two (or more) prime factors of n.

The IF problem is the trapdoor function underlying:

- the Rivest-Shamir-Adleman (RSA) cipher (§ 7.1.1).

Example 3.2 — IF example

It is easy to compute $7 \times 13 = 91$. However, given 91, it is difficult to understand that the factorization of $91 = 7 \times 13$. In fact, the simplest approach to factorize a number is through trial divisions, where we try to divide the number with increasingly greater prime numbers hoping to find a factor. Of course, doing trial divisions for 91 is quick, but doing trial divisions for numbers with hundreds of digits is not.

Warning 3.1 — The IF problem vs. quantum computing

Quantum computing can solve the IF problem using the Shor's algorithm [171] in polynomial time (that is, efficiently) — see § 19.5.2.

3.2 Composite Residuosity

The **Composite Residuosity (CR) problem** is a trapdoor function where the "easy part" is computing exponentiations while the "hard part" is computing the n-th root of a given number. The CR problem is intrinsically a problem only in groups — which in this case are often derived from fields (§ 2.4). More formally, given a finite prime field $\mathbb{F}_n$ — where n is the product of two prime numbers p and q, i.e., $n = p \times q$ — the multiplicative cyclic group of nonzero elements $(\mathbb{F}^*_{n^2}, \times)$, and an element $c \in \mathbb{F}^*_{n^2}$, is it (believed to be) hard to find an element $x \in \mathbb{F}^*_{n^2}$ so that $x^n \equiv c \pmod{n^2}$; computing x^n is easy, but recovering x from c — that is, computing the n-th root of c modulo n — is hard. The word "residuosity" refers to the fact that c is the "residue" of $x^n \pmod{n^2}$ — i.e., what remains after applying the modulus n^2 to x^n — while the term "composite" refers to the fact that n is composite — i.e., n is the product of p and q. The cyclic group $(\mathbb{F}^*_{n^2}, \times)$ may be built only on top of integers (where the operation is the exponentiation), not on top of elliptic curves (where the operation is point addition) — see § 2.5).
Note that not every element $c \in \mathbb{F}^*_{n^2}$ can be written as $x^n \pmod{n^2}$ for a given $x \in \mathbb{F}^*_{n^2}$. In other words, the n-th root of a given number may not exist in $\mathbb{F}^*_{n^2}$. Hence, the CR problem is often formulated as: given the modulus n and an element $c \in \mathbb{F}^*_{n^2}$, decide whether c is an n-residue modulo n^2. In other words, it is infeasible to distinguish n-th powers from random elements of $(\mathbb{F}^*_{n^2}, \times)$.

Remark 3.1 — CR and IF

The CR and IF (§ 3.1) problems are different albeit strongly connected. In fact, being able to factor n allows for trivially resolving the CR problem, and vice versa.

The CR problem is the trapdoor function underlying:

- the Paillier cipher (§ 7.1.2).

Example 3.3 — CR example

Consider the finite prime field $\mathbb{F}_{10}$ and the multiplicative group of nonzero elements $(\mathbb{F}^*_{100}, \times)$. In this context, it is easy to compute the 10-residue of 3 as $3^{10} = 59049 \equiv 49 \pmod{100}$ (hence, 49 is the 10-residue). However, given $c = 49$, it is difficult to understand that 49 is a 10-residue and $49 \equiv 3^{10} \pmod{100}$. In fact, the simplest approach to determine whether an element is a residue consists of trial-and-error computations — quick for a group with modulus 100, but infeasible for

larger groups. As another example, $c = 2$ is not a 10-residue, as no element in $\mathbb{F}^*_{100}$ raised to the 10th power gives 2 $\pmod{100}$.

Warning 3.2 — The CR problem vs. quantum computing

Quantum computing can solve any CR problem using the Shor's algorithm [171] in polynomial time (that is, efficiently) — see § 19.5.2.

3.3 Discrete Logarithm

The **DL problem** is a trapdoor function where the "easy part" is computing exponentiations while the "hard part" is inverting them by computing discrete logarithms. The DL problem is intrinsically a problem only in groups — which in this case are often derived from fields as it happens for elliptic curves (§ 2.5). More formally, given a cyclic group $(G, \bullet)$ with a generator g and an element h in G, is it (believed to be) hard to find the unique integer x such that $g \bullet x = h$; computing $g \bullet x$ is easy, but recovering x from h — that is, computing the discrete logarithm — is hard. The cyclic group $(G, \bullet)$ may be built on top of integers (where the operation is the exponentiation) or elliptic curves (where the operation is point addition). The DL problem based on integers is the trapdoor function underlying:

- the DH key agreement protocol (§9);
- the ElGamal cipher for both encryption and digital signatures (§ 7.1.3);
- the Integrated Encryption Scheme (IES) for hybrid encryption (§ 11.1.1);
- the Digital Signature Algorithm (DSA) for digital signatures (§ 8.2).

Example 3.4 — DL example

Consider the finite prime field $\mathbb{F}_{11}$ and, in particular, its multiplicative group of nonzero elements (i.e., $(\mathbb{F}^*_{11}, \times)$) and the generator $g = 2$. In this context, it is easy to compute $2^5 = 32 \equiv 10 \pmod{11}$. However, given $g = 2$ and $h = 10$, it is difficult to understand that $10 \equiv 2^5 \pmod{11}$. In fact, the simplest approach to compute discrete logarithms consists of trial-and-error computations — quick for a group with modulus 11, but infeasible for larger groups.

Warning 3.3 — The DL problem vs. quantum computing

Quantum computing can solve any DL problem using the Shor's algorithm [171] in polynomial time (that is, efficiently) — see § 19.5.2.

3.4 Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) refers to asymmetric ciphers (such as ElGamal or DSA — see §§ 7.1.3 and 8.2.1, respectively) using groups based on elliptic curves as the foundation for the DL problem [114, 136]. Hence, ECC is not itself an asymmetric cipher, but rather the umbrella term for any cipher employing elliptic curves.

Remark 3.2 — Groups based on elliptic curves vs. integers

Asymmetric ciphers based on elliptic curves (are believed to) guarantee the same level of security with respect to when the same ciphers use groups based on integers **while having smaller keys** (usually, one order of magnitude). Having smaller keys is relevant, as it implies lower memory and computational requirements. Thus, ECC is more suitable for use in resource-constrained environments such as smart cards (§12) — i.e., without dedicated mathematical co-processors or having limited energy available.

The trapdoor function in ECC is built upon the point addition operation over an elliptic curve $E(K)$ (§ 2.5), where the "easy part" is computing point addition operations — more precisely, summing a point G of the elliptic curve to itself a certain number of times d (where d is between 1 and the group order of $E(K)$) to obtain another point Q — while the "hard part" is inverting them — more precisely, understand how many times point addition was applied to G to obtain Q. More formally, the underlying intuition is the same as of the DL problem: given a cyclic group $(E(K),+)$ with a generator G and a point $Q \in E(K)$, is it (believed to be) hard to find the unique integer d such that $G \times d = Q$ — where $G \times d$ is the d-times addition of the point G to itself; computing $G \times d$ is easy, but recovering d from Q — which is called the Elliptic Curve Discrete Logarithm Problem (ECDPL), as it consists in finding d given Q and G — is hard. Consequently, in ECC the pair (G, Q) is usually the public key, while the integer d is usually the private key.

Remark 3.3 — Why recovering d is hard

One may wonder why computing $G \times d = Q$ is easy but computing point addition operations to G repeatedly until obtaining Q as output is hard. Apparently, both computations are the same, as they consist in carrying out point addition operations. However, it is possible to efficiently compute $G \times d$ using the standard "double-and-add" algorithm with complexity $O(\log d)$, while finding d given Q requires trying every possible repeated summation of G to itself. As an example, if $d = 34$, then one who knows d can compute $2G = G+G$, $4G = 2G+2G$, $8G = 4G+4G$, $16G = 8G+8G$, $32G = 16G+16G$, and finally $Q = 34G = 32G+2G$ (for a total of 6 operations), while one who does not know d would need to compute 33 operations (i.e., $G+G$, $2G+G$, $3G+G$, …, $33G+G$). While large d, trying every possible repeated summation of G to itself is simply computationally unfeasible.

ECC is used for the same ciphers as the DL problem based on integers (§ 3.3), that is:

- the Elliptic Curve Diffie-Hellman (ECDH) key agreement protocol (§9);
- the ElGamal cipher for both encryption and digital signatures (§ 7.1.3);
- the Elliptic Curves Integrated Encryption Scheme (ECIES) for hybrid encryption (§ 11.1.1);
- many ciphers for digital signatures such as Elliptic Curve Digital Signature Algorithm (ECDSA), Edwards-curve Digital Signature Algorithm (EdDSA), Schnorr signatures, and BLS signatures (§ 8.2).

Warning 3.4 — The ECDPL in ECC vs. quantum computing

Quantum computing can solve any ECDPL using the Shor's algorithm [171] in polynomial time (that is, efficiently) — see § 19.5.2.

> **Suggested Reading 3.1 — More information on ECC**
>
> The book "Guide to elliptic curve cryptography" by Hankerson et al. [2] is an excellent resource for a deep dive into ECC.

3.4.1 Bilinear Pairings

A **bilinear pairing** is a function that takes 2 inputs and returns 1 output. Typically, the 2 inputs are points on elliptic curves, while the output is an element of a finite field — (§ 2.4).
More formally, a bilinear pairing is a function $e : G_1 \times G_2 \to G_T$,[11] where G_1 and G_2 are (usually additive subgroups of points on) elliptic curves while G_T is a finite field — often, a finite non-prime field; in most cases, $G_1 = G_2$. In this context, a bilinear pairing e has the following properties:

- **bilinearity**: adding two inputs before the pairing is the same as pairing the inputs separately and then multiplying the results. Also, multiplying one of the inputs by a number before the pairing is the same as raising the result of the pairing to the power of that number. Informally, addition in G_1 or G_2 turns into multiplication in G_T, while multiplication in G_1 or G_2 turns into exponentiation in G_T. More formally, for all $a, b \in \mathbb{Z}$, $P \in G_1$, and $Q \in G_2$, then $e(aP, bQ) = e(P, Q)^{a \times b}$.[12] Expanding this formula into one argument at a time:
  - $e(P_1 + P_2, Q) = e(P_1, Q) \times e(P_2, Q)$;
  - $e(P, Q_1 + Q_2) = e(P, Q_1) \times e(P, Q_2)$;
  - $e(aP, Q) = e(P, Q)^a$;
  - $e(P, bQ) = e(P, Q)^b$;
- **non-degeneracy**: e is not always trivial: given a point $P \in G_1$ so that $P \neq \mathcal{O}_{G_1}$ (the point at infinity for G_1), then there exists at least one $Q \in G_2$ such that $e(P, Q) \neq 1_{G_T}$ — where 1_{G_T} is the identity element in G_T — and vice versa. In other words, non-degeneracy is meant to stop the completely useless case where $e(P, Q) = 1_{G_T}$ for every $Q \in G_2$ when $P \neq \mathcal{O}_{G_1}$; it is enough that some Q gives a non-1_{G_T} result, because that proves P can produce meaningful output. More formally, $\forall P \in G_1 \setminus \{\mathcal{O}_{G_1}\}$, $\exists Q \in G_2 : e(P, Q) \neq 1_{G_T}$, and symmetrically $\forall Q \in G_2 \setminus \{\mathcal{O}_{G_2}\}$, $\exists P \in G_1 : e(P, Q) \neq 1_{G_T}$;
- **efficient computability**: there exists an algorithm that computes $e(P, Q)$ efficiently for all $P \in G_1, Q \in G_2$.

> **Curiosity 3.1 — Why are they called bilinear pairings?**
>
> "Linear" because scaling inputs (with addition or multiplication) corresponds to scaling the result (with multiplication or exponentiation), "bi" because it is linear in both inputs, "pairing" because it takes a pair of elements from two groups G_1 and G_2 and pairs them to produce something in G_T.

Thanks to the aforementioned properties, bilinear pairings are used for advanced cryptosystems such as IBE, ABE, and Zero-Knowledge Proofs (ZKPs) (§§ 19.2.1 and 19.2.4 and **??**, respectively). In particular,

[11] G_T as the T stands for "target".
[12] This in the case G_1 and G_2 are additive groups, as it is usually the case. If G_1 and G_2 were multiplicative groups, then $e(P^a, Q^b) = e(P, Q)^{a \times b}$.

bilinearity allows for comparing results from two different sides of an equation and checking if they match — all without revealing the secret numbers used.

Example 3.5 — An example to show why bilinearity is important

Consider a generic pairing $e : G_1 \times G_2 \to G_T$, being $P \in G_1$ and $Q \in G_2$ generators for G_1 and G_2, respectively; e, P, and Q are public values. Now, assume two parties A and B — you can think of them as two parties called "Alice" (A) and "Bob" (B) — have two public values aP and bQ, while a and b are private numbers known by A and B only, respectively. If A and B want to check whether $a \stackrel{?}{=} b$ (modulo group order) without revealing either, A can compute and disclose $e(aP, Q)$ — by bilinearity, this is $e(P, Q)^a$ — while B can compute and disclose $e(P, bQ)$ — by bilinearity, this is $e(P, Q)^b$. Hence, A and B can compare the results: $e(aP, Q) \stackrel{?}{=} e(P, bQ) \iff e(P, Q)^a \stackrel{?}{=} e(P, Q)^b \iff a \stackrel{?}{=} b$. At the end, neither A and B learns the other party's private number, but both can verify whether the private numbers are the same (modulo group order). Note that the private numbers a and b may take different meanings depending on the cryptosystem in which they are used. In general, private numbers are private keys, while public values are public keys or known fixed parameters (§ 1.1).

Bilinear Pairings and KGAs. Some pairing-based cryptosystems expect a KGA who knows a main private key used to generate all other parties' private keys — such as IBE and ABE — while other pairing-based cryptosystems do not expect a KGA — e.g., BLS cipher and some ZKPs.

Example 3.6 — Bilinear pairings for digital signatures

Consider the signer's private key a, the corresponding public key aG (where G is a fixed generator), and the hash of the message to be signed mapped to an elliptic curve point P. In this context, the signature would be aP and the verification of the signature would amount at checking whether $e(\text{signature}, G) \stackrel{?}{=} e(P, \text{public key})$, which is equivalent to $e(aP, G) \stackrel{?}{=} e(P, aG)$ from which follows that $e(P, G)^a = e(P, G)^a$. This is exactly how the BLS digital signature cipher works (§ 8.2.4).

Example 3.7 — Bilinear pairings for data encryption

Consider a KGA with a main private key s and the corresponding main public key sP (where P is a fixed generator).[a] The public key of a party A with identity ID = "Alice" is denoted with Q_{ID} and anyone can compute it by hashing the identity and mapping the digest to an elliptic curve point $Q_{ID} = H(ID) \in G_1$. Differently, A's private key is denoted with d_{ID} and only the KGA can compute it as $d_{ID} = s \times Q_{ID}$ and communicate it security with A. Now, to encrypt a message m so that only A (and the KGA) can decrypt it, another party B can pick a random r and compute a shared secret $K = e(Q_{ID}, sP)^r$ to derive a symmetric key to encrypt m (§6). Then, B sends to A both the ciphertext and $U = rP$. Finally, A can compute $K' = e(d_{ID}, rP)$ which equals $e(sQ_{ID}, rP) = e(Q_{ID}, sP)^r$ by bilinearity, so $K' = K$. In other words, both A and B derive the same key without knowing s or revealing d_{ID}. This is exactly how the Boneh—Franklin IBE cipher works (§ 19.2.3).

[a] Note that the main private key s and the corresponding main public key sP have no special mathematical properties; many other key pairs $(s', s'P)$ could exist, but only the key pair bound to the designated KGA should be trusted by parties. In fact, whoever knows s can derive any private key, and whoever relies on sP assumes it was honestly generated from s without compromise.

Popular Bilinear Pairings. Defining a suitable bilinear pairing implemented concretely over elliptic curves is (very) complex. Therefore, as it happens with elliptic curves, there is a list of popular options

defined over pairing-friendly elliptic curves (e.g., BLS12-381 — see § 2.5):

- **Weil pairing**: one of the earliest pairings defined on elliptic curves;
- **Tate pairing**: faster to compute than Weil, used in practice in many schemes;
- **Ate pairing**: an optimization of Tate pairing, even faster;
- **Optimal Ate pairing**: state-of-the-art speed for certain curves;
- **R-Ate pairing**: another optimized variation.

4

SECURITY PROPERTIES

In the context of information security, a **security property** is a characteristic or feature of an application or of one of its components (e.g., a cryptographic algorithm or protocol) that contributes to the protection of data in one or more phases of its lifecycle — i.e., at rest, in transit, or in use. Security properties may be more or less relevant (or even undesirable) based on the requirements of the underlying scenario. Furthermore, security properties are sometimes opposed (e.g., non-repudiation vs. deniability) and thus cannot be achieved simultaneously. Below, we list security properties common in cryptography.

Confidentiality. Data can only be accessed by authorized agents. In cryptography, confidentiality means that a given ciphertext can be decrypted only by the intended recipient(s).

> **Suggested Reading 4.1 — On the difference between confidentiality and privacy**
>
> Confidentiality is not to be confused with the (related yet different) concept of privacy: while confidentiality concerns who can access what data, privacy concerns how data are used to derived information (see `https://resources.data.gov/glossary/data-vs.-information/` for a clarification of data vs. information). Traditional approaches to privacy are either Privacy-Enhancing Technologies (PETs) — usually involving advanced cryptographic constructions (e.g., see §§ 19.2.6 and 19.2.7) or confidential computing (§ 12.4) — and data anonymization techniques (not covered in this handbook). In both approaches, there is a trade-off between the degree of privacy achieved within a system and the level of data utility. Moreover, such approaches are (sometimes on purpose) often not sufficient to prevent arbitrary interference. In fact, there may be misuses and "function creeps", that is, request of unnecessary data or use of data beyond the intended purpose. Worryingly, PETs are nowadays being implemented by few big companies, that consequently can decide how to design, use, and update them. Summarizing, **privacy is not a goal but a mean with which to protect individuals [177], and it essentially amounts to data purpose limitation**.

Integrity. Any alteration of some data is easily noticeable — at least to the intended recipient(s). In general, integrity states that data can be modified by authorized agents only.

> ⚠ **Warning 4.1 — Confusion on the meaning of integrity**
> Integrity does not mean that data cannot be tampered with by an attacker. In other words, data can be modified by unauthorized agents even in the presence of cryptographic-based integrity mechanisms (e.g., digital signatures).

Availability. Data (and services) are readily available to authorized agents.

Authenticity. A specific instance of integrity stating that claims about the origin of data can be verified cryptographically. We can identify (at least) two kinds of authenticity:

- **sender authenticity** (also called *entity authenticity*): a specific instance of authenticity stating that the intended recipient(s) know who created a given message. This is often the case when using asymmetric cryptography (§7) to provide authenticity;
- **message authenticity** (also called *data origin authenticity* or *ciphertext authenticity* according to the context): a specific instance of authenticity stating that the intended recipient(s) know that whoever created a given message was authorized (e.g., had access to certain cryptographic material). This is often the case when using symmetric cryptography (§6) to provide authenticity. Message authenticity is sometimes also called *plausible deniability*.

> ⚠ **Warning 4.2 — Authenticity vs. authentication vs. authorization**
> Authenticity is not to be confused with the (related yet different) concepts of authentication and authorization: authentication consists in verifying the identity of a party — more generically, an agent — while authorization consists in verifying what actions an agent can perform on what resources and accordingly enforcing the resulting decision.

Public Verifiability. Anyone within a system can verify authenticity claims made on data. Specifically, public verifiability requires that no secret data (e.g., private or secret keys) is needed for such verification.

Non-Repudiation. a specific instance of public verifiability where the proof of authenticity may only have been created by using some secret data (e.g., private keys) reliably associated with a specific party who, therefore, cannot deny having produced it. Non-repudiation is a stronger security property than sender authenticity. In fact, non-repudiation not only requires that some data can be attributed to the sender (i.e., sender authenticity) but also ensures that this attribution holds up even if the sender disputes it. Hence, non-repudiation implies sender authenticity, but sender authenticity does not imply non-repudiation.

> ℹ **Remark 4.1 — Legal aspects of non-repudiation**
> In legal terms, non-repudiation typically implies that it is possible to hold one party to their commitments in a legal court. However, technical solutions said to offer non-repudiation often do not hold up in court because of various real-world gaps. For instance:
>
> - it may not be provable that the secret data (i.e., key) is correctly associated with the party

and that no other party has ever had access to it;

- such technical solutions do not adhere to relevant legal frameworks and policies establishing the legitimacy of signatures as evidence;
- a system holding the key, if compromised (e.g., a laptop infected by malware), may perform some actions that are not warranted by the associated party.

Usually, a quite complex system of technical and legal assurances is required to complement the cryptographic mechanism for non-repudiation (e.g., digital signatures) and fill the aforementioned gaps.

Unforgeability. An attacker cannot create a ciphertext or a digital signature that an honest verifier deems authentic. In the context of digital signatures (§8), unforgeability implies that an attacker cannot produce a digital signature that authenticates to — that is, seems to have been created by — another party without having the party's private key.

Remark 4.2 — Unforgeability vs. non-repudiation

Unforgeability is a technical security property of cryptographic algorithms, stating that a ciphertext or digital signature that authenticates to a specific symmetric or public key can have been created only by using the same symmetric key or the corresponding private key. Still, unforgeability provides no information on which party used the private or symmetric key — that is, unforgeability yields no notion of binding between keys and parties. Conversely, non-repudiation is a legal and practical security property related to how ciphertexts and digital signatures are created and verified. Non-repudiation implies unforgeability and also that a ciphertext or digital signature was created by the owner party of the key in a non-repudiable way, where ownership is usually specified through a PKI — see §14.

Deniability. Whoever created a ciphertext or digital signature can deny it. Deniability may be immediate or may be provided after a certain amount of time or steps in the underlying application (e.g., disclosure of previously secret data like private keys). Deniability is the complement to non-repudiation. In other words, if an application offers deniability, then it cannot offer non-repudiation at the same time — and vice versa.

Zero-Knowledge (ZK). A party (**prover**) can prove to another party (**verifier**) that a certain claim (mathematical **statement**) is true in such a way that the verifier learns nothing beyond the fact that the claim is true. ZK is a fundamental property of ZKPs and is discussed in more detail in **??**.

Verifiability. A party can verify that a cryptographic computation — the execution of which has been outsourced to a third party (such as a cloud provider) — has been executed correctly, without the first party having to perform the cryptographic computation themselves. Usually, the first party verifies the correctness of the output of the cryptographic computation non-interactively using a proof of correctness — often generated with ZKPs or HE (§ 19.2.7) — returned by the third party along with the aforementioned output.

Forward Secrecy. The compromise of a long-term asymmetric private key or session key — where the compromise happens in the future — does not compromise current or past session keys (and, consequently, the confidentiality of ciphertexts created in the past). Forward secrecy is relevant to data in transit but not to data at rest, where parties need to access (encrypted) stored data at any moment in the future, i.e., relying on a long-term key. Forward secrecy can be split into *sender forward secrecy* — that is, forward secrecy in case a key is compromised sender-side — and *recipient forward secrecy* — that is, forward secrecy in case a key is compromised recipient-side. A commonly adopted approach to achieve forward secrecy is employing single-use ephemeral key pairs in Diffie-Hellman Ephemeral (DHE) (§9) — while long-term key pairs should only be used when (sender) authenticity is required.

Remark 4.3 — Perfect forward secrecy

Perfect secrecy is sometimes also called *perfect forward secrecy*, although some deem the word "perfect" to be a misuse in this case.[a]

[a] https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/

Suggested Reading 4.2 — Relevance of forward secrecy

Michael Horowitz's post "Perfect Forward Secrecy can block the NSA from secure web pages, but no one uses it" — available at the link https://www.computerworld.com/article/2473792/perfect-forward-secrecy-can-block-the-nsa-from-secure-web-pages--but-no-one-uses-it.html — and Parker Higgins's post "Pushing for Perfect Forward Secrecy, an Important Web Privacy Protection" — available at the link https://www.eff.org/deeplinks/2013/08/pushing-perfect-forward-secrecy-important-web-privacy-protection — further discuss the relevance of forward secrecy to avoid mass surveillance.

Backward Secrecy. The compromise of a long-term asymmetric private key or session key — where the compromise happened in the past — does not compromise current or future session keys (and, consequently, the confidentiality of ciphertexts created in the future). In other words, backward secrecy implies that key establishment protocols can *self-heal* after the violation of a long-term asymmetric private key or session key. Backward secrecy is also called *future secrecy*.

Remark 4.4 — Security properties and KGAs

Although KGAs are typically trusted to manage keys on behalf of other parties, the presence of KGAs — according to how KGAs are actually used — may invalidate some security properties like authenticity and non-repudiation.

Notes on

HASH FUNCTIONS

last revision: 15 Sep 2025

5

HASH FUNCTIONS

A hash function is a function that takes an input of arbitrary length, applies a series of bitwise operations and compression techniques, and returns as output an array of bytes called **hash** or **digest** of (usually) fixed size[13]. Hash functions have several uses in computer science (e.g., indexing, error checking) as well as in cryptography (§ 5.1); in fact, there exist several hash functions designed to meet the requirements of different use cases. Anyway, well-designed hash functions satisfy the following 4 basic properties:

- **efficiency**: hash functions execute quickly even for large inputs;
- **uniformity**: digests are or appear to be uniformly distributed within the possible output space. Uniformity is essential to reduce the likelihood of *collisions* — that is, returning the same digest for two different inputs;

  > **Example 5.1 — A real-world analogy for uniformity**
  > Uniformity is like rolling a fair dice: each face of the dice has equal probability of landing face up.

- **determinism**: a given input always produces the same digest;

  > **Example 5.2 — A real-world analogy for determinism**
  > Determinism is like ordering a specific item from a vending machine. The machine should always dispense the same product when pressing the same button.

- **avalanche effect**: a tiny change in the input results in a drastically different output. Also, similar inputs should not yield similar digests.

[13] Some hash functions — such as Skein and Keccak/SHA-3 — may be configured to return outputs of variable length. However, the use of digests of variable size is uncommon

> **Example 5.3 — A real-world analogy for the avalanche effect**
> The avalanche effect is like the butterfly effect in urban traffic: a small event, like one car braking slightly on a highway, can cause a chain reaction of slowdowns and traffic congestion miles away.

Hash functions can be used for, e.g., indexing in hash tables, implementing bloom filters, error checking via checksums, and content-defined chunking. We report commonly used hash functions in Table 2. However, note that these hash functions should not be used in presence of attackers or malicious users. In fact, when additional security guarantees are needed, **Cryptographic Hash Function (CHF)** should be used instead (§ 5.1).

5.1 Cryptographic Hash Functions

A CHF — which, in a broader sense, can be seen as a cryptographic algorithm — is a hash function possessing 3 further security properties desirable for cryptographic use in addition to efficiency, uniformity, determinism, and the avalanche effect:

- **pre-image resistance**: a CHF is a (supposedly) one-way function (§3), meaning that it is computationally infeasible to find an input — which, in this context, is also called *pre-image* — producing a specific predetermined digest;

  > **Example 5.4 — A real-world analogy for pre-image resistance**
  > It is easy to shred a piece of paper into tiny fragments but it is difficult to reconstruct a piece of paper from the fragments.

- **second pre-image resistance** (also called *weak collision resistance*): given an input and the corresponding digest, it is computationally infeasible to find another input producing the same digest;

  > **Example 5.5 — A real-world analogy for second pre-image resistance**
  > It is difficult to find a fingerprint that unlocks a biometric scanner registered to someone else's finger.

- **collision resistance (also called *strong collision resistance*)**: it is computationally infeasible to find two inputs producing the same digest.

Table 2: Common hash functions (not to be used in cryptographic contexts)

hash function	strengths	weaknesses	use cases
DJB2	simple	high collision rate	basic hashing
MurmurHash	fast, good uniformity	hash flooding attacks	database indexing
CRC32	fast	high collision rate	error-checking, integrity
FNV Hash	simple, good uniformity	high collision rate	quick hashing
CityHash	fast, good uniformity	complex	large inputs

> **Example 5.6 — A real-world analogy for collision resistance**
> It is difficult to find two different persons with the same fingerprint.

CHF can be used to provide integrity (§4), usually by using a private or secret key to create or *sign* — that is, encrypt — the digest of a ciphertext (see Hash-based Message Authentication Codes (HMACs) in § 6.1.1 and digital signatures in §8). CHFs are also used in Key Derivation Functions (KDFs) (§ 13.1.1). We report commonly used CHFs in Table 3.

> **Remark 5.1 — Internal functioning of CHF**
> Older CHFs — such as MD5, SHA-1, and SHA-2 — are based on the **Merkle–Damgård construction** proposed by Ralph Merkle in 1979 [135]. Instead, the newer SHA-3 is based on a sponge construction (so called because it works in two steps: first *absorb* the whole input by dividing it in blocks and `xor`-ing them, then *squeeze* the output). Finally, BLAKE2 is based on (a variant of) the ChaCha20 stream cipher (§ 6.4.2).

> **Curiosity 5.1 — What is the difference between SHA-3 and Keccak?**
> SHA-3 is sometimes called **Keccak**. However, Keccak is the name of a family of CHFs created by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche which was submitted to the SHA-3 competition of National Institute of Standards and Technology (NIST) in 2007. Conversely, SHA-3 — besides the name of the competition — is the name of the Keccak variants standardized by NIST [143]. In other words, Keccak indicates the whole family of CHFs, while SHA-3 indicates a selected subset of Keccak's CHFs.

5.1.1 Keyed Cryptographic Hash Functions

A keyed CHF is an instance of CHF that — besides data — takes as input also a (usually symmetric) key which is used to produce the digest. The underlying intuition is that two parties knowing the key may exchange a message and its digest (created with a keyed CHF) as to ensure its integrity. Keyed CHF are a broad category that encompasses various constructions beyond just HMACs (§ 6.1.1); indeed, keyed CHF does not necessarily provide (ciphertext) authenticity (§4).

Table 3: Common Cryptographic Hash Functions (CHFs)

hash function	strengths	weaknesses
MD5	very fast, simple	insecure (collisions were found)[a]
SHA-1	fast, simple	insecure (collisions were found)[b]
SHA-2	mostly secure, widely used	length extension attacks
SHA-3	very secure, newer than SHA-2	slower than SHA-2, less adopted
BLAKE2[c]	very secure, faster than SHA-2	less adopted than SHA-2

[a] Peter Selinger: MD5 Collision Demo (`https://www.mscs.dal.ca/~selinger/md5collision/`)

[b] SHAttered (`https://shattered.io/`)

[b] BLAKE2 was announced in 2012 on top of BLAKE, but there is also BLAKE3 that was announced in 2020 on top of BLAKE2.

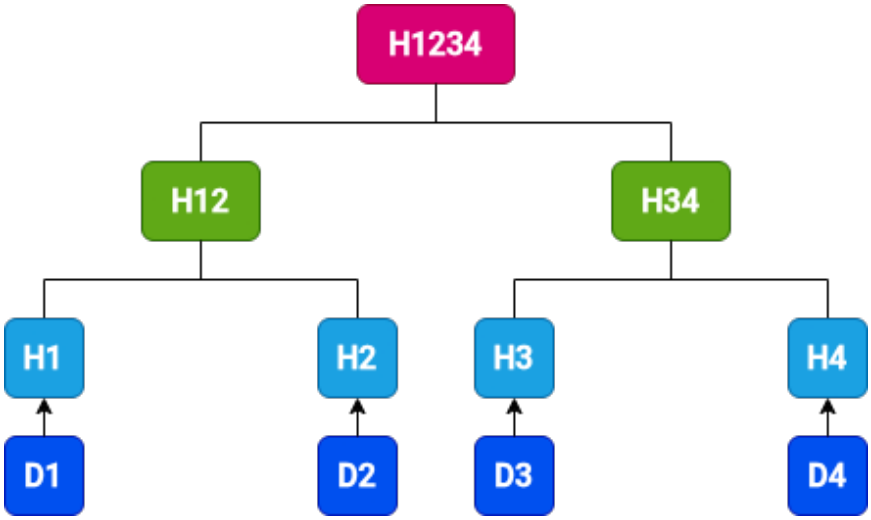

Figure 4: A simple Merkle tree

> **⚠ Warning 5.1 — Length extension attack**
>
> CHFs based on the Merkle–Damgård construction (e.g., MD5, SHA-1, and SHA-2) may suffer from the **length extension attack**: if an attacker knows the digest $H(m_1)$ of an unknown message m_1 and the length $len(m_1)$ of m_1, then the attacker can compute the digest of an extended message $(m_1 \parallel m_2)$ — for an attacker-chosen m_2 — without knowing m_1. This attack is an issue for keyed CHFs, as it implies that **an attacker can include extra information to m_1 and still produce a valid digest without knowing the key**. hence violating data integrity. The length extension attack is possible on CHFs based on the Merkle–Damgård construction because (*i*) these CHF process input data into fixed-size blocks sequentially and each block updates the CHFs' internal state — essentially, the output digest is just the latest internal state — hence appending new blocks is possible by continuing the hashing process starting from the digest (i.e., the last internal state) and (*ii*) padding is completely predictable once the attacker knows $len(m_1)$. Note that HMAC are not affected by the length extension attack — even when built upon CHF based on the Merkle–Damgård construction.

5.2 Merkle Trees

A **Merkle tree** is a cryptographic data structure used to efficiently verify the integrity of large datasets. Concretely, a Merkle tree is a binary tree (Figure 4) where each **leaf node** contains the digest of a **data block**, and each non-leaf node (or **intermediate node**) contains the digest of its two child nodes. The *root node*, known as the **Merkle root**, serves as a unique fingerprint of the entire dataset. For instance, in Figure 4 H1 is the digest of the data block D1, H34 is the digest of the concatenation of H3 and H4, and H1234 (the Merkle root) is the digest of the concatenation of H12 and H34. Any modification to any data block causes the Merkle root to change.

Merkle trees are widely employed in distributed systems — e.g., version control systems such as Git, blockchains (§ 19.4.1), distributed peer-to-peer networks such as BitTorrent, and distributed databases such as Cassandra — to ensure data integrity with minimal effort.

Notes on

SYMMETRIC CRYPTOGRAPHY

last revision: 15 Sep 2025

6 SYMMETRIC CRYPTOGRAPHY

Symmetric cryptography uses the same key — or keys trivially derivable from each other — as input to related cryptographic algorithms. Symmetric cryptography is commonly used on data at rest or in transit over an insecure communication channel for:

- data encryption (§ 6.4);
- creation of Message Authentication Codes (MACs) (§ 6.1).

Symmetric cryptography can provide confidentiality, integrity, and message authenticity; however, symmetric cryptography cannot provide sender authenticity or non-repudiation (§4). The combination of data encryption and MACs is called **Authenticated Encryption (AE)** (§ 6.2).

> **Remark 6.1 — Performance with respect to asymmetric cryptography**
> Symmetric cryptography is generally orders of magnitude (computationally) faster than asymmetric cryptography (§7).

6.1 Message Authentication Codes

A **MAC** — also known as a *cryptographic checksum* or *authentication tag* or simply *tag* — is a sequence of bytes created through a dedicated algorithm taking as input some data and a symmetric key (Figure 5). A MAC provides integrity and — if the symmetric key is known by a specific set of two or more parties — also message authenticity (§4). Differently from digital signatures (§8), MACs are verified using the same symmetric key employed to generate them. By themselves, MACs do not perform encryption; hence, MACs must be combined with (symmetric) encryption to provide confidentiality. There are two main approaches for creating MACs: hash-based (§ 6.1.1) and block cipher-based (§ 6.1.2).

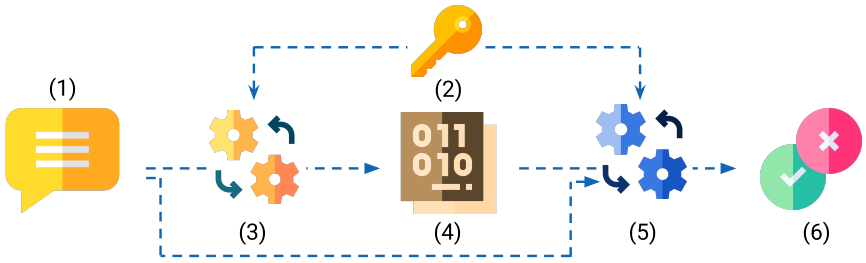

Figure 5: The process of creating and verifying MACs. The data over which the MAC is created (1) — any sequence of bytes of arbitrary length — and a symmetric key (2) are given as input to the MAC creation algorithm (3) which returns as output the MAC (4). The MAC verification algorithm (5) — which is usually very similar to the MAC creation algorithm (3) — takes as input the data (1), the MAC (4), and the symmetric key (2), and returns as outcome a boolean value (6) — either the MAC is valid or it is not.

6.1.1 Hash-based Message Authentication Codes

A **Hash-based Message Authentication Code (HMAC)** — also called *keyed-hash MAC* — is a MAC created by wrapping a Cryptographic Hash Function (CHF) (§ 5.1.1) into two keyed hashing operations over combination of a symmetric key and a given message [118]. HMACs often derive two keys — inner and outer — from the symmetric key, and expect two hash computations to resist length extension attacks (§ 5.1.1). HMACs are a specific instance of keyed CHF.

6.1.2 Cipher Block Chaining Message Authentication Codes

A **Cipher Block Chaining (CBC)-MAC** — also called *Cipher-based Message Authentication Code (CMAC)* — is a MAC created by applying a block cipher (§ 1.3) to a given message. Usually, the message is encrypted using a block cipher in CBC mode with zero Initialization Vector (IV); the last block is the MAC. CMACs are often used in systems where block ciphers are already being used.

6.2 Authenticated Encryption

By itself, symmetric encryption provides confidentiality only. Differently, AE combines symmetric encryption with MACs to provide integrity and message authenticity (recall that sender authenticity requires asymmetric cryptography instead — see §7). There exist three main composition approaches for symmetric encryption and MACs [37]:

- **Encrypt-then-MAC**: provides integrity at the ciphertext level. The MAC — being computed over the ciphertext — does not leak any information on the plaintext. Hence, this is the most ideal composition approach;

> **Warning 6.1 — Use different keys for encryption and MACs**
> Distinct and independent symmetric keys should be used for the encryption of the plaintext and for the derivation of the (keyed) MAC. Otherwise, an attacker might manipulate ciphertext bits in a way that causes predictable changes in the MAC, potentially leaking information or allowing forgery.

- **MAC-then-Encrypt**: provides integrity at the plaintext level (not at the ciphertext level). Therefore, one needs to decrypt the ciphertext to check whether it was tampered with. The MAC — being encrypted — does not leak any information on the plaintext;
- **Encrypt-and-MAC**: provides integrity at the plaintext level (not at the ciphertext level). Therefore, one needs to decrypt the ciphertext to check whether it was tampered with. Also, this composition approach may suffer from Chosen-Ciphertext Attacks (CCAs) (§17) on the symmetric encryption algorithm. Finally, the MAC may leak information on the plaintext — if not computed by adding further information, e.g., nonces or counters.

6.2.1 Authenticated Encryption with Associated Data

Authenticated Encryption with Associated Data (AEAD) is a specific instance of AE that allows adding **associated data** — that is, additional non-confidential information — to the ciphertext. Associated data is not encrypted, but their integrity is guaranteed by the MAC (usually, AEAD follows the encrypt-then-MAC composition approach). The associated data may be used for several reasons, e.g., to add contextual information (e.g., version numbers, nonces) and protect against replay attacks. In other words, an AEAD encryption algorithm receives as input a plaintext, a symmetric key, and the associated data, and outputs a ciphertext and a MAC. Similarly, an AEAD decryption algorithm receives as input a ciphertext, a symmetric key, a MAC, and the associated data, and outputs a plaintext (or an error if the MAC is not verified).

> **Remark 6.2 — Recommended approach to symmetric cryptography**
> The use of AEAD is generally recommended over the use of AE, which in turn is (highly) recommended over custom non-vetted combinations of simple symmetric encryption and MACs.

6.3 Format Preserving Encryption

Most symmetric ciphers (§ 6.4) do not necessarily preserve the format of the plaintext — intended as the set of allowed characters and overall structure — when encrypting. For instance, encrypting a credit card number usually produces a (pseudo-)random binary string rather than a sequence of digits only. The loss of format can pose challenges to existing (especially legacy) information systems that may struggle to accommodate changes in data format. To solve this problem, **Format Preserving Encryption (FPE)** — a category of symmetric ciphers which preserve plaintexts' format — was developed.
Early FPE ciphers were built on standard block ciphers (§ 1.3) where plaintext and (keystreams derived from) keys were defined over a restricted set of characters called **alphabet** [68]. Then, Black and Rogaway proposed the "cycle-following" method [47], which consists of repeating the encryption process iteratively until a ciphertext is produced that conforms to the desired format. The cycle-following method was refined over time and — at the time of writing — the state-of-the-art includes the FF1 and FF3-1 algorithms, recommended by National Institute of Standards and Technology (NIST) [81]; both FF1 and FF3 are instantiations of the FFX framework [38]. Another FPE standard is ANSI ASC X9.124 by major organizations including American Express, IBM, MasterCard, NIST, Thales, and VISA. [15]; ANSI ASC X9.124 specifies three FPE schemes derived from FF1 and FF3, with predefined secure parameters and configurations.

Remark 6.3 — Symmetric encryption vs. FPE

FPE is generally regarded as less robust and less efficient than non-FPE symmetric ciphers, as evidenced by the numerous vulnerabilities that have been identified [36, 93].

6.4 Ciphers for Symmetric Cryptography

Below, we briefly present a short list of renowned or modern symmetric cryptography ciphers.

Warning 6.2 — Old or insecure symmetric ciphers

There exist many symmetric ciphers such as RC4, DES, and 3DES. The use of such ciphers is discouraged. Advanced Encryption Standard (AES) and ChaCha20 are currently the most secure symmetric ciphers.

6.4.1 Advanced Encryption Standard

AES is a symmetric block cipher, hence typically used within a mode of operation (§ 1.3): for AEAD, AES is typically used with Galois/Counter Mode (GCM) or Counter with CBC-MAC (CCM). The block size of AES is always 128 bits, while there are 3 available key lengths: 128, 192, or 256 bits. The size of the IV depends on the mode of operation and it is 96 bits for GCM and CCM. As a final remark, the security of AES is not threatened by quantum (§ 19.5.2).

Curiosity 6.1 — What is the difference between AES and Rijndael?

AES is sometimes called **Rijndael**. However, Rijndael is the name of a family of symmetric block ciphers created by Belgian cryptographers Joan Daemen and Vincent Rijmen which supports 5 block size and key length options (128, 160, 192, 224, 256 bits) and was submitted to the AES competition of NIST in 1998. Conversely, AES — besides the name of the competition — is the name of the 128-bit-block variant of Rijndael standardized by NIST [144]. In other words, Rijndael indicates the whole family of ciphers, while AES indicates a selected subset of Rijndael's ciphers.

6.4.2 ChaCha20

ChaCha20 is a symmetric stream cipher (§ 1.3) with a 96-bit IV and key length fixed to 256 bits. ChaCha20 is usually used with the Poly1305 hash function (§5) for AEAD [146].

Curiosity 6.2 — The history of ChaCha20

In 2005, a symmetric cipher named **Salsa20** was proposed by Daniel J. Bernstein [39] with 64-bit IV and 64-bit counter.[a] A more secure variant of Salsa20 called **ChaCha20** was proposed by the same Bernstein in 2008 [40] — again with 64-bit IV and 64-bit counter. In the same year, Bernstein proposed a variant of Salsa20 called **XSalsa20** — where the "X" stands for "extended", as XSalsa20 extends the length of the IV from 64 to 192 bits [41]. In 2015, a reference implementation of ChaCha20 was published by the Internet Engineering Task Force (IETF) [192] which, however, changed the 64-bit IV and 64-bit counter to a 96-bit IV and 32-bit counter (the change is cryptographically irrelevant except for a reduction of the maximum message length that can be encrypted securely to 256 GB). Finally, in 2020 Scott Arciszewski proposed **XChaCha20** [17], an extension of the IETF version of ChaCha20 with an extended IV of 192 bits but still a 32-bit counter.

[a] Salsa20 produces keystreams in fixed-size blocks of 512 bits. The counter is an integer that increments by 1 for each 512-bit keystream block generated under the same pair of symmetric key and IV. Hence, the counter guarantees that within a single encryption operation, every keystream block uses a distinct input to the cipher and prevents two blocks in the same message from producing the same keystream.

AES vs. ChaCha20. AES and ChaCha20 have similarities and differences:

- they both support **AEAD**;
- they are both used in **Transport Layer Security (TLS) v1.3** [157] (§ 19.3.1). However, AES is also recommended by NIST [25], while **ChaCha20 is not**;
- AES is faster than ChaCha20 on devices equipped with **hardware acceleration for AES** (e.g., AES-NI[14] or similar) such as modern desktops and servers. However, it has been established that **ChaCha20 outperforms AES in software-only implementations**, as it is possible to infer from recent benchmarks [59].[15] Consequently, AES is usually preferred in modern desktops and servers, while ChaCha20 is usually preferred in mobile devices equipped with ARM-based CPUs [146];
- many software-only AES implementations are vulnerable to **side-channel attacks** (§17).

Ultimately, the choice of which symmetric cipher to use depends on the context.

⚠ Warning 6.3 — Committing vs. non-committing AEAD

AEAD outputs a ciphertext c and an authentication tag t by encrypting a plaintext m with a key k, nonce n, and associated data a. However, **basic AEAD may not necessarily guarantee commitment of (c, t) to (k, n, a, m)**. In other words, decrypting c with different key, nonce, or associated data may still work, hence c may decrypt into a different (but still valid) $m' \neq m$ [3]. Therefore, be sure to always use committing AEAD.

[14] Intel Advanced Encryption Standard Instructions (AES-NI) (https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html)

[15] Do the ChaCha: better mobile performance with cryptography (https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/)

Notes on

ASYMMETRIC CRYPTOGRAPHY

last revision: 15 Sep 2025

7

ASYMMETRIC CRYPTOGRAPHY

Asymmetric cryptography uses different keys as input to related cryptographic algorithms; usually, asymmetric cryptography involves the use of public-private key pairs — hence, it is often referred to as **Public-Key Cryptography (PKC)**. Asymmetric cryptography may be used for:

- data encryption (§ 7.1). Typically, public keys are used to encrypt data, while private keys are used to decrypt data;

  > **⚠ Warning 7.1 — Do not use asymmetric cryptography for direct data encryption**
  > There are many reasons why using asymmetric cryptography for direct data encryption is a bad idea (and, in fact, is almost never used in this way). The main reasons are that asymmetric cryptography is generally orders of magnitude (computationally) slower than symmetric cryptography (§6), can directly encrypt small messages only, and the concatenation of multiple blocks of ciphertexts may entail subtle security concerns. Therefore, asymmetric cryptography is commonly used for establishing symmetric keys (see below).

- establish symmetric keys between two or more parties over an insecure communication channel (**key establishment**) through either Diffie-Hellman (DH) (§9) or Key Encapsulation Mechanisms (KEMs) (§10);
- **digital signatures** (§8). Typically, private keys are used to create digital signatures, while public keys are used to verify digital signatures.

Asymmetric cryptography can provide confidentiality, integrity (e.g., with digital signatures), sender authenticity (when key pairs used for creating digital signatures are bonded to parties with digital certificates — see §14 — or key pairs are inherently bonded to parties — see §§ 19.2.1, 19.2.4 and 19.2.5) and non-repudiation (§4).

7.1 Ciphers for Asymmetric Encryption

Below, we briefly present a short list of renowned or modern asymmetric encryption ciphers. Note that the use of asymmetric cryptography ciphers for digital signatures is covered in § 8.2.

7.1.1 Rivest-Shamir-Adleman

Rivest-Shamir-Adleman (RSA) [159] is an asymmetric cipher which relies on the hardness of the Integer Factorization (IF) problem (§ 3.1). Being based on the IF problem, RSA involves a modulus n determined as the product of two prime numbers p and q, a public exponent e, and a private exponent d. The size of the modulus n in bits may vary and is usually one among 1024 (insecure), 2048, 3072, and 4096. A public RSA key is a tuple (n, e), while a private RSA key is a tuple (n, d).

> **Remark 7.1 — On the value of the public exponent e**
>
> The public exponent e is usually set to 65537. Besides other mathematical reasons, the binary representation of 65537 has only two bits set to 1 — i.e., 10000000000000001 — and this allows for carrying out the exponentiation operation for encryption faster. Also, 65537 is a Fermat number (see `https://en.wikipedia.org/wiki/Fermat_number`).

RSA can be used for both encryption and digital signatures (§8): RSAES-OAEP[16] is the best construction for encryption with padding, while RSASSA-PSS[17] is the recommended construction for digital signatures [107]. In theory, RSA can encrypt any message m so that m can be mapped to an integer between 0 (included) and the modulus n (excluded). Concretely, the message m is padded into a block whose length in bits equals that of the modulus n. Depending on the padding scheme, the maximum practical message size may vary (e.g., OAEP allows messages to be up to the length of the modulus n in bits - 528 — for instance, 1520 bits if using a 2048-bit modulus). In reality, RSA KEM (§10) — which avoids padding altogether — should be used for data encryption within hybrid cryptography (§11).

> **Warning 7.2 — Do not use RSA unless you really have a good reason to**
>
> The use of RSA — which is still popular in 2025 — is now heavily discouraged, as more secure or efficient alternatives (e.g., Elliptic Curve Cryptography (ECC) — see § 3.4) are available. In fact, albeit involving fairly easy arithmetic operations, RSA is very difficult to implement correctly due to concerns with prime numbers generation and choice of parameters (§ 3.1), and padding issues and side-channel attacks (§17).

> **Remark 7.2 — RSA and Homomorphic Encryption (HE)**
>
> RSA is multiplicative homomorphic. However, in practice the use of RSA for HE is very limited due to the fact that padding invalidates RSA's homomorphism (§ 19.2.7).

[16] RSAES-OAEP stands for *RSA Encryption Scheme — Optimal Asymmetric Encryption Padding* and is a secure way to encrypt small messages (or keys) with RSA while protecting against chosen-ciphertext attacks (§17)

[17] RSASSA-PSS stands for *RSA Signature Scheme with Appendix — Probabilistic Signature Scheme* and is a secure way to generate digital signatures with RSA by adding randomness (via a salt) and a provably secure padding construction to guard against forgery attacks (§ 8.1)

7.1.2 Paillier

Paillier [150] is an asymmetric cipher which relies on the hardness of the Composite Residuosity (CR) problem (§ 3.2). Being based on the CR problem, Paillier involves a multiplicative cyclic group (based on integers), a modulus n determined as the product of two prime numbers p and q, and a generator g (a common choice is $g = n+1$). The size of the modulus n in bits may vary and is usually one between 2048 and 3072. A public Paillier key is a tuple (n, g), while a private Paillier key is a tuple (λ, μ) — where λ is the least common multiple between $p-1$ and $q-1$ and μ is $\left(L(g^{\lambda} \bmod n^2)\right)^{-1} \bmod n$ with $L(u) = \frac{u-1}{n}$.
Paillier can be used for both encryption and digital signatures (§8). Differently from RSA, Paillier's encryption does not typically include padding. In theory, Paillier can encrypt any message m so that m can be mapped to an integer between 0 (included) and the modulus n (excluded). Concretely, the message m is mixed with a random value r picked randomly from the multiplicative cyclic group of nonzero elements $\mathbb{F}^*_n$.

> **Remark 7.3 — Paillier and HE**
> Paillier is additive homomorphic, hence it can be used in use cases such as secure voting (tallies without revealing individual votes), privacy-preserving data aggregation (e.g., summing encrypted sensor readings) and threshold cryptography (e.g., cryptographic computations require collaboration among multiple parties — see § 19.2.7.

7.1.3 ElGamal

ElGamal [82] is an asymmetric cipher which relies on the hardness of the Discrete Logarithm (DL) problem (§ 3.3). Being based on the DL problem, ElGamal involves a multiplicative cyclic group (either based on integers or elliptic curves — see § 2.5), a prime modulus p, and a generator g. The size of the modulus p in bits may vary and is usually one between 2048 and 3072 for integers and 256 or 384 for elliptic curves. A public ElGamal key is a value $h = g \bullet x$ — where the operation $\bullet$ may be either exponentiation or point addition depending on whether the group is based on integers or elliptic curves, respectively — while a private ElGamal key is the value x (which belongs to the group).
ElGamal can be used for both encryption and (although seldom) also for digital signatures (§8). Concerning encryption, ElGamal expects the encrypting party to choose an ephemeral (§ 1.1) private key k and derive a shared secret $s = h \bullet k$ — shared with respect to the recipient's public key h — which in turn is used to encrypt the message m as $c = m \bullet s$ (hence, m must be an element of the group). The ciphertext and the ephemeral public key $(c, g \bullet k)$ are sent to the recipient for decryption. Differently from RSA, ElGamal's encryption does not typically include padding.

> **Remark 7.4 — ElGamal and HE**
> ElGamal is multiplicative homomorphic, hence it can be used in use cases such as secure voting (tallies without revealing individual votes), privacy-preserving data aggregation (e.g., summing encrypted sensor readings) and threshold cryptography (e.g., cryptographic computations require collaboration among multiple parties — see § 19.2.7.

Remark 7.5 — ElGamal and other asymmetric ciphers

In a sense, ElGamal is a simpler version of Integrated Encryption Scheme (IES) (§ 11.1.1) — it is "simpler" because ElGamal's plaintexts must be (mapped to) an element of the group used in the underlying DL problem, limiting flexibility, while in IES this is not the case. Also, ElGamal is based on the same mathematics as the DH key agreement (§9). However, differently from DH, ElGamal immediately uses the shared secret to encrypt the message, while DH uses the shared secret as one of the inputs to a Key Derivation Function (KDF) (§ 13.1.1) to derive a symmetric key (§6). Finally, ElGamal is the basis of the Digital Signature Algorithm (DSA) for digital signatures (§ 8.2) — essentially, DSA is a refined and standardized variant of ElGamal with well-defined parameters.

8 DIGITAL SIGNATURES

Asymmetric encryption provides confidentiality, while digital signatures provide integrity. In essence, a **digital signature** (or simply signature) is a sequence of bytes created through a dedicated asymmetric algorithm taking as input some data and a private key (Figure 6). The act of creating a signature over some data is also referred to as *signing the data*. The public key associated with the aforementioned private key can be used for *verifying the signature*, thus verifying the integrity of the data.

8.1 Digital Signatures and Security Properties

By themselves, digital signatures provide integrity only. However, depending on how (public keys corresponding to) private keys used to create signatures are bonded to parties, signatures may provide further security properties such as sender authenticity and non-repudiation (§4). The two main mechanisms for binding public keys to parties are Public Key Infrastructures (PKIs) and the Web of Trust (WoT) (§14 and § 14.3, respectively).
In detail, a digital signature provides:

- *public verifiability* if anyone within a system can access the signature, the associated data, and the public key required to verify the signature;
- *sender authenticity* if the key pair used to create the signature — that is, the public key of the key pair — is somehow reliably bonded to a party. Usually, the party actively claims possession of the key pair and this claim is typically endorsed by one or more parties (which may be trusted by everyone within the system);
- *unforgeability* if attackers cannot produce signatures that authenticate to — that is, seem to have been created by — other parties without having the parties' private key;
- *non-repudiation* if it already provides sender authenticity, unforgeability, and the key pair used to create the signature — more precisely, the private key of the key pair — has always been under the sole control of the owner party.

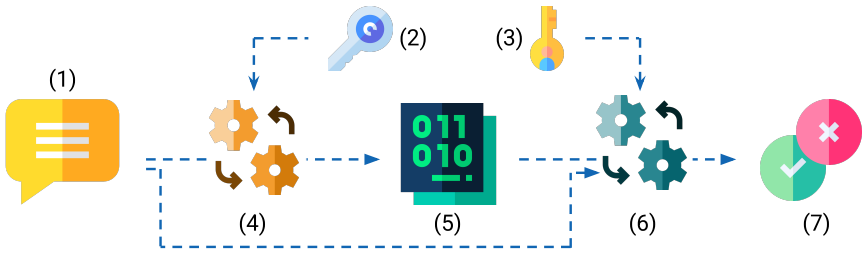

Figure 6: The process of creating and verifying digital signatures. The data to be signed (1) — any sequence of bytes of arbitrary length — and a private key (2) are given as input to the signature creation algorithm (4) which returns as output the signature (5). The signature verification algorithm (6) takes as input the data (1), the signature (5), and the public key (3) associated with the private key (1), and returns as outcome a boolean value (7) — either the signature is valid or it is not.

> **Remark 8.1 — Digital signatures and deniability**
>
> Typically, digital signatures are not used when deniability is required; in such cases, Message Authentication Codes (MACs) (§ 6.1) are used instead.

8.2 Ciphers for Digital Signatures

Below, we briefly present a short list of renowned or modern digital signature ciphers.

> **Warning 8.1 — Old or insecure digital signature ciphers**
>
> The use of DSA and RSA for creating digital signatures is discouraged, as more secure or efficient alternatives are available.

8.2.1 Elliptic Curve Digital Signature Algorithm

Elliptic Curve Digital Signature Algorithm (ECDSA) is a digital signature cipher based on elliptic curves (§ 2.5). Intuitively, ECDSA is an adaptation of an older (and now insecure) digital signature cipher, DSA, to the usage with elliptic curves. The choice of elliptic curve determines the security and the efficiency (in terms of both computational costs and key length) of ECDSA.

> **Warning 8.2 — ECDSA and malleability**
>
> ECDSA has partial malleability: it is possible to manipulate a digital signature to derive a different digital signature for the same data and public key — see §4. If not addressed with proper mitigations, such partial malleability may represent a subtle vulnerability for specific applications.

ECDSA has been adopted and specified by the following standards:

- the X9.62-1998 standard [13] (revised in 2005 and later) by the American National Standards Institute (ANSI) provides a specification of ECDSA, its parameters, and ASN.1 object identifiers;
- the FIPS 186-5 standard [142] by the National Institute of Standards and Technology (NIST) approves and places additional requirements on ECDSA (e.g., recommend specific curves for federal

use). In particular, this standard proposes an ECDSA variant (called deterministic or RFC6979) employing a hash function to prevent misuse of the nonce used during the creation of a digital signature. The variant concerns only the creation of digital signatures, hence it is compatible with vanilla ECDSA;

- the 14888 standard [98] by the International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) includes ECDSA and specifies key generation, signing, and verification processes;
- the 1363-2000 standard [96] by the Institute of Electrical and Electronics Engineers (IEEE) specifies ECDSA as one of several digital signature ciphers.

> **Remark 8.2 — Special features of ECDSA**
>
> Normally, the verification of a digital signature expects 3 inputs: the public key, the digital signature, and the data over which the digital signature was created (Figure 6). However, in ECDSA it is possible to recover a public key from a digital signature and the data (that is, the digital signature and the data allow for recovering the public key). This feature is usually exploited (e.g., in Bitcoin) to reduce the amount of bytes transmitted within a system where public keys are already known.

8.2.2 Schnorr Signature

The Schnorr signature is a digital signature cipher developed by Claus Schnorr in 1990 and based on either integers or (most often) elliptic curves; despite being robust and efficient, the Schnorr signature has seen limited adoption due to a patent which lasted until 2010 [169]. The Schnorr signature shares the same elliptic curves as ECDSA (hence, an ECDSA key pair can ideally be used for Schnorr signatures, and vice versa) and the security considerations on the nonce. However, differently from ECDSA, the Schnorr signature has a rigorous mathematical proof of security.

> **Remark 8.3 — Special features of Schnorr signatures**
>
> The Schnorr signature has a few variants that enable special features, the main ones being:
>
> - **fast batch verification**, that is, it is possible to verify multiple signatures in a single computation;
> - **signature aggregation**, that is, multiple signatures of the same data made with different private keys can be aggregated into a single signature.

8.2.3 Edwards-curve Digital Signature Algorithm

Edwards-curve Digital Signature Algorithm (EdDSA) is a digital signature cipher based on elliptic curves in twisted Edwards form (§ 2.5), which offer increased efficiency and security. EdDSA is based on Schnorr signatures and adds deterministic nonce generation[18] to solve many implementation vulnerabilities (see the Sony PlayStation 3 firmware update signing key[19]). At the time of writing (2025), EdDSA is a popular

[18] The nonce is the hash of a part of the private key and the message

[19] The Sony PlayStation 3 hack deciphered: what consumer-electronics designers can learn from the failure to protect a billion-dollar product ecosystem - EDN (https://www.edn.com/the-sony-

choice for many systems, and especially for securing internet connections — e.g., EdDSA is used in Signal (§ 19.3.2) and Transport Layer Security (TLS) (§ 19.3.1). Also EdDSA has a rigorous mathematical proof of security.
EdDSA has been adopted and specified by the following standards:

- the FIPS 186-5 standard [142] by the NIST (which also standardizes ECDSA);
- the RFC 8032 [42].

> **Remark 8.4 — Special features of EdDSA**
> Like Schnorr signatures, EdDSA allows for **fast batch verification**.

Extensions of the Edwards-curve Digital Signature Algorithm. EdDSA was extended with Extended Edwards-curve Digital Signature Algorithm (XEdDSA) and Verifiable random function Extended Edwards-curve Digital Signature Algorithm (VXEdDSA) by Trevor Perrin in the Signal protocol documentation in 2017.[20]. In brief, XEdDSA involves using the same private key for both creating digital signatures and for the DH exchange (§9); more in detail, XEdDSA provides EdDSA-compatible signatures using the same key pair format as X25519. VXEdDSA extends XEdDSA and transforms it into a Verifiable random function (VRF), a function that, for any input and a private key, produces a (pseudo)random output and a proof that such an output was correctly computed. Hence, anyone with the corresponding public key can verify the proof without learning the secret key. Differently from hash functions (§5), a VRF guarantees both unpredictability (i.e., no one can guess the output before seeing it) and verifiability (i.e., no one can forge a proof for a bogus output). VRFs are often used in services that need public and auditable randomness.

> **Warning 8.3 — XEdDSA and VXEdDSA and the key separation principle**
> XEdDSA and VXEdDSA purposely trade the ideal of key separation principle (§19) for a pragmatic design, but only after a very careful trade-off and design analysis. As benefits, reusing the same key reduces code complexity, key management overhead, attack surface, and storage requirements, at the cost of an "all-or-nothing" risk model.

8.2.4 Boneh-Lynn-Shacham Signature

The Boneh-Lynn-Shacham (BLS) signature is a digital signature cipher based on special elliptic curves that support a particular operation, called pairing (§ 3.4.1). The pairing operation is computationally expensive but enables short keys or short signatures and special signature creation algorithms. Also the BLS signature has a rigorous mathematical proof of security.

> **Remark 8.5 — Special features of BLS signatures**
> Like Schnorr signatures, BLS signatures allow for **fast batch verification** and **signature aggregation**. Moreover, BLS signatures support **threshold signatures** with fairly simple protocols. A

playstation-3-hack-deciphered-what-consumer-electronics-designers-can-learn-from-the-failure-to-protect-a-billion-dollar-product-ecosystem/)

[20] https://signal.org/docs/specifications/xeddsa/

> threshold signature is a special digital signature where the private key is not held by a single party but rather split among a number n different parties, and a number t of these parties (where $t \leq n$) is required to collaborate to produce a digital signature.

9 DIFFIE-HELLMAN

DH is an instance of key agreement protocol (§ 13.1.1) which relies on the hardness of the DL problem (§ 3.3). Two parties, each having a key pair, can use DH to derive a common shared secret sequence of bytes (often just called **shared secret**) over an insecure channel without having to directly exchange (parts of) the shared secret. The shared secret is then fed — combined with other inputs — to a KDF (§ 13.1.1) to derive a shared symmetric key (§6). Being based on the DL problem, DH involves a multiplicative cyclic group (either based on integers or elliptic curves — see § 2.5), a prime modulus p, and a generator g. The size of the modulus p in bits may vary and is usually one between 2048 and 3072 for integers and 256 or 384 for elliptic curves. A public DH key is a value $h = g \bullet x$ — where the operation $\bullet$ may be either exponentiation or point addition depending on whether the group is based on integers or elliptic curves, respectively — while a private DH key is the value x (which belongs to the group).

> **Suggested Reading 9.1 — More information on key establishment based on DL**
>
> NIST published a thorough overview and considerations on the recommended use of the DL problem for key establishment in [27].

The key pairs used by the parties can be either ephemeral or static, resulting in 4 basic modes for DH (Table 4):

- **static-static** — that is, both parties use long-term keys associated with their identity, e.g., through a PKI (§14). Knowing the other party's public key is required to prevent Man-in-the-Middle (MitM) attacks (§19). This mode is called **Authenticated Diffie-Hellman (ADH)** or **Static Diffie-Hellman (SDH)** and provides authenticity but not forward secrecy nor protection against replay attacks (unless contextual information is added for the derivation of the symmetric key — see Algorithm 1). Indeed, the two parties always derive the same shared secret if no other values (e.g., a nonce) are used in the key derivation step;
- **ephemeral-static** — that is, the sender uses a pair of ephemeral keys, while the recipient uses long-term keys. This mode is called **semi-ephemeral DH**, it is used in IES (§ 11.1.1) and provides one-sided authenticity (the sender can verify the authenticity of the recipient) but no forward secrecy

in case of recipient's (long-term) key compromise nor protection against replay attacks (unless the recipient remembers all previously seen ephemeral keys using, e.g., cryptographic accumulators);

- **static-ephemeral** — that is, the sender uses a pair of long-term keys, while the recipient uses ephemeral keys. This mode is called **semi-static DH**. In this mode, the recipient sends its ephemeral public key to the sender. The sender performs DH between the recipient's ephemeral public key and the sender's long-term private key. This mode protects against replay attacks (as it is the recipient that initiates the exchange of messages with a new ephemeral key) and provides one-sided authenticity (the recipient can verify the authenticity of the sender) but not forward secrecy in case of sender's (long-term) key compromise. Finally, this mode protects against Key Compromise Impersonation (KCI) attacks (§19);
- **ephemeral-ephemeral** — that is, both parties use ephemeral keys. This mode is called **Diffie-Hellman Ephemeral (DHE)** and provides forward secrecy (as ephemeral private keys are immediately disposed, thus we assume that the attacker cannot discover them and parties cannot leak them) but no authenticity (as ephemeral private keys are not associated with the identity of the corresponding party).

Example 9.1 — An example of a static-static DH key agreement

Using the DL problem over integers, consider a multiplicative cyclic group with prime modulus 17 and generator 2. Assume the first party A to have long-term private key $\mathbf{Pr_A} = 5$ and public key $\mathbf{Pu_A} = 15$ — that is, $g\text{^}\mathbf{Pr_A} \pmod p$ would be $2\text{^}5 \equiv 15 \pmod{17}$ — and the second party B to have long-term private key $\mathbf{Pr_B} = 7$ and public key $\mathbf{Pu_B} = 9$. Hence, assuming each party knows the other party's public key, A can derive the shared secret $\mathbf{k} = \mathbf{Pu_B}\text{^}\mathbf{Pr_A}$ as $9\text{^}5 \equiv 8 \pmod{17}$, an B can derive the shared secret $\mathbf{k} = \mathbf{Pu_A}\text{^}\mathbf{Pr_B}$ as $15\text{^}7 \equiv 8 \pmod{17}$. This is possible because both derivations amount at computing $g\text{^}\mathbf{Pr_B}\text{^}\mathbf{Pr_A}$.

Table 4: Security properties and protection from attacks in different DH modes

DH Mode	Security Properties				Attacks Protection		
	Sender Authenticity	Recipient Authenticity	Sender Forward secrecy	Recipient Forward secrecy	Replay Attack	KCI	MitM
static-static	✓	✓	✗	✗	✗	✗	✓
ephemeral-static	✗	✓	✓	✗	✗	✗	✗
static-ephemeral	✓	✗	✗	✓	✓	✓	✗
ephemeral-ephemeral	✗	✗	✓	✓	✓	✓	✗

> **Remark 9.1 — Combining DH modes**
>
> It is possible to achieve more security properties by using both ephemeral and long-term key pairs — thus, combining together (the shared secrets derived from) different DH modes. For instance, the combined use of ephemeral-ephemeral, static-ephemeral, and ephemeral-static modes is called Triple Diffie-Hellman (3DH) (§ 9.1) and provides mutual authenticity and forward secrecy.

Algorithm 1: Authenticated Diffie-Hellman

```
# Sender
function adh_sender(sender_pri_key, recipient_pub_key, context):

  # The use long-term keys provides authenticity
  var sss = dh(sender_pri_key, recipient_pub_key)

  # it is a best practice to inject contextual information
  var sym_key = kdf.derive_key(sss || context)
  return (sym_key)

# Recipient
function adh_recipient(sender_pub_key, recipient_pri_key, context):
  var sss = dh(recipient_pri_key, sender_pub_key)
  var sym_key = kdf.derive_key(sss || context)
  return sym_key
```

Finally, note that:

- DH implemented using the DL problem based on elliptic curves is called **Elliptic Curve Diffie-Hellman (ECDH)**;
- DHE implemented using the DL problem based on elliptic curves is called **Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)**;

> **Remark 9.2 — DH with more than two parties**
>
> DH can be generalized for more than two parties. The most straightforward approach is to apply the operation $\bullet$ on the chosen generator g for every party's private key once — in any order. Intuitively, with this approach the complexity increases with the number of parties — exponentially in the exchange of messages (since every party has to produce an intermediate value for each other party, hence creating a complete graph) and logarithmically in the number of operations (using a divide-and-conquer approach).

9.1 Triple Diffie-Hellman

The **3DH** key agreement protocol combines ephemeral and long-term key pairs to derive (three shared secrets that are then fed to a KDF to derive) a shared symmetric key [119]. In particular, each of the two parties uses DH *three times* — hence, *triple* DH (Algorithm 2): (i) ephemeral-ephemeral, (ii) static-ephemeral, and (iii) ephemeral-static mode. Afterwards, the KDF takes as input the three shared secrets as well as all public keys (so to bind the shared key to the public keys). 3DH guarantees authenticity — as long-term keys are used — and forward secrecy — as long as parties properly dispose of the ephemeral private keys. Finally, 3DH protects against MitM attacks.

Algorithm 2: Triple Diffie-Hellman

```
# From the sender's point of view
function 3dh(sender_pri_key, sender_pub_key, recipient_pub_key, recipient_eph_pub_key):
  var (sender_eph_pri_key, sender_eph_pub_key) = asym.generate_key_pair()
  var ees = dh(sender_eph_pri_key, recipient_eph_pub_key)
  var ses = dh(sender_pri_key, recipient_eph_pub_key)
  var ess = dh(sender_eph_pri_key, recipient_pub_key)
  var sym_key = kdf.derive_key(ees || ses || ess || sender_pub_key || sender_eph_pub_key ||
    recipient_pub_key || recipient_eph_pub_key)
  return sym_key
```

9.2 Extended Triple Diffie-Hellman

The Extended Triple Diffie-Hellman (X3DH) key agreement protocol — used in Signal (§ 19.3.2) — is an extension of 3DH designed for allowing two (or more) parties to *asynchronously* establish a shared symmetric key [129]. X3DH adds to 3DH the concept of **signed prekeys** and an (optional) fourth DH calculation with **one-time prekeys** (Algorithm 3); asynchronicity is achieved through the upload of cryptographic material to a server prior the execution of X3DH.

> **Curiosity 9.1 — Why are they called *signed prekeys* and *one-time prekeys*?**
>
> Prekeys are so called because they can be seen as keys that are published (to the server) before — that is, *prior* — starting the X3DH key agreement protocol. Signed prekeys are so called because they are signed by their owner to guarantee authenticity (that is, binding them to the identity of their owner), while one-time prekeys are so called because they can be used in one run of the X3DH key agreement protocol only and then deleted.

The use of ephemeral one-time prekeys provides forward secrecy (recall the discussion in §9) and prevents replay attacks — indeed, without the use of an ephemeral one-time prekey, the two parties would derive the same shared symmetric key at each run of the X3DH protocol. In both cases, it is recommended that the post-X3DH communication protocol negotiates a new symmetric key based on fresh random input (e.g., see cryptographic ratchets in § 19.5.7). Finally, all parties are supposed to regularly replenish the set of ephemeral one-time public prekeys stored in the server. Whenever no ephemeral one-time public prekey is available, the use of ephemeral signed prekeys provides similar but weaker security properties with respect to ephemeral one-time prekeys. Indeed, ephemeral signed prekeys (along with the corresponding signatures) are renewed at some interval only (e.g., once every week).

Algorithm 3: Extended Triple Diffie-Hellman

```
# Step 1 - the recipient (recip) uploads cryptographic material to the server
function setup(recip_pub_key, recip_pri_key, number_of_one_time_prekeys):
  var recip_pub_one_time_prekeys = []
  for i in range(1..number_of_one_time_prekeys):
    # The recipient stores recip_pri_one_time_prekey locally
    var (recip_pri_one_time_prekey, recip_pub_one_time_prekey) = asym.generate_key_pair()
    recip_pub_one_time_prekeys.add(recip_pub_one_time_prekey)

  # The recipient stores recip_pri_signed_prekey locally
  var (recip_pri_signed_prekey, recip_pub_signed_prekey) = asym.generate_key_pair()
  var signature_recip_pub_signed_prekey = asym.sign(recip_pub_signed_prekey, recip_pri_key)

  upload(recip_pub_key, recip_pub_signed_prekey, signature_recip_pub_signed_prekey,
    recip_pub_one_time_prekeys)

# Step 2 - the sender (send) sends the first message
```

```
function first_send(send_pub_key, send_pri_key, first_message):
  # The server allows the send to download one of the recip's one-time prekeys, if
  # one exists, and then delete it. Otherwise, recip_pub_one_time_prekey would be null
  var (recip_pub_key, recip_pub_signed_prekey, signature_recip_pub_signed_prekey,
    recip_pub_one_time_prekey) = download()

  asym.verify(recip_pub_signed_prekey, signature_recip_pub_signed_prekey, recip_pub_key)

  var (send_eph_pri_key, send_eph_pub_key) = asym.generate_key_pair()
  var ses = dh(send_pri_key, recip_pub_signed_prekey)
  var ess = dh(send_eph_pri_key, recip_pub_key)
  var ees = dh(send_eph_pri_key, recip_pub_signed_prekey)
  var eos = dh(send_eph_pri_key, recip_pub_one_time_prekey)
  var sym_key = kdf.derive_key(ses || ess || ees || eos)

  var ad = send_pub_key || recip_pub_key
  var first_ciphertext = aead.encrypt(sym_key, ad, first_message)
  send(send_pub_key, send_eph_pub_key, recip_pub_one_time_prekey, first_ciphertext)

# Step 3 - the recipient receives the first message
function first_receive(recip_pub_key, send_pub_key, send_eph_pub_key, first_ciphertext):
  # The recipient retrieves recip_pri_key, recip_pri_signed_prekey, and recip_pri_one_time_prekey
  var ses = dh(send_pub_key, recip_pri_signed_prekey)
  var ess = dh(send_eph_pub_key, recip_pri_key)
  var ees = dh(send_eph_pub_key, recip_pri_signed_prekey)
  var eos = dh(send_eph_pub_key, recip_pri_one_time_prekey)
  var sym_key = kdf.derive_key(ses || ess || ees || eos)

  var ad = send_pub_key || recip_pub_key
  var first_message = aead.decrypt(sym_key, ad, first_ciphertext)
```

10

KEY ENCAPSULATION MECHANISMS

The term **KEM** indicates a category of key transport protocols (§ 13.1.1). Differently from direct asymmetric encryption (Figure 7), a KEM expects a party to generate a random sequence of bytes whose length corresponds exactly to the length of a block of a given asymmetric cipher (such as RSA — see § 7.1) — hence, avoiding the risks of using padding (Algorithm 4). The random sequence of bytes will be fed to a KDF (§ 13.1.1) — possibly along with further contextual data — to derive a symmetric key which will be used for the actual data encryption (§6). The random sequence of bytes — from now on called **encapsulated key** — is encrypted with the recipient's public key. In this way, the recipient can decrypt the random sequence of bytes using the corresponding private key and derive the same symmetric key.

> **Remark 10.1 — KEM freshness**
>
> A generic KEM generates — or, better, is supposed to generate — a fresh symmetric key every time, since it involved the generation of a random sequence of bytes at every run.

Algorithm 4: Key Encapsulation Mechanism

```
# Sender
function kem_encapsulate(recipient_pub_key):
  var encapsulated_key = rand.generate_secure()
  var encrypted_encapsulated_key = asym.encrypt(encapsulated_key, recipient_pub_key)
  var sym_key = kdf.derive_key(encapsulated_key)
  return (sym_key, encrypted_encapsulated_key)

# Recipient
function kem_decapsulate(encrypted_encapsulated_key, recipient_pri_key):
  var encapsulated_key = asym.decrypt(encrypted_encapsulated_key, recipient_pri_key)
  var sym_key = kdf.derive_key(encapsulated_key)
  return sym_key
```

> **Remark 10.2 — DH vs. KEM**
>
> DH (§9) is an instance of key agreement protocol, while a KEM pertains to key transport protocols (hence, as shown in Figure 14, both DH and KEMs relate to key establishment). The main

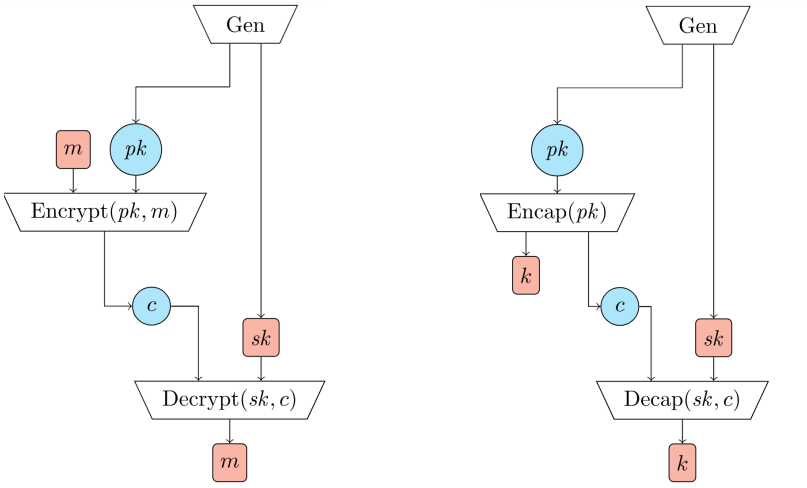

Figure 7: Direct asymmetric encryption (on the left) vs. KEM (on the right) — from https://en.wikipedia.org/wiki/Key_encapsulation_mechanism

difference between DH and KEMs is that DH **typically** expects both parties to contribute to the symmetric key derivation with random or private values — and only by combining both contributions can the shared secret be computed. Differently, KEMs expect one party only (the sender) to contribute with random or private values (as the recipient's public key is long-term, thus static) to derive a fresh symmetric key, encrypt (a precursor of) the symmetric key with the recipient's public key, and send the resulting ciphertext to the other party (the recipient); only at this point can the recipient decrypt (the precursor of) the symmetric key. Although it is true that **semi-ephemeral DH (i.e., ephemeral-static mode) is practically equivalent to a KEM**, the concepts of DH and KEMs are in general distinguished.

⚠ Warning 10.1 — KEM and the KCI attack

Straightforward KEMs — in which only the public key of the recipient is used — are vulnerable to the KCI attack (§19).

10.1 Authenticity in Key Encapsulation Mechanisms

Straightforward KEMs provide confidentiality but not sender authenticity. In other words, anyone can produce a valid ciphertext knowing the recipient's public key. To provide sender authenticity, the sender may either (*i*) create a digital signature of the ephemeral public key, the encapsulated key, or the symmetric key with a long-term key associated with the sender's identity (although this forces non-repudiation) — see §14 — or (*ii*) include the output of static-static or static-ephemeral DH mode in the derivation of the symmetric key [21] — hence, use DH instead of a KEM.

[21] https://soatok.blog/2020/04/21/authenticated-key-exchanges/

10.2 Key Encapsulation Mechanisms with Multiple Recipients

A KEM may be used to transport the same symmetric key to multiple recipients by leveraging key wrapping (§ 1.1). In other words, the sender generates a random symmetric key, and then wraps it separately for each recipient. Hence, (the symmetric keys derived from) the encapsulated keys are used as Key Encryption Keys (KEKs) rather than for directly encrypting data.

> **Suggested Reading 10.1 — More information on key encapsulation mechanisms**
>
> Soatok's blog post "KEM Trails — Understanding Key Encapsulation Mechanisms" — available at the link `https://soatok.blog/2024/02/26/kem-trails-understanding-key-encapsulation-mechanisms/` — is an excellent supplementary resource for KEMs.

Notes on

HYBRID CRYPTOGRAPHY

last revision: 15 Sep 2025

11 HYBRID CRYPTOGRAPHY

Hybrid cryptography refers to the combined use of distinct asymmetric and symmetric ciphers in a single application or context. Usually, hybrid cryptography is used for the encryption of data either at rest or in transit over an insecure communication channel. Consequently, hybrid cryptography is also called *hybrid encryption*.

At a high level, hybrid cryptography uses asymmetric ciphers to establish symmetric keys (§ 13.1.1 for a definition of key establishment), which are then used as input to symmetric ciphers for data encryption — thus, symmetric keys are Data Encryption Keys (DEKs) (§ 1.1), although asymmetric keys are not necessarily Key Encryption Key (KEK).

> **Remark 11.1 — Diffie-Hellman (DH) and Key Encapsulation Mechanisms (KEMs)**
> DH and standalone KEMs are not considered hybrid cryptography themselves, rather they are essential building blocks in hybrid cryptography constructions.

11.1 Ciphers for Hybrid Cryptography

Below, we briefly present a short list of renowned or modern hybrid cryptography constructions. At a high level, a hybrid cryptography construction may be based either on DH (§9) or KEMs (§10). However, as said in §10, semi-ephemeral DH is practically equivalent to a KEM.

> **Warning 11.1 — The simplest (and insecure) approach to hybrid cryptography**
> A naive approach to hybrid cryptography consists in encrypting a given plaintext with a (fresh and securely randomly generated) symmetric key, which in turn is encrypted with the recipient's public key (Algorithm 5). Intuitively, this approach — which may be seen as a simplified version of a KEM — does not allow for including contextual information, may require padding (hence introduce vulnerabilities), and does not expect using Key Derivation Functions (KDFs).

Algorithm 5: Naive Hybrid Encryption

```
# Sender
var sym_key = sym.generate_sym_key()
var encrypted_sym_key = asym.encrypt(sym_key, recipient_pub_key)
var ciphertext = sym.encrypt(plaintext, sym_key)
return (ciphertext, encrypted_sym_key)

# Recipient
var sym_key = asym.decrypt(encrypted_sym_key, recipient_pri_key)
var plaintext = sym.decrypt(ciphertext, sym_key)
```

11.1.1 Integrated Encryption Scheme

The **Integrated Encryption Scheme (IES)** is built on top of DH (§9) — hence, the Discrete Logarithm (DL) problem (§ 3.3). For this reason, IES is also called *Diffie-Hellman Integrated Encryption Scheme (DHIES)*. We can identify two instances of IES: **Discrete Logarithm Integrated Encryption Scheme (DLIES)** — when integers are used for the DL problem — and **Elliptic Curves Integrated Encryption Scheme (ECIES)** — when elliptic curves are used for the DL problem.
In both cases, IES consists of (usually) two parties carrying out the DH key agreement protocol and then using the shared secret to derive symmetric keys for communication (one key for symmetric encryption and a different key for computing Message Authentication Codes (MACs)). Typically, IES expects semi-ephemeral DH (i.e., ephemeral-static DH mode), where the sender uses a pair of ephemeral keys while the recipient uses long-term keys — hence, a straightforward KEM.

> **Curiosity 11.1 — Why is it called *integrated*?**
> IES includes the word "integrated" as it integrates DH, a KDF, a symmetric cipher, and a MAC.

IES is described in, among others, [174, 14, 5].

11.1.2 Data Encapsulation Mechanisms

The use of a symmetric cipher (§ 6.4) on the symmetric key derived from a KEM to encrypt data is called **Data Encapsulation Mechanism (DEM)**. Hence, the KEM-DEM paradigm is considered a hybrid cryptography construction.

> **Remark 11.2 — KEM-DEM and post-quantum cryptography**
> Many modern hybrid cryptography constructions rely on the KEM-DEM paradigm, especially in post-quantum cryptography (§ 19.5.2).

Notes on

PRAGMATIC ASPECTS

last revision: 15 Sep 2025

12 CRYPTOGRAPHIC MODULES

A **cryptographic module** is a set of hardware, software, firmware, or some combination thereof that implements cryptographic computations within a contiguous perimeter defining the physical or logical boundaries of the module [141]. **Software-based cryptographic modules** correspond to any kind of software implementing cryptographic computations for general-purpose hardware, while **hardware-based cryptographic modules** correspond to dedicated hardware implementing cryptographic computations through ad-hoc circuits [79]. Below, we briefly present popular hardware-based cryptographic modules (Table 5) and compare hardware- and software-based cryptographic modules (§ 12.6).

12.1 Hardware Security Modules

A Hardware Security Module (HSM) (Figure 8) is a physical device — either pluggable or standalone — specifically designed to provide a complete solution for algorithm execution and secure key management (§13). HSMs implement a dedicated operating system and allow for defining Access Control (AC) policies (§ 19.5.8) to mediate requests — sent on a secure channel — toward their Application Programming Interfaces (APIs). HSMs are thoroughly tested and built with specialized hardware (usually certified by third-party regulators — see §15) offering protection against physical, chemical, or mechanical tampering. Furthermore, HSMs are usually equipped with sensors to detect attacks (such as drilling, breaking open, or applying acid to the casing) and trigger proper countermeasures (e.g., automatic erase of all keys stored within the HSM) — this property is often called **tamper responsiveness**.

> **Curiosity 12.1 — How much does an HSM cost?**
>
> HSMs are expensive hardware costing on average between 10.000€ and 20.000€ (cheaper alternatives are of course available but may present security and performance trade-offs). For this reason, HSMs are usually used at the server side only as the first building block for implementing Key Management Systems (KMSs).

HSMs provide True Random Number Generators (TRNGs) (§ 19.5.4) based on physical processes (e.g., atomic decay or atmospheric noise). Besides, HSMs can execute (usually but not necessarily prede-

Figure 8: A generic HSM where the top lid was made transparent only for showing the internal circuitry; of course, the case of an HSM is typically made entirely with metal (image generated with OpenAI's Sora — https://sora.chatgpt.com/).

Figure 9: A generic Secure Element (SE) (image generated with OpenAI's Sora — https://sora.chatgpt.com/). Note that SEs come in a wide variety of fashions.

fined) algorithms securely through dedicated cryptoprocessors, and some new HSMs may even run arbitrary code. Finally, HSMs allows exporting keys, usually previously encrypted with a key derived from a Password-based Key Derivation Function (PBKDF) (§ 13.1.1); in this regard, it is important to note that, while preventing a Single Point of Failure (SPoF), exporting a key from a HSM degrades the security of that key (as an attacker may then be able to steal it and perform an offline guessing attack — see § 16.3).

12.2 Secure Elements

A SE (Figure 9) is a chip — either discrete, pluggable, or integrated — capable of storing secrets and executing preloaded algorithms packaged as (usually Java) applets. SEs almost always has limited storage capacity and processing speed — often because SEs receive power by induction. Besides, SEs are typically tamper-resistant and provide access restriction for trusted applications only.

> **Remark 12.1 — Formats for SEs**
>
> SEs have a wide range of implementations such as SIM cards, EMV chips, smart microSD, smart cards, cryptographic tokens, hardware digital wallets, and embedded circuits. SEs are either portable or embedded in mobile devices.

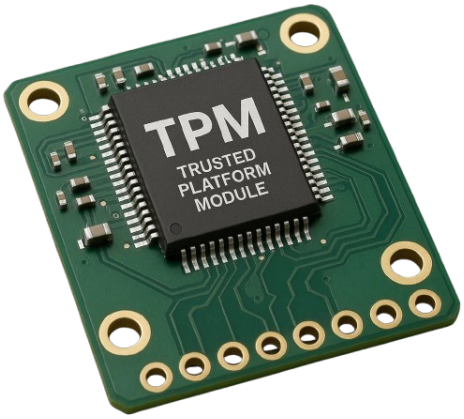

Figure 10: A generic Trusted Platform Module (TPM) (image generated with OpenAI's Sora — https://sora.chatgpt.com/).

12.3 Trusted Platform Modules

A TPM (Figure 10) is a chip either plugged or soldered into the motherboard of devices capable of storing secrets (usually keys, certificates, and passwords) and executing predefined algorithms through dedicated cryptoprocessors. TPMs are mainly used as hardware trust roots (§ 14.1.1) — TPMs do not have general computational capacity — and are typically equipped with:

- an **endorsement key** created during the manufacturing process. The endorsement key cannot be exported and can be used for authenticating the TPM — and, consequently, the device where the TPM is installed;
- a **storage root key**, either built-in or (more often) derived from the endorsement key and a password specified by the owner of the device where the TPM is installed. The storage root key cannot be exported and is typically used as Key Encryption Key (KEK) (§ 1.1);
- an **attestation identity key** used to protect the TPM against tampering attempts on its firmware through Message Authentication Codes (MACs) (§ 6.1).

TPMs are usually found in laptops and desktop computers, as their adoption in mobile devices has been hindered by computational, energy, and monetary constraints.

> **Curiosity 12.2 — The history of the TPM**
>
> The first version of the TPM dates back to 2003, while the latest TPM version (2.0) dates back to 2012 — the organization standardizing TPMs is the Trusted Computing Group (TCG).[a]
>
> [a] Trusted Platform Module (TPM) | Trusted Computing Group (TPM) (https://trustedcomputinggroup.org/work-groups/trusted-platform-module/)

12.4 Trusted Execution Environments

A TEE (Figure 11) is a dedicated area of the main processor of a device (laptops, desktop computers, mobile, and even Internet of Things (IoT) devices) — hence, separated from the operating system — allow-

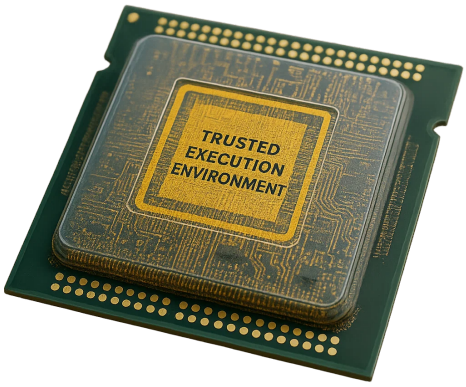

Figure 11: A generic Trusted Execution Environment (TEE) (image generated with OpenAI's Sora — https://sora.chatgpt.com/).

ing to run applications with access to the resources of the device itself securely. At a high level, TEEs can easily run arbitrary code and are often used for digital wallets, secure booting (although TPMs are more apt at that), mobile payments, and (especially) confidential computing[22] [140]. GlobalPlatform[23] is the main (non-profit) organization defining TEEs' architectures and supporting the standardization of TEEs's technical specifications and capabilities. Hence, although TEEs may be implemented with fundamentally different technologies — and their level of security may vary accordingly — basic features inherent to any (or at least the vast majority of) TEEs are [161]:

- **remote attestation**, which allows for verifying the integrity (hence, the trustworthiness) of the TEE and its contents; one possible implementation involves generating a report containing measurements of the TEE's status (e.g., deployed software, firmware) using cryptographic hash functions: the report is signed with a dedicated private key and sent to an **attestation authority** — a trusted third party that usually coincides with the manufacturer of the TEE — which verifies the signature of the report and compares the measurements contained therein against expected values;
- **isolated execution**, which protects data and software running within the TEE. More precisely, TEEs usually preserve the confidentiality and the integrity of data contained therein, and the integrity (but not the confidentiality) of the software executed therein — still, note that it is possible to preserve the confidentiality of sensitive code by, e.g., treating it as data consequently executed by a (public) interpreter or by decrypting (encrypted) sensitive code within the TEE (e.g., see [154]). Importantly, isolated execution may occur either at the **Virtual Machine (VM) level** or at the **application level** (the latter is also referred to as *process level*):[24]
  - *application-level isolated execution*: protection at the application level separates the execution of software (e.g., of an application) between two **worlds** or *realms*: the secure world (inside the TEE) and the normal world (the main processor). The (portion of the) software running inside the secure world is usually referred to as **trusted application** or *trustlet* (inspired

[22] Confidential Computing Consortium (https://confidentialcomputing.io/)
[23] GlobalPlatform: Securing the Digital Future (https://globalplatform.org/)
[24] Current Trusted Execution Environment landscape - Red Hat Emerging Technologies (https://next.redhat.com/2019/12/02/current-trusted-execution-environment-landscape/)

by the term “applet”), whereas the software running inside the normal world is referred to as **untrusted application**. Untrusted applications interface with the operating system and can interact with trusted applications only through well-defined interfaces. Trusted applications are generally limited in size, constituting a small Trusted Computing Base (TCB) (§ 18.3). Note that even users cannot access data and code of their own trusted applications — that is, users interact with the TEE as a black box. Finally, the development of software intended for application-level isolated execution typically relies on TEE-specific Software Development Kit (SDK) and APIs;

  - *VM-level isolated execution*: protection at the VM level targets software packaged as VMs (or containers) and protects them from the underlying hypervisor, providing isolation between the VMs and the operating system — and also among the VMs themselves. Consequently, the entire VM constitutes the TCB. Note that users have a certain degree of transparency and visibility on the (data and code running within) their own VMs. Finally, the development of software intended for VM-level isolated execution does not have any particular requirements and does not need to interact with any TEE's specific APIs;

- **secure channels**, which allows for establishing local (e.g., I/O) and remote communication channels for data transmission protected against external attacks (e.g., key-logging, replay, eavesdropping, tampering).

Intuitively, some TEE implementations — see § 12.4.1 — may offer further features (e.g., data sealing, resistance to side-channel attacks, obliviousness).

> **Remark 12.2 — On the practicality of developing software for TEEs**
>
> Developing software for TEEs, especially for those offering application-level isolated execution, often leads to lock in to specific SDKs, technologies, or features, and compatibility issues among software versions. Still, projects offering abstractions over TEEs to simplify development exist, e.g., Open Enclave SDK.[a]
>
> [a] SDK for developing enclaves (https://github.com/openenclave/openenclave)

12.4.1 Trusted Execution Environment Implementations

The authors in [140] report a list of TEE implementations among which we highlight:

- **TrustZone by ARM**,[25] a TEE for ARM processors offering application-level isolated execution especially popular in mobile and IoT devices;

- **Software Guard Extensions (SGX) by Intel**,[26] a TEE for Intel processors offering application-level isolated execution which has been deprecated in client-side processors in 2022 due to its (too) many vulnerabilities and — at the time of writing — is only available in server-side processors;[27]

[25] TrustZone technology for the ARMv8-M architecture Version 2.0 (https://developer.arm.com/documentation/100690/0200/ARM-TrustZone-technology)

[26] Intel Software Guard Extensions (https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html)

[27] Intel SGX deprecated in 11th Gen processors (https://community.intel.com/t5/Intel-

- **Sanctum** [70], a seminal academic prototype open-source TEE for RISC-V processors offering application-level isolated execution and introduced in 2016 as an alternative to proprietary solutions like Intel SGX. Sanctum has seen limited updates or development in recent years[28];
- **Trust Domain Extensions (TDX) by Intel**[29] and **Secure Encrypted Virtualization (SEV) by AMD**,[30] two TEEs for Intel and AMD processors, respectively, offering VM-level isolated execution often employed in cloud computing environments such as those of Azure[31] and Google;[32]
- **Keystone** [123],[33] an open-source TEE for RISC-V processors introduced in 2020 inspired from Sanctum;
- **Sancus** [147],[34] an open-source TEE comprising OpenMSP430 hardware extensions for IoT devices.

TEE implementations may have several re-configurations, extensions, or software abstractions known under different names. Hence, it is recommended to always investigate whether a supposedly new TEE implementation is actually directly built upon on an already existing TEE implementation (e.g., OP-TEE[35] is a popular software TEE that runs on ARM TrustZone).

12.5 Physical Unclonable Functions

A PUF (Figure 12) is a physical circuit that operates based on a challenge-response mechanism, that is, given an input (called **challenge**), a PUF returns a physically defined — and unpredictable a-priori — *digital fingerprint* output (called **response**). The challenge-response pairs are unique to each PUF; in other words, the same challenge produces the same response, while different challenges produce different responses. PUFs are based on unique physical variations occurring naturally during semiconductor manufacturing; this property means that PUFs are easy to manufacture but practically impossible to duplicate. PUFs allow for generating and storing keys — in the sense that PUFs, given the same challenge, can re-generate the same key. However, it is not possible to import cryptographic keys into PUFs.

Software-Guard-Extensions/Intel-SGX-deprecated-in-11th-Gen-processors/m-p/1351848)

[28] The MIT Sanctum processor top-level project (https://github.com/ilebedev/sanctum)

[29] Intel Trust Domain Extensions (Intel TDX)(https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html)

[30] AMD Secure Encrypted Virtualization (SEV) | AMD (https://www.amd.com/en/developer/sev.html)

[31] Azure Confidential VM options | Microsoft Learn (https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-machine-options)

[32] Confidential VM overview | Google Cloud (https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview)

[33] Keystone Enclave (QEMU + HiFive Unleashed) (https://github.com/keystone-enclave/keystone)

[34] Minimal OpenMSP430 hardware extensions for isolation and attestation (https://github.com/sancus-tee/sancus-core)

[35] About OP-TEE — OP-TEE documentation (https://optee.readthedocs.io/en/latest/general/about.html)

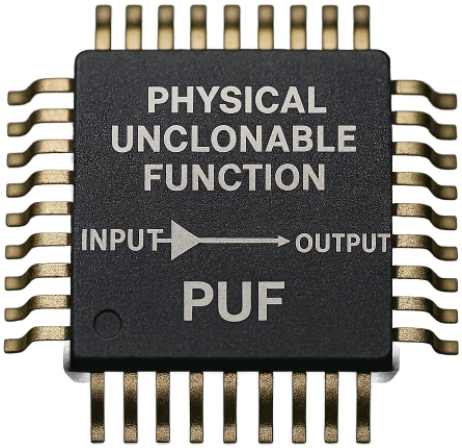

Figure 12: A generic Physical Unclonable Function (PUF) (image generated with OpenAI's Sora — `https://sora.chatgpt.com/`).

12.6 Hardware- vs. Software-Based Cryptographic Modules

The choice between hardware- and software-based cryptographic modules entirely depends on the underlying scenario (e.g., availability of hardware devices) and its requirements (e.g., usability vs. security). Nevertheless, it is still possible to provide a high-level comparison between the two kinds of cryptographic modules.

> **Remark 12.3 — Cryptographic modules security standards**
>
> There exist several standards for categorizing and certifying the protection levels — in terms of security and reliability — of both hardware- and software-based cryptographic modules (which are commonly certified with a combination of such standards): please refer to §15 for more details.

Table 5: Properties of Hardware-based Cryptographic Modules: a full, half, and empty circle (i.e., ●, ◐, ○) mean that the corresponding property is supported by the majority of the implementations, some of the implementations, or none of the implementations of the relative hardware-based cryptographic module, respectively. We define as **Portable** the property of a hardware-based cryptographic module of being easily removable, carried, and used in multiple devices, whereas **Key Usage** means that cryptographic computations (e.g., encryption) take place entirely within the module

Property	HSM	SE	TPM	TEE	PUF
Tamper Responsive	●	○	○	○	○
Portable	●	◐	○	○	○
Key Generation	●	●	●	●	●
Key Storage	●	●	●	●	◐
Key Usage	●	●	●	●	○
Key Export	●	○	●	●	○
Dedicated Processors	●	○	●	●	○
Run Arbitrary Code	◐	○	○	●	○

First, we note that managing sensitive (cryptographic) data and applications in a dedicated hardware environment provides high(er) performance — thanks to the use of dedicated cryptoprocessors and circuits — and, especially, high(er) security. Indeed, according to National Institute of Standards and Technology (NIST)'s security levels definition in [141], **software-based cryptographic modules can reach up to security level 2**, while **hardware-based cryptographic modules reach up to security level 4**. Moreover, TRNGs require the use of ad hoc appliances which are seldom available on the hosts running software-based cryptographic modules (§ 19.5.4). On the other hand, software-based cryptographic modules are highly portable, do not require (expensive) dedicated hardware, and are easier to develop and update.
As for disadvantages, hardware-based cryptographic modules (e.g., PUFs) may be a SPoF and, in general, it may be difficult to achieve redundancy of cryptographic material contained in hardware-based cryptographic modules without invalidating their security properties (e.g., as said in § 12.1, exporting keys from a HSM reduced the security of the keys). Hardware-based cryptographic modules are also opaque, meaning that it is usually impossible to inspect or monitor cryptographic computations executed within. Moreover, the intrinsic resilience to external modifications makes hardware-based cryptographic modules difficult to update in case vulnerabilities are found — which is not infrequent. Finally, it is important to note that, although manufacturers can release patches and fix vulnerabilities, often already existing hardware cannot be updated.

Remark 12.4 — On the (in)security of hardware-based cryptographic modules

Hardware-based cryptographic modules are not bullet-proof and have been successfully attacked several times and in very different ways: SEs are particularly exposed to side-channel attacks (§ 17.1), TEEs are often subject — besides to side-channel attacks — also to software-based and (micro)architectural attacks (see [140] for a nice survey), and HSMs has been vulnerable to firmware tampering and secret extraction attacks.[a] The history of the (in)security of hardware-based cryptographic modules does not mean that they are useless, but rather that their adoption should occur in full awareness of their limits and require a thorough risk assessment.[b]

[a] Black Hat USA 2019 | Briefings Schedule (https://www.blackhat.com/us-19/briefings/schedule/#everybody-be-cool-this-is-a-robbery-16233)

[b] risk assessment - Glossary | CSRC (https://csrc.nist.gov/glossary/term/risk_assessment)

13 KEY MANAGEMENT

Cryptographic protections such as symmetric encryption (§6) and digital signatures (§8) are more likely to be bypassed than cracked. In fact, as Kerckhoffs's principle entails (§1), the problem of protecting data and services in the realm of applied cryptography often reduces to the problem of guaranteeing **the secure management of keys** throughout their lifecycle.

> **Suggested Reading 13.1 — More information on key management**
>
> The NIST's series of special publications on key management [25, 26, 29] are an excellent — albeit complex and long — source of more information on key management. Note that a non-negligible part of the content below is taken from such special publications.

13.1 Key Management Lifecycle

The key management lifecycle considers 4 main key states (Figure 13): **pre-operational** (when keys are not available for use yet), **operational** (when keys are available for use), **post-operational** (when keys are available for special purposes only), and **destroyed** (when keys are no longer available). Each key state is then composed of **one or more phases**, such as generation, usage, revocation, and destruction of keys. Below, we present each of these phases, describing inherent protocols, algorithms, and best practices.

> **Remark 13.1 — Applicability of key management lifecycle in Figure 13**
>
> Intuitively, not all of the key states and phases described below may be applicable or relevant to all systems. Similarly, different key management lifecycles designed for different systems may consider different key states and phases (e.g.., key suspension) [132].

13.1.1 Pre-Operational Key State

In this key state, a key has been generated but is not available for use yet. Every newly created key is in the pre-operational key state by default.

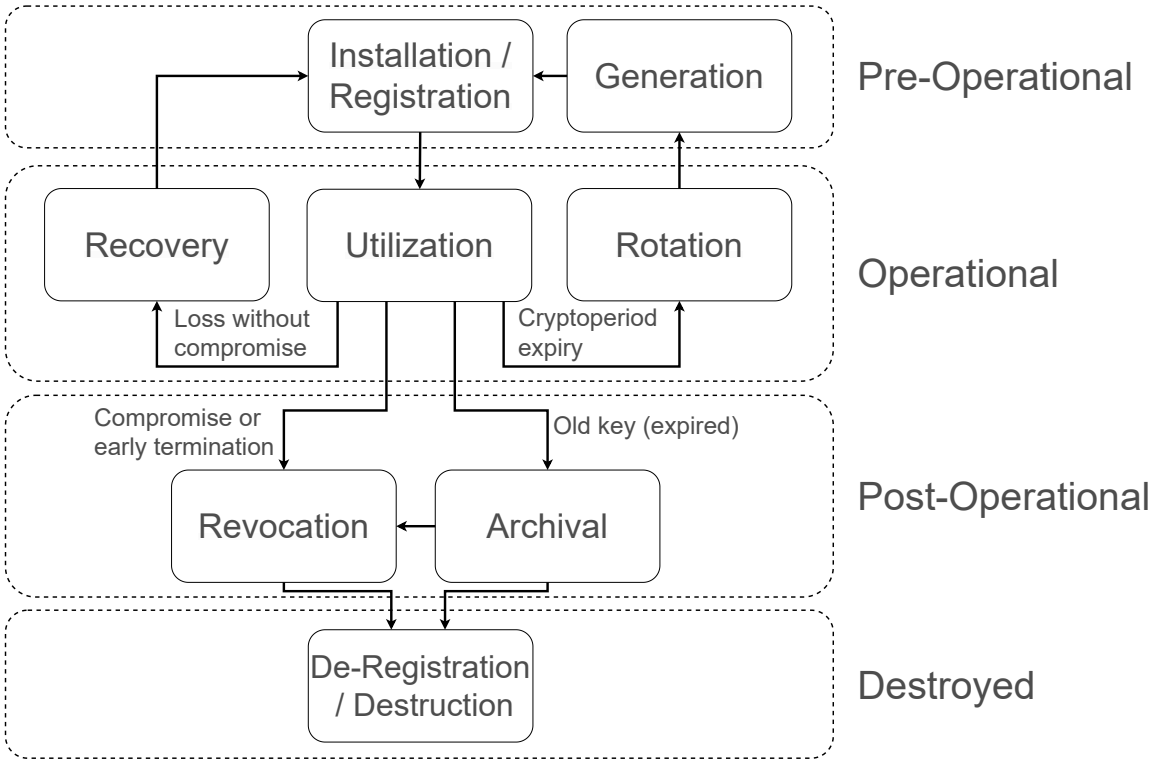

Figure 13: Key states and phases in the key management lifecycle

> **Remark 13.2 — Entity registration and initialization**
> Some key management lifecycles (e.g., [132]) may consider two additional phases in the pre-operational key state, i.e., **entity registration** and **entity initialization** — in this case, "entity" is the party to whom the key is associated. During registration, the entity's information is inserted in the system; for physical users, this phase usually correspond to the creation of digital identities [178], while for organizations this phase is usually a prerequisite for acquiring digital certificates [106] (§14). The initialization of entities, instead, corresponds to the installation and configuration of required cryptographic appliances (e.g., software applications, ciphers, and hardware-based cryptographic modules).

Key Generation. The generation of keys — either done locally for personal use or by a Key Generation Authority (KGA) on behalf of other parties — marks the beginning of key management. **Key generation** usually consists of a protocol whereby a key is created or derived by one or more interacting parties using a generic Key Derivation Function (KDF). At a high level, a KDF is an algorithm taking one or more inputs (where some of the inputs are usually secret) and returning as output a key. Common inputs to KDF are the output of Random Number Generator (RNG), other keys, and secrets such as passwords — in the latter case, we talk about PBKDF (which we describe below). Either directly or indirectly, each of these inputs includes a (more or less good) entropy source (§ 19.5.4). KDFs often involve the use of (keyed cryptographic) hash functions (§5).

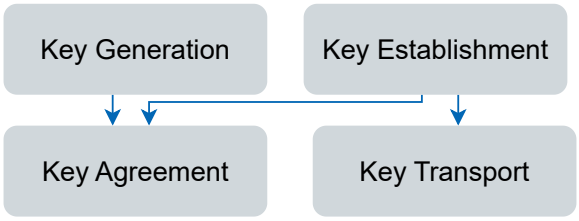

Figure 14: Key generation relationship graph [8]

Remark 13.3 — On the output of KDFs

Depending on the underlying cipher, the output of a KDF may either be immediately usable or need further processing (to, e.g., satisfy specific requirements of the cipher to avoid weak keys — see § 1.1). While symmetric ciphers usually present no such requirements, asymmetric ciphers may require their keys to possess specific properties (e.g., the Rivest-Shamir-Adleman (RSA) cipher requires the generation of two large primes — see § 7.1.1).

Detailed information about the context of the generation of keys (e.g., parameters, intended purpose) should be properly logged — especially for long-term keys. Protocols for key generation may be categorized as follows (Figure 14):

- **key generation**: a protocol whereby a key is created or derived using a generic KDF. The most common key generation protocols include:
  - the use of dedicated procedures to generate keys for specific algorithms. Such dedicated procedures may be implemented in software or by ad-hoc hardware circuits (§12);
  - the (possibly hierarchical) derivation of a key from another key;
  - the use of **mnemonic phrases**, that is, randomly generated sequences of dictionary words that form phrases that can be used as entropy source;

    Example 13.1 — A real-world example of mnemonic phrases usage

    BIP39[a] — developed in the context of Bitcoin — uses mnemonic phrases (also called *seed phrases*) of 12, 18, or 24 words that serves as a human-readable representation of random seeds used to generate private keys.

    [a] iancoleman/bip39: A web tool for converting BIP39 mnemonic codes (https://github.com/iancoleman/bip39)

  - the derivation of a (usually symmetric) key from a password or passphrase via a PBKDF;
- **key establishment**: a protocol whereby a key becomes available to two or more parties;
- **key agreement**: an instance of key generation or key establishment, it is a protocol whereby a key is derived from a function (i.e., a KDF) of information contributed by (or associated with) each of the involved parties, such that no party (or subset of parties) alone can predetermine the key while the other parties cannot — an example of key agreement is Diffie-Hellman (DH) (§9);

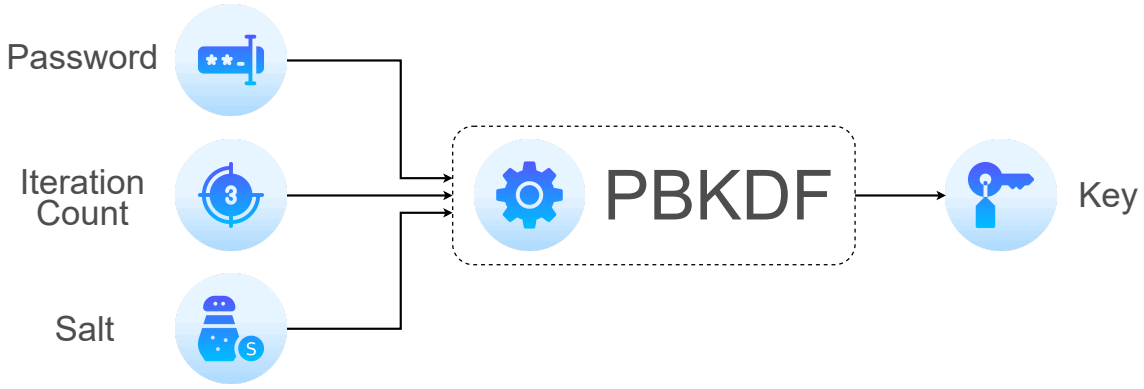

Figure 15: Generic Password-Based Key Derivation Function

- **key transport**: an instance of key establishment, it is a protocol whereby one party possessing a key — usually a KGA — securely transfers the key to other parties — an example of key transport are Key Encapsulation Mechanisms (KEMs) (§10).

> **Remark 13.4 — Mnemonic phrases vs. passwords and passphrases**
> Apart from being sequences of characters — including letters, digits, and possibly other symbols — mnemonic phrases are a different concept that serves a different purpose with respect to passwords and passphrases. A mnemonic phrase represents a random seed for generating private keys, while a password (or a passphrase) is a word (or sequence of words) kept secret by a user. Passwords are sometimes called *passcodes*, and a password including digits only is usually called *Personal Identification Number (PIN)*. Passphrases are longer than passwords, hence more secure.

PBKDFs are a specific instance of KDF where one of the inputs is a password (or passphrase). PBKDFs are more than simply applying generic Cryptographic Hash Functions (CHFs) to the password and use the resulting digest as a key — otherwise (*i*) they would always return the same key for the same password, (*ii*) attackers may guess (at least trivial) passwords easily using rainbow tables or performing brute force or dictionary attacks [183] (§ 16.3), and (*iii*) generic CHF are efficient, hence they would speed up the aforementioned attacks.

In fact, modern PBKDFs usually consider at least 3 inputs (Figure 15): a password, an iteration count, and a salt. The iteration count specifies how many times the CHF is applied — to make brute force attacks more time-consuming and resource-intensive — while the salt is a (non-secret and unique) random sequence of bytes that is combined (often concatenated) with the password before being hashed — to thwart pre-image attacks (§5). Also, PBKDF are not designed to be efficient. Recommended values for iteration count and salt may vary depending on the specific PBKDF [183].

> **Example 13.2 — Real-world examples of PBKDFs**
> BCRYPT [153], Argon2 [46], and PBKDF2 [108] are modern or widely used PBKDF.

> **Warning 13.1 — Limitations of PBKDF**
> One of the main drawbacks of using PBKDFs is that passwords tend to have low entropy. It has been estimated that a password of 8 characters including an uppercase and lowercase letter, a symbol,

and a digit has 33 bits of entropy, while a random sequence of 8 characters has 52 bits of entropy [110].

Suggested Reading 13.2 — More information on key generation

The NIST's special publication on key generation [28] is an excellent source of more information on key generation.

Key Installation and Registration. After generation, cryptographic material (e.g., keys, digital certificates) needs to be installed onto the devices from which it will be used *entity-side* — along with suitable metadata; the installation may be either manual (e.g., transfer from disk, read-only device, hardware token) or automated — granted a secure protocol is executed. Contextually, cryptographic material should also be registered *system-side* (e.g., registering a newly issued digital certificate). Concretely, key registration implies the recording of the related metadata. Whenever possible, the binding of cryptographic material with entities should be cryptographic (e.g., using a Public Key Infrastructure (PKI)). Note that ephemeral keys and short-lived keys (§ 1.1)) do not need to be registered — although it is usually recommended to log relevant related information.

13.1.2 Operational Key State

In this key state, a key is available for use for the intended purpose. Keys transition in the operational key state either directly following generation — if the keys were created for immediate use and the corresponding metadata were correctly associated — or upon reaching an activation date (e.g., the `notBefore` date in an X.509 digital certificate for a public-private key pair in a PKI [53] — see § 14.1). In both cases, this transition marks the beginning of the corresponding cryptoperiod.

Key Utilization. When in the operational key state, a key is available for use and may be either at rest, in transit, or in use; note that a key in use may also simultaneously be in transit and at rest [25]:

- **key at rest**: stored keys should be protected either by specific hardware-based cryptographic modules (§12) or — when managed by general purpose hardware (e.g., a laptop) — by software-based keystores, procedural controls, and physical isolation (e.g., secured rooms with isolated equipment). Intuitively, relying on hardware-based cryptographic modules is usually recommended, as it is the usage of key wrapping (§ 1.1) to preserve the confidentiality and integrity of keys. Furthermore, Data Encryption Keys (DEKs) should be stored in a separate location from the associated encrypted data. Finally, the availability of stored keys — particularly important for long-term keys — may be guaranteed through backups when possible (the aforementioned recommendations for secure storage of keys apply to their backups as well). A backup always refers to short-term storage during operational use and is usually implemented by duplicating keys;

  Warning 13.2 — Do not backup long-term private keys for signatures

  As an exception, the backup of private keys used for signatures is generally discouraged by NIST due to possibly loss of non-repudiation [25].

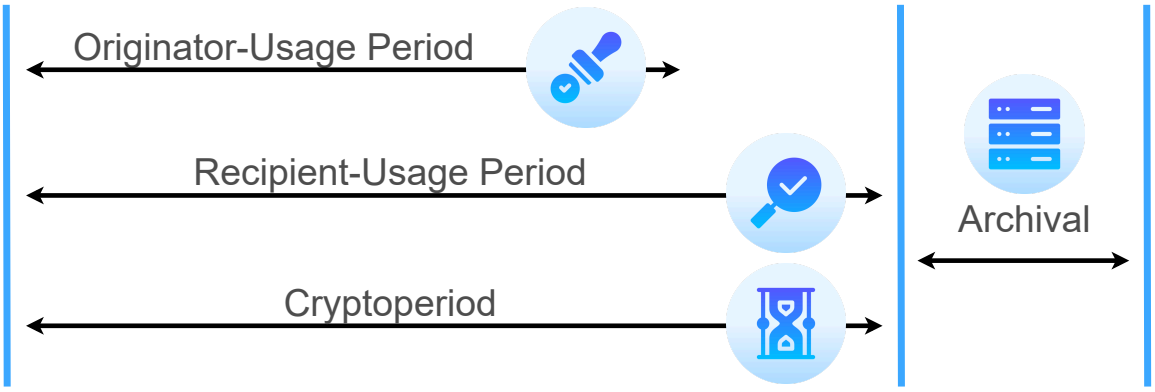

Figure 16: Cryptoperiod and Archival in the Key Management Lifecycle

- **key in transit**: a key may be in transit due to a key transport protocol — hence, the key is being distributed, either manually or automatically, to authorized parties (§ 13.1.1) — or due to backup or archival (§ 13.1.3). In any case, key wrapping should be applied to keys in transit — even if secure communication protocols such as Transport Layer Security (TLS) (§ 19.3.1) are used;
- **key in use**: the use of keys should occur within (possibly hardware-based) cryptographic modules only and be subject to AC policies (§ 19.5.8). Furthermore, it is a best practice to implement monitoring mechanisms to track and log key usage and to regularly review logs to detect unauthorized or suspicious activities.

> **Remark 13.5 — On keys splitting**
> Keys at rest, in transit, or in use may be split to improve security and reduce risks of compromise using Threshold Secret Sharing (TSS) or Secure Multi Party Computation (SMPC) schemes — see §§ 19.2.8 and 19.5.3, respectively. Nonetheless, each share of a key should anyway be protected as just described.

All keys have a **cryptoperiod** after which the keys *expire* and should not be used anymore — except for special purposes. Indeed, over time, keys may be leaked, compromised, or found vulnerable (the keys themselves or their ciphers); defining a cryptoperiod allows for preventing and mitigating these risks and the corresponding security threats. A cryptoperiod may be either temporal (e.g., number of days) or based on the amount of data processed (e.g., encrypted) to limit the amount of information available for cryptanalysis. As shown in Figure 16, a cryptoperiod typically comprises an **originator-usage period** and a **recipient-usage period**: the former is the period during which keys can apply cryptographic protections (e.g., encryption, creation of digital signatures), while the latter is the period during which keys can process cryptographic protections. The recipient-usage period is usually equivalent to the cryptoperiod, while the originator-usage period is often shorter.

> **Remark 13.6 — How to compute cryptoperiods**
> There is no (widely adopted) formula for calculating cryptoperiods exactly, as these depend on several (often subjective) factors such as the sensitivity of the data, the risk of attacks, the cost of generating and distributing new keys (and, possibly, re-encrypting data), the strength of the

chosen cipher, the methodology for storing keys, and the size of keys; for more details and some examples, we refer the interested reader to [25].

Key Rotation. Keys may need to be rotated due to (presumed or real) leakages and cryptoperiods. When new keys are generated independently of old keys, we talk about **re-keying**; **key update** otherwise. Re-keying is often computationally more intensive than key update; for instance, re-keying a DEK implies re-encrypting the associated data — either immediately (i.e., *eager re-encryption*) or the first time the data needs to be decrypted (i.e., *lazy re-encryption*). Intuitively, lazy re-encryption does not apply to compromised keys. However, key update entails a higher security risk; for instance, if an attacker compromises a key, knowing the update process, keys derived from the compromised key could easily be determined [25]. In other words, key update is strongly discouraged. In any case, key rotation should be synchronized and happen simultaneously for all instances of the keys being rotated.

Remark 13.7 — Key rotation and PBKDF

When a key that has to be rotated was generated through a PBKDFs, it is recommended to also change the corresponding password.

Key Recovery. Keys may be lost or become unusable due to, e.g., accidental or intentional modifications and deletions, hardware damage, or corruption. Under the assumption that a lost or unusable key was not compromised, that key may be recovered if deemed necessary either through backups or archives (§§ 13.1.2 and 13.1.3). Even a compromised key may be recovered, but only for processing (previously existing) cryptographic protections after a careful evaluation of inherent risks.

Suggested Reading 13.3 — More information on key recovery

Appendix B of the NIST's special publication on key management [25] contains further information on key recovery.

13.1.3 Post-Operational Key State

In this key state, a key should be available for use for special purposes only. In particular, keys in this state should not be used for applying cryptographic protections, but only for processing (previously existing) cryptographic protections — if strictly necessary. For instance, a public key may be used to verify a digital signature created before the cryptoperiod of the corresponding private key ended to settle disputes involving repudiation. Similarly, a KEK may be used to allow re-encrypting wrapped keys. In general, the use of keys in the post-operational key state should aim at processing cryptographic protections and renewing them with new keys. Keys transition in the post-operational key state when expired (§ 13.1.2) — hence, when they need to be archived — or compromised — hence, when they need to be revoked.

Key Archival. As shown in Figure 16, when (the associated cryptoperiod) expired and not suspected to have been compromised, keys may be archived to be used for the aforementioned special purposes. Archival always refers to long-term storage where post-operational keys — together with the associated

metadata — are stored offline with the appropriate protections (e.g., key wrapping, AC policies). Note that archives may (and should) have backups.

Key Revocation. A key may be revoked mainly for 2 reasons: (*i*) the key — in any key state of phase — is suspected to have been compromised or (*ii*) the authorized use of the key is terminated before the end of its cryptoperiod (e.g., because the owner of the key has left the system, or the device containing the key has been removed from service) [25]. In both cases, a notification that a key has been revoked should be sent to all entities that have been, are, or would be using the revoked key. The notification should contain at least the identifier of the key, the timestamp, and the motivation for the revocation. The latter is particularly useful to allow recipient entities to determine how to behave toward the revoked key.

> **Warning 13.3 — Revocation of a long-term private key for signatures**
> If a public key used for verifying signatures is revoked because its owner left the system, signatures created before the revocation can still be considered trustworthy, hence valid. If the same public key is revoked because of suspected compromise, then there is the need to carefully assess whether to trust previously created signatures; this assessment — and possible consequent signature verification operations — should be conducted in a highly controlled environment with a clear image of the possible consequences.

A key revoked due to compromise is usually rotated through re-keying (§ 13.1.2). Finally, the way notifications are broadcasted depends on the key and the underlying system; for instance, Certificate Revocation Lists (CRLs) would be used in the context of a PKI for asymmetric keys, while a Compromised Key List (CKL) may be used for symmetric keys [29].

13.1.4 Destroyed Key State

In this key state, a key is no longer available for use. Keys transition in the destroyed key state — sometimes also called **obsolete** — when created but never used (i.e., directly from the pre-operational phase) or when no longer used for the special purposes considered in the post-operational key state (i.e., for processing previously existing cryptographic protections) or after having being revoked.

Key De-Registration and Destruction. When in the destroyed key state, keys need to be de-registered and then destroyed. The de-registration of a key implies marking all related records to indicate that the key was destroyed — contrary to what one may think, these records should not be deleted, as they may be useful for audit purposes. The destruction of a key implies that all copies of the key — both in backups and archives — are destroyed, that is, securely erased so that they cannot be recovered by either physical or electronic means. Key destruction strictly applies to private and secret keys, while public keys may be either retained or destroyed. Finally, a compromise of a destroyed key could be discovered after destruction: in this case, the event should be recorded [31].

> **Remark 13.8 — Entity de-registration**
> Similarly to key, entities should be de-registered when no longer involved in the system. Records related to a de-registered entity should be properly marked but not deleted.

14 BINDING KEYS TO PARTIES

Binding key to parties essentially requires trust, whose establishment and management may vary according to the underlying scenario and involves several aspects such as preliminary assumptions, policies and responsibilities, legal agreements and rights, and further human factors. The main **trust models** which define the establishment and management of trust for binding key to parties are the following:

- **PKI** — also known as *hierarchical trust model* (§ 14.3);
- **Web of Trust (WoT)** (§ 14.2);
- blockchain-based (§ 19.4.1);
- hybrid models combining elements of the aforementioned trust models.

14.1 Digital Certificates

All trust models encode and validate trust relationships through some sort of digital certificates. In general, a **digital certificate** (or simply certificate) is an electronic document cryptographically binding a key pair with a unique identity (or pseudo-identity) of a party (e.g., users, services and agents in general). A certificate usually contains the public key, the unique identity of the party, further attributes of the party, and a digital signature (§8) of the certificate itself. Such a digital signature may have been created with the private key corresponding to the public key contained in the certificate — in which case, the certificate is said to be **self-signed** — or with the private key of a (often widely trusted or authoritative) third party typically called **Certificate Authority (CA)**. A certificate usually has a start and an end time for its validity — often (but not necessarily) corresponding to the cryptoperiod of the key pair (§ 13.1.2) but can even be revoked in advance (§ 14.4).

> **⚠ Warning 14.1 — Verifying possession of private keys**
>
> A certificate binding a party to a public key allows the party to claim control of the corresponding private key. Therefore, the procedure for releasing a certificate needs to include a verification step, that is, ensure that the recipient party actually controls the private key. Obviously, the lack of such

a verification step invalidates any authenticity claims.

14.1.1 Validation of Digital Certificates

The validation of a certificate is a complex procedure, but roughly involves (at least) the following steps:

1. check the digital signature of the certificate;
2. compare the validity start and end time of the certificate against the current time;
3. ensure the certificate has not been revoked — this step may vary depending on the revocation procedure employed (§ 14.4);
4. establish a transitive trust path linking the certificate to a predefined **trust root** — this step surely varies depending on the chosen trust model. This trust path is often called **trust chain** (especially in PKI);
5. consider policy and application-level checks (e.g., the certificate may be valid but not properly formatted, the policy does not accept it due to the use of insecure chosen ciphers, the attributes contained in the certificate do not match the intended usage).

Remark 14.1 — More on the establishment of trust paths

One of the steps to validate a certificate is establishing a trust path: this step is not only cryptographic but also concerns trust assumptions and trust endorsements. In fact, a party who wants to validate a certificate (called **validator**) ultimately needs to have assumed trust on other parties in the real world — that is, having defined one or more trust roots. **Without any prior real-world trust assumptions, is it impossible to have trust**. A trust root may either be another party (e.g., a user) or an organization (i.e., CAs), and the definition of a trust root varies depending on the underlying trust model. Also, each party may have its own trust roots. Consequently, the establishment of a trust path is the (often recursive or iterative) procedure through which the validator ensures that the certificate being validate is (either directly or indirectly) endorsed by one or more (or enough) trust roots.

Warning 14.2 — Misinterpreting certificates

A valid certificate does not inherently imply that the owner of the certificate is trusted, but only that some trust root endorsed the key pair and the attributes contained therein. Moreover, these attributes are often misinterpreted, yielding to misplace trust. For instance, assuming that a certificate for a website guarantees the honesty and reliability of the website owner is a common misconception. In fact, scam websites often exhibit certificates whose attributes (e.g., website name and url) resemble those of the original websites (a practice called **domain name mismatch**). In this sense, an `https` connection only guarantees the security of the communication, not the authenticity or trustworthiness of the website owner.

14.2 Public Key Infrastructure

The PKI is a (mostly) centralized trust model comprising hardware, software, parties, policies, documents and workflows supporting the management — in terms of creation, issuance, distribution, usage, storage, and revocation — of digital certificates.

> **Remark 14.2 — Certificates in PKI**
> The most widely adopted format for certificates in PKIs is the X.509 standard [53] — see https://www.itu.int/rec/T-REC-X.509 — a very complex format that yielded several implementation errors in the past (e.g., see https://www.imperialviolet.org/2014/02/22/applebug.html.)

14.2.1 Registration Authorities

The creation of a party's certificate in a PKI is subject to the verification of the party's possession of the private key (§ 14.1) — as well as the correctness of further attributes that will added to the certificate — by a **Registration Authority (RA)**. Such a verification may be manual or automated, more or less rigorous, take more or less time, and be more or less expensive depending on the level of assurance required for the certificate. For instance, industry guidelines commonly distinguish among extended validation certificate (high assurance), organization validation certificate (medium assurance), and domain validation certificate (low assurance).

> **Remark 14.3 — Domain validation certificates in the real world**
> RFC8555 [155] specifies the Automatic Certificate Management Environment (ACME) for the automatic provisioning of domain validation certificates for websites, and the Let's Encrypt nonprofit organization implements ACME for free use — see https://letsencrypt.org/

14.2.2 Certificate Authorities

The actual creation and issuing of certificates in a PKI is carried out by **CAs** — usually a well-known organization or governing body. A CA may possess one or more key pairs which may (or may not) be endorsed by other CAs. We can distinguish among the following types of CAs:

- **subordinate CA**: a CA using its key pair to issues certificates to parties. The key pair of a subordinate CA is endorsed by an intermediate or root CA;
- **intermediate CA**: a CA using its key pair to issues certificates to intermediate or subordinate CAs. The key pair of an intermediate CA is endorsed by an intermediate or root CA;
- **root CA**: a CA using its key pair to issues certificates to intermediate or subordinate CAs. The key pair of a root CA is not endorsed by any other CA.

> **Remark 14.4 — Certificates of root CAs**
> Certificates of root CAs are normal certificates which are pre-installed in operating systems and browsers as trust roots (see https://certifi.io/). Although it is usually the case, these certificates do not need (and sometimes are not) self-signed. However, it may happen that the certificate of a root CA is signed by another root CA — usually for migration reasons.

Typically, a PKI has few root CAs and many intermediate and subordinate CAs for added security and scalability.

Suggested Reading 14.1 — An initiative against maliciously or mistakenly issued certificates

In the past, it happened that root CAs were compromised or mistakenly issued certificates to "wrong" parties. Hence, a consortium of organizations (e.g., Google, Apple, Cloudflare) launched the *Certificate Transparency* ecosystem to make the issuance of website certificates transparent and verifiable and monitor and audit existing certificates through several independent logs — see `https://certificate.transparency.dev/`.

14.2.3 Validation Authorities

The Validation Authority (VA) in a PKI is responsible for answering inquiries by parties about the real time validity of certificates.

Remark 14.5 — Centralized vs. decentralized PKI

Albeit inherently hierarchical, a PKI may be more or less centralized depending on the number of root CAs and their relationships. In general, a more centralized PKI — which is usually the case — simplifies certificate management and has a clear trust chain, at the cost of introducing SPoFs, low scalability, and limited flexibility. Conversely, a more decentralized PKI has few or none SPoFs and great scalability and flexibility, at the cost of greater complexity and consequently security concerns.

14.3 Web of Trust

The WoT is a cumulative, decentralized, and transitive trust model where anyone can verify and endorse the association between public keys and parties — similarly to a reputation system. The WoT enables a decentralized network of trust to grow organically, relying on overlapping trust relationships rather than centralized parties.

Remark 14.6 — Certificates in WoT

The most widely adopted format for certificates in WoT is the Pretty Good Privacy (PGP) certificate format — see `https://www.ietf.org/archive/id/draft-bre-openpgp-samples-01.html`.

Remark 14.7 — Indirect trust in the WoT

Being a transitive trust model, in the WoT a party `A` may trust another party `C` either directly — by issuing a PGP-formatted digital certificate for the key pair of the party `C` as normally — or indirectly. In the latter case, the party `A` trusts the party `C` if there exists a chain of trust between the party `A` and the party `C`. In other words, trust in one party extends to others parties based on a chain of endorsements. For example, if the party `A` directly trusts a party `B`, and the party **B** directly trusts the party `C`, then the party `A` indirectly trusts the party `C` — even without direct interaction between the party `A` and the party `C`. Intuitively, parties can set thresholds on the length of the chain of trust they accept.

A comparison between PKI and WoT is provided in Figure 17

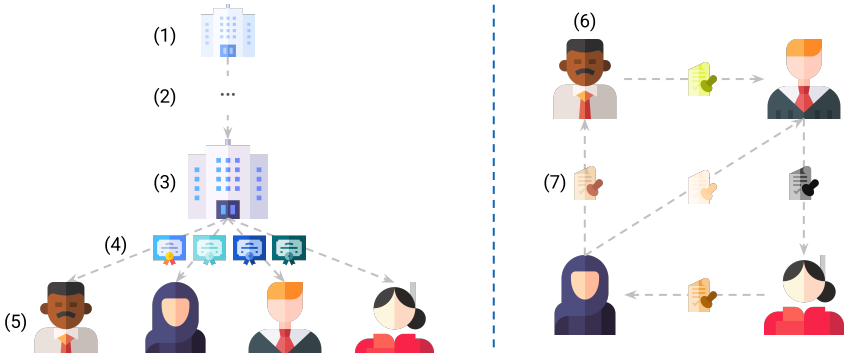

Figure 17: PKI vs. WoT. In a PKI, one or more root CAs (1) issue X.509-formatted digital certificates to other CAs. Subordinate CAs (3) issue digital certificates (4) only to parties (5) — a party may represent an entire organization, a part of an organization, or, seldom, a single user. Between root CA and subordinate CAs, there may be zero or more intermediate CAs (2). The key pairs of an intermediate CA are endorsed by root or other intermediate CAs. Usually, an intermediate CA issues digital certificates only to intermediate or subordinate CAs and not to parties. Conversely, in WoT a party (6) receives PGP-formatted digital certificates by other parties.

14.3.1 Pretty Good Privacy

PGP is both a suite of cryptographic ciphers (mainly for encryption and creation of digital signatures) and the name of their original implementation by Phil Zimmermann[36]) in 1991. PGP was later standardized as OpenPGP [149] — see `https://www.openpgp.org/` — and its reference implementation is GnuPG — see `https://gnupg.org`. OpenPGP is the main standard supporting WoT — from another point of view, WoT is the trust model used in OpenPGP to establish and manage trust.

> **Warning 14.3 — On the (in)security of PGP**
>
> Some deem PGP to be outdated, badly designed, and with serious vulnerabilities — see `https://latacora.micro.blog/2019/07/16/the-pgp-problem.html` and `https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/`.

14.4 Digital Certificates Revocation

A certificate can be **revoked in advance**; common motivations include the fact that the private key corresponding to the public key contained in the certificate has been compromised (or it is deemed so) and the party to which the key pair is binded has misbehaved or has left the system. Common revocation procedures are:

- (root) CAs make available and periodically update **CRLs** — that is, lists of (the serial numbers of) revoked certificates — that can be downloaded by parties and checked locally. This procedure is simple and widely used in PKIs but also in WoT. Rather than publishing a full CRL, CAs may also publish **delta CRLs** ("delta" in the sense of "difference") which contain incremental changes with respect to a full CRL;

[36] `https://philzimmermann.com/`

- parties make available **Distributed Revocation Lists (DRLs)** across multiple servers to improve efficiency and availability. This procedure is usually employed in WoT;
- the **Online Certificate Status Protocol (OCSP)** [162], in which parties can send the serial number of a certificate to an OCSP responder (usually run by a root CA) which returns a signed answer stating whether the certificate has been revoked in real time. The owner of a certificate may even send the answer of an OCSP responder along the certificate itself — a practice called **OCSP Stapling** — so that parties do not have to query the OCSP responder; of course, the stapled OCSP responder's answer may be valid for a short period only;
- the **Server-based Certificate Validation Protocol (SCVP)** [85], in which parties offload the entire validation of a certificate (including the revocation checking) to an SCVP server;
- CAs issue short-lived certificates with very brief validity (e.g., hours or days), thus eliminating the need for revocation in most cases (although incurring in a frequent re-issuance overhead).

Intuitively, each procedure presents a trade-off between efficiency, scalability, privacy (e.g., OCSP may leak which websites a party is visiting), and security (e.g., CRL has low bandwidth consumption but local CRLs may not always be synchronized, while OCSP requires high bandwidth but is always up to date). In any case, as for the key revocation phase in the key management lifecycle (§ 13.1.3), the information that a certificate has been revoked should include at least the identifier of the certificate, the timestamp, and the motivation for the revocation; the latter is particularly useful to allow other parties to determine how to behave toward the revoked certificate. Finally, in case it is not possible to check whether a certificate has been revoked, the best practice is to consider it revoked (or at least require user intervention).

> **Remark 14.8 — Revocation in cryptography**
>
> The above content strictly concerns revocation of digital certificates only; revocation in cryptography is a much larger and multifaceted topic.

15 STANDARDS AND CERTIFICATIONS

Below, we report a list (of categories) of standards and certifications most relevant to security in applied cryptography:

- **general cryptographic standards**:
  - *Federal Information Processing Standard (FIPS) 140-3 [141]*, which defines security requirements for (software- and) hardware-based cryptographic modules (§12) on 4 increasing levels of security — with level 4 being the highest — including testing for side-channel and fault injection attacks (§ 17.1);
  - *ISO/IEC 15408 (Common Criteria*[37]*) [99]*, which proposes a security assurance framework for evaluating cryptosystems by defining protection profiles and 7 security assurance levels — ranging from EAL1 to EAL7, with EAL7 being the highest — which often address physical attack resilience (e.g., side-channel and fault injection);
  - *ISO/IEC 19790 [101]* which, similarly to FIPS 140-3, concerns security and testing requirements for cryptographic modules covering aspects such as module interfaces, roles, services, authentication, and side-channel and fault injection attacks;
- **cryptographic standards specific to side-channel and fault injection attacks**:
  - *ISO/IEC 17825 [100]*, which discusses testing methods for the mitigation of non-invasive side-channel attacks;
  - *ISO/IEC 20085 [102]*, which evaluates methods for the physical security of cryptographic modules;
- **cryptographic standards specific to data remanence**:
  - *NIST SP 800-88 [112]*, which presents guidelines for media sanitization and specifies methods to prevent data remanence in cryptographic hardware (§ 17.1.3);

[37] Common Criteria (https://www.commoncriteriaportal.org/index.cfm)

– *ISO/IEC 27040 [105]*, which presents storage security standards addressing residual data risks and secure wiping methods.

Furthermore, we highlight that there exist certification programs providing a standardized approach for testing and validating the security of cryptographic algorithm implementations:

- the NIST Cryptographic Algorithm Validation Program (CAVP);[38]
- the EMVCo Security Evaluation program for financial applications[39];
- the PSA Certified;[40]
- the GlobalPlatform certification schemes (e.g., SESIP) [41].

Nation organizations relevant to the topic of standards and certifications of the security of cryptographic algorithm implementation are

- the Agence nationale de sécurité des systèmes d'information (ANNSI)[42] — the French National Cybersecurity Agency;
- the Federal Office for Information Security (BSI)[43] — the German Federal Office for Information Security;
- the National Cybersecurity Agency (ACN)[44] — the Italian National Cybersecurity Agency;
- the NIST[45] and the National Security Agency (NSA)[46] — organizations in the United States.

> **⚠ Warning 15.1 — Trustworthiness of standardization bodies**
>
> Trust in standards and certifications derives from their technical merits as well as the underlying standardization bodies and countries. In other words, always verify which organization authored and reviewed a standard or certification — and remain alert to political or commercial influences [133].

Further organizations relevant to cryptography are

- International Association for Cryptologic Research (IACR)[47] — a non-profit scientific organization for research in cryptology;
- Internet Engineering Task Force (IETF)[48] — a standards development organization for the Internet;

[38] Cryptographic Algorithm Validation Program | CSRC (https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program)
[39] EMVCo Security Evaluation Process (https://www.emvco.com/resources/emvco-security-evaluation-process/)
[40] PSA Certified (https://www.psacertified.org/)
[41] GlobalPlatform Certification Schemes (https://globalplatform.org/globalplatform-certification-schemes/)
[42] French Cybersecurity Agency (ANSSI) (https://cyber.gouv.fr/en)
[43] Federal Office for Information Security (https://www.bsi.bund.de/EN/Home/home_node.html)
[44] ACN (https://www.acn.gov.it)
[45] National Institute of Standards and Technology (https://www.nist.gov/)
[46] National Security Agency | Central Security Service (https://www.nsa.gov/)
[47] The International Association for Cryptologic Research (https://www.iacr.org/)
[48] IETF | Internet Engineering Task Force (https://www.ietf.org/)

- European Union Agency for Cybersecurity (ENISA)[49] — the European Union's agency for cybersecurity;
- Open Web Application Security Project (OWASP)[50] — a nonprofit foundation that works to improve the security of software.

Finally, further standards may need to be considered depending on the country and the underlying scenario, e.g., the Payment Card Industry Data Security Standard (PCI-DSS)[51] for financial services, the Health Insurance Portability and Accountability Act (HIPAA)[52] for healthcare services in the US, and the General Data Protection Regulation (GDPR)[53] for privacy in Europe.

Remark 15.1 — On the timeliness of standards and certifications

Standards and certifications take time (i.e., years) to be written and updated, while developments in cryptography are frequent. Hence, be sure to refer to up-to-date standards and — in any case — always check their content is still relevant. Moreover, beware that there may be perception gaps and differences in priorities between the designers and the practitioners of standards [120].

Warning 15.2 — Watch out for conflicts

Standards may sometimes conflict with each other. For instance, as of 2025, IETF RFC 8446 [157] — which defines TLS v1.3 (§ 19.3.1) — considers ChaCha20-Poly1305 (§ 6.4.2) secure, while the NIST does not (yet).

[49] Home | ENISA (https://www.enisa.europa.eu/)
[50] OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation (https://owasp.org/)
[51] PCI Data Security Standard (PCI DSS) (https://www.pcisecuritystandards.org/standards/pci-dss/)
[52] HIPAA Home | HHS.gov (https://www.hhs.gov/hipaa/index.html)
[53] GDPR compliance (https://gdpr.eu/)

Notes on

SECURITY

last revision: 15 Sep 2025

16 SECURITY IN CRYPTOGRAPHY

As discussed in §§ 2 and 3, cryptography is ultimately based on mathematics. Therefore, the theoretical design and logical robustness of cryptographic algorithms and protocols are often scrutinized to reveal any weaknesses or attacks.

However, the **application of cryptography** in real-world scenarios yields security concerns other than only those related to its mathematical underpinnings. Even more, cryptography in general is a necessary but insufficient condition for guaranteeing the security of sensitive data and services [133]. In fact, **(applied) cryptography may break for a variety of reasons** not strictly related to mathematics: implementation errors and vulnerabilities, (unintentionally introduced) backdoors, misconfigurations and misuses, wrong security assumptions or standard definitions, side-channel attacks, and social engineering attacks, to mention a few.

Below, we provide an overview of security in cryptography starting from its most generic aspects: the definition of a threat model (§ 16.1), the consequent choice of attack model (§ 16.2), and a taxonomy of attacks (§ 16.3).

16.1 Threat Modeling

The fundamental step for discussing security concerns within applied cryptography is the definition of a **threat model**.

> **Remark 16.1 — Definition of threat modeling**
>
> Threat modeling is a critical (usually iterative) process common to many cyber-physical systems — not only cryptography. A threat model defines the system under consideration, its security assumptions (e.g., presence of shared public parameters, trusted bulletin boards, or a Public Key Infrastructure (PKI) — see §14), and characteristics and capabilities of attackers in the context of the system, the components thereof, and their interactions. The ultimate objective of a threat model is to **identify potential threats and the associated risk** (e.g., likelihood and impact), **propose and prioritize mitigations** (§§ 17.2 and 17.2.4 and §18), and **verify** that the proposed mitigations do address the identified threats (§ 17.2.4).

The following categorization identifies high-level characteristics and capabilities of attackers in the context of applied cryptography:

- **passive vs. active attackers**: passive attackers eavesdrop on communication or monitor the behavior of systems without altering them or the data container therein, while active attackers can interfere with communications and systems by injecting, modifying, or deleting messages and data, or by manipulating their behavior to achieve their objectives;
- **remote vs. local attackers**: remote attackers operate from a distance and can interact with the system through well-defined interfaces over network communications only (e.g., either internet or intranet), while local attackers stay in close physical proximity to or directly within systems;
- **single-trace vs. multi-trace attackers**: single-trace attackers may rely on a single trace or instance of cryptographic computations to extract secret data, while multi-trace attackers can exploit patterns or correlations over multiple traces or instances to improve their chances of success or refine their results;
- **external vs. internal attackers**: external attackers conduct attacks and impersonate authenticated parties from outside the system, while internal attackers (sometimes called *malicious insiders*) are authenticated or compromised parties that, either maliciously or accidentally, endanger security from within the system.

The latest threat models in applied cryptography have been considering hybrid attackers (i.e., combining multiple capabilities and categories of attacks — see § 16.3) [22] and, especially, attackers relying on machine learning and artificial intelligence consistently [180, 92, 87, 74].

> **Remark 16.2 — Established threat models**
> Renown and widely used threat models are available for analyzing the security of protocols (§19). Conversely, to the best of our knowledge, there are no established threat models for analyzing the security of algorithm implementation — net of the aforementioned categorization (§17).

> **Suggested Reading 16.1 — More information on threat modeling in cryptography**
> A thorough survey on the topic of threat modeling in cryptography is available in [77].

16.2 Attack Models

With a threat model, it is possible to accordingly derive an **attack model** for mathematical and algorithmic attacks describing specific methods or approaches attackers may employ to (usually) retrieve (portions of) private or secret keys through cryptanalysis (§1).

> **Remark 16.3 — Threat models vs. attack models**
> Threat models establish the capabilities and resources of attackers, while attack models operate within the constraints of a given threat model. Hence, depending on the threat model, attack models may be more or less powerful (Figure 18).

Some common attack models are:

- **Ciphertext-Only Attack (COA)**: attackers have access to one or more ciphertexts only and not to the corresponding plaintexts. Brute force attacks or exhaustive key search attacks (§ 16.3) are attacks in the COA attack model;
- **Known-Plaintext Attack (KPA)**: attackers have access to a limited number of ciphertext-plaintext pairs;
- **Chosen-Plaintext Attack (CPA)**: attackers can choose a number of plaintexts for which obtain the corresponding ciphertext. The possibility to choose arbitrary plaintexts allows attackers to explore edge cases and trigger particularly significant behaviors. The CPA attack model entails that the attacker has access to a so-called **encryption oracle**.[54] The **Adaptive Chosen-Plaintext Attack (CPA2)** derives from CPA and extends it by allowing attackers to choose plaintexts adaptively, i.e., attackers can choose plaintexts based on previous ciphertext-plaintext pairs observed. Consequently, CPA is also referred to as **CPA1**;
- **Chosen-Ciphertext Attack (CCA)**: attackers can choose a number of ciphertexts of which receive the corresponding plaintext. The CCA attack model entails that the attacker has access to a so-called **decryption oracle**. The **Adaptive Chosen-Ciphertext Attack (CCA2)** derives from CCA and extends it by allowing attackers to choose ciphertexts adaptively, i.e., attackers can choose ciphertexts based on previous ciphertext-plaintext pairs observed. Consequently, CCA is also referred to as **CCA1**.

> **Example 16.1 — A real-world example of a CCA2 attack**
>
> A practical attack in the CCA2 attack model is the Bleichenbacher attack [51] against the Rivest-Shamir-Adleman (RSA) encryption standard Public Key Cryptography Standards (PKCS)#1 [107] — see also `https://royalholloway.ac.uk/media/9116/gageboyleisg.pdf`.

> **Remark 16.4 — CCA and CPA**
>
> CCA usually includes CPA, although some distinguish the two attack models and refer to their combination as **chosen-plaintext/ciphertext attack**.

> **Remark 16.5 — Attack models for symmetric and asymmetric cryptography**
>
> Attackers cannot create ciphertexts in symmetric cryptography (§6), as symmetric keys are secret. Conversely, attackers can create ciphertexts in asymmetric cryptography (§7), as public keys are known. Hence, symmetric cryptography does not typically require security against attackers operating in the CCA attack model — although the most secure symmetric ciphers do.

[54] The term "oracle" indicates a black-box function available for use (in this case, by the attacker) whose inner functioning is, however, opaque. For example, an encryption oracle allows for encrypting any plaintext, while a decryption oracle allows for decrypting any ciphertext — but not accessing private or secret keys. The term oracle comes from the idea of a mythological oracle, an all-knowing entity accepting questions and replying the truth.

Figure 18: The COA attack model requires the most effort and skills by attackers to be successful; if a system resists an attacker operating in the COA model, then it provides low security. Conversely, the CCA attack model requires the least effort and skills by attackers to be successful; if an system resists an attacker operating in the CCA model, then it provides high security.

16.2.1 Security Goals

Attack models can be paired with (ideally any) security goals. Some common security goals are:

- **indistinguishability (IND)**: consider a ciphertext c obtained by applying an algorithm to one plaintext randomly chosen among two plaintexts $m1$ and $m2$ defined by an attacker. An algorithm achieves the indistinguishability goal if the attacker — given c — cannot determine which plaintext was chosen to produce c with probability significantly better than that of random guessing (intuitively, if an attacker could determine which plaintext was used, it would mean that the ciphertext leaks information about the plaintext);

  > **Remark 16.6 — Naming the combination of an attack model with a security goal**
  >
  > The combination of an attack model with a security goal is usually denoted with the hyphenation of their acronyms (e.g., IND-CPA is the indistinguishability security goal under the CPA attack model).

  > **Example 16.2 — Indistinguishability and attack models**
  >
  > IND-CPA means that the attacker can preliminarily choose a number of plaintexts for which obtain the corresponding ciphertext both before and after executing the algorithm over either $m1$ or $m2$ to obtain c. Similarly, IND-CCA means that the attacker can preliminarily choose a number of ciphertexts for which obtain the corresponding plaintext both before and after executing the algorithm over either $m1$ or $m2$ to obtain c — note that IND-CCA includes IND-CPA, i.e., the attacker can also preliminarily choose a number of plaintexts for which obtain the corresponding ciphertext. IND-CPA2 and IND-CCA2 allows the attacker to adaptively choose plaintexts and ciphertexts, respectively.

- **One-Wayness (OW)**: consider a ciphertext c obtained by applying an algorithm to a plaintext m. An algorithm achieves the one-wayness goal if an attacker — given c — cannot recover m with probability significantly better than that of randomly choosing a plaintext m' among the set of all plaintexts $\mathcal{P}$;
- **Pseudorandomness (PRF)**: consider a ciphertext c obtained by applying an algorithm to a plaintext m chosen by an attacker and a random value r whose length is the same of c. An algorithm achieves the pseudorandomness goal if the attacker — given c and r — cannot distinguish between c and r with probability significantly better than that of random guessing;

> **Remark 16.7 — Use cases for the pseudorandomness goal**
> The pseudorandomness goal may be relevant for encrypted communications in which deniability (§4) and resilience to cryptanalysis (§1) is important, storage systems prioritizing data hiding in disk encryption, and steganography in general (§1).

- **Non-Malleability (NM)**: consider a ciphertext c obtained by applying an algorithm to a plaintext m chosen by an attacker. An algorithm achieves the non-malleability goal if an attacker — given c — cannot produce a different ciphertext c' whose plaintext bears any relation to m.

Security goals may be comparable, hence achieving certain security goals may imply achieving others. For instance, indistinguishability implies one-wayness and non-malleability implies indistinguishability — although this depends on the chosen attack model (e.g., NM-CCA2 is equivalent to IND-CCA2). Also, IND-CPA is equivalent to the semantic security (§ 1.3) and confidentiality (§4) properties, while IND-CCA is equivalent to the non-malleability and integrity properties.

16.3 Attacks Taxonomy

Below, we list the most common categories of attacks within the context of applied cryptography.

Misconfiguration, incorrect usage, and misuse attacks. This category of attacks exploits weaknesses arising from the improper configuration or use of algorithms rather than flaws inherent in the algorithms themselves or their implementation. This category of attacks includes secret data reuse (e.g., using the same key for multiple purposes or reusing initialization vectors or nonces), low entropy in the generation of (supposedly) random values (§ 19.5.4), use of weak keys (§ 1.1), and improper key management (§13).

Incorrect implementation attacks. This category of attacks exploits vulnerabilities — due to bugs in the code or non-adherence to the original theoretical specification — in the implementation of algorithms.

> **Example 16.3 — A real-world example of an incorrect implementation attack**
> An example of such a category of attacks is the key reinstallation attack on WPA2 [186].

Supply chain attacks. This category of attacks aims to introduce vulnerabilities in hardware, software, or third-party components of cryptographic modules before their deployment. These attacks typically involve introducing malicious (and often subtle) modifications to the design and manufacture of hardware and circuits or tampering with firmware or software (e.g., open-source libraries) and their distribution to introduce cryptographic backdoors (§ 18.3).

> **Remark 16.8 — Supply chain attacks to cryptography algorithms**
> The practice of inserting cryptographic backdoors in (usually asymmetric) algorithms or implementations while preserving their functional correctness is called **kleptography** [194].

Human factor attacks. This category of attacks targets individuals rather than weaknesses in algorithms or hardware. This category of attacks includes social engineering (e.g., phishing), coercive attacks (e.g., rubber-hose cryptanalysis), malicious insiders (e.g., evil maid attacks), and also insecure workarounds driven by poor usability.

Quantum attacks. This category of attacks relies on quantum computers to efficiently solve mathematical problems that are assumed to be hard for classical computers (§ 19.5.2).

Guessing attacks. This category of attacks includes approaches based on (sometimes educated) guesses and exhaustive trial-and-error methods. This category of attacks includes **brute force attacks** (also called *exhaustive key search*) — systematically attempting all possible secret or private keys until the correct ones are discovered — **dictionary attacks** — systematically attempting a precompiled list of words, phrases, or common passwords (exploiting users' tendencies to choose easily guessable passwords) until the correct ones are discovered — and **rainbow table attacks** — relying on pre-computed tables caching the outputs of a specific Cryptographic Hash Function (CHF) used for conducting preimage attacks (§ 5.1).

> **Warning 16.1 — Guessing attacks are relevant**
>
> Although guessing attacks may seem trivial, they are a serious threat and should not be underestimated — especially given that the computational power available to attackers keep increasing (see **Moore's law**). Intuitively, the higher the computational power, the faster a guessing attack is. Also, dedicated techniques for improving over random guessing (that is, doing educated guessing) are often available for specific ciphers to reduce the key search space and further accelerate guessing attacks. Remember that cryptography is ultimately based on one-way and trapdoor functions which are **computationally** — **not theoretically** — hard to invert (§3). As a consequence and to mitigate the risk of guessing attacks, standardization bodies such as National Institute of Standards and Technology (NIST) periodically recommend to increase the length of private and secret keys to enlarge the key search space.

Mathematical and algorithmic attacks. This category of attacks aims to break the foundations of algorithms by solving the underlying hard mathematical problems (§2) or trying to find weaknesses through cryptanalysis. This category of attacks includes factorization and discrete-logarithm solvers, differential cryptanalysis — that is, analyzing patterns in outputs for different inputs — and linear cryptanalysis — that is, attempting to find linear approximations of the relationship among plaintexts, ciphertexts, and keys (§ 16.2).

Side-channel attacks. This category of attacks aims to infer secret data such as private keys from side effects caused by the execution of algorithms. This category of attacks includes power analysis attacks and timing attacks (§ 17.1.1).

Fault injection attacks. This category of attacks consists of deliberately introducing faults into a cryptographic module to cause it to behave in unexpected ways, and consequently leak secret data. This category of attacks includes glitching attack (§ 17.1.2).

Data remanence attacks. This category of attacks aims to recover residual physical traces of secret data that remain on a storage medium after use. This category of attacks includes cold boot attacks and forensics recovery (§ 17.1.3).

Protocol attacks. This category of attacks targets the composition and use of algorithms for providing security properties such as confidentiality, integrity, and authenticity (§4) in the context of protocols (§19).

> **Remark 16.9 — White-box cryptography**
>
> Besides the aforementioned categories, a quite recent category of attacks is the so-called **white-box cryptography**, an extreme attack scenario in which the attacker has full unrestricted access to an algorithm implementation. This category of attacks was first proposed in 2002 [67] and has many applications, including Digital Rights Management (DRM) and protection of cryptographic keys in the presence of malware. However, due to the huge power given to the attacker in white-box cryptography, there have been little to no practical results yet.

17 SECURE IMPLEMENTATION OF ALGORITHMS

The proper implementation of cryptographic algorithms plays a crucial role in upholding their security. Indeed, besides bugs leading to incorrect functional behavior, a poorly implemented algorithm may present vulnerabilities and (unintentionally) introduce backdoors that may nullify the security of the whole system.
Below, we further discuss this topic by discussing attacks on the implementation of algorithms (§ 17.1) and best practices to mitigate or even invalidate such attacks (§ 17.2).

17.1 Attacks on the Implementation of Cryptographic Algorithms

We now present those 3 categories of attacks (already mentioned in § 16.3) in which the way an algorithm is implemented is relevant, i.e., side-channel attacks (§ 17.1.1), fault injection attacks (§ 17.1.2), and data remanence attacks (§ 17.1.3).

> **Remark 17.1 — Classification of side-channel attacks**
> Depending on the criteria used to distinguish different categories of attacks, fault injection attacks and data remanence attacks are sometimes seen as a specific instance of side-channel attacks (e.g., see [151, 176, 140]).

17.1.1 Side-Channel Attacks

At a high level, **side-channel attacks** measure side effects — generally referred to as **leakage** — of cryptographic computations to infer (portions of) secret data such as seeds, private keys, and internal states. The term side-channel attack is an **umbrella term** that encompasses several — and possibly fundamentally different — approaches for attacking the implementation of algorithms by gathering (separately or collectively) various kinds of leakages: timing, power consumption, radio and electromagnetic leaks, and even acoustic emissions, to mention a few. Nonetheless, all side-channel attacks are generally carried out by:

1. identifying secret data-dependent cryptographic computations;
2. recording traces caused by such computations;
3. analyzing such traces to recover secret data.

While side-channel attacks are typically passive attacks, some may be invasive in the sense that they require accessing cryptographic modules (§12) and operating on them (e.g., insert a probe on a wire to read electric signals) [4].

> **Remark 17.2 — Signal-to-Noise Ratio (SNR)**
> A central concept in the effectiveness of side-channel attacks is the **SNR**, which quantifies the relationship between useful information (**signal**) and irrelevant data or interference (**noise**) within the monitored leakage. In other words, the SNR reflects how distinguishable the useful information is from the irrelevant data or the interference: a high SNR means the signal is prominent and therefore easy to isolate, while a low SNR means the noise is prominent, posing greater difficulty in isolating the signal. Consequently, side-channel attacks aim to maximize the SNR, while mitigations (§ 17.2) aim to either eliminate or reduce the signal (e.g., by reducing sources of signal or eliminating the relationship between the signal and the secret data, respectively) or increase the noise.

Power Analysis Attacks. This type of side-channel attack relies on measuring the **power consumption** of a cryptographic module during the execution of cryptographic computations. In fact, depending on how algorithms are implemented, secret data (e.g., private keys) may influence power consumption, therefore creating correlations that an attacker may analyze. Intuitively, attackers must have access to the cryptographic module and often be able to control its execution to conduct power analysis attacks [160] — hence, attackers must be local (§ 16.1). At a high level, power analysis attacks may be [89, 88] *simple* or *differential*: in the former case, single-trace local attackers try to directly (and often visually) correlate power consumption with cryptographic computations to reveal secret data, while in the latter case multi-trace local attackers rely on statistical methods (e.g., average) to correlate small variations in power consumption with secret data while collecting multiple measurements for different inputs. Simple power analysis attacks usually require a high SNR and are effective when secret data-dependent behavior in power consumption is apparent, while differential power analysis attacks — by using statistical methods (as well as signal processing and error correction) — are effective even with low SNR.

> **Remark 17.3 — Latest trends for power analysis attacks**
> Starting from the seminal work in [89], several power analysis attacks were discussed in the literature such as the Collide+Power attack [117],[a]. Recently, many power analysis attacks are combining power analysis with machine learning and artificial intelligence [124] and targeting post-quantum algorithms [109] (§ 19.5.2).

[a] https://collidepower.com/

Radio and Electromagnetic Analysis Attacks. This type of side-channel attack relies on capturing radio or electromagnetic emissions from hardware to infer currently ongoing cryptographic

computations and secret data within cryptographic modules. Radio and electromagnetic analysis attacks are strictly related to power analysis attacks, especially because **radio and electromagnetic emissions are a direct consequence of power consumption**. It follows that radio and electromagnetic analysis attacks expect an analogous threat model (i.e., require local attackers), use techniques equivalent to those of power analysis attacks, and are similarly divided between simple and differential. However, radio and electromagnetic analysis attacks are more complex and difficult to carry out, having achieved lower maturity [163]. Conversely, sampling and probing for radio and electromagnetic analysis attacks is easier than for power analysis attacks, as the former requires neither physical access nor depackaging/decapsulation (i.e., exposing the silicon of chips).

Acoustic Analysis Attacks. This type of side-channel attack relies on measuring **ultrasonic sounds and vibrations** emitted from hardware to then infer secret data. Such ultrasonic sounds and vibrations are usually not human-audible and come from internal electronic components (e.g., capacitors and inductors) [86]. Similarly to power, radio, and electromagnetic analysis attacks, acoustic analysis attacks may be simple and differential.

> **Remark 17.4 — Threat model for acoustic analysis attacks**
> Acoustic analysis attacks may not always require attackers to be local when cryptographic modules are operating near attacker-controlled microphones; this is often the case when considering cryptographic modules within laptops.

Thermal Imaging Attacks. This type of side-channel attack relies on the observation of **residual heat and heat patterns** on hardware — usually through thermal imaging cameras capturing infrared images — to then infer secret data. Intuitively, thermal imaging attacks, regardless of whether they are simple or differential, require local attackers. Like other side-channel attacks (i.e., acoustic, radio, and electromagnetic) whose traces are linearly correlated with power consumption, thermal imaging attacks are complex and difficult to carry out, especially if considering the limitations posed by the physical properties of thermal conductivity and capacitance [95]. For this reason, thermal imaging attacks are often carried out in combination with other kinds of side-channel attacks [9].

Optical Analysis Attacks. This type of side-channel attack relies on the phenomena of **photonic emission** from transistors when executing cryptographic computations [84, 166] to then infer secret data. Optical analysis attacks require local attackers and may be simple or differential. In terms of equipment, an optical analysis attack is comparable in price to a power analysis attack [63].

Timing Attacks. This type of side-channel attack relies on monitoring **variations in the execution time** of cryptographic computations to infer secret data [116]. Timing attacks are among the most studies, effective, and impactful side-channel attacks. Even worse, timing attacks — whether simple or differential — can be conducted either locally and even remotely over the internet [56, 73].

> **Example 17.1 — A famous example of a timing attack**

In 1996, Paul Kocher demonstrated how to recover private RSA keys (§ 7.1.1) via a timing attack — see [116]

There are multiple reasons why cryptographic computations — when implemented — may take more or less time depending on secret data:

- *generic operations*: some (especially mathematical) operations may execute faster or slower according to the input data and their implementation, e.g., exponentiations for the Discrete Logarithm (DL) problem (§ 3.3) implemented with the square-and-multiply algorithm and fast-failing comparison functions like `memcmp()`[55] in the C programming language;
- *conditional branches and jumps*: making the execution (of a portion of the implementation) of a cryptographic algorithm dependent on secret data is one of the most common timing attack vulnerabilities. Moreover, to provide more concurrency when extra resources are available, modern processors often speculatively execute past branches and jumps by assuming the corresponding condition will be true. However, **speculative execution** either prevents (by performing operations in advance) or adds delays (by reverting changes) according to whether conditions are actually true. Especially when conditions are tied to secret data, speculative execution may result in timing and memory attacks (since processors may speculatively load secret data into memory before checking the process' permissions);

  Example 17.2 — Real-world speculative execution timing attacks
  Two famous attacks exploiting speculative execution are Spectre [115] and Meltdown [125]. Note that, while being mainly based on speculative execution, Spectre and Meltdown also leverage leakage from cache behavior.

- *memory access patterns*: a large sub-category of timing attacks is based on memory access patterns and, in particular, **cache behavior**. At a high level, cache timing attacks rely on subtle timing variations in cache access patterns (e.g., cache hits and misses) which may ultimately depend on secret data. Intuitively, the way a cryptographic algorithm is implemented may make it more or less susceptible to cache timing attacks.

  Example 17.3 — Real-world memory access patterns timing attacks
  Advanced Encryption Standard (AES) (§ 6.4.1) often uses lookup tables for efficiency. However, memory access patterns for these tables depend on the secret key, hence they may leak secret data as presented in [181, 20, 97].

  There exist many types of cache timing attacks, **all of which require an active attacker**; the main 4 are:

  - `Evict+Time` [148] (and then in [181]): the attacker ensures specific cache lines are evicted (i.e., removed) from the cache. Then, when a cryptographic computation involving secret data is executed, the attacker measures the time taken by the execution of the cryptographic computation and determines — according to the access time — whether the required secret

[55] https://en.cppreference.com/w/c/string/byte/memcmp

data (or instructions) were not in the cache. `Evict+Time` is often used when cache access patterns depend on secret data;

- `Prime+Probe` [148] (and then in [181]): the attacker fills (i.e., primes) the cache with their own data, ensuring the cache lines are in a known state. Therefore, when a cryptographic computation involving secret data is executed, it is likely that some of the attacker's data get evicted. The attacker can then measure the time it takes to access their own data, determining — according to the access time — what data was evicted. `Prime+Probe` is often used on shared caches in multi-core or multi-tenant environments (e.g., cloud computing);
- `Flush+Reload` [193, 97]: the attacker flushes a specific cache line (i.e., removes it from the cache) using instructions like `clflush`. When a cryptographic computation involving secret data is executed, the attacker accesses that specific cache line again, determining — according to the access time — whether that cache line was evicted. We note that `Flush+Reload` is more precise than `Prime+Probe` and it is often used when specific memory accesses correlate with secret data-dependent computations;
- `Flush+Flush` [1]: a more complex yet stealthier variation of the `Flush+Reload` attack where the attacker flushes the cache once, the cryptographic computation involving secret data is executed, and then the attacker flushes the cache again while calculating the time taken to perform the second flush — the rationale being that the time spent in flushing depends on what cache lines has been loaded by the cryptographic computation.

As a final remark, note that timing attacks can be introduced unintentionally with compiler optimizations and Instruction Set Architecture (ISA) hardware characteristics — see §18 for more details.

> **Suggested Reading 17.1 — More practical information on side-channel attacks**
>
> Soatok's blog post "Soatok's Guide to Side-Channel Attacks" and Jean-Philippe Aumasson's GitHub post "Cryptocoding" — available at the links `https://soatok.blog/2020/08/27/soatoks-guide-to-side-channel-attacks/` and `https://github.com/veorq/cryptocoding` — provide a good excursus on side-channel attacks with code examples.

17.1.2 Fault Injection Attacks

At a high level, **fault injection attacks** — firstly introduced in [45] — aim to intentionally introduce hardware faults during the execution of cryptographic computations, usually to infer (portions of) secret data but also to bypass security checks or control flows (e.g., by flipping bits, skipping or repeating operations), cause incorrect data fetch (e.g., from the cache), and understand how a cryptographic module behaves under induced faults [24]. Similarly to side-channel attacks (§ 17.1.1), the term fault injection attack denotes a **category of attacks** encompassing different approaches for introducing faults, e.g, glitching attacks [80], laser and optical fault injection [170], electromagnetic pulses, and temperature variations [66]. Fault injection attacks are carried out by local and active attackers and are obviously invasive (except for some software-based fault injection attacks), and — similarly to many side-channel attacks — can be either simple or differential. Although fault injection attacks do not always require direct contact to introduce faults, depackaging/decapsulation is sometimes needed. Finally, fault injection attacks commonly target hardware-based cryptographic modules like Secure Element (SE) and Trusted Platform Module

(TPM) (and similar embedded systems) by affecting either processors (i.e., computations) or memory (i.e., data).

Glitching Attacks. This type of fault injection attack relies on introducing faults as either **power or voltage glitches** [54] — that is, sudden and short-lived interruptions or fluctuations in the power supply of a circuit — or as **clock glitches** — that is, irregularities in clock signal resulting in the speed up or delay of computations — to cause computational errors or timing issues [83].

> **Suggested Reading 17.2 — More information on glitching attacks**
>
> The NCCgroup's series of blogposts "An Introduction to Fault Injection" — available at the link `https://www.nccgroup.com/us/research-blog/an-introduction-to-fault-injection-part-13/` — provides further details on fault injection attacks based on glitching.

Laser and Optical Fault Injection. This type of fault injection attack relies on directing high-precision and timely laser [173] and optical [167] beams on specific locations of a circuit to introduce faults through **localized heating or photoelectric effects**.

Electromagnetic Pulses. This type of fault injection attack relies on emitting electromagnetic pulses (or creating electromagnetic fields) on specific locations of a circuit to **induce voltages or currents** and introduce faults. Electromagnetic pulses are usually less precise (and potentially more damaging) than laser and optical beams [66].

Temperature Variations. This type of fault injection attack relies on altering the temperature (i.e., through extreme cooling or heating) of specific locations of a circuit to **alter electrical characteristics** and introduce faults [95]. The rationale behind temperature variation attacks is that electronic components — like transistors and diodes — may slow down, speed up, or change their behavior when subject to significant temperature variations.

Software-Based Fault Injection. This type of fault injection attack relies on **exploiting software and firmware vulnerabilities and bugs to introduce faults remotely**. Software-based fault injection attacks are particularly worrisome as they do not require local attackers, hence can even target cloud-hosted servers and virtual machines [138, 195]. The most common software-based fault injection attack is **Rowhammer** [139, 111], which aims to flip particular bits (belonging to a process performing cryptographic computations) in DRAM memory by repeatedly accessing nearby bits (belonging to a process controlled by the attacker) in adjacent rows of memory cells. The Rowhammer attack is feasible due to the fact that memory cells are getting closer and closer in modern DRAM memory, leading to isolation and interference issues.

17.1.3 Data Remanence Attacks

At a high level, **data remanence attacks** aim to retrieve data whose residues persist on cryptographic modules after the execution of cryptographic computations. Such residues may be found in both volatile

memory (i.e., RAM), non-volatile memory (e.g., flash memory, SSDs, and magnetic hard drives), and even caches and buffers at the processor level. Data remanence attacks may rely on file deletion operations of operating systems (which typically remove entries of deleted files rather than actually overwriting deleted file contents), physical properties of storage media (e.g., RAM) allowing recovery of previous contents, or on applying rapid cooling techniques (e.g., liquid nitrogen) on memory chips to recover data that would have otherwise been erased during a normal shutdown process (in this last case, the attacker is required to be active). Data remanence attacks are often carried out in combination with temperature variations (§ 17.1.2) [16]. Again, the term data remanence attack denotes a **category of attacks** encompassing different approaches and techniques for recovering data, e.g., cold boot attacks and forensics recovery.

❄ Cold Boot Attacks. This type of data remanence attack relies on the **physical properties of RAM memory** due to which data persist in RAM memory for a short period (e.g., seconds to minutes) after the system is powered off. In cold boot attacks, the RAM memory is often physically removed from the system and analyzed on a different device. Alternatively, attackers may forcefully and abruptly reboot a system using a lightweight operating system from a removable disk to then dump the contents of the RAM memory [90].

Forensic Recovery. This type of data remanence attack relies on **forensics methods** to observe traces of data that should have been deleted and consequently recover the data themselves [189]. Forensic recovery attacks exploit residual data traces on rewritable optical media (e.g., CDs, DVDs) and non-volatile memory (i.e., hard drives and tapes through residual magnetic patterns and flash memory or SSDs with wear-leveling).

> **Remark 17.5 — On the practicality of implementation attacks**
>
> Attacks on the implementation of cryptographic algorithms may be more or less practical depending on the context — which should be reflected by the definition of a suitable threat model (§ 16.1) — and the system under consideration. In fact, conducting such attacks against complex systems such as laptops and servers is inherently more complex than conducting them against isolated hardware-based cryptographic modules like SE and TPM and embedded systems alike (§12): when the system is small and the hardware to be attacked relatively simple, then these attacks are more likely to succeed. Although it is often the case that software vulnerabilities (e.g., generic malware) are simpler to exploit than vulnerabilities in algorithm implementation, this does not mean that these attacks are impractical. On the contrary, several researchers highlighted the practicality of such attacks — even for modestly equipped and skilled attackers — and called for increased awareness among developers and organizations [66, 76, 127, 56, 55].

17.2 Security Best Practices

We now introduce the main high-level principles and methods for implementing cryptographic algorithms securely by mitigating and minimizing their exposure to the attacks described in § 17.1. Intuitively, the adoption of the following principles and methods cannot be considered sufficient to prevent all attacks.

> **Remark 17.6 — Security best practices and programming languages**
> The choice of the programming language for implementing cryptographic algorithms has a great impact on the relevance, practicality, and impact of the security best practices presented below — see § 18.1.

17.2.1 Secret-Independent Execution

Ideally, no side-channel attack is possible if every observable feature of the execution of sensitive cryptographic computations (e.g., mathematical operations with variable complexity, branching, access to data) is **statistically independent of secret data**. In particular, **variable complexity** is a prominent cause of timing and power-related attacks, as it provides a direct correlation between observable behavior and secret data. Similarly, **branching** can reveal information either indirectly due to instruction caching — which may cause timing leakage — or directly by varying the amount or type of operations performed — which may cause any kind of leakage. Finally, **data access** depending on secret data (e.g., as discussed for lookup tables in AES — see § 17.1.1) can cause timing and cache leakages due to the (unpredictable) physical management of virtual memory. For instance, large arrays — whose elements are contiguous in virtual memory — could be actually stored on multiple pages that are mapped to different physical memory (and even in different chips) and consequently have different access times.

> **Remark 17.7 — On the difficulty of achieving secret-independent execution**
> Unfortunately, full secret-independent execution is hardly achievable in practice. In fact, secret-independent execution does not only depends on the original implementation of an algorithm, but also on the multiple layers — roughly 2: compilation/interpretation (§18) and execution (recall the concept of speculative execution and the Spectre and Meltdown attacks in § 17.1.1) — between the original implementation written by the developers and the corresponding instructions ultimately executed by the processor. As a corollary, it is better (although more costly) to check the secret-independent execution of compiled machine code rather than the original implementation.

17.2.2 Constant-Time Coding

Constant-time coding refers to writing code explicitly aimed at thwarting timing (and, collaterally, other side-channel) attacks. Some operations are naturally constant-time: many stream ciphers (§ 1.3) — such as ChaCha20 (§ 6.4.2) — employ only bitwise operations like `xor`-ing, addition, and rotation; modern processors already execute these operations in constant time.

> **Warning 17.1 — Misunderstanding the meaning of constant-time coding**
> Constant-time coding **does not** prescribe that algorithms must execute in a fixed amount of time, but rather that secret data within an algorithm must not influence the use of underlying hardware resources.

The inherent constant-time nature of bitwise operations is also exploited in the **bitslicing** technique [134], which consists of expressing a function in terms of single-bit operations (i.e., AND, XOR, OR, NOT) in a logic circuit and exploiting the bigger registries of modern processors (usually 64 bits) to parallelize the execution of such single-bit operations. For instance, an 8-bit value in a bitsliced implementation

is stored in 8 variables (one for each bit), but a 64-bit architecture would allow processing up to 64-bit values at the same time.

To make an operation constant-time in presence of conditions (i.e., branches), it is common to compute both (or all) branches of the operation and then discard the useless result(s).[56] An instance of this approach is the **Montgomery ladder** [137] for scalar multiplication of a point Q by a number d in Elliptic Curve Cryptography (ECC) (§§ 2.5 and 3.4). Simplifying, in the case of ECC, the Montgomery ladder carries out the scalar multiplication by processing d (represented in binary) one bit at a time. In general, point addition is done only if the bit is 1; however, the Montgomery ladder always performs point addition on a temporary variable (even if the bit is 0) and then checks through branching (e.g., an `if` conditional instruction) whether to disregard the temporary variable or swap its value with the actual variable. Notably, even such branching and swap need to be constant-time; this is done via **constant-time conditional swapping** usually implemented with bitwise XOR and AND bitwise operations.

Warning 17.2 — Consider other non-constant time operations

Besides mathematical operations — and, in general, code written directly by the developers of an algorithm — developers should also check whether third-party code (e.g., libraries) is constant-time. For instance, comparisons — e.g., checking equality among two sequences of bytes — are usually not constant time but instead **fast failing**: they compare pairs of bytes until they differ, possibly not reaching the end of the two sequences. Hence, most implementations of comparisons leak which is the first byte where two sequences differ. Notably, some libraries are aware of the importance of constant-time coding and thereby offer constant-time functions (like the `CRYPTO_memcmp` comparison function in OpenSSL — see `https://docs.openssl.org/3.3/man3/CRYPTO_memcmp/`).

Suggested Reading 17.3 — More information on constant-time coding

The Chosen Plaintext Consulting's and Red Hat's blog posts "A beginner's guide to constant-time cryptography" and "The need for constant-time cryptography" — available at the links `https://www.chosenplaintext.ca/articles/beginners-guide-constant-time-cryptography.html` and `https://research.redhat.com/blog/article/the-need-for-constant-time-cryptography/` — provide more information on (the relevance of) constant-time coding. Moreover, Thomas Pornin gives a good primer on constant-time cryptography in relation to his library BearSSL.[a]

[a] BearSSL - Constant-Time Crypto (`https://www.bearssl.org/constanttime.html`)

Suggested Reading 17.4 — A collection of constant-timeness verification tools

The Centre for Research on Cryptography and Security at at Masaryk University curates a list of tools — available at the link `https://crocs-muni.github.io/ct-tools/` — for testing and verification of constant-timeness of programs.

17.2.3 Masking and Randomization

One of the simplest defenses against many side-channel attacks — like timing, electromagnetic, and power analysis — is to lower the SNR (§ 17.1.1) by **injecting noise**. For example, it is possible to exe-

[56] Go Assembly Mutation Testing (`https://words.filippo.io/assembly-mutation/`)

cute operations in parallel on multiple cores while simultaneously adding dummy or decoy operations. However, lowering the SNR simply makes attacks harder but does not prevent them — especially if measurements are precise and repeated enough times to filter out noise through statistics (i.e., differential attacks). Hence, noise injection is not a very effective defense.[57]

Better (but not perfect) protection can be achieved through **masking**, which consists in mixing secret data with random values (e.g., through `xor`-ing) and using the masked secret data to execute operations — intuitively, the operations have to be modified so that correct results can still be computed, and masks should never be reused.

> **Example 17.4 — Examples of masking and randomization**
>
> In RSA it is possible to blind the message by multiplying it with a random number or the secret exponent by exploiting Fermat's little theorem.[a] Similarly, in ECC it is possible to exploit Jacobian and projective coordinates to blind point coordinates [182].[b]
>
> [a] https://en.wikipedia.org/wiki/Fermat%27s_little_theorem
> [b] https://en.wikipedia.org/wiki/Jacobian_curve

17.2.4 Hardware Hardening

Despite the fact that hardware-based cryptographic modules (§12) are usually not enough to prevent (or sometimes even mitigate) the attacks discussed in § 17.1, their use is often considered a security best practice. In fact, even though being typically isolated from the main processors — which reduces noise and gives the attacker cleaner and precise measurements — **hardware-based cryptographic modules allows for implementing defenses not easily achievable through software alone**. For instance, many hardware-based cryptographic modules are shielded to reduce electromagnetic and sound leakage and embed power filtering to reduce power signal leakage. Moreover, randomizers for the power supply, the clock (called *clock jitter*), and memory access (e.g., see Kilopass' Secretcode memory[58]) are often available. Similarly, hardware-based cryptographic modules allows for balanced circuitry, that is, circuits that balance the amount of power used for a given data value or operation to prevent power analysis attacks. An even more sophisticated technique — that comes at the cost of sensibly increased computational cost — is **threshold implementation** and operates on a gate level: this technique exploits algorithms originally designed for Secure Multi Party Computation (SMPC) (§ 19.2.8) to split secret data into multiple variables so that secret data are never used directly. Then, gates performing non-linear operations are replaced by a cluster of gates called **gadgets** which perform the same operation but have all inputs masked — not dissimilarly to masking (§ 17.2.3). Intuitively, threshold implementation is often not viable in practice.

17.2.5 Memory Zeroization

To protect against data remanence attacks (§ 17.1.3) and memory leaks in general, it is recommended to immediately delete secret data when no longer needed. A common approach is to overwrite variables

[57] https://crypto.stackexchange.com/a/59901
[58] https://www.chipestimate.com/SIDE-CHANNEL-ATTACKS--How-Differential-Power-Analysis-DPA-and-Simple-Power-Analysis-SPA-Works-/Kilopass-Technology-a-part-of-Synopsys/Technical-Article/2017/09/19

containing secret data with a string of zero bytes of the same length — hence, **zeroization**. Another related recommendation is to minimize the time secret data is kept in memory and similarly minimize opportunities for dispersion.

> **Remark 17.8 — On the difficulty of achieving memory zeroization**
> Unfortunately, memory zeroization is hindered by many factors. First, the programming language itself may not fully support zeroization (e.g., those based on garbage collection — see §18). Also, processors may copy the value of variables into registries, stacks, or even swap them to non-volatile memory supports; it is rather difficult — when not impossible — to zeroize every copy of some secret data. Finally, processor optimizations may prevent zeroization altogether.

Historically, several vulnerabilities found in the implementation of cryptographic algorithms were related to memory management in general (e.g., buffer overflows) which depends on the chosen programming language — see § 18.1).

17.2.6 Resistance to Fault Injection

A first strategy against fault injection attacks (§ 17.1.2) is to structure the code such that it **fails closed**: the execution should fall back to a closed (i.e., secure) state in all plausible fault scenarios, and only enter an open (i.e., insecure) state when an unlikely set of conditions are satisfied. In other words, it must never be possible to enter an insecure state by skipping or glitching any single instruction. A second strategy is **resiliency through redundancy**: critical operations should be performed more than once and their results compared, and the execution should safely fail in case inconsistencies are found.

> **Example 17.5 — An example of resiliency through redundancy**
> Boolean values may be replaced with multi-bit values with alternate bits set and used for checks.

A third strategy is to introduce **random delays after externally observable events** (i.e., inter-chip communications, power-ups) to make it more difficult to time glitches.

17.2.7 Key Limit

The probably most effective mitigation against side-channel attacks is to implement a hard limit on **key usage** — a concept different than that of cryptoperiod (§ 13.1.2). The intuition is that, by stopping using a key and replacing it before enough measurements about its usage can be gathered, it is possible to prevent the attacker from performing an effective analysis and recovering useful information. In fact, (especially differential) side-channel attacks require a statistically significant number of data points to be successful. Intuitively, key limit presents a trade-off between security and usability, as it forces frequent key rotations.

17.2.8 Testing

Any implementation — especially that of cryptographic algorithms — should be properly reviewed and tested for (both functional correctness and) security with one or (better) more among the following approaches:

> **Suggested Reading 17.5 — More information on software security**
>
> § 17.2.8 focuses on testing for the implementation of cryptographic algorithms; on the subject of generic software security, we refer the interested reader to the resources of the NIST [165, 175], the International Organization for Standardization (ISO) [103, 104], the Open Web Application Security Project (OWASP) [21, 191], and the European Union Agency for Cybersecurity (ENISA) [7].

- **static analysis**: analyze the implementation without execution to detect vulnerabilities and ensure adherence to security best practices. While static analysis can detect vulnerabilities early in the development process and (ideally) achieve 100% coverage, such an approach may produce false positives and relies on the expertise of the developers and the capabilities of the tools employed. As for examples of methodologies and tools for static analysis, KLEE[59] [60, 69] allows for checking memory safety and proper deallocation through symbolic execution, CodeSonar[60] provides thorough static analysis in source and binary code, and True Code[61] detects fault injection vulnerabilities;
- **dynamic analysis (attack simulation)**: execute the code and/or emulate real-world attacks to analyze the resulting runtime behavior under controlled conditions and identify vulnerabilities. In the latter case, developers assume the role of the attacker, with the noticeable advantage of already knowing the attack surface and operating in favorable conditions (e.g., not needing depackaging/decapsulation — see § 17.1.1). While attack simulation provides practical evidence of eventual vulnerabilities, the coverage of such an approach is limited by the developers' offensive capabilities. As for examples of methodologies and tools for dynamic analysis, project Wycheproof[62] contains test vectors against known attacks, ctgrind[63] checks whether code is constant-time, Inspector SCA,[64] Inspector FI,[65], and ChipWhisperer[66] allows for conducting side-channel and fault injection attacks;
- **formal methods**: mathematically prove the security of the implementation against one or more attacks. Formal methods usually require modeling the system under test and specifying the desired security properties in a formal language, using a tool for conducting automated analysis and identifying (proofs of) vulnerabilities, and iteratively refinement. While formal methods offer mathematical assurance of security, such an approach can be resource-intensive and require deep expertise on the developers' end. As for examples of methodologies and tools for formal methods,

[59] Documentation · KLEE (https://klee-se.org/docs/)
[60] Static Application Security Testing (SAST) Software Tool | CodeSonar (https://codesecure.com/our-products/codesonar/)
[61] True Code - Automated embedded Software Security checks (https://www.riscure.com/security-tools/true-code)
[62] Project Wycheproof https://github.com/C2SP/wycheproof
[63] ctgrind (https://github.com/agl/ctgrind?tab=readme-ov-file)
[64] Side Channel Analysis Security Solutions - Riscure (https://www.riscure.com/security-tools/inspector-sca/)
[65] Fault Injection Security Tools - Riscure (https://www.riscure.com/security-tools/inspector-fi/)
[66] ChipWhisperer - the complete open-source toolchain for side-channel power analysis and glitching attacks (https://github.com/newaetech/chipwhisperer)

MaskVerif[67] [32] provides automatic verification of side-channel countermeasures based on masking, EasyCrypt[68] [34, 33] manages security proofs of cryptographic constructions, Vale[69] allows for writing in a C-like formal pseudo-language and then directly generate assembly code, ct-verif[70] [10] validates constant-time coding, and FIVER[71] [158] analyzes and verifies implementations of countermeasures against fault-injection attacks;

> **Suggested Reading 17.6 — More information on formal methods for cryptography**
>
> A survey of approaches and tools used by cryptographers and their features can be found in [23].

- **fuzzing**: automatically generate and test a large number of inputs to identify unexpected behavior or vulnerabilities in the implementation. While fuzzing is inherently scalable and can uncover edge cases, such an approach may also not detect all security issues, especially those requiring specific attack vectors. As for examples of methodologies and tools for fuzzing, AFL (American Fuzzy Lop)[72] allows for fuzzing generic software, LibFuzzer[73] provides in-process, coverage-guided, evolutionary fuzzing, FuzzTest [74] allows for writing and executing fuzz tests for C++ software, and True Code simulates faults in virtual hardware;
- **machine learning and artificial intelligence**: detect vulnerabilities (e.g., by automatically analyzing logs, traces, and execution flows for suspicious patterns), optimize attack strategies (e.g., by suggesting likely weak spots), and analyze the behavior of the implementation to identify anomalies. While machine learning and artificial intelligence can uncover non-obvious vulnerabilities and minimize the developers' effort, such an approach requires high-quality data for training and significant computational resources. SCAAML (Side Channel Attacks Assisted with Machine Learning)[75] is an example of a tool based on machine learning and artificial intelligence;
- **compliance testing**, which aims to validate the security of the implementation against established security standards (§15).

Developers should carefully assess the most suitable approach according to their context (e.g., expertise, threat model — see § 16.1 — and programming language — see §18) and possibly combine multiple approaches. For instance, developers may use static analysis and formal methods for verifying foundational

[67] maskVerif (https://cryptoexperts.com/maskverif/)
[68] EasyCrypt (https://www.easycrypt.info/)
[69] Introduction — Vale: Verified Assembly Language for Everest documentation (https://project-everest.github.io/vale/intro.html)
[70] GitHub - imdea-software/verifying-constant-time (https://github.com/imdea-software/verifying-constant-time)
[71] GitHub - Chair-for-Security-Engineering/FIVER (https://github.com/Chair-for-Security-Engineering/FIVER)
[72] GitHub - google/AFL: american fuzzy lop - a security-oriented fuzzer (https://github.com/google/AFL)
[73] libFuzzer – a library for coverage-guided fuzz testing. — LLVM 20.0.0git documentation (https://llvm.org/docs/LibFuzzer.html)
[74] google/fuzztest (https://github.com/google/fuzztest)
[75] GitHub - google/scaaml (https://github.com/google/scaaml/)

security properties and attack simulation and fuzzing for practical testing of the implementation — together with machine learning and artificial intelligence to enhance attack simulations and analysis, and compliance testing for ensuring adherence to industry standards, regulatory requirements, and organizational security policies.

> **Suggested Reading 17.7 — More information on testing for cryptographic code**
> Further resources relevant to the topic of testing the security of the implementation of cryptographic algorithms can be found in [43, 57, 152, 130, 187].

> **Suggested Reading 17.8 — Evaluating side-channel and fault injection attacks resistance**
> The German Federal Office for Information Security (BSI) has published the AIS 46 series of guidelines — available at the link `https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Kryptografie/Seitenkanalresistenz/seitenkanalresistenz_node.html` — on evaluating side-channel and fault injection resistance of algorithm implementation.

17.2.9 Bad Practices

Besides the aforementioned best practices, there exist also bad practices in implementing algorithms. The most notorious are:

- **security through obscurity**, already discussed in §1;
- **rolling your own crypto**, that is, programmers who prefer developing their own implementation of algorithms rather than relying on well-established and vetted cryptographic libraries. After reading §17, it should be clear how difficult it is to implement algorithms securely — not to mention bugs and inefficiencies. For instance, an empirical study of vulnerabilities in cryptographic libraries ("You Really Shouldn't Roll Your Own Crypto") found that only 27.2% of vulnerabilities in cryptographic libraries are actually cryptographic-related issues, while systems-level bugs are a greater security concern with memory safety issues (§ 18.1) constituting 37.2% of vulnerabilities [52]. In fact, note that there are experts who pursue security in algorithm implementation as a full-time profession;[76]
- **bloated code base**, that is: the more the code, the more vulnerabilities and errors are present. For this reason, modern cryptographic libraries tend to have a small code base (e.g., TweetNaCl[77] has 1,000 lines of code) hence allowing for complete and thorough security audits. Also, a larger code base usually implies many Application Programming Interfaces (APIs), increasing the risk of misusing and misconfiguration — especially by non-expert developers.

> **Suggested Reading 17.9 — Further low-level bad practices**
> Greg Rubin's blog post "Crypto Gotchas!" — available at the link `https://gotchas.salusa.dev/` — provides a very interesting list of low-level bad practices concerning applied cryptography.

[76] `https://words.filippo.io/full-time-maintainer/`
[77] `https://tweetnacl.cr.yp.to/`

18 PROGRAMMING LANGUAGES AND SOFTWARE

Below, we discuss the impact that the choice of the programming language (§ 18.1) — and, consequently, compiler (or interpreter) and configuration options (§ 18.2) — may potentially have on the security of the implementation of cryptographic algorithms. In other words, we investigate how the choice of the programming language and compiler or interpreter can reduce or amplify the risk of an attack (§ 17.1) or inherently provide — or makes more difficult to implement — a security best practice (§ 17.2).

18.1 Programming Languages for Security

Different programming languages naturally have distinct characteristics and approaches to memory management, code optimization, and execution strategies. Hence, each programming language balances efficiency, usability, security, and abstraction differently according to the intended target audience, purpose, and applications. When considering the implementation of cryptographic algorithms specifically, we identify the following aspects as relevant to security:

- **memory safety**, which comprises mechanisms and practices that a programming language adopts by design to ensure proper and secure management of secret data and prevent memory corruption vulnerabilities such as:
  - *automatic bounds checking*, to prevents buffer overflows (both in read and in write) by ensuring that array and memory accesses are within valid bounds;
  - *null safety*, to eliminate the risk of dereferencing null pointers;
  - *ownership and borrowing*, to enforce clear memory ownership rules to avoid use-after-free and double-free errors;
  - *type safety*, to ensure that variables are used only in ways consistent with their declared types, reducing the risk of unintended behavior;
  - *memory zeroization*, to wipe or overwrite data explicitly (§ 17.2.5);

- *memory integrity*, to provide the possibility to mark specific pieces of data as immutable or constant after initialization while also providing integrity checks;

> **Example 18.1 — A real-world memory safety vulnerability**
>
> A famous vulnerability related to memory safety is the **Heartbleed Bug**,[a] a bug in the OpenSSL implementation of the heartbeat extension [156] for Transport Layer Security (TLS) (§ 19.3.1). The heartbeat extension was intended to allow for keeping TLS connections alive even without continuous data transfer. However, the lack of bounds checking allowed attackers to read out of bounds in memory areas highly likely to store sensitive data such as private keys.
>
> [a] Heartbleed Bug (`https://heartbleed.com/`)

- **deterministic behavior**, which expects instructions and runtime routines to behave deterministically (as much as possible) thanks to:
  - *language complete specification*, to ensure no edge case leads to non-deterministic behaviors;
  - *constant-time*, to make the execution time of instructions in libraries and algorithms — like comparisons and arithmetic operations — constant regardless of their input (§ 17.2.2);
  - *predictable memory management*, to eliminate runtime routines — like garbage collection — that may introduce timing variability and random behavior;
  - *secure error handling*, to handle runtime errors securely without leaking sensitive information and avoid unintended behavior from uncaught or mishandled exceptions and signals;
  - *cross-platform consistency*, to guarantee that the source code behaves the same across different platforms, preventing platform-specific vulnerabilities and abstracting platform-specific hardware features;
- **concurrency safety**, which ensures consistency and integrity of operations and data in concurrent environments (e.g., through atomicity and prevention of race conditions, respectively);
- **safe defaults**, which encourage developers to use secure defaults and require explicit declarations (i.e., opt-in) for employing new or untested features or practices that bypass security mechanisms, ensuring conscious use;
- **ecosystem and community support**, which allows developers to use off-the-shelf robust, vetted and well-documented libraries, tools, and frameworks for cryptographic development.

We now analyze how mainstream and popular programming languages relate to the aforementioned security aspects and report our findings in Table 6.

C and C++. C and C++ are widely popular — almost dominant — for implementing cryptographic algorithms due to their widespread adoption, efficiency, and low-level control which allows for fine-tuning and optimizing implementations (especially important in resource-constrained environments).

In fact, many well-known cryptographic libraries such as OpenSSL,[78] libgcrypt,[79] and NaCl[80] are written in C and C++. **On the other hand**, C and C++ are inherently less secure than other programming languages. First, C and C++ lack built-in mechanisms for memory safety: manual memory management, buffer overflow, dangling (null) pointers, and use-after-free have often been the source of serious vulnerabilities. Moreover, undefined behavior present in many cases in the languages specification[81] lead to non-deterministic behavior. Furthermore, developers often need to handle synchronization manually through locks, semaphores, and mutexes, and uninitialized variables and unsafe typecasting are common — although C++ provides stronger type safety than C. Finally, many functions in libraries and algorithms (e.g., `memcmp()`[82]) are not constant-time.

> **Suggested Reading 18.1 — More information on how to program in C securely**
> Jean Philippe Aumasson's "Cryptocoding" — available at the link `https://github.com/veorq/cryptocoding` — is a neat collection of best practices with code examples for C.

Java. With respect to C and C++, Java offers safer memory management thanks to bounds checking, strong type safety, and automatic garbage collection which significantly reduces the risk of memory corruption attacks — but also **thwarts predictable memory management** (also, null pointer exceptions are a common concern in Java). Moreover, Java provides a rich set of tools for safe multi-threading and concur-

[78] OpenSSL (https://www.openssl.org/)
[79] Libgcrypt (https://gnupg.org/software/libgcrypt/index.html)
[80] NaCl (https://nacl.cr.yp.to/)
[81] Undefined behavior - cppreference.com (https://en.cppreference.com/w/cpp/language/ub)
[82] memcmp - cppreference.com (https://en.cppreference.com/w/c/string/byte/memcmp)

Table 6: Security aspects of languages for the implementation of cryptographic algorithms

Aspect	Programming Language						
	C/C++	Java	Kotlin	Python	Rust	JS	Go
automatic bounds checking	○	●	●	●	●	○	●
null safety	○	○	●	◐	●	○	●
ownership and borrowing	○	○	○	○	●	○	○
type safety	◐	◐	◐	●	●	●	●
memory zeroization	○	○	○	○	●	○	○
memory integrity	○	◐	◐	○	●	○	●
language complete specification	○	●	●	○	●	○	●
constant-time	○	◐	◐	○	●	○	◐
predictable memory management	○	○	○	○	●	○	●
secure error handling	○	◐	●	○	●	○	●
cross-platform consistency	◐	●	●	●	●	●	●
concurrency safety	○	●	●	○	●	○	●
safe defaults	○	◐	●	○	●	○	●
ecosystem and community support	●	●	●	●	◐	●	◐

rency, such as synchronized blocks and thread-safe collections. Finally, we note that Java inherently provides strong cross-platform capabilities — by relying on the Java Virtual Machine (JVM) — and includes a structured approach to the provisioning of cryptographic functions thanks to the Java Cryptography Architecture (JCA).

Kotlin. Building on top of Java, Kotlin introduces immutable data classes and null safety as core language features, thereby **improving memory safety and integrity** and preventing null pointer dereferences. Moreover, Kotlin enhances secure error handling and provides a modern approach to handling concurrency safely and efficiently through coroutines. Finally, Kotlin benefits from the Java ecosystem while building its own growing community and libraries.

Python. Excellent in educational contexts and for rapid prototyping due to its simplicity and abstraction, Python typically relies on libraries implemented in lower-level languages (e.g., C) for cryptographic functions. For instance, both cryptography.io[83] and M2Crypto[84] act like wrappers over OpenSSL. Also PyCryptodome[85] — which explicitly aims to implement cryptographic algorithms in pure Python — relies on C for some cryptographic constructions, e.g., block ciphers (§ 1.3). In fact, **the high abstraction of Python makes it difficult** to achieve fine-grained control over timing and memory, both of which are critical for implementing cryptographic algorithms securely. Then, concurrency safety is limited by the Global Interpreter Lock (GIL), reducing exposure to concurrency issues but also limiting performance. Finally, Python has a large ecosystem with numerous libraries and frameworks to rely upon.

Rust. Prioritizing memory safety without sacrificing low-level control and performance, Rust is **an increasingly popular (and sensible) candidate for implementing cryptographic algorithms**. The ownership and borrowing model[86] — albeit complex to learn — ensures memory safety at compile time, preventing vulnerabilities such as buffer overflows, use-after-free, and double-free errors. At the same time, developers may rely on language core features, libraries, and functions (e.g., `subtle`, `constant_time_eq`) to improve the constant-time quality of the code. Moreover, Rust provides mechanisms for memory zeroization (e.g., `zeroize`) and type safety. Finally, while smaller than other languages, Rust's ecosystem is rapidly expanding with security-focused libraries and increasing support for cryptographic tools.[87]

JS. Until a few years ago (~2023), web applications typically relied directly on JavaScript (JS) for the provisioning of cryptographic algorithms required for client-server interaction. For instance, the popular CryptoJS library[88] implements many cryptographic algorithms directly in JS. Nonetheless, the same de-

[83] pyca/cryptography (https://cryptography.io/)

[84] GitHub - mcepl/M2Crypto: OpenSSL for Python (both 2.x and 3.x) (generated by SWIG) (https://github.com/mcepl/M2Crypto)

[85] GitHub - Legrandin/pycryptodome: A self-contained cryptographic library for Python (https://github.com/Legrandin/pycryptodome)

[86] What is Ownership? - The Rust Programming Language (https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html)

[87] GitHub - rust-secure-code/projects: Contains a list of security related Rust projects (https://github.com/rust-secure-code/projects)

[88] GitHub - brix/crypto-js: JavaScript library of crypto standards (https://github.com/brix/crypto-js)

velopers of CryptoJS remark that modern browsers employ native cryptographic libraries which should be preferred over JS (to the point that active development of CryptoJS has even been discontinued). The built-in **Web Crypto API**[89] is nowadays the standard used in JS.

Go. Known for its simplicity, performance, and built-in concurrency support (i.e., `goroutines`), Go is also an **ideal candidate for the implementation of cryptographic algorithms**. In fact, Go provides memory safety by including automatic bounds checking and type safety. On the other hand, Go uses garbage collection which mitigates memory corruption attacks but prevents predictable memory management. Finally, security practices are encouraged by default, but some manual effort is required for developing constant-time code.

Conclusion. While efficiency often dominates the discussion about the implementation of cryptographic algorithms, security should be highly regarded as well: Table 6 shows how C and C++, Java, and Python are not ideal, Kotlin may be a fairly good choice, JS may be the only option for web applications, and Go and Rust are by far the best choices.

Remark 18.1 — Other factors to consider for an educated choice

Further factors that developers should consider for making an educated choice regarding what programming language to use for implementing cryptographic algorithms are:

- **further programming languages**: some popular programming languages (e.g., Ruby, Perl) were not analyzed in Table 6 and may be considered by developers;
- **developers' expertise**: intuitively, developers must be familiar with a programming language to use it proficiently;
- **development environment**: the availability of advanced Integrated Development Environments (IDEs), debuggers, linters, and testing frameworks may improve the quality — and, consequently, the security — of the code;
- **legacy systems ans integration**: the need to develop code which is compatible with other (possibly legacy) systems and software may favor the adoption of a programming language over another;
- **long-term viability**: an ideal programming language should be subject to active development, proper (i.e., stable and predictable) updates, and be guaranteed long-term support by major technological organizations;
- **target application**: the environment, devices (e.g., embedded systems, mobile devices), and purposes for which the code should be developed must be considered when choosing the programming language.

Warning 18.1 — Lack of literature concerning programming languages for security

Although some researchers have investigated the (in)security of specific characteristics of specific programming languages in the implementation of cryptographic algorithms — i.e., with a very focused scope— to the best of our knowledge a comprehensive analysis is missing in the literature. For an empirical investigation on the subject of the relationship between security and program-

[89] Web Crypto API - Web APIs | MDN (https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API)

ming languages in the wild (not only concerning cryptography) we refer the reader to [72].

18.2 Compilers and Interpreters Options for Security

Even though a suitable programming language is selected (§ 18.1) and all attacks (§ 17.1) relevant to the considered threat model (§ 16.1) have been properly addressed through the adoption of security best practices (§ 17.2), compilers and interpreters — also according to their configuration options — may play a decisive role in fostering (or, more often, undoing) security. For instance, mainstream C compilers usually apply aggressive optimizations that can inadvertently break constant-time cryptography [172]. A (very recent and) more detailed and thorough investigation on the topic of how compilers may (or, better, often) break constant-time implementations can be found in [168].
In general, it is difficult to control (or anticipate) the behavior of compilers and interpreters. Moreover, configuration options either heavily rely on developer expertise or do not always allow developers to express requirements concerning, e.g., code optimization, as they would. In other words, **it is advisable to choose a programming language whose compiler or interpreter (e.g., one among those available) impacts code structure as little as possible**, and then use specific approaches and tools (§ 17.2.8) to verify and validate the security of cryptographic algorithm implementation.

18.3 Trusted Computing Base

The Trusted Computing Base (TCB) of a system (e.g., an application, a cryptographic module) is the minimal subset of hardware, firmware, and software constituting the system — as well as their configuration — **that must function correctly and remain uncompromised for the system's security guarantees to hold**. In other words, misbehaviors and vulnerabilities in a system's TCB directly impact the system's security, while errors and vulnerabilities in the remaining system's hardware, firmware, and software do not impact its security (although they might affect functionality or availability).

Example 18.2 — A real-world analogy for TCB

Consider as a system the front door of a house: the TCB may comprise the lock mechanism (e.g., the tumbler, bolt, and casing), the lock key, and the door frame, while the doorknob, the welcome mat, and all decorations are not part of the TCB.

Clearly identifying the TCB of a system allows for narrowing down the scope, complexity, and cost of security analysis and testing. Similarly, minimizing the size of the TCB is fundamental to reduce the attack surface of a system. Intuitively, **cryptographic algorithms are (almost) always included in the TCB**.

Remark 18.2 — The concept of attack surface

The attack surface of a system is the sum total of all the points or interfaces — such as APIs and their parameters, open network ports, files reads, lines of code and third-party libraries — through which attacker can attempt to compromise the system or extract data. Obviously, a small attack surface is preferable.

Security properties particularly relevant to a TCB are:

- **tamper-evidence**: reliable indicators (e.g., flags, logs, alerts, safe-fail modes) should make it clear whether the TCB of a system was compromised. Concretely, this can be achieved, e.g., through digital signatures (§8), Message Authentication Codes (MACs) (§ 6.1), and features of hardware-based cryptographic modules such as secure boot (§12);
- **auditability**: any qualified evaluator — e.g., the system's developers themselves, external authorized auditors, or the broader community in open-source projects — should be able to inspect each component of a TCB to verify it was correctly written, compiled, and is running as intended. Naturally, the greater the transparency of a TCB — net of inherently complexities (e.g., in hardware designs and firmware) — the higher the confidence in its security;
- **measurability**: means should be available to produce measurements of a TCB and ensure it corresponds to the intended definition. The remote attestation capability of Trusted Execution Environment (TEE) (§ 12.4) is an example of such means, as the comparison of the digest (§5) of the TCB of an open-source software instance against the digest computed by the developers.

⚠ Warning 18.2 — Beyond TCB: the supply chain

Even though a system has a small, verified, and robust TCB, there still are security concerns to be well-aware of. In particular, although a good TCB may provide confidence on the security of a system, **without a secure supply chain such confidence is simply delusive**. In system security, the supply chain of a system comprises processes, tools, and relationships by which the components of the system — software, firmware, and hardware — are developed, built, delivered, and updated. A supply chain is deemed secure if every component originates from a trusted source, was not tampered with, and is properly versioned and signed. Otherwise, a good TCB may anyway end up containing undocumented features, malicious code, and backdoors (e.g., injected during the build process). In this sense, a famous speech was delivered by Ken Thomson during his Turing Award lecture in 1984 and published in his article "Reflections on Trusting Trust" [179], which discusses the possibility of including self-replicating backdoors into compilers so to propagate them, invisibly embedded, into all programs compiled with it (including the next versions of the same compiler). Unfortunately, ensuring the security of a supply chain is a daunting task, and trustworthiness is often assumed rather than verified. The Intel Transparent Supply Chain (TSC)[a] is an example of an attempt of securing the supply chain concerning selected Intel products by providing visibility and traceability of hardware components, firmware, and systems, and using TPMs (§ 12.3) for secure booting and measurements' comparison.

[a] Transparent Supply Chain (https://www.intel.com/content/www/us/en/security/security-practices/transparent-supply-chain.html)

19 SECURE DESIGN OF PROTOCOLS

A **cryptographic protocol** is a protocol employing cryptography within the exchange of messages among two or more parties — and, possibly, also cryptographic computations local to each party (§ 1.5 for more details). Depending on the underlying scenario, cryptographic protocols may be designed to provide different security properties such as confidentiality, integrity, and authenticity (§4 for a more complete list).
However, cryptographic protocols may fail to provide the desired security properties for a variety of reasons, the main ones being:

- the use of inadequate cryptographic algorithms (either misconfigured, misused, or outdated) — this is discussed more in detail in §§ 5.1, 6.4, 7.1 and 8.2 where we list cryptographic algorithms for hash functions, symmetric cryptography, and asymmetric cryptography;
- vulnerabilities in the implementation of the cryptographic protocol itself or in the employed cryptographic algorithms or libraries (§17) or in the programming language used to implement them (§18);
- flaws in the design of the cryptographic protocol — **this is the focus of the current note**.

19.1 Design of Protocols

The design of cryptographic protocols comprises several steps, which we illustrate below.

1. Scenario and Security Properties. First, the designers of a cryptographic protocol gather detailed information about the scenario and elicit desired security properties by engaging stakeholders. Notably, this step has more to do with **requirements engineering** rather than with technical aspects — and is always more complex than it seems. Relevant information and considerations resulting from this step are the list of interacting parties, assets to protect, use cases, regulatory constraints, and the operational environment (e.g., performance constraints on computational resources, latency, and bandwidth).

> **Warning 19.1 — Interdependencies among security properties**
> Beware that some security properties may have interplays (e.g., integrity vs. sender authenticity vs. ciphertext authenticity) and even conflict with each other (e.g., non-repudiation vs. deniability).

2. Assumptions and Threat Model. Then, the designers explicitly state all assumptions related to trust (e.g., which parties are trusted) and security (e.g., on cryptographic material available to those parties and on cryptographic keys binding — see §14). At the same time, the designers define a threat model (§ 16.1) for enumerating and prioritizing possible attacks (e.g., see §§ 16.3 and 19.1.2) to the cryptographic protocol present in the considered scenario, thereby ensuring that the protocol addresses realistic risks rather than hypothetical or impractical ones. A good threat model should be realistic, comprehensive, and consistent, and it should be updated regularly to reflect the changing environment and threats.

> **Example 19.1 — Threat models for cryptographic protocols**
> Communication channels are often assumed to be controlled by an attacker. Starting from this assumption, two threat models are commonly used for cryptographic protocols. The **Dolev-Yao** threat model [78] is widely used for discussing (and formally verifying) the security of protocols [188]. The Dolev-Yao threat model defines an attacker — i.e., the Dolev-Yao attacker — who has full control over communication channels, i.e., the attacker can intercept, modify, inject, delete, and replay any message. Moreover, the Dolev-Yao attacker can execute and interleave multiple executions (or **runs**) of protocols. However, the Dolev-Yao attacker cannot affect single components, break cryptography (e.g., through cryptanalysis, brute-forcing, or side-channel attacks — see § 17.1), or affect the physical world in any way. In other words, the Dolev-Yao threat model assumes perfect cryptography, operating in the symbolic model (§ 19.2). Another widely used threat model in the context of protocols is **Canetti-Krawczyk** [62] which defines an attacker with the same capabilities as the Dolev-Yao attacker; in addition, the Canetti-Krawczyk attacker can also conduct physical attacks to potentially retrieve secret data from cryptographic modules, operating in the computational model (again, see § 19.2).

3. Informal Flow and Algorithms Selection. Once the context and the objectives are clear, the designers can draft an informal description of the cryptographic protocol in plain language or simple diagrams (e.g., message sequence charts): this amounts at defining the messages (in terms of syntax, semantics, and temporization) and the cryptographic algorithms to employ (e.g., hash functions, symmetric and asymmetric ciphers). Usually, this is a top-down process where details are refined iteratively to balance security properties with performance constraints (actually, the whole design process is often iterative).

> **Suggested Reading 19.1 — The Alice and Bob notation**
> Dating back to 1978, the **Alice and Bob notation** is a simple, intuitive, and popular graphical language for the description and specification of the interactions between system components (e.g., such as cryptographic protocols). Quinn DuPont's and Alana Cattapan's website "A History of The World's Most Famous Cryptographic Couple" — available at the link `https://cryptocouple.com/` — details the major events in the "lives" of Alice and Bob, from their birth in 1978 onward.

4. Formal Specification and Security Analysis. Afterwards, the designers translate the informal design using a formal language or pseudocode to avoid ambiguities. Having a formal specification allows for then applying formal methods (§ 19.2) to prove the security of the cryptographic protocol under the defined threat model.

5. Configuration, Implementation, and Deployment. Lastly, the designers can define a suitable configuration (e.g., cipher suite choices, key lengths, timestamp windows, error-handling semantics), choose suitable cryptographic libraries, and deploy the cryptographic protocol.

> **Remark 19.1 — Further points of attention**
>
> Besides the aforementioned 5 steps, the secure design of cryptographic protocols needs to take into account key management (§13), proper randomness sources (§ 19.5.4), clear deployment and usage guidelines, and update and deprecation mechanisms.

19.1.1 Best Practices

Even years after their deployment, many cryptographic protocols were found to have vulnerabilities and weaknesses; learning from these mistakes, practitioners and researchers defined best practices for their design. Unfortunately, there is not a unique list of best practices formally agreed upon by everyone; nonetheless, below we make an effort to review the main (i.e., most important) design best practices to avoid common pitfalls when designing cryptographic protocols.

> **Example 19.2 — A famous example of a vulnerable cryptographic protocol**
>
> A replay attack on the symmetric version of the Needham-Schroeder key exchange protocol (comprising 5 messages only) went undetected for around 3 years; even more notably, a Man-in-the-Middle (MitM) attack on the public key version of the Needham-Schroeder protocol (comprising 7 messages only[a]) was discovered after around 17 years.[b]
>
> [a] The original version of the public-key Needham-Schroeder protocol in 1978 [145] assumed a trusted server for key distribution within a 7-step protocol. However, later treatments (notably, Lowe's 1995 attack [128]) abstract away the server, focusing on the core 3-message exchange.
>
> [b] Needham–Schroeder protocol - Wikipedia (`https://en.wikipedia.org/wiki/Needham\T1\textendashSchroeder_protocol`)

Map Cryptographic Protections with Security Properties. Map each cryptographic protection (e.g., using a table) to the security properties it provides — albeit it may seem obvious — rather than relying on intuition or colloquial terminology. Subtle protocol flaws may derive from mismatches between cryptographic protections and security properties, e.g., assuming that encryption alone yields integrity, that freshness or ephemeral keys imply authenticity, or that MAC (§ 6.1) provides non-repudiation. It is also important to rigorously check that the composition of cryptographic protections does cover every required security property.

Simplicity and Clarity. Being arguably (almost) always part of the TCB of a system (§ 18.3), design cryptographic protocols as simple and minimal as possible. Every feature, message field, or option must

have a clear security purpose. However, note that simplicity and clarity do not mean lacking functionality.

The Human Factor. Design cryptographic protocols with human comprehension and usability in mind; this includes both end users and implementers. In fact, even the most secure cryptographic protocol can be compromised by user error, misuse, misconfiguration, or misunderstanding. Concretely, use secure defaults and reduce opportunities for incorrect choices and ensure that any required user actions (e.g., confirming an identity, accepting a key, or verifying a fingerprint) are clear and guided. Provide friendly interfaces, meaningful error messages, clear documentation, and, wherever applicable, training materials or educational cues: if users do not understand what a security warning means, they may ignore it. Similarly, if implementers do not understand the specification of a cryptographic protocol, they may create insecure variants.

Freshness. Include freshness to ensure that messages are recent and not reused to thwart reflection and replay attacks. Common methods to include freshness in cryptographic protocols are nonces and timestamps: nonces are non-secret random or sequential numbers (either way, the important thing is to avoid collisions) used only once and protect the parties that generated them,[90] while timestamps ensure message timeliness (with an allowable clock skew) for all parties involved. Intuitively, timestamps are tricky to use as they inherently have a time window tolerance (e.g., a few seconds) that requires synchronization among parties and entails a short-lived vulnerability to replay attacks.

Channel Binding. Include context information (e.g., identities of sender and recipient, session identifiers, purpose) in each message — typically, in Authenticated Encryption with Associated Data (AEAD) authentication tags (§ 6.2.1) — to explicit under what circumstances a message is valid (and useless elsewhere) and thwart impersonation, rebinding, reflection, message insertion, and reordering attacks. In other words, implicit context (e.g., assuming the recipient "knows" what a message means based on order) should be avoided. Similarly, it is a best practice to "chain" cryptographic computations through the injection of context information (e.g., encrypt data with committing AEAD using a symmetric key established from a combination of asymmetric keys and parties' identifiers).

> **Example 19.3 — Channel binding in real-world protocols**
>
> Modern cryptographic protocols such as TLS v1.3 and Signal (§§ 19.3.1 and 19.3.2) inject context information in key establishment or in AEAD computations. In fact, channel binding applies also to layered authentication, i.e., whenever a cryptographic protocol runs on top of another (e.g., Signal over TLS) in a secure channel [190].

[90] A nonce protects the party that generated it against replay attacks. For instance, consider a party A that needs to submit a query to a party B: A generates a nonce n_A and sends it along with the query to B. Then, B replies by sending the answer to the query and the nonce n_A to A. Now, A is sure that the answer is "fresh", given that it includes the newly generated nonce n_A (assuming no collisions happened and that no attacker can replace nonces in messages). However, B is not protected against replay attacks — unless B keeps a list of all nonces ever received and checks the list every time it receives a new query to detect replay attacks; however, this kind of protection against replay attacks is often impractical due to the complexity in handling a continuously growing list. Therefore, it is more common for B to generate a nonce n_B as well (or to keep the list of received nonces short by requiring queries to include timestamps and consequently preserve a nonce in the list only if within the allowed clock skew).

AEAD. If symmetric cryptography is needed, use (committing) AEAD ciphers such as AES-Galois/Counter Mode (GCM) and ChaCha20-Poly1305 (§§ 6.4.1 and 6.4.2) to thwart ciphertext forgery and tampering attacks.

CHFs. If digests are needed, use robust CHFs such as SHA-3 and BLAKE2 to thwart collision, preimage, and length extension attacks (§ 5.1). Moreover, include context information or a prefix as input to CHFs to avoid cross-protocol collision issues.

> **Warning 19.2 — Using CHFs for integrity**
> If digests are required for integrity, do not "roll your own" integrity checks by hashing data and comparing the resulting digests manually — instead, use Hash-based Message Authentication Code (HMAC) or digital signatures as appropriate (§ 6.1.1 and §8).

Public-Key Cryptography (PKC). If asymmetric cryptography is needed, use robust asymmetric ciphers for encryption (§ 7.1) as the basis for hybrid cryptography (§11) and robust asymmetric ciphers for digital signatures (§ 8.2).

Ephemeral Keys. Include ephemeral keys (§ 1.1) to provide forward secrecy during key establishment (§ 13.1.1) — typically when deriving (symmetric) session keys — and mitigate long-term confidentiality violation attacks.

> **Example 19.4 — Ephemeral keys in real-world protocols**
> The use of ephemeral keys to provide forward secrecy is enforced by default in modern cryptographic protocols such as TLS v1.3 and Signal. Moreover, Signal allows for providing limited forward secrecy by using short-term asymmetric key pairs.

Key Separation or Single Purpose. Prefer using distinct keys for distinct purposes (e.g., encryption vs. authentication vs. digital signatures vs. different communication directions – client-to-server and server-to-client), even if derived from a common secret through Key Derivation Functions (KDFs) with proper domain separation, e.g., injecting labeled context strings (§ 13.1.1). In fact, reusing keys across different cryptographic algorithms can weaken their security guarantees against cryptanalysis and amplify the impact of key disclosure [25]. However, note that the same key may sometimes be used to provide multiple security properties (e.g., confidentiality and ciphertext authenticity in AEAD), though almost always in the context of a single cipher.

> **Remark 19.2 — Ciphers not respecting the key separation principle**
> There are ciphers which disregard key separation for a more pragmatic design, but only after a very careful trade-off and design analysis — see § 8.2.3 for some examples and more details.

Security Over Backward Compatibility. Do not sacrifice security for backward compatibility — that is, do not include obsolete cryptographic algorithms or configurations to accommodate outdated systems: prioritizing or even allowing backward compatibility may (re-)introduce vulnerabilities. If legacy

devices are present, need to communicate, and cannot be updated, consider offering a separate legacy mode if absolutely necessary (and remember to isolate the legacy mode from the main secure mode).

> **Example 19.5 — TLS and backward compatibility**
>
> TLS v1.2 favored backward compatibility with TLS v1.1 by considering 45 cipher suites — inheriting serious vulnerabilities — while the latest TLS v1.3 favors security by considering 5 cipher suites only and removing insecure options (such as RSA key establishment).

Crypto-Agility. Cryptographic protocols must be capable of swapping out cryptographic algorithms without a complete redesign and with minimal disruption. In fact, cryptographic algorithms deemed robust may become weak due to cryptanalytic breakthroughs or new capabilities (e.g., quantum-computing — see § 19.5.2). Consequently, include versioned algorithm identifiers (possibly standard) and cipher suites negotiation secured against downgrade attacks [64] and be algorithm-neutral in design.

> **Remark 19.3 — Trade-offs between crypto-agility and complexity**
>
> If necessary (e.g., the cryptographic protocol targets a wide generic audience), crypto-agility would prescribe to offer more options for cipher suites to avoid scenarios in which a single cipher gets broken and leaves no alternative. However, this would add further complexity, increase the TCB (§ 18.3), and ultimately enlarge the attack surface. As a unique optimal choice does not exist, the designers of a cryptographic protocol should identify the best trade-off between crypto-agility and complexity on a case-by-case basis.

Post-Quantum Cryptography. If expected by the threat model, design protocols to include — either during the first design or also in the future — post-quantum cryptographic algorithms (e.g., by planning for larger keys, message sizes, and digital signatures) to thwart "harvest now, decrypt later" attacks (§ 19.5.2).

Standards. Consider standards (§15) and white papers of reputable organizations (Appendix B) when designing cryptographic protocols to avoid known mistakes and enhance interoperability, security, and unambiguity. Finally, being consistent with standards also helps with compliance.

Performance Constraints. Be mindful of performance constraints (e.g., low power, low bandwidth, high latency, limited UI, low battery or CPU) tied to the specific application or environment (e.g., Internet of Things (IoT) vs. web vs. mobile vs. automotive) in which a cryptographic protocol will be deployed: cryptographic protocols impractical in their environment will either fail to be adopted or, worse, be implemented in an insecure way to "make them work". This may imply prioritizing security properties (e.g., forward secrecy and deniability for instant messaging protocols, authenticity and confidentiality for IoT) according to the considered threat model. Still, we recommend never compromising core security principles for performance.

Testing and Verification. Analyze and test the security of cryptographic protocols by conducting attack simulations or using formal methods and verification tools (§ 19.2). If possible, engage in security audits or invite cryptographers to review the design of cryptographic protocols.

Ratcheting. Ratcheting mechanisms (§ 19.5.7) ensure that cryptographic protocols can recover from key compromise over time. Often, an attacker who obtains currently used private or secret keys may decrypt all future messages. In this context, ratcheting mechanisms allow cryptographic protocols to self-heal after compromise — that is, once the attacker is out of the loop, future messages become secure again.

Error Handling. Handle errors due to failures, inconsistent states, and unforeseen situations properly by preventing information leakage and defining clear recovery or abort procedures to ensure predictable and safe behavior. Protocol-level error messages must be uniform and generic, revealing only that an operation failed without exposing details such as whether a decryption failed due to an incorrect padding value, an expired key, or a malformed ciphertext. Error messages in logs may contain more information, but still must not disclose sensitive information (e.g., private or symmetric cryptographic keys).

19.1.2 Categories of Attacks

In order to design secure cryptographic protocols, one should first have an idea of what the (at least most common) attacks are. Compiling an exhaustive list is difficult — as each cryptographic protocol may present peculiar vulnerabilities and weaknesses — but we can identify the following high-level attack categories:

- **passive eavesdropping attacks**: the attacker listens to communications without altering them to read unencrypted data or perform traffic analysis (e.g., to identify patterns in timing, frequency, or length of messages);
- **MitM attacks**: standing between two (or more) communicating parties, the attacker exploits poor authenticity to impersonate one or both parties by intercepting and possibly altering exchanged messages. Intuitively, the legitimate parties are usually not aware of the MitM attacker;
- **downgrade attacks**: the attacker forces parties engaging in a cryptographic protocol to fall back to outdated or less secure protocol versions and cipher suites, often by interfering during the negotiation or handshake phase[91];

> **Example 19.6 — Real-world downgrade attacks**
>
> Two famous downgrade attacks are the POODLE (Padding Oracle on Downgraded Legacy Encryption)[a] and the FREAK (Factoring RSA Export Keys)[b] attacks on TLS.
>
> [a] SSL 3.0 Protocol Vulnerability and POODLE Attack | CISA (https://www.cisa.gov/news-events/alerts/2014/10/17/ssl-30-protocol-vulnerability-and-poodle-attack)
>
> [b] FREAK SSL/TLS Vulnerability | CISA (https://www.cisa.gov/news-events/alerts/2015/03/06/freak-ssltls-vulnerability)

[91] A handshake in a cryptographic protocol is the initial exchange of messages between two (or more) parties whose goals are usually to authenticate one another, negotiate protocol parameters, and establish fresh secrets.

- **replay attacks**: the attacker delays, repeats, or reuses valid protocol messages — often originated by legitimate parties — in different and possibly simultaneous runs of cryptographic protocols. For instance, an attacker might capture an authentication token and send it again to impersonate the original sender at a later time. Two sub-categories of replay attacks are **reflection attacks** — where the attacker tricks a party into unknowingly responding to its own messages — and **rebinding attacks** — where the attacker reuses messages in different contexts or for different purposes than originally intended;
- **oracle attacks**: the attacker takes advantage of normal protocol responses (e.g., error messages, response codes) or their characteristics (e.g., timing differences) to make a legitimate party — in this context called **oracle** — reveal or leak secret data unknowingly, often as a encryption or decryption service. Typically, oracle attacks require multiple interactions;

  > **Example 19.7 — Real-world oracle attacks**
  >
  > A classic example of oracle attacks are **padding oracle attacks**: the attacker sends malformed encrypted messages to a server and observes whether the server returns a padding error; by varying the ciphertext and watching the server's behavior, the attacker can gradually deduce the correct plaintext or encryption key. An instance of a padding oracle attack is the Bleichenbacher attack [51] already discussed in § 16.2.

- **Denial of Service (DoS) attacks**: the attacker aims to violate the availability of a system. In reality, DoS attacks typically do not concern cryptography, except when a cryptographic protocol is (badly) designed and allows the attacker to repeatedly trigger costly cryptographic computations (e.g., by flooding a server with bogus handshake initiations);
- **online guessing attacks**: the attacker conducts guessing attacks by actively interacting with a system, e.g., repeatedly attempting to authenticate using a different password or to decrypt some data by guessing a decryption key. Intuitively, online guessing attacks are relatively easy to detect and mitigate (e.g., with lockout policies and rate limiting — which however may yield DoS attacks);
- **offline guessing attacks**: the attacker obtains cryptographically-protected data (e.g., an encrypted file, a password hash) and then attempts to break the cryptographic protection offline via a trial-and-error approach. Offline guessing attacks are often regarded as passive attacks and are effective against cryptographic protocols employing low-entropy sources (§ 19.5.4). Brute force, dictionary, and rainbow table attacks are instances of offline guessing attacks (§ 16.3);
- **side-channel attacks**: the attacker relies on leakage of cryptographic computations to infer (portions of) secret data such as seeds, private keys, and internal states. In cryptographic protocols, the most relevant side-channel attacks are remote timing attacks, where the attacker measures how long a party takes to respond (or differences in response times) to infer secret data. Side-channel attacks are thoroughly discussed in § 17.1.1;
- **"harvest now, decrypt later" attacks**: the attacker records encrypted data hoping to decrypt it in the future when more computational power will likely be available (see the Moore's law in § 16.3 and, especially, pos-quantum cryptography in § 19.5.2). Note that in the context of "harvest now, decrypt later" attacks, forward secrecy does not help in protecting future data.

- **Key Compromise Impersonation (KCI) attacks**: the KCI attack assumes that an attacker was able to compromise the long-term private key of a party used in key establishment (e.g., such as Diffie-Hellman (DH) — see §9). A natural consequence is that now the attacker can pretend to be that party when communicating with everyone else. However, in cryptographic protocols suffering from KCI attacks, the attacker can also pretend to be anyone else when communicating with that party (due to the symmetric nature of DH). Preventing KCI attacks generally involves either an interactive authenticity protocol or digitally signing messages (but forcing then non-repudiation, which may not be a desired security property). A cryptographic protocol suffering from KCI attacks is Signal;
- **Unknown Key-Share (UKS) attacks**: the attacker manages to trick one party A into believing to have established a shared secret key with another party B while in reality A established the key with a third party C. Instead, B correctly believes to have established the key with A. Usually, a UKS attack does not allow the attacker to learn the key; still, the attacker can abuse A's misassociation. UKS attacks are typically possible when parties' identities are included neither in the KDF used to obtain the shared secret key nor in key confirmation messages;
- **Truncation attacks**: the attacker drops an arbitrary amount of messages without the communicating parties noticing it.

> **Example 19.8 — A real-world truncation attack**
>
> A recent example of truncation attack is the Secure Shell (SSH) (§ 19.3.3) **Terrapin attack** [35]:[a] by carefully adjusting the sequence numbers during the handshake, the attacker can remove an arbitrary amount of messages sent by the client or server at the beginning of the secure channel, downgrading the connection's security by truncating the extension negotiation message (RFC8308 [44]).
>
> [a] Terrapin Attack (https://terrapin-attack.com/)

As a final remark, note that it is the threat model that identifies which subset of the aforementioned attack categories is relevant for a given cryptographic protocol.

19.2 Security of Protocols

Following the best practices described in § 19.1.1 and watching out for the attacks listed in § 19.1.2 is not enough to claim that the design of a cryptographic protocol is secure: a security analysis is essential in that sense. The security analysis of cryptographic protocols includes (Figure 19):

- **formal methods**: mathematically rigorous and usually tool-supported, formal methods use models formalizing participants, messages, and attackers to prove security properties or find attacks by conducting exhaustive reasoning, or even pen-and-paper proofs (typically game-based proofs written in prose);
- **informal methods**: expert-driven and heuristic-based, informal methods rely on manual reasoning, peer review, and empirical testing.

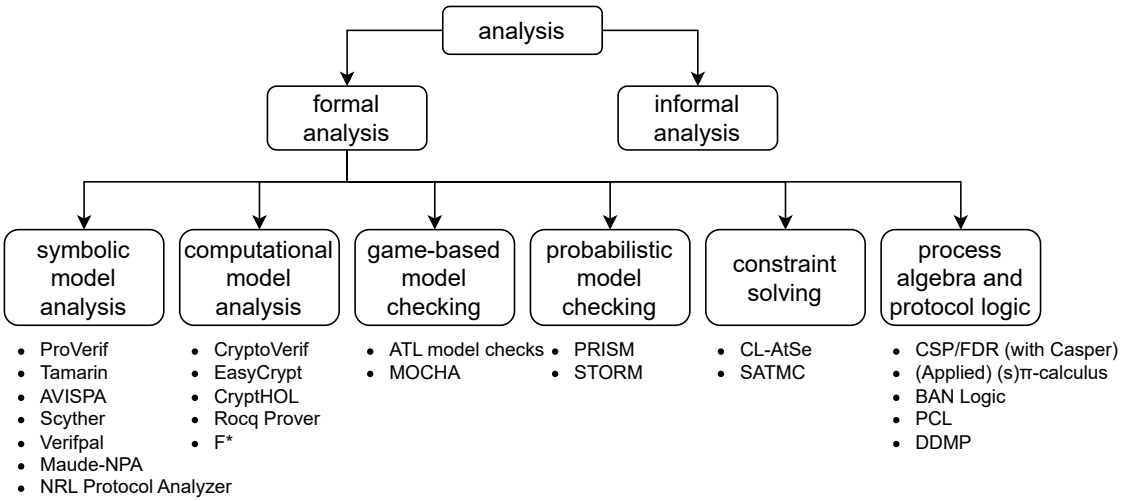

Figure 19: A (probably incomplete) taxonomy of methods for cryptographic protocols security analysis

Although being useful — especially in early design — informal methods lack the coverage guarantees of formal verification. Hence, formal methods are usually preferred (although complementing them with informal methods is often a good idea). For this reason, below we focus on describing the major categories of formal methods for the security analysis of cryptographic protocols (Table 7 for a high-level comparison). Note that the boundaries among the following categories may be blurred, hence a specific method may belong to multiple categories. In particular, symbolic and computational model analysis may be seen as macro-categories.

Symbolic Model Analysis. This (macro)category abstracts messages and cryptographic algorithms as symbolic terms that an attacker (usually following the Dolev-Yao threat model — see § 19.1) can manipulate but not break. In other words, symbolic model analysis is **aimed specifically at cryptographic protocols**, not cryptographic algorithms, which instead are treated as black boxes where only some properties (e.g., correctness — see § 1.2) are preserved. For instance, symmetric encryption (§6) is often modeled with two functions $\texttt{enc}(m, k)$ and $\texttt{dec}(c, k)$ where the former stands for the encryption of a message m with a key k, the latter stands for the decryption of a ciphertext c with a key k, and the following equality holds $\texttt{dec}(\texttt{enc}(m, k), k) = m$.

> **Remark 19.4 — Limits of symbolic model analysis**
>
> Symbolic model analysis is suited for finding logical flaws but **cannot capture** probabilistic or complexity-theoretic aspects (e.g., brute force or cryptanalysis — see § 16.3) which instead are the focus of computational model analysis.

In symbolic model analysis, a cryptographic protocol is usually modeled through a specific language, and the resulting model is analyzed by **tools that implement automated reasoning** to search for attacks by considering all[92] possible (multiple and parallel) protocol runs, messages and states,[93] and interleavings;

[92] Modern tools (e.g., ProVerif, Tamarin) use special algorithms to efficiently handle infinite state spaces — but their execution may take a lot of time.

[93] A **state** is like a snapshot of the currently active protocol runs (including which step of their execu-

the final goal is to find runs (also called **traces**) that contradict security properties.

Remark 19.5 — State explosion

A common problem in symbolic model analysis is **state explosion**, that is, the rapid (and often combinatorial) growth of the number of states due to multiple/parallel/interleaved protocol runs. In other words:

- symbolic model analysis tools typically allow for an unbounded (or very large) number of protocol runs;
- each run creates new symbols (e.g., fresh nonces, session identifiers, and keys) that can be arbitrarily interleaved with symbols of other runs and with the attacker's actions (e.g., replaying, forging, delaying of messages);
- even a small number of protocol runs (<10) can produce millions of states due to interleavings and message variations.

As a result, symbolic model analysis may take too long or even never finish. Indeed, **the correctness of security protocols is an undecidable problem**.

Example 19.9 — Tools for symbolic model analysis

Major tools for symbolic model analysis are:

- **ProVerif** [49]:[a] widely used and state-of-the-art, supports several security properties;
- **Tamarin** [131]:[b] widely used and state-of-the-art, supports automated search and interactive proof steps, natively handles advanced cryptographic equational theories and global state;
- **Verifpal** [113]:[c] a new(er) tool aimed at students and beginners, provides an intuitive modeling language emphasizing usability to reduce user modeling errors while still performing rigorous analysis, and can export models to ProVerif for cross-verification;
- **AVISPA** [18]:[d] provides a common input language (HLPSL) with many back-end analyzers such as OFMC, CL-AtSe, and SATMC (historically relevant, not the state-of-the-art anymore);
- **Scyther** [71]:[e] provides efficient attack discovery through pattern-based state-space search (historically relevant, not the state-of-the-art anymore).

Other tools not particularly relevant anymore but still worth mentioning are **Maude-NPA** and the **NRL Protocol Analyzer**.

[a] ProVerif (https://bblanche.gitlabpages.inria.fr/proverif/)
[b] Tamarin prover: Home (https://tamarin-prover.com/)
[c] Verifpal: Cryptographic Protocol Analysis for Students and Engineers (https://verifpal.com/)
[d] AVISPA (https://avispa-project.org/)
[e] The Scyther Tool | Cas Cremers (https://people.cispa.io/cas.cremers/scyther/)

Computational Model Analysis. This (macro)category uses complexity theory and probabilistic reasoning to prove that any attacker with feasible resources has only a negligible probability of violating

tion they reached and associated symbols), messages in transit, and the attacker's knowledge (e.g., keys, nonces, ciphertexts).

the security properties of the protocol under stated assumptions. Hence, differently from symbolic model analysis, messages are strings of bits and cryptographic algorithms (which, consequently, are functions from strings of bits to strings of bits) are not unbreakable and can only resist polynomial-time attackers under certain stated hardness assumptions (e.g., the DL problem — see § 3.3).[94] As a result, computational model analysis is typically more complex, detailed, and mathematically intensive than symbolic model analysis, albeit it offers stronger security guarantees (i.e., real cryptographic security, not just logical correctness) and thus is often used by cryptographers [50].

In computational model analysis, security proofs for cryptographic protocols are usually based either on games or on simulations:

- **game-based proofs** model security properties as an indistinguishability (§ 16.2.1) or win/lose games which — via a series of game transformations — are reduced to a known hard problem (e.g., the DL problem);
- **simulation-based proofs** typically consist in defining an ideal functionality (that is, a specification that models perfect security) and then demonstrating that the real cryptographic protocol simulates the ideal functionality even in the presence of an attacker. A preeminent framework in this context is Universal Composability (UC),[95] which provides strong composition guarantees: protocols proven UC-secure remain secure even when composed with other protocols in arbitrary ways [61]. In the UC framework, the attacker has capabilities similar to the Canetti-Krawczyk attacker, although it is often considered to be more powerful due to its capability of performing arbitrary protocol compositions and concurrent executions. Finally, proving the UC security of a cryptographic protocol is very complex.

Example 19.10 — Tools for computational model analysis

Major tools for computational model analysis are:

- **CryptoVerif** (*game-based*) [48]:[a] widely used, reduces security goals of cryptographic protocols to underlying mathematical problem (§3), may require guidance for complex protocols;
- **EasyCrypt** (*game-based*) [33]:[b] supersedes CertiCrypt,[c] widely used, is an interactive proof assistant (i.e., less automated than CryptoVerif) tailored for cryptographic proofs, supports probability distributions, failure events, oracles, and attacker definitions, and can interact with Satisfiability Modulo Theories (SMT) solvers (e.g., Coq) for some subgoals;
- **CryptHOL** (*game-based*) [126]:[d] new interactive proof assistant specific for Isabelle/HOL.

Automation for UC simulation-based proofs is not mature yet. Progress has been modest and often limited to specific cases, e.g., with generic proof assistants such as the Rocq Prover[e] — formerly known as Coq — Isabelle/HOL,[f] and F*,[g] which are highly rigorous but require significant manual effort and expertise.

[a] CryptoVerif (`https://bblanche.gitlabpages.inria.fr/CryptoVerif/`)

[94] In computational model analysis, the attacker is usually modeled as a probabilistic polynomial-time Turing machine.

[95] "Universally composable" security — often written as UC security — is the property that a given protocol satisfies within the UC framework. In other words, UC security is what one proves about a protocol using the UC framework.

[b] EasyCrypt (https://www.easycrypt.info/)
[c] EasyCrypt/certicrypt: CertiCrypt Coq Framework (https://github.com/EasyCrypt/certicrypt)
[d] CryptHOL - Archive of Formal Proofs (https://www.isa-afp.org/entries/CryptHOL.html)
[e] Welcome to a World of Rocq (https://rocq-prover.org/)
[f] Isabelle (https://isabelle.in.tum.de/)
[g] F*: A Proof-Oriented Programming Language (https://fstar-lang.org/)

Game-Based Model Checking. This category models protocol analysis as a two-player game on a state graph. In this case, "game-based" refers to modeling a cryptographic protocol and the attacker within a state-transition system, then using temporal logic model checking to verify strategy-dependent security properties (e.g., fairness, timing) expressed as game objectives.

Example 19.11 — Tools for game-based model checking

An influential formalism for game-based model checking tools is Alternating-time Temporal Logic (ATL) [12], while a notable tool is **MOCHA** [11].[a]

[a] MTC (Models and Theory of Computation): Mocha Project (https://ptolemy.berkeley.edu/projects/embedded/research/mocha/)

Probabilistic Model Checking. This category is an extension of traditional model checking for probabilistic state machines or Markov decision processes, and consists in quantifying probabilities in finite or parameterized models for anonymity, randomness, or performance-security trade-offs.

Remark 19.6 — Probabilistic model checking for cryptographic protocols

Probabilistic model checking of cryptographic protocols usually requires abstracting cryptography to small probability events (since real cryptographic probabilities are negligible but not exactly zero). For this reason, probabilistic model checking is more common in the analysis of privacy protocols (onion routing, mixing networks) and quantum cryptography protocols, where randomness and probability are intrinsic and the properties (like "anonymity level" or "key distribution success rate") are quantitative.

Example 19.12 — Tools for probabilistic model checking

Notable tools for probabilistic model checking are **PRISM** [121][a] and **Storm** [91].[b]

[a] PRISM - Probabilistic Symbolic Model Checker (https://www.prismmodelchecker.org/)
[b] Storm – A Modern Probabilistic Model Checker – Home (https://www.stormchecker.org/)

Constraint Solving. This category consists in reducing security analysis to logical constraints analysis: rather than exploring all traces, constraint solving encodes the existence of an attack trace as a satisfiability problem. Constraint solving is often an integral part of automated analyzers (e.g., AVISPA, Tamarin).

Example 19.13 — Tools for constraint solving

Notable tools for constraint solving are **CL-AtSe** [185] and **SATMC** [19].[a]

[a] SATMC - Security & Trust (https://st.fbk.eu/tools/SATMC/)

Process Algebra and Protocol Logic. This category relies on formal languages to model and analyze protocols: although representing a concept on its own, **process algebra**[96] and protocol logic are often used as a modeling foundation for automated analyzers. Process algebra describes concurrent interactions and message flows, while protocol logic uses logical systems tailored to security.

Example 19.14 — Formalisms for process algebra and protocol logic

An influential formalism for process algebra is π-calculus (and its extensions applied π-calculus and spi-calculus[a] tailored for cryptographic protocols [6]), while an influential formalism for protocol logic is the Burrows-Abadi-Needham (BAN) logic (more popular in the past) [58] and the Protocol Composition Logic (PCL) [75].

[a] The Spi Calculus (https://piazza.com/class_profile/get_resource/hzdnubyxvfx3sg/i3rz8bcdpsdwi)

Suggested Reading 19.2 — More information on π-calculus

Jeannette M. Wing's document "FAQ on π-Calculus" — available at the link https://www.cs.tufts.edu/~nr/cs257/archive/jeannette-wing/pi.pdf — is a good primer on π-calculus.

The aforementioned categories often complement, generalize, and build upon each other (e.g., tools for symbolic methods are often based on constraint solving). More importantly, their application is not limited to the security analysis of cryptographic protocols; in that regard, **symbolic methods** are the easiest and excel at rapid automated analysis under idealized assumptions, while **computational methods and process algebra** provide higher assurance with greater effort. Specialized formal methods (e.g., game-based model checking, probabilistic verification, and protocol logic) address niches that general tools might not cover (e.g., fairness, anonymity).

Warning 19.3 — Important limitations of formal methods

Formal methods often produce models of cryptographic protocols and, by definition, models abstract many real-world issues. Hence, important concerns in applied cryptography (e.g., key management, side-channels attacks, programming languages, TCB — see §§ 13, 17 and 18), which lie outside the scope of formal methods, should still be considered and addressed properly.

Suggested Reading 19.3 — More information on techniques for verification of cryptographic protocols

The article "Security Protocol Verification: Symbolic and Computational Models" [50] provides a survey of techniques for analyzing the security of cryptographic protocols with symbolic and

[96] In general, an algebra is a mathematical structure including a set of values, a set of operations on these values, and possibly algebraic properties such as commutativity and associativity. In a process algebra, processes are values and parallel composition is a commutative and associative operation on processes — see https://www.cs.tufts.edu/~nr/cs257/archive/jeannette-wing/pi.pdf.

computational model analysis as well as verifying the security of protocols implementations.

Table 7: High-level comparison of formal methods categories for investigating the security of cryptographic protocols. Please note that the table is a simplification and the information it contains may be approximate and surely not comprehensive

category	Automation	Scalability	Expressiveness	Most suitable for	Adoption
symbolic	high	high	logic (symbolic)	design-stage	widespread
computational (game)	medium/low	medium/low	hardness assumptions	high assurance	academic
computational (UC)	low	low	extremely expressive	protocol composition	niche
game model checking	high	medium	strategic properties	fairness	niche
prob. model checking	high	medium	probabilistic events	quantitative security	niche in academic
constraint solving	high	high	algebraic properties	attack discovery	widespread
process algebra and protocol logic	medium (may vary)	medium	extremely expressive (may vary)	refinement and sub-protocol analysis	academic

Notes on

FUTURE CONTENT

last revision: 15 Sep 2025

Future versions of this handbook — or a new different handbook — will hopefully include content on the following topics (and more).

19.2.1 Identity-based Cryptography

19.2.2 Ciphers for Identity-based Cryptography

19.2.3 Boneh—Franklin

19.2.4 Attribute-based Encryption

19.2.5 Certificateless Cryptography

19.2.6 Functional Encryption

19.2.7 Homomorphic Encryption

19.2.8 Secure Multi-Party Computation

19.2.9 Searchable Encryption

19.3 Popular Cryptographic Protocols

19.3.1 Transport Layer Security

19.3.2 The Signal Protocol

19.3.3 The Secure Shell Protocol

19.4 Popular Applications of Cryptography

19.4.1 Blockchain

19.5 Other Topics

19.5.1 Re-Encryption

19.5.2 Post-Quantum Cryptography

19.5.3 Secret Sharing and Threshold Schemes

19.5.4 Randomness

19.5.5 Lattice-based Cryptography

19.5.6 Cryptographic Accumulators

19.5.7 Cryptographic Ratchets

19.5.8 Access Control

19.5.9 Post-Quantum Cryptographic Hash Functions

20

REFERENCES

[1] *Flush+Flush: A Fast and Stealthy Cache Attack*, pages 279–299. Springer International Publishing. ISSN: 0302-9743, 1611-3349.

[2] *Guide to Elliptic Curve Cryptography*. Springer Professional Computing. Springer-Verlag.

[3] *On Committing Authenticated-Encryption*, pages 275–294. Springer Nature Switzerland. ISSN: 0302-9743, 1611-3349.

[4] *Power Analysis Attacks*. Springer US.

[5] IEEE standard specifications for public-key cryptography - amendment 1: Additional techniques, 2004. ISBN: 9780738140049.

[6] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47. ACM.

[7] Advancing Software Security in the EU. Standard, ENISA, Athens, GR, April 2020.

[8] Paul C. van Oorschot Alfred J. Menezes and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 5 edition, 1996.

[9] Abdullah Aljuffri, Marc Zwalua, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, and Mottaqiallah Taouil. Applying thermal side-channel attacks on asymmetric cryptography. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(11):1930–1942.

[10] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association.

[11] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427, pages 521–525. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[12] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713.

[13] American National Standards Institute. Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm, 1998. Reaffirmed 2005.

[14] American National Standards Institute. Public key cryptography for the financial services industry - key agreement and key transport using elliptic curve cryptography, 2001. Reaffirmed 2017.

[15] American National Standards Institute. Symmetric key cryptography for the financial services industry format preserving encryption - part 1: Definitions and mode, 2023.

[16] Nikolaos Athanasios Anagnostopoulos, Tolga Arul, Markus Rosenstihl, Andre Schaller, Sebastian Gabmeyer, and Stefan Katzenbeisser. Low-temperature data remanence attacks against intrinsic SRAM PUFs. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 581–585. IEEE.

[17] S. Arciszewski. Xchacha: extended-nonce chacha

and aead_xchacha20_poly1305 draft-irtf-cfrg-xchacha-03, 2020.

[18] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. Von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, volume 3576, pages 281–285. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[19] Alessandro Armando, Roberto Carbone, and Luca Compagna. SATMC: A SAT-based model checker for security-critical systems. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413, pages 31–45. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[20] C. Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for AES key retrieval in cache access attacks. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 261–275. IEEE.

[21] OWASP Application Security Verification Standard (ASVS) — v4.0.3. Standard, OWASP Foundation, Inc, Maryland, US, September 2024.

[22] Maria Azees, Pandi Vijayakumar, and Lazarus Jegatha Deboarh. EAAP: Efficient anonymous authentication with conditional privacy-preserving scheme for vehicular ad hoc networks. *IEEE Transactions on Intelligent Transportation Systems*, 18(9):2467–2476, February 2017.

[23] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE symposium on security and privacy (SP)*, pages 777–795. IEEE, 2021.

[24] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076.

[25] Elaine Barker. Recommendation for key management: part 1 - general, 2020.

[26] Elaine Barker and William C Barker. Recommendation for key management:: part 2 – best practices for key management organizations, 2019.

[27] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography, 2018.

[28] Elaine Barker, Allen Roginsky, and Richard Davis. Recommendation for cryptographic key generation, 2020.

[29] Elaine B. Barker and Quynh H. Dang. Recommendation for key management part 3: Application-specific key management guidance, 2015.

[30] Elaine B. Barker and John M. Kelsey. Recommendation for random number generation using deterministic random bit generators, 2015.

[31] Elaine B. Barker, Miles Smid, and Dennis Branstad. A profile for u. s. federal cryptographic key management systems, 2015.

[32] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and Francois-Xavier Standaert. maskVerif: Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, volume 11735, pages 300–318. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[33] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604, pages 146–166. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[34] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841, pages 71–90. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[35] Fabian Baumer, Marcus Brinkmann, and Jörg Schwenk. Terrapin attack: Breaking SSH channel integrity by sequence number manipulation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7463–7480, Philadelphia, PA, August 2024. USENIX Association.

[36] Mihir Bellare, Viet Tung Hoang, and Stefano Tessaro. Message-recovery attacks on feistel-based format preserving encryption. In *Proceedings of the 2016*

ACM SIGSAC Conference on Computer and Communications Security, pages 444–455. ACM, 2016.

[37] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, volume 1976, pages 531–545. Springer Berlin Heidelberg, 2000. Series Title: Lecture Notes in Computer Science.

[38] Mihir Bellare, Phillip Rogaway, and Terence Spies. The ffx mode of operation for format-preserving encryption, February 2010.

[39] Daniel J. Bernstein. The salsa20 stream cipher, slides of talk at ecrypt stvl workshop on symmetric key encryption, 2005.

[40] Daniel J. Bernstein. Chacha, a variant of salsa20. page 6, 01 2008.

[41] Daniel J. Bernstein. Extending the salsa 20 nonce. page 14, 2008.

[42] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. Edwards-curve digital signature algorithm (eddsa), January 2017.

[43] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Everest: Towards a verified, drop-in replacement of HTTPS. *LIPIcs, Volume 71, SNAPL 2017*, 71:1:1–1:12.

[44] D. Bider. Extension Negotiation in the Secure Shell (SSH) Protocol, March 2018.

[45] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1294, pages 513–525. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[46] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 292–302, 2016.

[47] John Black and Phillip Rogaway. Ciphers with arbitrary finite domains. In Bart Preneel, editor, *Topics in Cryptology — CT-RSA 2002*, volume 2271, pages 114–130. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[48] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207.

[49] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, pages 82–96. IEEE.

[50] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, volume 7215, pages 3–29. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[51] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1462, pages 1–12. Springer Berlin Heidelberg, 1998. Series Title: Lecture Notes in Computer Science.

[52] Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries. Version Number: 1.

[53] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, May 2008.

[54] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: Optimizing voltage fault injection attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 199–224.

[55] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10:113122–113130.

[56] David Brumley and Dan Boneh. Remote timing attacks are practical. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[57] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. SoK: Design tools for side-channel-aware implementations. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 756–770. ACM.

[58] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36.

[59] Heiko Bühler, Andreas Walz, and Axel Sikora. Benchmarking of symmetric cryptographic algorithms on

a deeply embedded system. *IFAC-PapersOnLine*, 55(4):266–271, 2022. 17th IFAC Conference on Programmable Devices and Embedded Systems PDES 2022 — Sarajevo, Bosnia and Herzegovina, 17-19 May 2022.

[60] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[61] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[62] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, volume 2332, pages 337–351. Springer Berlin Heidelberg, May 2002.

[63] Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. Photonic side channel attacks against RSA. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 74–78. IEEE.

[64] Lily Chen. Considerations for achieving cryptographic agility: Strategies and practices.

[65] Lily Chen, Dustin Moody, Andrew Regenscheid, Angela Robinson, and Karen Randall. Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters.

[66] Tinghung Chiu and Wenjie Xiong. SoK: Fault injection attacks on cryptosystems. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 64–72. ACM.

[67] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. A white-box des implementation for drm applications. In *ACM Workshop on Digital Rights Management*, pages 1–15. Springer, 2002.

[68] Institute Computer. Guidelines for implementing and using the nbs data encryption standard, 1981-04-01 1981.

[69] Ricardo Corin and Felipe Andrés Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, volume 6542, pages 58–72. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[70] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[71] Cas J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123, pages 414–418. Springer Berlin Heidelberg. ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science.

[72] Roland Croft, Yongzheng Xie, Mansooreh Zahedi, M. Ali Babar, and Christoph Treude. An empirical study of developers' discussions about security challenges of different programming languages. *Empirical Software Engineering*, 27(1):27.

[73] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security*, 12(3):1–29.

[74] Debayan Das, Anupam Golder, Josef Danial, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. X-deepsca: Cross-device deep learning side channel attack. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[75] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358.

[76] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, P. Orsatelli, Philippe Maurine, and Assia Tria. Injection of transient faults using electromagnetic pulses Practical results on a cryptographic system, 2012. Journal of Cryptology ePrint Archive: Report 2012/123.

[77] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. The role of the adversary model in applied security research. *Computers & Security*, 81:156–181.

[78] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[79] Doug Egan, Ashish Kurmi, Paul Rich, Michael Roza, and Mike Schrock. Key management in cloud services: Understanding encryption's desired outcomes and limitations, 2020.

[80] Jean-Max Dutertre, Jacques J.A. Fournier, Amir-Pasha Mirbaha, David Naccache, Jean-Baptiste Rigaud, Bruno Robisson, and Assia Tria. Review of fault injection mechanisms and consequences on

countermeasures design. In *2011 6th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE.

[81] Morris Dworkin. Recommendation for block cipher modes of operation: Methods for format-preserving encryption, 2025.

[82] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[83] Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. An on-chip glitchy-clock generator for testing fault injection attacks. *Journal of Cryptographic Engineering*, 1(4):265–270, 2011.

[84] J. Ferrigno and M. Hlaváč. When AES blinks: introducing optical side channel. *IET Information Security*, 2(3):94–98.

[85] T. Freeman, R. Housley, A. Malpani, D. Cooper, and W. Polk. Server-Based Certificate Validation Protocol (SCVP), December 2007.

[86] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic cryptanalysis. *Journal of Cryptology*, 30(2):392–443.

[87] Anupam Golder, Debayan Das, Josef Danial, Santosh Ghosh, Shreyas Sen, and Arijit Raychowdhury. Practical approaches toward deep-learning-based cross-device power side-channel attack. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2720–2733.

[88] Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, volume 1666, pages 398–412. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[89] Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, volume 1666, pages 388–397. Springer Berlin Heidelberg, 1999. Series Title: Lecture Notes in Computer Science.

[90] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98.

[91] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *International Journal on Software Tools for Technology Transfer*, 24(4):589–610.

[92] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Applications of machine learning techniques in side-channel attacks: a survey. *Journal of Cryptographic Engineering*, 10(2):135–162.

[93] Viet Tung Hoang, Stefano Tessaro, and Ni Trieu. The curse of small domains: New attacks on format-preserving encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, volume 10991, pages 221–251. Springer International Publishing, 2018. Series Title: Lecture Notes in Computer Science.

[94] Vincent C Hu. Overview and considerations of access control based on attribute encryption.

[95] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications*, volume 8419, pages 219–235. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[96] IEEE Computer Society. Ieee standard specifications for public-key cryptography, December 2000. Includes Amendment IEEE Std 1363a-2004.

[97] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688, pages 299–319. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[98] Information security, cybersecurity and privacy protection Evaluation criteria for IT security Part 1: Introduction and general model, 2006.

[99] Information security, cybersecurity and privacy protection Evaluation criteria for IT security Part 1: Introduction and general model, August 2022.

[100] Information technology — Security techniques — Testing methods for the mitigation of non-invasive attack classes against cryptographic moduless. Standard, International Organization for Standardization, Geneva, CH, January 2024.

[101] Information technology - Security techniques - Security requirements for cryptographic modules, August 2012.

[102] IT Security techniques — Test tool requirements and test tool calibration methods for use in testing non-invasive attack mitigation techniques in cryptographic modules. Standard, International Organization for Standardization, Geneva, CH, October 2019.

[103] Programming languages — Avoiding vulnerabilities in programming languages. Standard, International Organization for Standardization, Geneva, CH, October 2024.

[104] Information technology — Security techniques — Application security. Standard, International Organization for Standardization, Geneva, CH, November 2011.

[105] Information technology — Security techniques — Storage security. Standard, International Organization for Standardization, Geneva, CH, 2024.

[106] Joint Task Force Interagency Working Group. Security and privacy controls for information systems and organizations, 2020. Edition: Revision 5.

[107] Ed. K. Moriarty. PKCS #1: RSA Cryptography Specifications Version 2.2, November 2016.

[108] Burt Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0, September 2000.

[109] Tendayi Kamucheka, Michael Fahr, Tristen Teague, Alexander Nelson, David Andrews, and Miaoqing Huang. Power-based side channel attack analysis on PQC algorithms, 2021.

[110] Patrick Gage Kelley, Saranga Komanduri, Michelle L. Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *2012 IEEE Symposium on Security and Privacy*, pages 523–537. IEEE, 2012-05.

[111] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372. IEEE.

[112] Richard Kissel, Andrew Regenscheid, Matthew Scholl, and Kevin Stine. Guidelines for media sanitization.

[113] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *Progress in Cryptology – INDOCRYPT 2020*, volume 12578, pages 151–202. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[114] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[115] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE.

[116] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109, pages 104–113. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[117] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking inaccessible data with software-based power side channels. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7285–7302, Anaheim, CA, August 2023. USENIX Association.

[118] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. Technical report, 1997.

[119] Caroline Kudla and Kenneth G. Paterson. Modular security proofs for key agreement protocols. In Bimal Roy, editor, *Advances in Cryptology - ASIACRYPT 2005*, volume 3788, pages 549–565. Springer Berlin Heidelberg, 2005. Series Title: Lecture Notes in Computer Science.

[120] Shreyas Kumar, Evelyn Croww, and Guofei Gu. Demystifying the perceptions gap between designers and practitioners in two security standards. In *2024 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE.

[121] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806, pages 585–591. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[122] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security, January 2016.

[123] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: an open

framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16. ACM.

[124] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Power analysis attack: an approach based on machine learning. *Int. J. Appl. Cryptol.*, 3(2):97–115, June 2014.

[125] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56.

[126] Andreas Lochbihler. Crypthol. *Archive of Formal Proofs*, May 2017. https://isa-afp.org/entries/CryptHOL.html, Formal proof development.

[127] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys*, 54(6):1–37.

[128] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133.

[129] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 283:10, 2016.

[130] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: grey box' modelling for instruction leakages. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 199–216, USA, 2017. USENIX Association.

[131] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044, pages 696–701. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[132] A. J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press series on discrete mathematics and its applications. CRC Press, 1997.

[133] Alfred Menezes and Douglas Stebila. Challenges in cryptography. *IEEE Security & Privacy*, 19(2):70–73, 2021.

[134] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. Usuba: Optimizing & trustworthy bitslicing compiler. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, pages 1–8. ACM, 2018.

[135] Ralph C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, Stanford, CA, June 1979.

[136] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, volume 218, pages 417–426. Springer Berlin Heidelberg, 1986.

[137] P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[138] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE.

[139] Koksal Mus, Yarkın Doröz, M. Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering TLS signing keys via rowhammer faults. Cryptology ePrint Archive, Paper 2022/1669, 2022.

[140] Antonio Muñoz, Ruben Ríos, Rodrigo Román, and Javier López. A survey on the (in)security of trusted execution environments. *Computers & Security*, 129:103180. Publisher: Elsevier BV.

[141] National Institute of Standards and Technology. Security requirements for cryptographic modules, 2019.

[142] National Institute of Standards and Technology. Digital signature standard (dss), December 2023.

[143] National Institute of Standards and Technology (US). SHA-3 standard : permutation-based hash and extendable-output functions, 2015.

[144] National Institute of Standards and Technology (US). Advanced encryption standard (AES), 2023.

[145] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999.

[146] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols, June 2018.

[147] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Sympo-*

sium (USENIX Security 13), pages 479–498, Washington, D.C., August 2013. USENIX Association.

[148] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860, pages 1–20. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[149] J. Winter P. Wouters, D. Huigens and Y. Niibe. OpenPGP, July 2024.

[150] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, volume 1592, pages 223–238. Springer Berlin Heidelberg, 1999. Series Title: Lecture Notes in Computer Science.

[151] Frank Piessens and Paul C. Van Oorschot. Side-channel attacks: A short tour. *IEEE Security & Privacy*, 22(2):75–80.

[152] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. Fault attacks, injection techniques and tools for simulation. In *2015 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE.

[153] Niels Provos and David Mazieres. Bcrypt algorithm. In *USENIX*, 1999.

[154] Davide Quarta, Michele Ianni, Aravind Machiry, Yanick Fratantonio, Eric Gustafson, Davide Balzarotti, Martina Lindorfer, Giovanni Vigna, and Christopher Kruegel. Tarnhelm: Isolated, transparent & confidential execution of arbitrary code in ARM's TrustZone. In *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks*, pages 43–57. ACM.

[155] D. McCarney R. Barnes, J. Hoffman-Andrews and J. Kasten. Automatic Certificate Management Environment (ACME), March 2019.

[156] M. Tuexen R. Seggelmann and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension, February 2012.

[157] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, August 2018.

[158] Jan Richter-Brockmann, Aein Rezaei Shahmirzadi, Pascal Sasdrich, Amir Moradi, and Tim Güneysu. FIVER – robust verification of countermeasures against fault injections. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 447–473.

[159] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[160] Thomas S. and Ezzy A. Dabbish. Investigations of power analysis attacks on smartcards. In *USENIX Workshop on Smartcard Technology (Smartcard 99)*, Chicago, IL, May 1999. USENIX Association.

[161] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, pages 57–64. IEEE, 2015-08.

[162] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP, June 2013.

[163] Asanka Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics. *Digital Investigation*, 29:43–54.

[164] K A Scarfone, W Jansen, and M Tracy. Guide to general server security, 2008.

[165] K A Scarfone, M P Souppaya, A Cody, and A D Orebaugh. Technical guide to information security testing and assessment. Edition: 0.

[166] Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple photonic emission analysis of AES. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428, pages 41–57. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[167] Jorn-Marc Schmidt and Michael Hutter. Optical and em fault-attacks on crt-based rsa: Concrete results. In *Austrochip 2007*, pages 61–67. Verlag der Technischen Universität Graz, 2007. Austrochip 2007 : 15th Austrian Workhop on Microelectronics ; Conference date: 11-10-2007 Through 11-10-2007.

[168] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking bad: How compilers break constant-time~implementations. Version Number: 1.

[169] Claus-Peter Schnorr. Method for identifying subscribers and for generating and verifying electronic signatures in a data exchange system, February 1991.

Filed February 24 1989; anticipated expiration in Februrary 2010.

[170] Bodo Selmke, Johann Heyszl, and Georg Sigl. Attack on a DFA protected AES by simultaneous laser fault injections. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 36–46. IEEE.

[171] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Comput. Soc. Press, 1994.

[172] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE.

[173] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523, pages 2–12. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[174] Jerome Solinas and David E. Fu. Elliptic Curve Groups modulo a Prime (ECP Groups) for IKE and IKEv2 , June 2010.

[175] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. Secure software development framework (SSDF) version 1.1 : recommendations for mitigating the risk of software vulnerabilities.

[176] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: A case study for mobile devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488.

[177] Theresa Stadler and Carmela Troncoso. Why the search for a privacy-preserving data sharing mechanism is failing. *Nature Computational Science*, 2(4):208–210. Publisher: Springer Science and Business Media LLC.

[178] David Temoshok. Digital identity guidelines, 2022.

[179] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763. Publisher: Association for Computing Machinery (ACM).

[180] Benjamin Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 107–131.

[181] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71.

[182] Michael Tunstall and Marc Joye. Coordinate blinding over large prime fields. In *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12*, pages 443–455. Springer, 2010.

[183] M S Turan, E B Barker, W E Burr, and L Chen. Recommendation for password-based key derivation :: part 1: storage applications, 2010.

[184] Meltem Sönmez Turan, Elaine Barker, John Kelsey, Kerry A McKay, Mary L Baish, and Mike Boyle. Recommendation for the entropy sources used for random bit generation, 2018.

[185] Mathieu Turuani. The CL-atse protocol analyser. In Frank Pfenning, editor, *Term Rewriting and Applications*, volume 4098, pages 277–286. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[186] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328. ACM.

[187] Nikita Veshchikov and Sylvain Guilley. Use of simulators for side-channel analysis. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 104–112. IEEE.

[188] Mohammad Wazid, Ashok Kumar Das, Rasheed Hussain, Giancarlo Succi, and Joel J.P.C. Rodrigues. Authentication in cloud-driven IoT-based big data environment: Survey and outlook. *Journal of Systems Architecture*, 97:185–196, August 2019.

[189] Steve H. Weingart. Physical security devices for computer subsystems: A survey of attacks and defenses. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965, pages 302–317. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[190] N. Williams. On the Use of Channel Bindings to Secure Channels, November 2007.

[191] OWASP Web Security Testing Guide (WSTG) — v4.2. Standard, OWASP Foundation, Inc, Maryland, US, December 2024.

[192] A. Langley Y. Nir. ChaCha20 and Poly1305 for IETF Protocols, May 2015.

[193] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD:

A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[194] Adam Young and Moti Yung. The dark side of "black-box" cryptography or: Should we trust capstone? In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109, pages 89–103. Springer Berlin Heidelberg, 1996. Series Title: Lecture Notes in Computer Science.

[195] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based fault injection using selective state reset. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1135–1151, Philadelphia, PA, August 2024. USENIX Association.

A

ACRONYMS

ispc Intel SPMD Program Compiler

3DH Triple Diffie-Hellman

ABAC Attribute-Based Access Control

ABE Attribute-based Encryption

AC Access Control

ACIR Abstract Circuit Intermediate Representation

ACME Automatic Certificate Management Environment

ACN National Cybersecurity Agency

ADH Authenticated Diffie-Hellman

AE Authenticated Encryption

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

AIR Algebraic Intermediate Representations

AKEM Authenticated Key Encapsulation Mechanism

ANNSI Agence nationale de sécurité des systèmes d'information

ANSI American National Standards Institute

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ATKEM Authenticated Tag Key Encapsulation Mechanism

ATL Alternating-time Temporal Logic

AWS Amazon Web Services

BAN Burrows-Abadi-Needham

BLS Boneh-Lynn-Shacham

BSI Federal Office for Information Security

BYOK Bring-Your-Own-Key

CA Certificate Authority

CAC Cryptographic Access Control

CAVP Cryptographic Algorithm Validation Program

CBC Cipher Block Chaining

CCA Chosen-Ciphertext Attack

CCA2 Adaptive Chosen-Ciphertext Attack

ccDEM compactly committing Data Encapsulation Mechanism

CCM Counter with CBC-MAC

CFB Cipher Feedback

CHF Cryptographic Hash Function

CISC Complex Instruction Set Computer

CKL Compromised Key List

CKMS-SP Cryptographic Key Management System Security Policy

CL-PKC Certificateless Public-Key Cryptography

CLA Cloud Security Alliance

CMAC Cipher-based Message Authentication Code

COA Ciphertext-Only Attack

CP-ABE Ciphertext-Policy Attribute-Based Encryption

CPA Chosen-Plaintext Attack

CPA2 Adaptive Chosen-Plaintext Attack

CR Composite Residuosity

CRL Certificate Revocation List

CRS Common Reference String

CTF Captuer the Flag

CTR Counter

CYOK Control-Your-Own-Key

DEK Data Encryption Key

DEM Data Encapsulation Mechanism

DH Diffie-Hellman

DHE Diffie-Hellman Ephemeral

DHETM Diffie-Hellman Encrypt-then-MAC

DHIES Diffie-Hellman Integrated Encryption Scheme

DL Discrete Logarithm

DLIES Discrete Logarithm Integrated Encryption Scheme

DoS Denial of Service

DRL Distributed Revocation List

DRM Digital Rights Management

DSA Digital Signature Algorithm

DSL Domain-Specific Language

DV-NIZKs Designated-Verifier Non-Interactive Zero-Knowledge proofs

EAX Encrypt-Then-Authenticate with Xor

ECB Electronic Codebook

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie-Hellman

ECDHE Elliptic Curve Diffie-Hellman Ephemeral

ECDPL Elliptic Curve Discrete Logarithm Problem

ECDSA Elliptic Curve Digital Signature Algorithm

ECIES Elliptic Curves Integrated Encryption Scheme

ECRYPT European Network of Excellence in Cryptology

EdDSA Edwards-curve Digital Signature Algorithm

EKM External Key Manager

ENISA European Union Agency for Cybersecurity

EPIC Explicitly Parallel Instruction Computing

FIPS Federal Information Processing Standard

FPE Format Preserving Encryption

FPGA Field-Programmable Gate Array

FRI Fast reed soloman Interactive oracle proofs of proximity

GCM Galois/Counter Mode

GDPR General Data Protection Regulation

GIL Global Interpreter Lock

HE Homomorphic Encryption

HIPAA Health Insurance Portability and Accountability Act

HKDF HMAC-based key derivation function

HMAC Hash-based Message Authentication Code

HSM Hardware Security Module

HYOK Hold-Your-Own-Key

IACR International Association for Cryptologic Research

IAM Identity and Access Management

IBE Identity-based Encryption

IDE Integrated Development Environment

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

IES Integrated Encryption Scheme

IETF Internet Engineering Task Force

IF Integer Factorization

IND indistinguishability

IoT Internet of Things

IP Interactive Proof

IPA Inner-Product-based Arguments

ISA Instruction Set Architecture

ISO International Organization for Standardization

IV Initialization Vector

JCA Java Cryptography Architecture

JS JavaScript

JVM Java Virtual Machine

KCI Key Compromise Impersonation

KDF Key Derivation Function

KEK Key Encryption Key

KEM Key Encapsulation Mechanism

KGA Key Generation Authority

KMIP Key Management Interoperability Protocol

KMS Key Management System

KP-ABE Key-Policy Attribute-Based Encryption

KPA Known-Plaintext Attack

LRW Liskov, Rivest, and Wagner

LTO Link Time Optimization

MAC Message Authentication Code

mAKEM Multiple Authenticated Key Encapsulation Mechanism

mATKEM Multiple Authenticated Tag Key Encapsulation Mechanism

MISC Minimal Instruction Set Computer

MitM Man-in-the-Middle

mKEM Multiple Key Encapsulation Mechanism

MPCitH Multi Party Computation-in-the-Head

MPK Main Public Key

MSK Main Secret Key

NIST National Institute of Standards and Technology

NM Non-Malleability

NSA National Security Agency

OASIS Organization for the Advancement of Structured Information Standards

OCB Offset Codebook Mode

OCSP Online Certificate Status Protocol

OFB Output Feedback

OISC One-Instruction Set Computer

OPA Open Policy Agent

OW One-Wayness

OWASP Open Web Application Security Project

PBKDF Password-based Key Derivation Function

PCI-DSS Payment Card Industry Data Security Standard

PCI-DSS Payment Card Industry Data Security Standard

PCL Protocol Composition Logic

PETs Privacy-Enhancing Technologies

PGP Pretty Good Privacy

PIN Personal Identification Number

PKC Public-Key Cryptography

PKCS Public Key Cryptography Standards

PKI Public Key Infrastructure

PoS Proof-of-Stake

PoW Proof-of-Work

PRF Pseudorandomness

PRNG Pseudorandom Number Generator

PUF Physical Unclonable Function

R1CS Rank-1 Constraint System

RA Registration Authority

RBAC Role-Based Access Control

RISC Reduced Instruction Set Computer

RNG Random Number Generator

RSA Rivest-Shamir-Adleman

S3 Simple Storage Service

SaaS Software-as-a-Service

SCVP Server-based Certificate Validation Protocol

SDH Static Diffie-Hellman

SDK Software Development Kit

SE Secure Element

SEV Secure Encrypted Virtualization

SGX Software Guard Extensions

SHA Secure Hash Algorithm

SIMD Single Instruction, Multiple Data

SIV Synthetic Initialization Vector

SLA Service-Level Agreement

SMPC Secure Multi Party Computation

SMT Satisfiability Modulo Theories

SNR Signal-to-Noise Ratio

SPoF Single Point of Failure

SRS Structured Reference String

SSH Secure Shell

TCB Trusted Computing Base

TCG Trusted Computing Group

TDX Trust Domain Extensions

TEE Trusted Execution Environment

TKEM Tag Key Encapsulation Mechanism

TLS Transport Layer Security

TPM Trusted Platform Module

TRNG True Random Number Generator

TSC Transparent Supply Chain

TSS Threshold Secret Sharing

UC Universal Composability

UKS Unknown Key-Share

VA Validation Authority

VLIW Very Long Instruction Word

VM Virtual Machine

VRF Verifiable random function

VXEdDSA Verifiable random function Extended Edwards-curve Digital Signature Algorithm

WoT Web of Trust

X3DH Extended Triple Diffie-Hellman

XACML eXtensible Access Control Markup Language

XEdDSA Extended Edwards-curve Digital Signature Algorithm

XTS XEX-based Tweaked Codebook with Ciphertext Stealing

ZK Zero-Knowledge

ZK-SNARKs Zero-Knowledge Succinct Non-interactive ARguments of Knowledge

ZK-STARKs Zero-Knowledge Scalable Transparent ARguments of Knowledge

ZKP Zero-Knowledge Proof

B

FURTHER RESOURCES

Below, we collect further helpful resources for those who wish to deepen their cryptographic knowledge, in particular books (Appendix B.1), online courses (Appendix B.2), hands-on challenge platforms (Appendix B.3), newsletters and podcasts (Appendix B.4), useful websites and reading guides (Appendix B.5), and blogs (Appendix B.6).

B.1 Books

The following books are ordered by latest edition (most recent first):

- **Serious Cryptography: A Practical Introduction to Modern Encryption** by *Jean-Philippe Aumasson* (latest edition being the 2nd and published on August 2024 with 376 pages): breaking down fundamental math concepts behind ciphers without shying away from how and why they work (and how they can fail), it says to be "a practical guide to the past, present, and future of cryptographic systems and algorithms". Print book and Ebook can be bought at `https://nostarch.com/seriouscrypto` (No Starch Press);
- **Understanding Cryptography: A Textbook for Students and Practitioners** by *Christof Paar, Jan Pelzl, and Tim Guneysu* (latest edition being the 2nd and published on May 2024 with 564 pages): focusing on covering (nearly) all symmetric, asymmetric, and post-quantum cryptographic algorithms, it says to be a "must-have book [] indispensable as a textbook for graduate and advanced undergraduate courses". Print book and Ebook can be bought at `https://www.cryptography-textbook.com/` (Springer);

> **Remark B.1 — Video lectures on Understanding Cryptography**
> Christof Paar's video lectures — available at the link `https://www.youtube.com/@introductiontocryptography4223/videos` — are an excellent companion to the Understanding Cryptography book, mixing theory with practical demos.

- **A Graduate Course in Applied Cryptography** by *Dan Boneh and Victor Shoup* (latest edition being the 0.6th and published on January 2023 with 1130 pages): a rigorous, graduate-level book emphasizing provable security, it says to be "about mathematical modeling and proofs to show that a particular cryptosystem satisfies the security properties attributed to it" and to "describe dozens of cryptographic algorithms, give practical advice on how to implement them, and show how they can be used to solve security problems". PDF freely available at the link https://toc.cryptobook.us/;
- **Real-World Cryptography** by *David Wong* (latest edition being the 1st and published on September 2021 with 400 pages): a hands-on book focused on applying cryptographic techniques in real systems, it says to teach "practical techniques for day-to-day work as a developer, sysadmin, or security practitioner". Print book, Ebook, and Audiobook can be bought at https://www.manning.com/books/real-world-cryptography (Manning Publications Co);
- **Introduction to Modern Cryptography** by *Jonathan Katz and Yehuda Lindell* (latest edition being the 3rd and published on December 2020 with 648 pages): an academic book offering a rigorous treatment of cryptography from a theoretical standpoint focused on formal security models. PDF can be bought at https://www.taylorfrancis.com/books/mono/10.1201/9781351133036/introduction-modern-cryptography-yehuda-lindell-jonathan-katz (CRC Press);
- **Crypto 101** by *Laurens Van Houtven* (latest edition being incomplete and published on 2017 with 223 pages): an open-source book on cryptography, it says to be "an introductory course on cryptography, freely available for programmers of all ages and skill levels" and extensively "focus on breaking cryptography", teaching how common flaws can be exploited. PDF freely available at the link https://www.crypto101.io/.

> **Suggested Reading B.1 — Older books**
>
> The following books are perhaps outdated but surely still worth mentioning:
>
> - Cryptography Engineering: Design Principles and Practical Applications by Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno (latest edition being the 1st and published on March 2010 with 384 pages: Print book and Ebook can be bought at https://www.schneier.com/books/cryptography-engineering/ (Wiley);
> - Handbook of Applied Cryptography by Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone (latest edition being the 5th and published on August 2001 with 810 pages — latest errata update on 2014): PDF freely available at the link https://cacr.uwaterloo.ca/hac/ (CRC Press);
> - Applied Cryptography by Bruce Schneier (latest edition being the 2nd and published on January 1996 with 784 pages): Print book and Ebook can be bought at https://www.schneier.com/books/applied-cryptography/ (John Wiley & Sons).

B.2 Online Courses

The following online courses are ordered alphabetically (note that we include only courses available to everyone; university courses targeting students are not included):

- **Cryptography** by *University of Maryland, Jonathan Katz*: an online course that explains the basis of traditional cryptography. Freely available at the link https://www.coursera.org/learn/cryptography;
- **Cryptography I** by *Stanford University, Dan Boneh*: a very popular online course that provides a broad introduction to cryptographic primitives and how to use them correctly. The course starts from basic symmetric ciphers and works up to public-key cryptography. Later modules examine real protocols and common implementation mistakes, and an optional programming project lets students experiment with breaking a toy system. Freely available at the link https://www.coursera.org/learn/crypto/.

> **Warning B.1 — Cryptography II was removed**
>
> The course Cryptography I used to have a follow up named **Cryptography II** covering more advanced topics like Zero-Knowledge Proofs (ZKPs). However, Cryptography II was recently removed from the online course platform.[a]
>
> [a] Cryptography II by Dan Boneh has been removed from Coursera, any alternative sources? : r/crypto (https://www.reddit.com/r/crypto/comments/1deuruf/cryptography_ii_by_dan_boneh_has_been_removed/)

B.3 Hands-On Challenge Platforms

The following hands-on challenge platforms are ordered alphabetically:

- **CryptoHack**: a gamified platform for learning cryptography by breaking bad implementations of Advanced Encryption Standard (AES), Rivest-Shamir-Adleman (RSA), and Elliptic Curve Cryptography (ECC). The platform is organized as Captuer the Flag (CTF)-style challenges, escalating from `xor` ciphers to lattices. Notably, the platform is active and regularly updated with new challenges. Freely available at the link https://cryptohack.org/;
- **Cryptopals Crypto Challenges**: a famous (fixed) set of programming challenges to demonstrate real-world attacks on cryptographic algorithms (not side-channel attacks, though), spanning from `xor` ciphers and AES in Electronic Codebook (ECB) mode up to RSA and ECC. Freely available at the link https://cryptopals.com/;

> **Suggested Reading B.2 — Other Platforms**
>
> The following platforms are perhaps smaller in scope but surely still worth mentioning:
>
> - id0-rsa, freely available at the link https://id0-rsa.pub/;
> - OverTheWire community's Krypton, freely available at the link https://overthewire.org/wargames/krypton/;
> - HackTheBox Crypto challenges, , freely available at the link https://help.hackthebox.com/en/articles/5185436-how-to-play-challenges.

B.4 Newsletters and Podcasts

The following newsletters and podcasts are ordered alphabetically (newsletters first):

- **Crypto-Gram**: a "free monthly newsletter providing summaries, analyses, insights, and commentaries on security technology" by Bruce Schneier. Despite the name, Crypto-Gram is not only about cryptography, but instead it covers broader cybersecurity. Crypto-Gram often highlights recent cryptographic breaches or debates (e.g., encryption backdoors, algorithm policy) with a balanced and clear analysis. Freely available at the link https://www.schneier.com/crypto-gram/;
- **Cryptography Dispatches**: a newsletter by Filippo Valsorda where each issue is a short essay focusing on a specific topic of applied cryptography or recent development. Started around 2020, this newsletter is valuable for providing digestible "dispatches" from the world of applied cryptography in plain language. Freely available at the link https://buttondown.com/cryptography-dispatches/;
- **Cryptography FM**: a podcast hosted by Nadim Kobeissi and Lúcás Meier offering "in-depth, substantive discussions on the latest news and research in applied cryptography". The latest episode dates back to February 2023. Freely available at the link https://cryptography.fireside.fm/;
- **Security Cryptography Whatever**: a podcast hosted by David Adrian, Deirdre Connolly, and Thomas Ptacek where "some cryptography and security people talk about security, cryptography, and whatever else is happening". Episodes range from deep dives into specific protocols or vulnerabilities to casual chats about industry news. The podcast is active as of 2025. Freely available at the link https://securitycryptographywhatever.com/.

B.5 Useful Websites and Reading Guides

The following websites and guides are ordered alphabetically:

- **Crypto Gotchas!**: a curated list of common mistakes and pitfalls in cryptography by Greg Rubin. The list enumerates concerns like "keys wearing out", poor entropy, and padding oracles attacks. Essentially, Crypto Gotchas! is a checklist of what not to do in applied cryptography, with links to blog posts or papers. Freely available at the link https://gotchas.salusa.dev/;
- **r/crypto**: a cryptography community/wiki on Reddit that serves as a beginner's guide, FAQ, and Q&A. Freely available at the link https://www.reddit.com/r/crypto/wiki/index/;
- **The Stick-Figure Guide to AES**: a famous illustrated blog post published in 2009 by Jeff Moser that provides a "layman's introduction to cryptography and AES". The blog post uses intuitive analogies and even stick-figure cartoons to explain how AES works internally, allowing for understanding AES encryption at a conceptual level (e.g., what are S-boxes and mix-columns) without too many mathematical details. Freely available at the link http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html;
- **Timeline of Cryptography**: a Wikipedia article that provides a chronological timeline of important events in the history of cryptography, from classical ciphers in Roman era up to recent developments. Freely available at the link https://en.wikipedia.org/wiki/Timeline_of_cryptography.

B.6 Blogs

The following blogs are ordered alphabetically:

- **A Few Thoughts on Cryptographic Engineering**: a blog by prof. Matthew Green containing "some random thoughts about crypto. Notes from a course I teach. Pictures of my dachshunds". This blog is among the most respected for applied cryptography, offering long-form explanations of things like how Apple's iPhone encryption works, analyses of protocol designs, critiques of government policies, and timely explainers of new vulnerabilities. Freely available at the link https://blog.cryptographyengineering.com/;
- **Chosen Plaintext's blog**: a blog by Chosen Plaintext Consulting (a security consulting company) which features deeply technical articles on cryptography engineering. Many of the posts focus on secure implementation and cryptographic pitfalls (e.g., "A beginner's guide to constant-time cryptography" and "Cryptography Red Flags"). Almost all posts date back to 2015. Freely available at the link https://www.chosenplaintext.ca/blog/;
- **Dhole Moments**: a blog by Soatok covering software, security, and cryptography from the perspective of an independent researcher. The content is practitioner-oriented; for example, Soatok has articles debunking bad cryptographic advice, explaining secure protocol design, and reviewing the cryptography in messaging apps. Freely available at the link https://soatok.blog/author/soatok/;
- **Kelby Ludwig's blog**: a blog containing technical posts on specific cryptographic topics suc as ECC, the Hidden Number Problem, Elliptic Curve Digital Signature Algorithm (ECDSA) quirks, and lattice-based cryptography. The latest post dates back to March 2021. The blog is not a general beginner blog, but it is great for niche topics. Freely available at the link https://kel.bz/;
- **Latacora's Blog**: a blog by Latacora (a security consulting company) giving practical guidance on cryptography for developers, especially for what concerns the selection of cryptographic algorithms, secure defaults, libraries, and post-quantum cryptography as well. Freely available at the link https://www.latacora.com/blog/;
- **Neil Madden's blog**: "Thoughts on application security, applied crypto, philosophy and logic", often deep diving into cryptographic implementations, protocols, and vulnerabilities in an approachable style — especially for developers. Freely available at the link https://neilmadden.blog/;
- **Ryan Castellucci's blog**: "posts on computer security, programming, systems administration, electronics, and general geekery". The blog has not been frequently updated lately, but the existing content is insightful, especially on topics of practical cryptanalysis. Freely available at the link https://rya.nc/.

> **Suggested Reading B.3 — Other Blogs**
>
> David Wong also used to curate a large list of blogs related to cryptography — available at the link https://github.com/mimoo/crypto_blogs — although the latest update to the list dates back to 2021.