Computer Security
Theory

Christian Rossi

Academic Year 2023-2024

Abstract

The course topics are:

- Introduction to information security.
- A short introduction to cryptography.
- Authentication.
- Authorization and access control.
- Software vulnerabilities.
- Secure networking architectures.
- Malicious software.

Contents

1 Introduction **1**
- 1.1 Basic security requirements . . . 1
- 1.2 Definitions . . . 1
- 1.3 Ethical hacking . . . 2

2 Cryptography **3**
- 2.1 Introduction . . . 3
  - 2.1.1 History . . . 3
  - 2.1.2 Definitions . . . 4
- 2.2 Computational security . . . 5
- 2.3 Pseudorandom number generators . . . 6
  - 2.3.1 Pseudorandom function . . . 7
  - 2.3.2 Pseudorandom permutation . . . 7
  - 2.3.3 Standard block ciphers . . . 8
- 2.4 Plaintext encryption . . . 8
  - 2.4.1 Chosen plaintext attacks . . . 9
- 2.5 Data integrity . . . 10
  - 2.5.1 Message authentication codes . . . 11
  - 2.5.2 Cryptographic hashes . . . 12
- 2.6 Asymmetric cryptosystems . . . 13
  - 2.6.1 Diffie-Hellman key agreement . . . 14
  - 2.6.2 Public key encryption . . . 14
  - 2.6.3 Digital signatures . . . 16
- 2.7 Keys handling . . . 18
  - 2.7.1 Digital certificates . . . 18
  - 2.7.2 Certification authorities . . . 18
- 2.8 State of the art . . . 19
- 2.9 Shannon's information theory . . . 19
  - 2.9.1 Entropy . . . 20
  - 2.9.2 Minimum entropy . . . 20

A The x86 architecture **22**
- A.1 Introduction . . . 22
  - A.1.1 History . . . 22
  - A.1.2 Von Neumann architecture . . . 22
- A.2 Features . . . 23
- A.3 Syntax . . . 24

A.3.1 Basic instructions 24
A.3.2 Conventions 26
A.4 Program layout and functions 26
A.4.1 Stack 27
A.4.2 Functions handling 28

CHAPTER 1

Introduction

1.1 Basic security requirements

The fundamental security principles, known as the CIA paradigm for information security, outline three key requirements:

- *Confidentiality*: only authorized entities can access information.
- *Integrity*: information can only be modified by authorized entities in authorized ways.
- *Availability*: information must be accessible to all authorized parties within specified time limits.

It's worth noting that the availability requirement can sometimes conflict with the other two, as higher availability exposes the system for longer durations.

1.2 Definitions

Definition (*Vulnerability*)**.** A vulnerability is a flaw that can be exploited to violate one of the constraints of the CIA paradigm.

Definition (*Exploit*)**.** An exploit is a specific method of leveraging one or more vulnerabilities to achieve a particular objective that breaches the constraints.

Definition (*Asset*)**.** An asset is anything of value to an organization.

Definition (*Threat*)**.** A threat is a potential event that could lead to a violation of the CIA paradigm.

Definition (*Attack*)**.** An attack is a deliberate use of one or more exploits with the aim of compromising a system's CIA.

Definition (*Threat agent*)**.** A threat agent is any entity or factor capable of causing an attack.

Definition (*Hacker*)**.** A hacker is an individual with advanced knowledge of computers and networks, driven by a strong curiosity and desire to learn.

Definition (*Black hats*)**.** Malicious hackers are commonly referred to as black hats.

1.3 Ethical hacking

White hats, also known as security professionals or ethical hackers, are tasked with:

- *Identifying vulnerabilities.*
- *Developing exploits.*
- *Creating attack-detection methods.*
- *Designing countermeasures against attacks.*
- *Engineering security solutions.*

Since no system is invulnerable, it's crucial to assess its risk level. This involves evaluating the potential damage due to vulnerabilities and threats through the concept of risk:

Definition (*Risk*)**.** Risk is a statistical and economic evaluation of potential damage resulting from the presence of vulnerabilities and threats:

$$\text{Risk} = \text{Asset} \times \text{Vulnerabilities} \times \text{Threats}$$

Assets and vulnerabilities can be managed, but threats are independent variables.

To ensure system security, a balance must be struck between cost and reducing vulnerabilities and containing damage. The costs of securing a system can be categorized as direct and indirect. Direct costs include management, operational, and equipment expenses, while indirect costs, which often form the larger portion, stem from:

- *Reduced usability.*
- *Slower performance.*
- *Decreased privacy* (due to security controls).
- *Lower productivity* (as users may be slower).

It's important to note that simply spending more money on security may not always resolve the issue.

In real-world systems, setting boundaries is essential, meaning that a portion of the system must be assumed as secure. These secure parts consist of trusted elements determined by the system developer or maintainer. For example, the level of trust in a particular system can be determined at the software, compiler, or hardware level.

CHAPTER 2

Cryptography

2.1 Introduction

Definition (*Cryptography*)**.** Cryptography refers to the field of study concerned with developing techniques that enable secure communication and data storage in the presence of potential adversaries.

Cryptography offers several essential features, including:

- *Confidentiality*: ensures that data can only be accessed by authorized entities.
- *Integrity/freshness*: detects or prevents tampering or unauthorized replays of data.
- *Authenticity*: certifies the origin of data and verifies its authenticity.
- *Non-repudiation*: ensures that the creator of data cannot deny their responsibility for creating it.
- *Advanced features*: includes capabilities such as proofs of knowledge or computation.

2.1.1 History

Cryptography has a history as ancient as written communication itself, originating primarily for commercial and military purposes. Initially, cryptographic algorithms were devised and executed manually, using pen and paper.

The early approach to cryptography involved a contest of intellect between cryptographers, who devised methods to obscure messages, and cryptanalysts, who sought to break these ciphers.

A significant development occurred in 1553 when Bellaso pioneered the idea of separating the encryption method from the key.

In 1883, Kerchoff formulated six principles for designing robust ciphers:

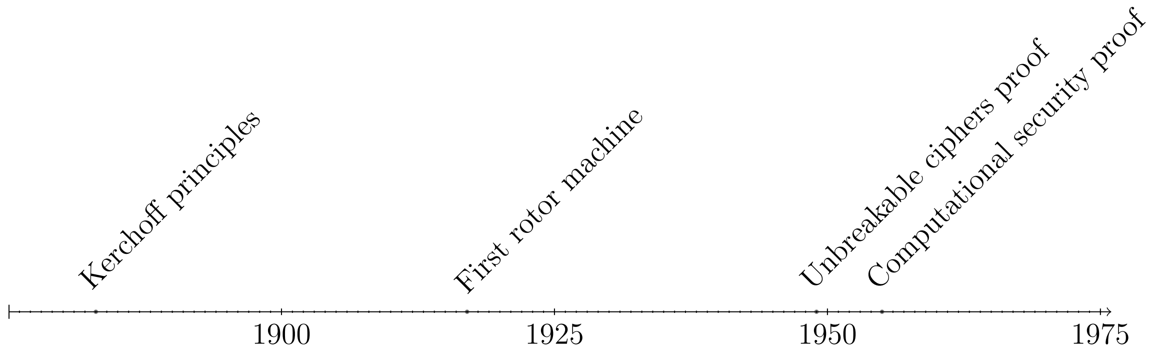
1. The cipher should be practically, if not mathematically, unbreakable.
2. It should be possible to disclose the cipher to the public, including enemies.

3. The key must be communicable without written notes and changeable at the discretion of correspondents.
4. It should be suitable for telegraphic communication.
5. The cipher should be portable and operable by a single person.
6. Considering the operational context, it should be user-friendly, imposing minimal mental burden and requiring a limited set of rules.

The landscape of cryptography underwent a significant transformation in 1917 with the introduction of mechanical computation, exemplified by Hebern's rotor machine, which became commercially available in the 1920s. This technology evolved into the German Enigma machine during World War II, whose encryption methods were eventually deciphered by cryptanalysts at Bletchley Park, contributing significantly to the Allied victory.

After World War II, in 1949 Shannon proved that a mathematically secure ciphers exists.

Following World War II, in 1949, Shannon demonstrated the existence of mathematically secure ciphers. Subsequently, in 1955, Nash proposed the concept of computationally secure ciphers, suggesting that if the interaction of key components in a cipher's determination of ciphertext is sufficiently complex, the effort required for an attacker to break the cipher would grow exponentially with the length of the key ($\mathcal{O}(2^\lambda)$), surpassing the computational capabilities of the key owner ($\mathcal{O}(\lambda^2)$) for sufficiently large key lengths (λ).



2.1.2 Definitions

Definition (*Plaintext space*)**.** A plaintext space P is the set of possible messages $ptx \in P$.

Definition (*Ciphertext space*)**.** A ciphertext space C is the set of possible ciphertext $ctx \in P$.

It's worth noting that the ciphertext space C may have a larger cardinality than the plaintext space P.

Definition (*Key space*)**.** A key space K is the set of possible keys.

The length of the key often correlates with the desired level of security.

Definition (*Encryption function*)**.** An encryption function $\mathbb{E}$ is a mapping that takes an element from the plaintext space P and a key from the key space K, and produces an element from the ciphertext space C:

$$\mathbb{E} : P \times K \to C$$

Definition (*Decryption function*)**.** A decryption function $\mathbb{D}$ is a mapping that takes an element from the ciphertext space C and a key from the key space K, and yields an element from the plaintext space P:

$$\mathbb{D} : C \times K \to P$$

2.2 Computational security

The objective of ensuring confidentiality is to prevent unauthorized individuals from comprehending the data. Various methods can compromise confidentiality:

- Passive interception by an attacker.
- Knowledge of a set of potential plaintexts by the attacker.
- Data manipulation by the attacker to observe the reactions of an entity capable of decryption.

Definition (*Perfect cipher*)**.** In a perfect cipher, for any plaintext ptx in the plaintext space P and any corresponding ciphertext ctx in the ciphertext space C, the probability of the plaintext being sent is equal to the conditional probability of that plaintext given the observed ciphertext:

$$\mathrm{P}(ptx\ sent = ptx) = \mathrm{P}(ptx\ sent = ptx | ctx\ sent = ctx)$$

In other words, observing a ciphertext $c \in C$ provides no information about the corresponding plaintext it represents.

Theorem 2.2.1 (Shannon 1949)**.** *Any symmetric cipher $\langle P, K, C, \mathbb{E}, \mathbb{D} \rangle$ with $|P| = |K| = |C|$, achieves perfect security if and only if every key is utilized with equal probability $\frac{1}{|K|}$, and each plaintext is uniquely mapped to a ciphertext by a unique key:*

$$\forall\,(ptx, ctx) \in P \times C, \exists! k \in K \text{ such that } \mathbb{E}\,(ptx, k) = ctx$$

Example:
Let's consider P, K, and C as sets of binary strings. The encryption function selects a uniformly random, fresh key k from K each time it's invoked and computes the ciphertext as $ctx = ptx \oplus k$.

Gilbert Vernam patented a telegraphic machine in 1919 that implemented $ctx = ptx \oplus k$ using the Baudot code. Joseph Mauborgne proposed utilizing a random tape containing the key k.

Combining Vernam's encryption machine with Mauborgne's approach results in a perfect cipher implementation.

It's crucial to understand that while a cipher may achieve perfect security, this doesn't necessarily mean it's practical or user-friendly. Managing key material and regularly changing keys can be exceptionally challenging.

In practice, perfect ciphers often face vulnerabilities due to issues such as key theft or reuse. Additionally, the generation of truly random keys has historically been problematic, leading to potential vulnerabilities and breaches.

In practical terms, ensuring the security of a cipher involves ensuring that a successful attack would also require solving a computationally difficult problem efficiently. The most commonly utilized computationally hard problems for ciphers include:

- Solving a generic nonlinear Boolean simultaneous equation set.
- Factoring large integers or finding discrete logarithms.

- Decoding a random code or finding the shortest lattice vector.

These problems cannot be solved faster than exponential time. However, with some hints, they can become easier to solve within polynomial time.

At this juncture, proving computational security involves the following steps:

1. Define the ideal attacker's behavior.
2. Assume a specific computational problem is difficult.
3. Prove that any non-ideal attacker would need to solve the difficult problem.

The attacker is typically represented as a program capable of accessing given libraries that implement the cipher in question. The security property is defined as the ability to respond to a specific query. The attacker succeeds if it breaches the security property more frequently than would be possible through random guessing.

2.3 Pseudorandom number generators

To expand the key for use in a Vernam cipher with a finite-length key, we require a pseudo-random number generator (PRNG). We assume that the attacker's computational capability is limited to poly(λ) computations.

Definition (*Cryptographically safe pseudorandom number generators*)**.** A cryptographically secure pseudorandom number generator is a deterministic function:

$$\text{PRNG} : \{0,1\}^{\lambda} \to \{0,1\}^{\lambda+I}$$

where I is an expansion factor, such that the output of the PRNG cannot be distinguished from a uniformly random sample $\{0,1\}^{\lambda+I}$ with computational complexity $\mathcal{O}(\text{poly}(\lambda))$.

In practice, cryptographic pseudo-random number generators (CSPRNGs) are considered as candidates because there is no conclusive evidence supporting the existence of a definitive pseudo-random number generator (PRNG) function. Demonstrating the existence of a CSPRNG would imply $\mathcal{P} \neq \mathcal{NP}$.

Developing a CSPRNG from scratch is feasible but not the usual approach due to inefficiency. Typically, they are constructed using another fundamental element called Pseudorandom Permutations (PRPs), which are derived from PseudoRandom Functions (PRFs).

To randomly select a function, we start by considering the set:

$$F = \left\{ f : \{0,1\}^{in} \to \{0,1\}^{out}, in, out \in \mathbb{N} \right\}$$

A uniformly randomly sampled $f \xleftarrow{\$} F$ can be represented by a table with 2^{in} entries, each entry being *out* bits wide:

$$|F| = \left(2^{out}\right)^{2^{in}}$$

> **Example:**
> For instance, if $in = 2$ and $out = 2$, the function set $F = \{f : \{0,1\}^2 \to \{0,1\}^1\}$ consists of the 16 Boolean functions with two inputs. Each function is represented by a 4-entry truth table. The total number of functions is 16, corresponding to the $2^4 = 16 = \left(2^1\right)^{2^2}$ tables.

2.3.1 Pseudorandom function

Definition (*Pseudorandom function*)**.** A pseudorandom function (PRF) is denoted as:

$$prf_{seed} : \{0,1\}^{in} \to \{0,1\}^{out}$$

Where it takes an input and a λ-bit seed.

Consequently, prf_{seed} is entirely determined by the seed value. It cannot be distinguished from a random function:

$$f \in \left\{f : \{0,1\}^{in} \to \{0,1\}^{out}\right\}$$

within polynomial time in λ. In other words, given $a \in \{f : \{0,1\}^{in} \to \{0,1\}^{out}\}$, it is computationally infeasible to determine which of the following is true:

- $a = prf_{seed}(\cdot)$ with seed $\xleftarrow{\$} \{0,1\}^{\lambda}$.
- $b \xleftarrow{\$} F$, where $F = \{f : \{0,1\}^{in} \to \{0,1\}^{out}\}$.

2.3.2 Pseudorandom permutation

Definition (*Pseudorandom permutation*)**.** A pseudorandom permutation is a bijective pseudorandom function defined as:

$$prf_{seed} : \{0,1\}^{len} \to \{0,1\}^{len}$$

It is characterized solely by its seed value and cannot be distinguished from a random function within poly(λ). This permutation represents a rearrangement of all possible strings of length *len*. In practical terms:

- It operates on a block of bits and yields another block of equal size.
- The output appears unrelated to the input.
- Its behavior is entirely determined by the seed, akin to a key in conventional cryptography.

However, there is no formally proven pseudorandom permutation because its existence would imply $\mathcal{P} \neq \mathcal{NP}$. Construction of such a pseudorandom permutation typically involves three steps:

1. Compute a small bijective Boolean function f with input and key.
2. Compute f again between the previous output and the key.
3. Repeat the second step until satisfaction.

PRP selection Modern Pseudorandom Permutations (PRPs) often emerge from public competitions, where cryptanalytic techniques help identify and eliminate biases in their outputs, ensuring robust designs.

These PRPs are commonly known as block ciphers. A block cipher is considered broken if it can be distinguished from a PRP with less than 2^{λ} operations, achieved by:

- Deriving the input corresponding to an output without knowledge of the key.
- Determining the key identifying the PRP or narrowing down plausible options.

- Detecting non-uniformities in their outputs.

The key length λ is chosen to be sufficiently large to render computing 2^λ guesses impractical. For different security levels:

- Legacy-level security typically employs λ around 80.
- For a security duration of five to ten years, λ is set to 128.
- Long-term security requires λ of 256.

2.3.3 Standard block ciphers

The Advanced Encryption Standard (AES) operates on a 128-bit block size and offers three key lengths: 128, 192, and 256 bits. Chosen as a result of a three-year public competition by NIST on February 2, 2000, AES emerged as the preferred standard out of 15 candidates and has since been standardized by ISO. Modern processor architectures such as ARMv8 and AMD64 include dedicated instructions to accelerate the computation of AES.

The predecessor to AES, known as the Data Encryption Standard (DES), was established by NIST in 1977. DES operated with a relatively short 56-bit key length, leading to security concerns. It was bolstered through triple encryption, effectively achieving an equivalent security level of $\lambda = 112$. Although still present in some legacy systems, DES has been officially deprecated.

2.4 Plaintext encryption

Encrypting plaintexts with a length less than or equal to the block size using a block cipher is effective. This method can be expanded by employing multiple blocks with a split-and-encrypt approach, also known as Electronic CodeBook (ECB) mode.

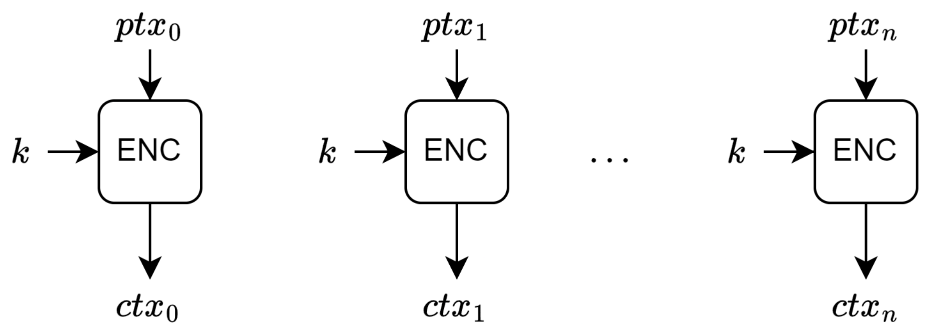


Figure 2.1: ECB encryption mode

However, this technique becomes problematic when there is redundancy within plaintext segments, as the resulting ciphertext may still reveal patterns. This vulnerability arises from the deterministic nature of ECB encryption, where identical plaintext blocks produce identical ciphertext blocks, making it susceptible to certain cryptographic attacks.

To address the issue of pattern visibility in ciphertexts caused by redundancy in plaintext segments, we can employ a counter to differentiate the strings submitted to each block during encryption. This counter, unique for each block, helps mitigate the predictability inherent in traditional encryption modes.

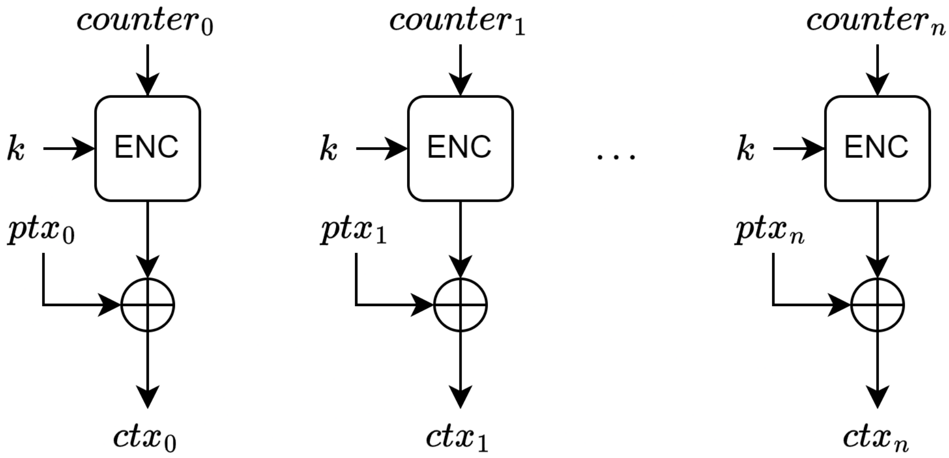


Figure 2.2: CTR encryption mode

This method ensures that even if plaintext blocks are repeated, the resulting ciphertext blocks are different due to the unique counter values assigned to each block.

2.4.1 Chosen plaintext attacks

Now, let's consider a scenario where the attacker has access to both the ciphertext and a portion of the plaintexts. In this type of attack, the attacker is familiar with a series of plaintexts that undergo encryption, and their objective is to determine the specific plaintext being encrypted.

In an ideal situation, the attacker should not be able to distinguish between two plaintexts of equal length when provided with their encrypted versions. Such scenarios frequently occur in contexts like managing data packets within network protocols and discerning between encrypted commands sent to a remote host.

The Counter (CTR) mode of operation is vulnerable to Chosen-Plaintext Attacks (CPA) due to its deterministic encryption process. To enhance security and achieve decryptable non-deterministic encryption, we can implement the following steps:

1. *Rekeying*: change the encryption key for each block using a mechanism like a ratchet, ensuring that each block's encryption is independent and unpredictable.

2. *Randomize the encryption*: introduce (removable) randomness into the encryption process by altering the mode of employing PseudoRandom Permutations (PRPs). This randomization enhances the unpredictability of the ciphertext, making it more resistant to cryptanalysis.

3. *Nonce usage*: utilize numbers used once (NONCEs) to introduce additional variability into the encryption process. In the case of CTR mode, a NONCE is chosen as the starting point for the counter. This NONCE can be public, adding an extra layer of unpredictability to the encryption.

By implementing these measures, we can significantly enhance the security of the encryption process and mitigate vulnerabilities associated with deterministic encryption modes like CTR.

Symmetric ratcheting The term ratcheting is derived from the mechanical device called a ratchet, which allows movement in one direction while preventing backward movement. Similarly, in symmetric ratcheting, the encryption keys are ratcheted forward in a manner that prevents an attacker from decrypting past messages even if they compromise the current key.

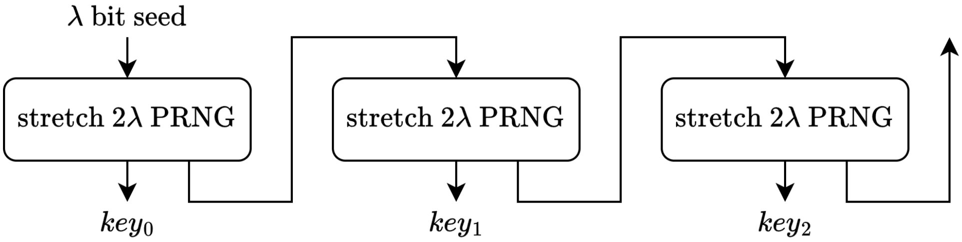


Figure 2.3: Ratcheting

Symmetric ratcheting ensures that even if an attacker manages to compromise the current encryption key, they cannot decrypt past messages or predict future messages due to the frequent key updates. This technique effectively limits the impact of key compromise and strengthens the security of encrypted communication over time.

Chosen plaintext attacks secure encryption Secure encryption schemes are designed to withstand CPA by ensuring that an attacker cannot gain any useful information about the encryption key or plaintexts, even if they have access to ciphertexts for chosen plaintexts. This is accomplished by utilizing the NONCE in conjunction with the Counter (CTR) mode of operation.

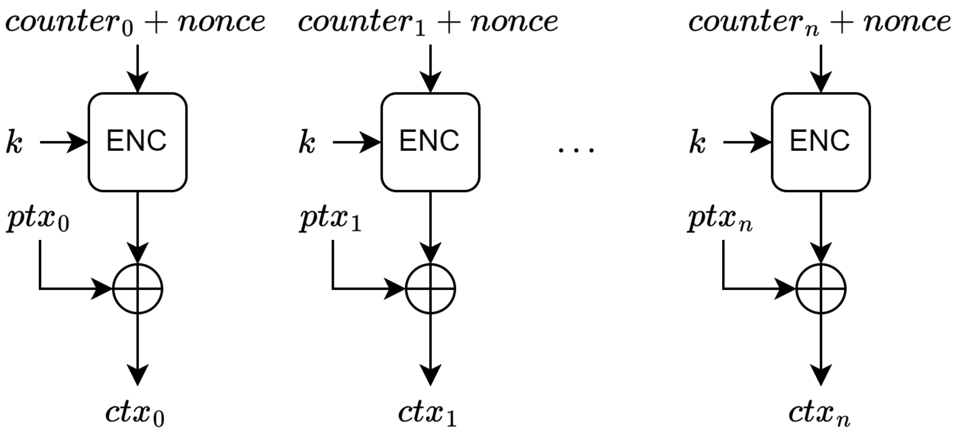


Figure 2.4: Secure ctr

2.5 Data integrity

Malleability refers to the ability to make alterations to the ciphertext, without knowledge of the encryption key, resulting in predictable modifications to the plaintext. This characteristic can be exploited in various ways to launch decryption attacks and manipulate encrypted data.

However, malleability can also be leveraged as a desirable feature, as seen in homomorphic encryption schemes.

To mitigate malleability, it is crucial to design encryption schemes that are inherently non-malleable and incorporate mechanisms to ensure data integrity against attackers. While current encryption schemes primarily provide confidentiality, they do not detect changes in the ciphertext effectively.

To address this limitation, a small piece of information known as a tag can be added to the encrypted message, allowing for integrity testing of the encrypted data itself. Simply adding the tag to the plaintext before encryption is not sufficient, as Message Authentication Codes (MACs) are required for proper data authentication.

2.5.1 Message authentication codes

A message authentication code consists of a pair of functions:

- `compute_tag(string, key)`: generates the tag for the input string.
- `verify_tag(string, tag, key)`: verifies the authenticity of the tag for the input string.

In an ideal attacker model, the attacker may possess knowledge of numerous message-tag pairs but should be unable to forge a valid tag for a message they do not already know. Additionally, tag splicing from valid messages should also be prevented.

CBC-MAC Cipher block chaining message authentication code (CBC-MAC) is a method for generating a fixed-size authentication tag from variable-length messages using a block cipher in CBC mode. Here's how CBC-MAC works:

1. *Initialization*: CBC-MAC operates on fixed-size blocks of data, so if the message is not a multiple of the block size, padding is applied to make it fit. The MAC is initialized with a zero or an initial value.
2. *Block Encryption*: the message is divided into blocks of equal size. Each block is encrypted using the block cipher in CBC mode. The ciphertext of each block is then XORed with the next plaintext block before encrypting the next block.
3. *Finalization*: once all blocks are encrypted, the last ciphertext block becomes the MAC.

CBC-MAC possesses several noteworthy characteristics. It is computationally efficient, requiring only a single pass through the message. The MAC generates a fixed-length authentication tag determined by the block size of the underlying block cipher. Additionally, CBC-MAC offers collision resistance, making it extremely difficult to find two different messages that produce the same MAC.

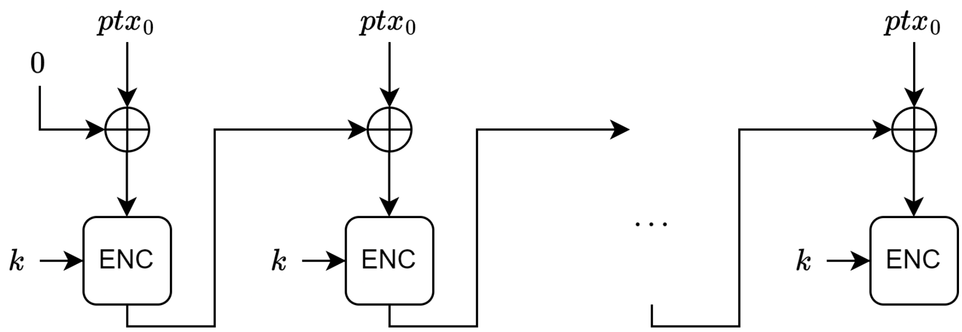


Figure 2.5: CTR encryption mode

CBC-MAC is widely used in practice for message authentication in various cryptographic protocols and applications, including network protocols, file authentication, and secure messaging systems. However, it is important to use CBC-MAC correctly and securely to avoid potential vulnerabilities.

MAC usages HTTP cookies serve as a form of "note to self" for HTTP servers, providing a means to store information locally within a user's browser. However, it is crucial that this information remains unaltered between server reads. To address this concern, the server employs a process where it computes a tag for the cookie using a cryptographic function, denoted as `compute_tag(cookie, k)`. This tag is then stored alongside the corresponding cookie as a pair (cookie, tag), ensuring the integrity and authenticity of the cookie's contents.

2.5.2 Cryptographic hashes

Ensuring the integrity of a file typically involves either comparing it bit by bit with an intact copy or reading the entire file to compute a message authentication code. However, it would be highly advantageous to verify the integrity of a file using only short, fixed-length strings, regardless of the file's size, thereby simplifying the process and reducing computational overhead. Unfortunately, a significant obstacle arises due to the inherent lower bound on the number of bits required to accurately encode a given content without any loss of information. This limitation presents a challenge when attempting to devise a method for efficiently testing the integrity of files.

A cryptographic hash function, denoted as $H : \{0,1\}^* \rightarrow \{0,1\}^I$, is designed such that the following computational problems are difficult to solve:

1. Given a digest $d = H(s)$, determining the original input s (first preimage).
2. Given both an input s and its corresponding digest $d = H(s)$, finding another input r (where $r \neq s$) that produces the same digest ($H(r) = d$) (second preimage).
3. Finding two distinct inputs r and s (where $r \neq s$) that yield the same digest ($H(r) = H(s)$) (collision).

In an ideal scenario, the performance of a concrete cryptographic hash function can be summarized as follows:

1. Finding the first preimage requires approximately $O(2^d)$ hash computations, involving guessing potential inputs s.

2. Finding the second preimage similarly demands around $O(2^d)$ hash computations, involving guessing potential inputs r.

3. Discovering a collision involves approximately $O(2^{2d})$ hash computations.

The resulting output bitstring of a hash function is commonly referred to as a digest.

Hash functions For preferred cryptographic hash functions, consider utilizing SHA-2 and SHA-3. SHA-2, developed privately by the NSA, offers digest sizes of 256, 384, and 512 bits. SHA-3, on the other hand, emerged from a public design contest akin to AES and boasts digest sizes ranging from 256 to 512 bits. Both SHA-2 and SHA-3 are currently unbroken and enjoy wide standardization by bodies such as NIST and ISO.

Conversely, it's advisable to steer clear of SHA-1 and MD-5. SHA-1, with its fixed 160-bit digest size, has been compromised for collisions, achievable in around 2^{61} operations. MD-5, which is known to be severely broken, allows for collisions with just 2^{11} operations, with public tools readily accessible for generating collisions. MD-5 is particularly vulnerable to collisions with arbitrary input prefixes, achievable in approximately 2^{40} operations.

Usage Hash functions serve various purposes, including:

- *Pseudonymized matching*: employed in scenarios like signal's contact discovery, where hashes of values are stored and compared instead of the actual values themselves.

- *MAC construction*: hash functions are integral in generating Message Authentication Codes (MACs), where a tag is produced by hashing both the message and a secret string. Verification involves recomputing the same hash and comparing it with the original tag. HMAC (Hash-based Message Authentication Code) is a widely adopted method, standardized in RFC 2104 and NIST FIPS 198. It utilizes a generic hash function as a plug-in, denoted as HMAC-hash name. Examples include HMAC-SHA1, HMAC-SHA2, and HMAC-SHA3.

- *Forensic applications*: hash functions are crucial in forensic investigations. For instance, only the hash of a disk image obtained can be documented in official reports, ensuring data integrity and facilitating verification processes.

2.6 Asymmetric cryptosystems

Desirable features include the ability to establish a short secret agreement over a public channel, send messages confidentially over an authenticated public channel without sharing secrets with recipients, and authenticate actual data.

The solution lies in asymmetric cryptosystems, which revolutionized cryptography. Before 1976, methods relied on human carriers or physical signatures. Then, innovations such as the Diffie-Hellman key agreement in 1976, public key encryption in 1977, and the introduction of digital signatures in the same year paved the way for modern cryptographic solutions.

Note Until now, the most effective attack method has been enumerating the secret parameter. This approach suffices for modern block ciphers, with the best attack requiring approximately

$O(2^\lambda)$ operations. However, asymmetric cryptosystems depend on complex computational problems, where brute-forcing the secret parameter is not the optimal strategy. For instance, factoring a number of λ bits demands computational effort approximately $O\left(e^{k(\lambda)^{\frac{1}{3}}\cdot\log(\lambda)^{\frac{2}{3}}}\right)$. It's crucial to note that comparing the sizes of security parameters rather than their actual complexities can lead to misleading conclusions.

2.6.1 Diffie-Hellman key agreement

The objective of the Diffie-Hellman key agreement protocol is to enable two parties to securely share a secret value using only public messages. Assuming an attacker model where interception of communications is possible, but tampering is not, and relying on the Computational Diffie-Hellman (CDH) assumption, the protocol operates under the following principle:

- Let $(\mathsf{G},\cdot) \equiv \langle g\rangle$ represent a finite cyclic group, with two randomly sampled numbers a and b from the set $\{0,\ldots,|\mathsf{G}|\}$, where the length of a ($\lambda = \text{len}(a)$) is approximately logarithmic to the size of G.
- Given g^a, g^b, the computational complexity of finding g^{ab} is significantly greater than polynomial in the logarithm of the size of G.
- The most effective current attack strategy involves solving either for b or a, known as the discrete logarithm problem.

Example:
Let's consider two users, A and B:

- User A selects a random number a from the set $\{0,\ldots,|\mathsf{G}|\}$ and sends g^a to user B.
- User B selects a random number b from the set $\{0,\ldots,|\mathsf{G}|\}$ and sends g^b to user A.
- User A receives g^b from user B and computes $\left(g^b\right)^a$.
- User B receives g^a from user A and computes $(g^a)^b$.

Because the finite cyclic group $(\mathsf{G},\cdot)$ is commutative, we can observe that $\left(g^b\right)^a = (g^a)^b$.

In practical implementations, a subgroup $(\mathbb{Z}_N^*,\cdot)$ is chosen from $(\mathsf{G},\cdot)$, where $\mathbb{Z}_N^*$ denotes the set of integers modulo n

In this scenario, breaking the computational Diffie-Hellman assumption requires a minimum of:

$$\min\left(O\left(e^{k\log(n)^{\frac{1}{3}}\cdot\log(\log(n))^{\frac{2}{3}}}\right), O\left(2^{\frac{\lambda}{2}}\right)\right)$$

Alternatively, when utilizing elliptic curve points with dedicated addition, breaking the computational Diffie-Hellman assumption takes $O\left(2^{\frac{\lambda}{2}}\right)$ time.

2.6.2 Public key encryption

In public key encryption, distinct keys are utilized for decryption and encryption purposes. It is computationally challenging to accomplish two tasks: decrypting a ciphertext without access to the private key and computing the private key solely from the public key.

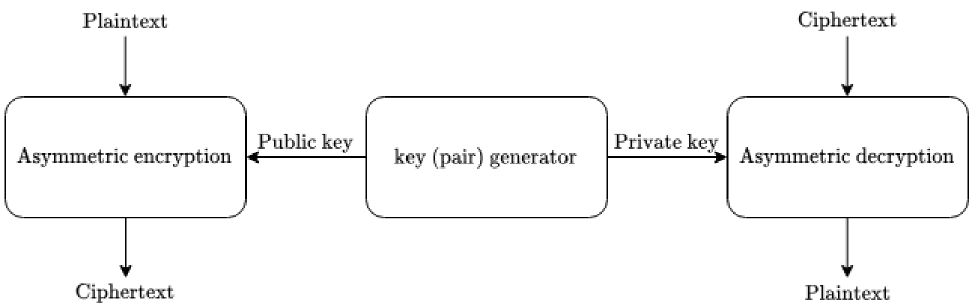


Figure 2.6: Key encryption

Algorithm Rivest, Shamir, Adleman (RSA) is a groundbreaking encryption algorithm introduced in 1977. It supports message and key sizes ranging from 2048 to 4096 bits, ensuring robust security against modern computational threats. Originally patented, RSA's intellectual property rights have since expired, fostering widespread adoption and further development within the cryptographic community. One notable advantage of RSA is its capability to encrypt messages without expanding ciphertext, ensuring efficient transmission and storage of encrypted data. Additionally, RSA encryption with a fixed key exhibits pseudorandom permutation (PRP) properties, enhancing its versatility and applicability in various cryptographic scenarios.

The ElGamal encryption scheme, conceived in 1985, offers a versatile cryptographic solution characterized by its flexibility and patent-free nature. ElGamal encryption accommodates keys spanning either the k-bit range or hundreds of bits, contingent upon the chosen variant, allowing for tailored security configurations to suit various applications. Free from patent encumbrances, the ElGamal scheme has gained traction as a viable alternative to RSA, particularly in scenarios where patent restrictions were a consideration. A distinctive attribute of ElGamal encryption is its ciphertext, which typically spans twice the size of the plaintext. Despite this expansion, its widespread adoption attests to its effectiveness in safeguarding sensitive data and communications.

Usage Key encapsulation is a cryptographic technique used to securely transmit secret keys between parties over an insecure communication channel. In this method, the secret key is encapsulated or wrapped within another encryption layer using a public key algorithm. The recipient, possessing the corresponding private key, can then decrypt and extract the encapsulated key.

> **Example:**
> Let's consider a scenario where there exists a public channel between users A and B, and the attacker can only observe but not alter the communication. Subsequently, user B randomly selects a bitstring s from the set (k_{pri}, k_{pub}) encrypts it using k_{pub}, and forwards the resulting ciphertext to user A. User A, possessing the corresponding private key k_{pri}, decrypts the ciphertext and retrieves the bitstring s.
>
> The process is then repeated with the roles of users A and B swapped. Consequently, both users obtain separate secrets. Although user B alone determines the value of the shared secret s, combining the two secrets derived from the exchanged messages yields analogous security guarantees to those of a conventional key agreement protocol.

Using an asymmetric cryptosystem, user B encrypts a message for user A without the requirement of pre-shared secrets. Theoretically, user B and user A could rely solely on an asymmetric cryptosystem for their communication needs. However, in practice, this method would prove highly inefficient. Asymmetric cryptosystems operate significantly slower compared to their symmetric counterparts, with performance degradation ranging from 10 to 1000 times.

Modern encryption Hybrid encryption schemes represent a strategic blend of asymmetric and symmetric cryptography techniques. In this approach, asymmetric algorithms are utilized for key transport or agreement, facilitating secure key exchange between parties. Meanwhile, symmetric algorithms are employed to encrypt the bulk of the data, ensuring efficient and swift encryption of large volumes of information. This concept serves as the cornerstone for all contemporary secure transport protocols, embodying a harmonious integration of both cryptographic methodologies to deliver robust and effective encryption for secure communication.

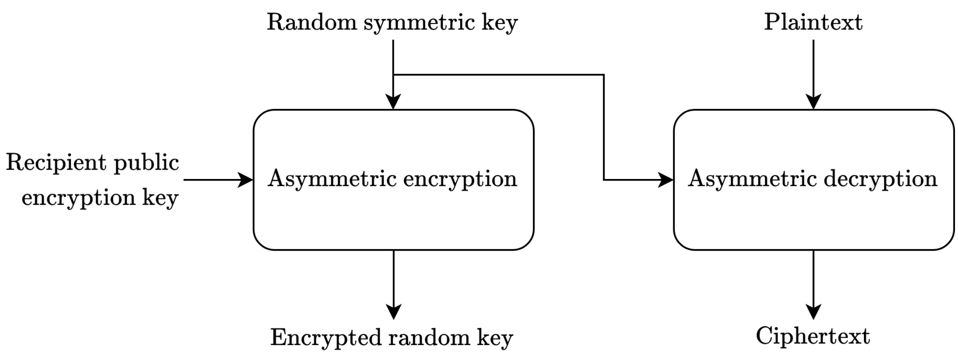


Figure 2.7: Modern encryption

2.6.3 Digital signatures

Authenticating data serves as a crucial aspect in the establishment of secure hybrid encryption schemes. It ensures that the public key utilized by the sender corresponds accurately to the intended recipient. Additionally, the ability to verify the authenticity of data without relying on a pre-shared secret is highly desirable. Digital signatures play a pivotal role in achieving data authentication objectives:

- They offer robust evidence linking data to a specific user, enhancing data integrity.
- Verification of digital signatures does not necessitate a shared secret, simplifying the authentication process.
- Properly generated digital signatures cannot be repudiated by the user, ensuring accountability.
- Asymmetric cryptographic algorithms underpin digital signatures, providing a solid foundation for their security.
- It has been formally demonstrated that achieving non-repudiation without digital signatures is impractical, reinforcing their indispensable role in data authentication.

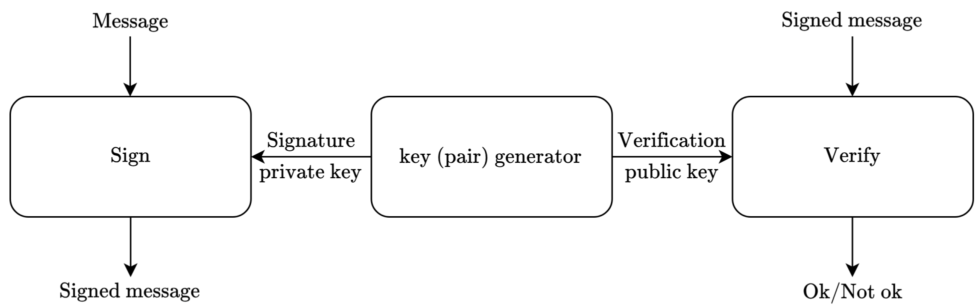


Figure 2.8: Digital signature

The computational challenges inherent in digital signatures encompass several key aspects:

- Signing a message without possessing the signature key, which includes attempting to splice signatures from unrelated messages.
- Computing the signature key when provided only with the verification key.
- Attempting to derive the signature key solely from signed messages, without access to additional information.

Algorithms In 1977, Rivest, Shamir, and Adleman (RSA) introduced a groundbreaking cryptographic method. This method employs a singular hard-to-invert function to craft both an asymmetric encryption scheme and a signature, with distinct message processing for each. Notably, the process of signing is significantly slower than verification, roughly around 300 times slower. This innovative approach has been standardized in NIST DSS (FIPS-184-4), underscoring its widespread adoption and importance in modern cryptographic practices.

The Digital Signature Standard (DSA) draws its foundations from adjustments made to signature schemes initially proposed by Schnorr and ElGamal. It, too, has been formalized in NIST DSS (FIPS-184-4), reflecting its establishment as a recognized cryptographic protocol. Notably, in DSA, the processes of signature creation and verification unfold at comparable speeds, distinguishing it from some other cryptographic methods.

Usages Digital signatures serve various purposes:

- Authenticating digital documents: in order to enhance efficiency, digital signatures frequently entail signing the hash of a document rather than the document itself. However, the assurance of the signature's reliability relies on the robustness of both the signature and hash algorithms.
- Authenticating users: digital signatures present an alternative approach to user authentication, serving as a viable replacement for traditional password-based logins. During this procedure, the server retains the user's public verification key, typically acquired during the account creation phase. Upon authentication requests, the server initiates the client to sign a lengthy, randomly generated bitstring, referred to as a challenge. Successful verification of the challenge signature by the client serves as compelling evidence of identity to the server.

2.7 Keys handling

The issue of securely binding public keys to user identities is paramount in both asymmetric encryption and digital signatures. Failure to authenticate public keys can lead to serious consequences, including susceptibility to Man-in-the-Middle attacks in asymmetric encryption and the potential for unauthorized signature generation by malicious actors.

To ensure the authenticity of public keys, an additional layer of verification, often in the form of another signature, is necessary. This additional signature serves as a guarantee of the legitimacy of the public-key/identity pairing. To facilitate this process, there is a requirement for a standardized format for distributing these signed pairs securely across systems.

2.7.1 Digital certificates

Digital certificates serve the purpose of associating a public key with a specific identity. This identity can be represented as an ASCII string for human interpretation or as either the Canonical Name (CNAME) or IP address for machine understanding. Additionally, these certificates outline the intended usage of the public key they contain, eliminating any potential ambiguities when a key format is suitable for both encryption and signature algorithms.

Furthermore, digital certificates include a designated time frame during which they are deemed valid. One of the most commonly adopted formats for digital certificates is detailed in the ITU X.509 standard.

2.7.2 Certification authorities

The certificates are signed by a trusted third party, known as the Certificate Authority (CA). This CA's public key is authenticated using another certificate. This process extends even to self-signed certificates, which must be trusted beforehand.

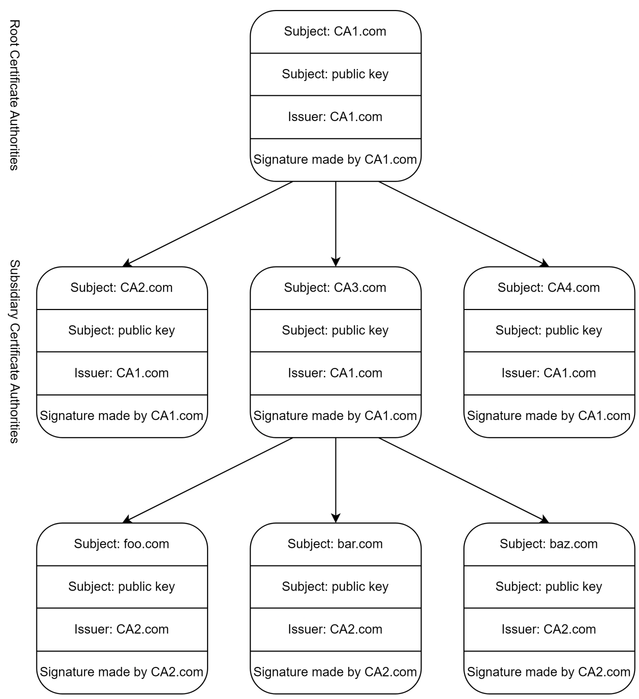


Figure 2.9: Certificate authorities hierarchy

2.8 State of the art

The contemporary secure communication protocols such as TLS, OpenVPN, and IPSec utilizes the following structure:

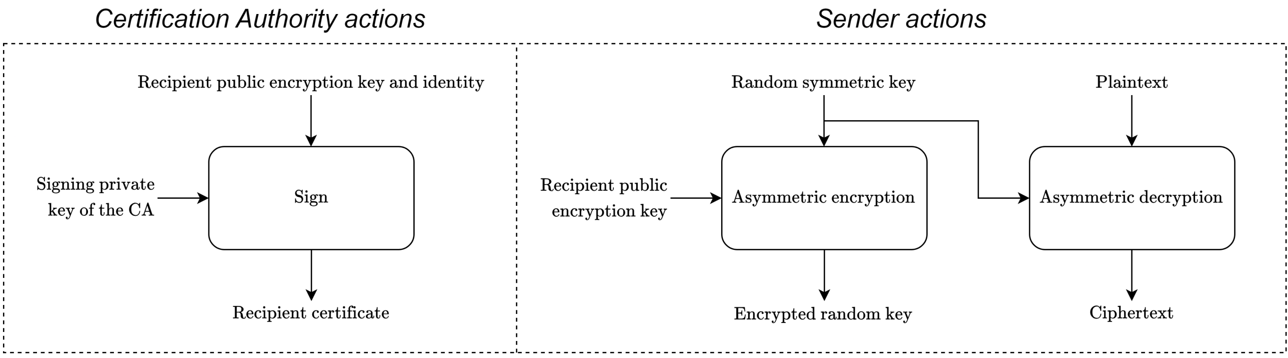


Figure 2.10: Secure communication protocols

Quantum computers With quantum computing some computationally challenging problems will become less hard, prompting a reassessment of their difficulty. There's a notable shift away from cryptosystems built upon factoring and discrete logarithm. Instead, there's a growing focus on exploring alternatives, currently undergoing standardization as of April 2022.

Compute on encrypted data Performing computations on encrypted data is feasible; however, it tends to be moderately to severely inefficient.

Physical access If the attacker gains physical access to the device executing the cipher (or can remotely measure it), it is essential to consider side-channel information within the attacker model.

2.9 Shannon's information theory

Shannon's information theory provides a mathematical framework for understanding communication and quantifying information.

Communication occurs between two endpoints:

- The sender comprises an information source and an encoder.
- The receiver consists of an information destination and a decoder.

Information is transmitted through a channel in the form of a sequence of symbols from a finite alphabet.

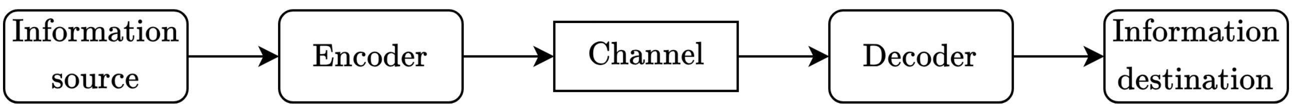


Figure 2.11: Shannon's communication structure

The receiver exclusively receives information through the channel. Until the symbol arrives, there remains uncertainty about what the next symbol will be. Consequently, we represent the sender as a random variable. Hence, obtaining information is akin to determining an outcome of a random variable $\mathcal{X}$, and the quantity of information relies on the distribution of $\mathcal{X}$. Encoding involves mapping each outcome to a finite sequence of symbols: more symbols are necessary when transmitting more information.

2.9.1 Entropy

We require a non-negative measure of uncertainty. The combination of uncertainties should correspond to adding entropies.

Definition (*Entropy*)**.** Let $\mathcal{X}$ be a discrete random variable with n outcomes in $\{x_0, \ldots, x_{n-1}\}$, where $\mathrm{P}(\mathcal{X} = x_i) = p_i$ for all $0 \leq 1 \leq n$. The entropy of $\mathcal{X}$ is given by:

$$H(\mathcal{X}) = \sum_{i=0}^{n-1} -p_i \log_b (p_i)$$

The unit of measurement for entropy is contingent on the base b of the logarithm, where the typical case for $b = 2$ is bits.

Example:

The random variable $\mathcal{X}$ represents a sequence of 6 uniform random letters (with 6^{26} combinations). In this case, the entropy is calculated as:

$$H(\mathcal{X}) = \sum_{i=0}^{6^{26}-1} -\frac{1}{6^{26}} \log_b \left(\frac{1}{6^{26}}\right) \approx 28.2b$$

On the other hand, if $\mathcal{X}$ represents a uniform selection from six-letter English words (with 6^6 combinations), the entropy is computed as:

$$H(\mathcal{X}) = \sum_{i=0}^{6^{6}-1} -\frac{1}{6^{6}} \log_b \left(\frac{1}{6^{6}}\right) \approx 12.6b$$

Theorem 2.9.1. *It is possible to encode the outcomes n of independent and identically distributed random variables, each with entropy $H(\mathcal{X})$, using at least $nH(\mathcal{X})$ bits per outcome. Encoding with fewer than $nH(\mathcal{X})$ bits will result in loss of information.*

Consequently, achieving arbitrary compression of bitstrings without loss is unattainable, necessitating cryptographic hashes to discard certain information.

Additionally, the task of guessing a piece of information (equivalent to one outcome of $\mathcal{X}$) is no less challenging than guessing a bitstring of length $H(\mathcal{X})$, disregarding momentarily the effort involved in decoding the guess.

2.9.2 Minimum entropy

Definition (*Min-entropy*)**.** The min-entropy is a measure of the most conservative assessment of the unpredictability of a set of outcomes. It is defined for $\mathcal{X}$ as:

$$H_\infty(\mathcal{X}) = -\log(\max_i p_i)$$

In essence, it represents the entropy of a random variable with a uniform distribution, where each outcome has a probability of $\max_i p_i$.

It's worth noting that guessing the most common outcome of $\mathcal{X}$ is no less challenging than guessing a bitstring of length $H_\infty(\mathcal{X})$.

Example:
Consider the random variable $\mathcal{X}$ defined as:

$$\mathcal{X} = \begin{cases} 0^{128} & \text{with probability} \frac{1}{2} \\ a & \text{with probability} \frac{1}{2^{128}} \end{cases}$$

Here, $a \in 1\{0,1\}^{127}$. Predicting an outcome shouldn't be too difficult: just predict 0^{128}:

$$H(\mathcal{X}) = \frac{1}{2}\left(-\log_2\left(\frac{1}{2}\right)\right) + 2^{127}\frac{1}{2^{128}}\left(-\log_2\left(\frac{1}{2^{128}}\right)\right) = 64.5b$$

$$H_\infty(\mathcal{X}) = -\log_2\left(\frac{1}{2}\right) = 1b$$

Min-entropy indicates that guessing the most common output is as difficult as guessing a single bit string.

APPENDIX A

The x86 architecture

A.1 Introduction

The Instruction Set Architecture (ISA) serves as the abstract blueprint for a computer architecture, outlining its logical structure. It encompasses essential programming elements like instructions, registers, interrupts, and memory architecture. Importantly, the ISA may deviate from the physical microarchitecture of the computer system in practice.

A.1.1 History

The x86 Instruction Set Architecture (ISA) originated in 1978 as a 16-bit ISA with the Intel 8086 processor. Over time, it transitioned into a 32-bit ISA with the Intel 80386 in 1985. Finally, in 2003, it advanced to a 64-bit ISA with the AMD Opteron processor.

Characterized by its Complex Instruction Set Computing (CISC) design, the x86 ISA retains numerous legacy features from its earlier iterations.

A.1.2 Von Neumann architecture

Von Neumann architecture, named after mathematician and physicist John von Neumann, is a conceptual framework for designing and implementing digital computers. It consists of four main components:

1. *Central Processing Unit* (CPU): this is the brain of the computer, responsible for executing instructions. It contains an arithmetic logic unit (ALU) for performing arithmetic and logical operations, and a control unit that fetches instructions from memory, decodes them, and controls the flow of data within the CPU.

2. *Memory*: Von Neumann computers have a single memory space that stores both data and instructions. This memory is divided into cells, each containing a unique address. Programs and data are stored in memory, and the CPU accesses them as needed during program execution.

3. *Input/Output* (I/O) devices: these devices allow the computer to interact with the external world. Examples include keyboards, monitors, disk drives, and network interfaces.

Data is transferred between the CPU and I/O devices through input and output operations.

4. *Bus*: the bus is a communication system that allows data to be transferred between the CPU, memory, and I/O devices. It consists of multiple wires or pathways along which data travels in the form of electrical signals.

In Von Neumann architecture, programs and data are stored in the same memory space, and instructions are fetched from memory and executed sequentially by the CPU. This architecture is widely used in modern computers and forms the basis for most general-purpose computing devices. However, it has some limitations, such as the Von Neumann bottleneck, where the CPU is often waiting for data to be fetched from memory, leading to inefficiencies in performance.

The memory is structured into cells, with each cell capable of holding a numerical value ranging from -128 to 127.

A.2 Features

The x86 architecture employs several general-purpose registers, including EAX, EBX, ECX, EDX, ESI, EDI (utilized as source and destination indices for string operations), EBP (serving as the base pointer), and ESP (acting as the stack pointer). Additionally, it incorporates:

- The instruction pointer (EIP) in x86 architecture remains inaccessible directly, but undergoes modification through instructions like `jmp`, `call`, and `ret`. Its value can be retrieved from the stack, known as the saved IP. This register is 32 bits in size and serves as a holder for boolean flags that convey program status, including overflow, sign, zero, auxiliary carry (BCD), parity, and carry. These flags indicate the outcome of arithmetic instructions and play a crucial role in controlling program flow. In terms of program control, the direction flag manages string instructions, dictating whether they auto-increment or auto-decrement. Additionally, EIP controls system operations pertinent to the operating system.
- Program status and control are managed by the EFLAGS register.
- Segment registers are also utilized in the architecture.

The core data types include:

- *Byte*: 8 bits
- *Word*: 2 bytes
- *Dword* (Doubleword): 4 bytes (32 bits)
- *Qword* (Quadword): 8 bytes (64 bits)

Assembly language is unique to each Instruction Set Architecture (ISA) and directly corresponds to binary machine code. The process of converting assembly language instructions into machine code is illustrated in the diagram below:

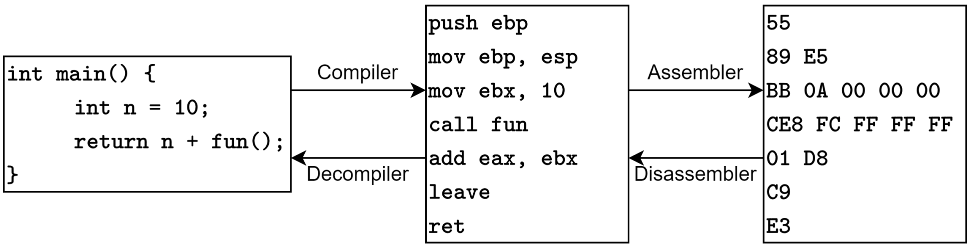


Figure A.1: From source code to machine code

A.3 Syntax

In the x86 architecture, two primary syntaxes are commonly used:

- Intel syntax: This is the default syntax in most Windows programs.
- AT&T syntax: This syntax is default in most UNIX tools.

The Intel syntax is the simpler of the two. Additionally, in x86, instructions have variable length.

A.3.1 Basic instructions

Data transfer Data transfer is accomplished using the following command:

```
mov destination, source
```

Where:

- `source`: immediate value, register, or memory location.
- `destination`: register or memory location.

This command facilitates basic load and store operations, allowing for register-to-register, register-to-memory, immediate-to-register, and immediate-to-memory transfers. It's important to note that memory-to-memory transfers are invalid in every instruction.

Addition and subtraction Addition and subtraction operations are executed using the following commands:

```
// destination = destination + source
add destination, source
// destination = destination - source
sub destination, source
```

Where:

- `source`: immediate value, register, or memory location
- `destination`: register or memory location

It's important to note that the size of the `destination` operand must be at least as large as the `source` operand.

Multiplication Multiplication is performed using the following commands:

```
// destination = implied_op * source
mul source
// signed multiplication
imul source
```

Here, `source` represents a register or memory location. Depending on the size of `source`, the implied operands are as follows:

- First operand: AL, AX, or EAX.
- Destination: AX, DX:AX, EDX:EAX (twice the size of `source`).

Division Division is carried out using the following command:

```
div source
// signed division
idiv source
```

Here, `source` represents a register or memory location. These commands compute both the quotient and remainder. The implied operand for the division operation is EDX:EAX.

Logical operators To perform logical operations such as negation or bitwise operations, the following commands are used: `neg`, `and`, `or`, `xor`, and `not`.

Compare and test To compare two operands or perform bitwise AND operation between them, the following commands are used:

```
// computes op1 - op2
cmp op1, op2
// computes op1 AND op2
test op1, op2
```

These operators set the flags ZF (Zero Flag), CF (Carry Flag), and OF (Overflow Flag) based on the result of the operation but discard the actual result.

Conditional jump Conditional jumps are executed using the following command:

```
j<cc> address or offset
```

This command jumps to the specified address or offset only if a certain condition `<cc>` is met. The condition is checked based on one or more status flags of EFLAGS and can include conditions such as O (overflow), NO (not overflow), S (sign), NS (not sign), E (equal), Z (zero), and NE (not equal).

Other possible jump instructions include:

```
// jump if zero
jz
// jump if greater than
jg
// jump if less than
jlt
```

These instructions allow for conditional branching based on specific conditions evaluated by the processor's status flags.

Unconditional jump Unconditional jumps are executed using the following command:

```
jmp address or offset
```

This command unconditionally transfers control to the specified address or offset by setting the Instruction Pointer (EIP) to the designated location.

The offset can also be relative, causing the EIP to be incremented or decremented by the specified offset value.

Load effective address The load effective address instruction is performed with the following syntax:

```
lea destination, source
```

In this command:

- `source` represents a memory location.
- `destination` denotes a register.

Functionally similar to a `mov` instruction, `lea` doesn't access memory to retrieve a value. Instead, it calculates the effective address of the `source` operand and stores it in the `destination` register, effectively storing a pointer rather than a value.

No operations The `nop` instruction simply advances to the next instruction without performing any operation. Its hexadecimal opcode, `0x90`, is widely recognized. This command holds significant utility in exploitation scenarios.

Interrupts and syscall Interrupts return an integer ranging from 0 to 255. System calls are invoked using the instructions `syscall` in Linux and `sysenter` in Windows.

A.3.2 Conventions

In x86 architectures, a convention known as endianness is employed. This convention dictates the sequential ordering of bytes within a data word in memory.

Big endian Big endian systems store the most significant byte of a word in the lowest memory address.

Little endian Little endian systems store the least significant byte of a word in the lowest memory address.

Note IA-32 architecture follows the little endian convention.

A.4 Program layout and functions

The mapping of an executable to memory in Linux involves several sections:

- `.plt`: this section contains stubs responsible for linking external functions.
- `.text`: this section contains the executable instructions of the program.

- `.rodata`: this section holds read-only data contributing to the program's memory image.
- `.data`: this section holds initialized data contributing to the program's memory image.
- `.bss`: this section holds uninitialized data contributing to the program's memory image. The system initializes this data with zeros when the program starts running.
- `.debug`: this section holds symbolic debugging information.
- `.init`: this section holds executable instructions contributing to the process initialization code. It executes before calling the main program entry point (typically named main for C programs).
- `.got`: this section holds the global offset table.

The program memory layout is depicted in the following simplified diagram:

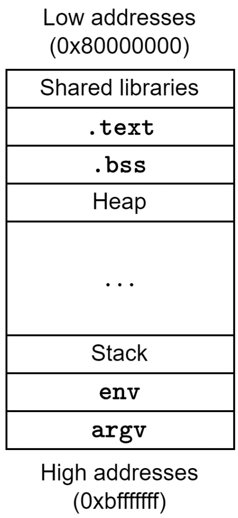


Figure A.2: Simplified program memory layout

A.4.1 Stack

The stack operates on a Last In, First Out (LIFO) principle and is crucial for managing functions, local variables, and return addresses in programs. Its management is facilitated through the use of the ESP register (stack pointer). It's important to note that the stack grows towards lower memory addresses, which means it extends downward in the address space.

Push To insert a new element into the stack, the following command is used:

```
push immediate or register
```

This command places the immediate or register value at the top of the stack and decrements the ESP by the operand size.

Push To remove an element from the stack, the following command is used:

```
pop destination
```

This command loads a word from the top of the stack into the destination and then increases the ESP by the operand's size.

A.4.2 Functions handling

When encountering a `call` instruction, the address of the next instruction is pushed onto the stack, and then the address of the first instruction of the called function is loaded into the EIP register.

Upon encountering a `ret` instruction, the return address previously saved by the corresponding `call` is retrieved from the top of the stack.

At the start of a function, space must be allocated on the stack for local variables. This region of the stack is known as the stack frame. The EBP register serves as a pointer to the base of the function's stack frame. At the function's entry point, the following steps are typically taken:

1. Save the current value of EBP onto the stack.

2. Set EBP to point to the beginning of the function's stack frame.

Upon encountering a `leave` instruction, the caller's base pointer (EBP) is restored from the stack.

Conventions Conventions dictate the method of passing parameters (via stack, registers, or both), the responsibility for cleaning up parameters, the manner of returning values, and the designation of caller-saved or callee-saved registers.

The high-level language, compiler, operating system, and target architecture collaboratively establish and adhere to a specific calling convention, which is an integral part of the Application Binary Interface (ABI).

In x86 C compilers, the declaration conventions are governed by the `cdecl` modifier. Although the `cdecl` modifier can be explicitly used to enforce these conventions, the standard rules dictate that:

- Arguments are passed through the stack in a right-to-left order.

- Parameter cleanup is the responsibility of the caller, who removes the parameters from the stack after the called function concludes.

- The return value is stored in the EAX register.

- Caller-saved registers encompass EAX, ECX, and EDX, while other registers are considered callee-saved.

In x86 C compilers, the calling conventions follow the `fastcall` modifier. While explicitly using the `_fastcall` modifier enforces these conventions, the standard guidelines dictate that:

- Parameters are passed in registers: rdi, rsi, rdx, rcx, r8, and r9, with subsequent parameters passed on the stack in reverse order (caller cleanup).

- Callee-saved registers include rbx, rsp, rbp, r12, r13, r14, and r15.

- Caller-saved registers (scratch) encompass rax, rdi, rsi, rdx, rcx, r8, r9, r10, and r11.

- The return value is stored in rax. If the return value is 128-bit, it's stored across rax and rdx registers.