# Blockchains and Distributed Ledgers
# Lecture Notes

Damiano Abram

February 2026

# Contents


**1 Basic Mathematical and Cryptographic Notions** 2**
- 1.1 A Basic Recap of Probability Theory** . . . 2
  - 1.1.1 Some Fundamental Distributions** . . . 3
- 1.2 Turing Machines** . . . 4
- 1.3 Hash Functions** . . . 6
  - 1.3.1 Random Oracles** . . . 7
- 1.4 Digital Signatures** . . . 7

**2 The Consensus Problem** 10
- 2.1 Byzantine Agreement and Secure Broadcast . . . 10
  - 2.1.1 Byzantine Agreement . . . 10
  - 2.1.2 Secure Broadcast . . . 13
  - 2.1.3 The Dolev-Strong Broadcast Protocol . . . 15
  - 2.1.4 Bracha's Asynchronous Secure Broadcast Protocol . . . 17
  - 2.1.5 Impossibility of Asynchronous Broadcast without Honest Supermajority . . . 20

**3 Chain-Based Consensus** 22
- 3.1 A Simple Construction in the Synchronous Setting . . . 23
- 3.2 Proof of Work . . . 23
  - 3.2.1 Permissionless Consensus . . . 24
  - 3.2.2 Mining . . . 24
  - 3.2.3 The Longest Chain Principle . . . 26
  - 3.2.4 Quantifying the Computational Cost . . . 29
  - 3.2.5 The Security Analysis . . . 31
  - 3.2.6 Joining the Protocol . . . 38
  - 3.2.7 Advantages and Disadvantages of Proof-of-Work Consensus . . . 39

**4 Decentralised Economies and Smart Contracts** 41
- 4.1 The UTXO System . . . 42
  - 4.1.1 The UTXO System in Bitcoin . . . 43
- 4.2 The Account-Based System . . . 45
- 4.3 Smart Contracts . . . 46
- 4.4 Transaction Fees and Rewards . . . 47


**How to Read these Notes.** Students of Blockchains and Distributed Ledgers should assume that all sections in these notes are by default included in the exam. Exceptions are marked by appending the symbol * or ** to the name of the section. The single asterisk * denotes optional material for motivated and curious students. Double asterisks ** indicate preliminary material that is not the primary object of the course; students are nevertheless expected to have some basic familiarity with these concepts because of other courses they undertook in their study paths.

# Chapter 1

# Basic Mathematical and Cryptographic Notions**

In this first chapter, we will introduce the basic mathematical and cryptographic tools on which blockchains and distributed ledgers base their security. These will be the Swiss Army knife of this course.

## 1.1 A Basic Recap of Probability Theory**

In this section, we recall some basic notions and results in probability theory. We start by recalling that discrete random variables are random variables that can assume only a finite number of values.

**Definition 1** (Support of a random variable)**.** *Let $\mathcal{X}$ be a set. A random variable $X$ has support $\mathcal{X}$ if*

$$\Pr[X \in \mathcal{X}] = 1.$$

*We say that $X$ is discrete if it has a finite support. Otherwise, we say that $X$ is continuous.*

In the case of discrete random variable $X$, its distribution is often identified with the list $\{\Pr[X = x]\}_x$ of probabilities associated with the elements in its support. For continuous random variables, this is not possible (typically, $\Pr[X = x] = 0$ for every $x$). Their probability distribution is instead identified by their density function.

**Definition 2** (Density function)**.** *A random variable $X$ over the (measurable) domain $\mathcal{X}$ has density function $f : \mathcal{X} \to \mathbb{R}$ if, for every (measurable) subset $A \subseteq \mathcal{X}$,*

$$\Pr[X \in A] = \int_A f(x)\, dx.$$

We also recall the definition of independent random variables.

**Definition 3** (Independent random variables)**.** *Let $X_1, \ldots, X_n$ be random variables. We say that $X_1, \ldots, X_n$ are independent if, for every sets $A_1, \ldots, A_n$, we have that*

$$\Pr[X_1 \in A_1, \ldots, X_n \in A_n] = \Pr[X_1 \in A_1] \cdot \ldots \cdot \Pr[X_n \in A_n].$$

We conclude by recalling the notion of average.

**Definition 4** (Average)**.** *Let $X$ be a discrete random variable over $\mathbb{R}$. The average (or mean, or expected value) of $X$ is*

$$\mathbb{E}[X] = \sum_{x \in \mathbb{R}} x \cdot \Pr[X = x].$$

*If $X$ is instead a continuous random variable over $\mathbb{R}$ with density function $f$, the average (or mean) of $X$ is*

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot f(x)\, dx,$$

*if this exists.*

It turns out that the average sum of *independent* random variables is equal to the sum of their expectations. This is however not true when the variables are not independent.

**Lemma 1.** *Let $X_1, \ldots, X_n$ be $n$ independent discrete random variable over $\mathbb{R}$. Let $S := \sum_{i=1}^{n} X_i$. Then*

$$\mathbb{E}[S] = \sum_{i=1}^{n} \mathbb{E}[X_i].$$

Finally, we recall the following fundamental result, which states that that the sum of $n$ independent random variables is unlikely to assume values that are far from its average.

**Theorem 1** (Chernoff Bound)**.** *Let $X_1, \ldots, X_n$ be independent random variables with support $\{0, 1\}$ and let $S := \sum_{i=1}^{n} X_i$. Let $\mu = \mathbb{E}[S]$. Then, for every $\delta \in [0, 1]$, we have*

$$\Pr[X \leq (1 - \delta)\mu] \leq \exp(-\delta^2 \mu / 2).$$

We also recall the following bound

**Lemma 2.** *For every $x \geq 0$, it holds that*

$$1 \geq \exp(-x) \geq 1 - x.$$

### 1.1.1 Some Fundamental Distributions**

We start by recall the simplest distribution: the Bernoulli distribution. This is a discrete random variable with support $\{0, 1\}$. This can be used for instance to model the tossing of a coin.

**Definition 5** (Bernoulli distribution)**.** *We say that a random variable $X$ is distributed according to a Bernoulli distribution of parameter $p \in [0, 1]$ if*

$$\Pr[X = 1] = p, \qquad \Pr[X = 0] = 1 - p.$$

*In particular, the mean of $X$ is $p$.*

Next, we recall the binomial distribution. This is again a discrete random variable but its support consists of all integers between 0 and some parameter $n$.

**Definition 6** (Binomial distribution)**.** *We say that a random variable $X$ is distributed according to a Binomial distribution with $n$ trials and success probability $p \in [0, 1]$ if, for every $k \in \{0, 1, \ldots, n\}$, we have*

$$\Pr[X = k] = \binom{n}{k} p^k (1 - p)^{n-k}.$$

*In particular, the mean of $X$ is $pn$.*

The binomial distribution is linked to the Bernoulli distribution by the following lemma.

**Lemma 3.** *Let $p \in [0, 1]$ be a parameter and suppose that $X_1, \ldots, X_n$ are independent random variables that are all distributed according to a Bernoulli distribution of parameter $p$. Then, $S := \sum_{i=1}^{n} X_i$ is distributed according to a Binomial distribution with $n$ trials and success probability $p$.*

We recall another distribution that is strongly linked to the Bernoulli distribution.

**Definition 7** (Geometric distribution)**.** *We say that a random variable $X$ is distributed according to a geometric distribution with parameter $p \in [0,1]$ if, for every $k \in \mathbb{N} \setminus \{0\}$, we have*

$$\Pr[X = k] = (1-p)^{k-1} \cdot p.$$

*In particular, the mean of $X$ is $1/p$.*

The following lemma states that the geometric distribution can be used to describe how many times we need to sample from Bernoulli distribution until we obtain the first success.

**Lemma 4.** *Let $p \in [0,1]$ be a parameter and suppose that $(X_i)_{i \in \mathbb{N} \setminus \{0\}}$ are independent random variables that are all distributed according to a Bernoulli distribution of parameter $p$. Define $S := \min_i \{i | X_i = 1\}$. Then, $S$ is distributed according to a geometric distribution with paramater $p$.*

We recall another fundamental discrete distribution which is called the *the Poisson distribution*. For instance, this can be used to model the number of customers that enter in a shop in a interval of time of length $\Delta$ (assuming that the probability of receiving a new customer remains the same at each instant of the day and this is independent of the number of previous and current customers). For this instantiation, the parameter $\lambda$ of the Poisson distribution is taken to be $\lambda_0 \cdot \Delta$ where $\lambda_0$ is the expected number of customers per unit of time.

**Definition 8** (Poisson distribution)**.** *We say that a random variable $X$ is distributed according to a Poisson distribution of parameter $\lambda > 0$ if, for every $k \in \mathbb{N}$, we have*

$$\Pr[X = k] = \frac{\lambda^k \exp(-\lambda)}{k!}$$

*In particular, the mean of $X$ is $\lambda$.*

We conclude with the only continuous probability distribution we will use in this course: the *exponential distribution*. This can be, for instance, used to model the waiting time between a customer and the next, where the parameter $\lambda$ measures the expected number of new customers per unit of time.

**Definition 9** (Exponential distribution)**.** *We say that a random variable $X$ is distributed according to an exponential distribution of parameter $\lambda > 0$ if its probability density function is*

$$f(x) = \begin{cases} \lambda \cdot \exp(-\lambda x) & x \geq 0, \\ 0 & x < 0. \end{cases}$$

*In particular, the mean of $X$ is $\lambda^{-1}$ and*

$$\Pr[X \leq x] = \begin{cases} 1 - \exp(-\lambda x) & x \geq 0, \\ 0 & x < 0. \end{cases}$$

## 1.2 Turing Machines**

In this section, we will formalise some basic notions of computability and complexity theory. This will be needed to fully understand the formal definitions of the cryptographic primitives we are going to use in this course. Practitioners who are not interested in understanding the mathematical aspects of distributed ledgers are welcome to skip this section.

We start by recalling the notion of Turing machine, which sometimes, we improperly call *algorithm*. Looking ahead, the reason why we are interested in this definition is the Church-Turing thesis, which conjectures that all computable functions can be computed by a Turing machine.

**Definition 10** (Turing machine)**.** *A (uniform) Turing machine $M$ consists of a function*

$$F : \big(\{0,1\} \cup \{\bot\}\big) \times Q \longrightarrow \{0,1\} \times \big(Q \cup \{q_{\mathsf{halt}}\}\big) \times \{\mathsf{L}, \mathsf{R}\}$$

*where $Q$ is a finite non-empty set, containing the special state $q_{\mathsf{init}}$. We call $F$* the transition function*.*

Essentially, $Q$ denotes a set of states. The execution of a Turing machine is described by a head and a state. Initially, the head points to the first element of the input $x$, whereas the state is set to $q_{\mathsf{init}}$. At each step, the machine reads the bit $b$ indicated by the head and updates its value and that of the state using the transition function $F$. Finally, the head is moved by one position in the direction $\mathsf{L}/\mathsf{R}$ indicated by the transition function $F$. Notice, that the head does not need to always point to entries of $x$: by continuously moving to the right, it can move past it; by moving to the left, it can move before it. In the first case, the read character will be $\bot$; in the second case, it will be a random bit. Either way, we will immediately overwrite this value with the output of the transition function $F$. The execution of the machine will terminate when the updated state becomes $q_{\mathsf{halt}}$. The output will correspond to the updated binary string.

**Definition 11** (Execution of a Turing machine)**.** *Let $x \in \{0,1\}^*$ be a binary string. We define $M(x)$ as the output of the following procedure.*

1. *For every $i \in [|x|]$, set $t[i] \leftarrow x[i]$.*
2. *Set $s \leftarrow 0$, $b \leftarrow x[0]$, $q \leftarrow q_{\mathsf{init}}, T \leftarrow 0$.*
3. *Compute $(b', q', d) \leftarrow F(b, q)$ and update the state of the machine $q \leftarrow q'$, $t[s] \leftarrow b'$, $T \leftarrow T + 1$. Then, set $s \leftarrow s - 1$ if $d = \mathsf{L}$, otherwise set $s \leftarrow s + 1$. If $t[s]$ is not defined yet, set $t[s] \leftarrow \bot$, when $s \geq 0$, and $t[s] \xleftarrow{\$} \{0,1\}$, when $s < 0$. Finally, update $b \leftarrow t[s]$. Repeat this step until $q = q_{\mathsf{halt}}$.*
4. *Output $t$.*

*We call the final value of $T$ the running time of $M$ on input $x$. The vector $t$ is often called* tape*.*

How many steps does a Turing machine require in order to derive the output? In general, there is no algorithm that can tell this. Some Turing machines terminate after a number of steps that is independent of the length of the input $|x|$, some require time that grows proportionally with a power of $|x|$ (e.g., $|x|^3$), some require exponential time (e.g., $2^{|x|}$), some never stop! In blockchains and distributed ledgers, we will focus on efficient machine where the running time is upper bounded by a power of the input length. We informally say that these machines are *efficient*. Of course, it is debatable whether an algorithm that runs in time $|x|^{100}$ is actually efficient, however, when $|x|$ becomes big, this will be significantly better than any algorithm that runs in time $2^{|x|}$, or even $|x|^{\log|x|}$.

**Definition 12** (Polynomial-time Turing machine)**.** *We say that a Turing machine $M$ runs in polynomial time if there exists a polynomial $p(X)$ such that, for every input $x \in \{0,1\}^*$, the running time of $M$ on input $x$ is at most $p(|x|)$.*

There are many ways in which Turing machines can be defined. As a matter of fact, the definition above will most likely be different from any other resource you may compare these notes with. What matters, however, is that the definition of polynomial-time machine are all equivalent: if an algorithm is polynomial-time with respect to the definition in these notes, it will also be polynomial-time with respect to any other valid definition of Turing machine (and vice-versa).

Next, we define what deterministic Turing machines are. Essentially, these are those that never read any position on the left of the input.

**Definition 13** (Deterministic Turing machine)**.** *We say that a Turing machine $M$ is deterministic if for every input $x \in \{0,1\}^*$, the execution of $M$ on $x$ never sets the position of the header $s$ to a negative value. When a Turing machine is not deterministic we say that it is probabilistic.*

We call the random bits read by a probabilistic machine *the random coins* of the execution.

## 1.3 Hash Functions**

Informally speaking, a hash function is a (deterministic) function $\mathsf{Hash} : \mathcal{X} \to \mathcal{Y}$ for which it is hard to find *collisions*, i.e., pairs of distinct elements $x_0, x_1 \in \mathcal{X}$ such that $\mathsf{Hash}(x_0) = \mathsf{Hash}(x_1)$. Without any further requirements, obtaining a function with this property is straightforward. For instance, if $\mathcal{X} = \mathcal{Y}$, we can consider the identity function $\mathsf{id} : x \mapsto x$. Finding collisions for this function is simply impossible because they don't exist!

That is why we additionally require hash functions to be *compressing*, i.e., $|\mathcal{X}| > |\mathcal{Y}|$. For any such function, collisions always exist, but we want that finding any of them either requires a disproportionate amount of computations or a disproportionate amount of good luck.

**A formal definition.** We provide a formal mathematical definition of hash function.

**Definition 14** (Hash Function)**.** *Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets such that $|\mathcal{X}| > |\mathcal{Y}|$. Let $T \in \mathbb{N}$ and $\epsilon \in [0, 1]$ be two values. A $(T, \epsilon)$-secure (keyless) hash function with domain $\mathcal{X}$ and range $\mathcal{Y}$ is a deterministic polynomial-time algorithm* $\mathsf{Hash}$ *such that, for every (uniform) probabilistic algorithm $\mathcal{A}$ that runs in time at most $T$, it holds that*

$$\Pr\left[\begin{array}{c} x_0 \neq x_1 \\ \mathsf{Hash}(x_0) = \mathsf{Hash}(x_1) \end{array} \middle| (x_0, x_1) \xleftarrow{\$} \mathcal{A} \right] \leq \epsilon.$$

*The above probability is taken over the randomness of the algorithm $\mathcal{A}$.*

Essentially, to understand the above definition, we should view the algorithm $\mathcal{A}$ as an adversary whose goal is to find a collision for the hash function: if $\mathcal{A}$ runs in time at most $T$, the probability of succeeded will remain smaller than $\epsilon$. The higher the value of $T$ and the lower the value of $\epsilon$, the more secure the hash function. Typically, we are interested in hash functions where $T$ and $\epsilon$ guarantee that, even if we take the most powerful computer on earth and we run it for a billion times the age of the universe, the probability of finding a collision is as low as the probability that a meteorite hits you in the next second: essentially, it is (almost) impossible!

**Theoretical attacks.** In order to understand how a secure hash function looks like, let's try to break one! Take any hash function $\mathsf{Hash} : \mathcal{X} \to \mathcal{Y}$, where $|\mathcal{X}| \geq |\mathcal{Y}|$. For simplicity, suppose that $\mathcal{X}$ is finite (in many hash functions, e.g, $\mathsf{SHA2}$ or $\mathsf{SHA3}$, that is not the case, however, the attacks we are going to discuss still apply as we can always restrict the domain to a finite subset). The most straightforward attack would be to sample $x_0 \xleftarrow{\$} \mathcal{X}$ and $x_1 \xleftarrow{\$} \mathcal{X}$, hoping that $\mathsf{Hash}(x_0) = \mathsf{Hash}(x_1)$. It does not matter how $\mathsf{Hash}$ is constructed, it is not too hard to prove that the attack succeeds with probability $p \geq |\mathcal{Y}|^{-1} - |\mathcal{X}|^{-1}$. When $|\mathcal{X}| \gg |\mathcal{Y}|$, this probability roughly becomes $|\mathcal{Y}|^{-1}$.

Of course, in case of a failure, we can expect the attacker to retry. This leads to the following generalisation: the adversary samples $N$ random values $x_1, \ldots, x_N$ from $\mathcal{X}$ and check whether any pair gives a collision. By the union bound, the probability of success becomes

$$\Pr\left[\exists i, j : \begin{array}{c} x_i \neq x_j \\ \mathsf{Hash}(x_i) = \mathsf{Hash}(x_j) \end{array}\right] \leq \sum_{i,j \in [N]} \Pr\left[\begin{array}{c} x_i \neq x_j \\ \mathsf{Hash}(x_i) = \mathsf{Hash}(x_j) \end{array}\right] \leq N^2 \cdot p.$$

This turns out to be a tight bound as it can be proven that if $N \leq \sqrt{2/p}$, it holds that

$$\Pr\left[\exists i, j : \begin{array}{c} x_i \neq x_j \\ \mathsf{Hash}(x_i) = \mathsf{Hash}(x_j) \end{array}\right] \geq \frac{N(N-1)}{4} \cdot p.$$

**Summary.** From our analysis, we learn an important lesson:

*in order to be secure, a hash function needs to have a big range $\mathcal{Y}$!*

For example, the hash functions in use today, such as SHA2 or SHA3, have range $\mathcal{Y} = \{0,1\}^{256}$. The probability that two random inputs have the same digests is conjectured to be extremely small, specifically $p \approx 1/2^{256}$. In order to have good enough chances of finding a collision, an attacker would need to sample $N \approx 2^{128}$ inputs (that is why we say that the standard versions of SHA2 and SHA3 have 128 bits of security). Since they would need to evaluate Hash for each such value, this is an astronomic amount of computation and memory, far exceeding what today's best computers could ever achieve! Even if one would have enormous luck (say like winning the lotto twice in a row), the computation would require several geological eras.

### 1.3.1 Random Oracles**

The outputs of many hash functions behave somewhat *random* and *unpredictably*. These constructions are often used as *random oracles*. It is important to know that there is nothing random in a hash function: its outputs are just the result of a deterministic computation. However, for good random oracle instantiations, if we run any "interesting" statistics (e.g., average, variance, etc.) on any arbitrarily chosen set of outputs of the hash function, the results are essentially the same as if we run these on uniformly random strings.

Unfortunately, there is no formal definition of what a good random oracle is other than this handwavy description (that is why random oracles are somewhat contentious topic in cryptography). The concept has nevertheless been extremely influential in practice and today lies at the foundation of Internet security. Although not all hash functions would be suitable to be used as a random oracle, the most popular ones such as SHA2 and SHA3 behave sufficiently well for this purpose.

## 1.4 Digital Signatures**

Digital signatures are a fundamental tool for message authentication. Formally, a signature consists of a tag $s$ that links a message $m$ to the signer, identified through a public key pk: anybody holding $m$, $s$ and pk can verify that $s$ is indeed consistent with $m$ and pk. Deriving such tag $s$ is general extremely hard unless we know a secret key sk, which is generated by the signer together with pk. To summarise, a signature scheme guarantees that, if we receive a valid signature $s$ on a message $m$ with public key pk, this was generated by somebody that knows the secret counterpart sk of pk. In other words, as long as the signer keeps their secret key sk private, nobody can issue signatures that verify under pk other than the signer themselves.

This suggest the following use-case scenario. Imagine that we have a group of users $P_1, \ldots, P_n$ that intend to communicate over the Internet. They start by locally generating signature key pairs: each party $P_i$ will generate a random key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$, they disclose $\mathsf{pk}_i$ to everybody else, and (fundamentally) they keep $\mathsf{sk}_i$ secret. Every time that, in the future, $P_i$ wants to send a message $m$ to somebody else, they will append a signature $s$ on $m$ generated using $\mathsf{sk}_i$. In this way, $s$ proves to any recipient that the message $m$ was sent indeed by $P_i$ and not by somebody else that tries to impersonate them. For example, this prevents the following attacks:

**Change the message:** Imagine that the attacker controls the closest router to $P_i$. As soon as $P_i$ sends $(m, s)$ to everybody else, the attacker intercepts the message and sends $(m', s)$ to the other parties, where $m' \neq m$.

**Change the sender:** Imagine again that the attacker controls the closest router to $P_i$. As soon as $P_i$ sends $(m, s)$ to everybody else, the attacker intercepts the message and sends $(m, s)$ to the other parties, claiming that this was sent sent by another party $P_j$.

In both attacks, the recipients will notice that $s$ is not consistent with the claimed message and sender. Consequently, they will reject the communication.

**A formal definition.** We are ready to formally define this primitive. We start by defining the syntax and the correctness property.

**Definition 15** (Digital Signature)**.** *A digital signature scheme consists of a triple of polynomial time algorithms* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *with the following syntax:*

$\mathsf{KeyGen}$**:** *This algorithm is probabilistic, and takes no input. The output is a key pair* $(\mathsf{pk}, \mathsf{sk})$*.*

$\mathsf{Sign}$**:** *This algorithm is oftentimes probabilistic, and takes as input the secret key* $\mathsf{sk}$ *and a message* $m$*. The output is a signature* $s$*.*

$\mathsf{Verify}$**:** *This algorithm is deterministic, and takes as input a public key* $\mathsf{pk}$*, a message* $m$ *and a signature* $s$*. The output is a bit: 0 stands for "*reject*", 1 stands for "*accept*".*

*We additionally ask that, for every message* $m$*, it holds that*

$$\Pr\left[\mathsf{Verify}(\mathsf{pk}, m, s) = 1 \middle| \begin{array}{l}(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen} \\ s \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, m)\end{array}\right] = 1.$$

Essentially, the above property states that the verification algorithm $\mathsf{Verify}$ always accepts the signatures produced by the signing algorithm $\mathsf{Sign}$, on input the private counterpart of $\mathsf{pk}$.

Notice that so far we have not yet discussed the security requirements, i.e., that if we do not know the secret key, generating valid signatures that verify under a given public key $\mathsf{pk}$ either requires an extreme amount of computation or an extreme amount of luck. Additionally, we want this to hold even if the attacker were to receive signatures generated by the honest signer. Indeed, while the attacker tries to forge, the honest signer will continue to issue valid signatures to authenticate their communication. We should expect that some of these, if not all, will become known to the attacker. These must not make their task any easier: forging new signatures must still remain an intractable task! We formalise these properties as follows.

**Definition 16** (Unforgeability)**.** *Let* $T \in \mathbb{N}$ *and* $\epsilon \in [0, 1]$ *be two values. We say that a signature scheme* $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *is* $(T, \epsilon)$*-unforgeable if, for every polynomial time algorithm* $\mathcal{A}$ *that runs in time at most* $T$*, it holds that*

$$\Pr\left[\begin{array}{c}\mathsf{Verify}(\mathsf{pk}, m, s) = 1 \\ m \notin \mathcal{Q}\end{array}\middle| \begin{array}{l}(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen} \\ (m, s) \xleftarrow{\$} \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(\mathsf{pk})\end{array}\right] \leq \epsilon.$$

*Above, we use* $\mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}$ *to mean that* $\mathcal{A}$ *has unrestricted access to the oracle that, on input any message* $m'$*, responds with the signature* $s' \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, m')$*. In other words,* $\mathcal{A}$ *can query the oracle for signatures on messages of their choice.*[1] *This can be done as many times as the adversary wants. We denote the set of queried messages by* $\mathcal{Q}$*.*

Observe that the adversary $\mathcal{A}$ knows valid signatures for all the messages in $\mathcal{Q}$. These were provided by the oracle! That is why the above definition asks that the probability that $\mathcal{A}$ outputs a valid signature on a message *not in* $\mathcal{Q}$ remains smaller than $\epsilon$.

**Theoretical attacks.** As for hash functions, to understand how a secure signature scheme looks like, it is a good idea to try to attack one. There are two straightforward attempts one could try:

[1] We assume that the Turing machine describing $\mathcal{A}$ has a special state $q_{\mathsf{query}}$. When that state is reached, we expect the queried message to be encoded on the tape starting from position 0. We provide the response to $\mathcal{A}$ by "manually" encoding it on the tape starting from the first empty position on the right. At that point, we resume the execution of $\mathcal{A}$ starting from the moment it reached state $q_{\mathsf{query}}$. The operations computed by the oracle and the encoding of the response are not counted towards the running time of $\mathcal{A}$.

**Guess the secret-key:** Here, the attacker samples a random secret key from the key space $\mathcal{K}$, hoping to be lucky enough to obtain the private counterpart of $\mathsf{pk}$. That would allow them to efficiently generate as many forgery as they want. The success probability is $|\mathcal{K}|^{-1}$. For instance, if $\mathcal{K} = \{0,1\}^{128}$, the probability of finding the correct secret key at the first try is $2^{-128}$, which is extremely low. Of course, one should expect the attacker to repeat this attack many times. However, to make the success probability remotely high (say with probability $1/2$), the attacker would need to retry for $\approx 2^{128}$ times. This is an astronomic amount of computations! Observe that if the attacker were to guess the key correctly, they could easily verify it: just use the candidate secret key to sign a fixed message $m$ and see whether the signature verifies under $\mathsf{pk}$.

**Guess a signature:** The other straightforward attack is to try to directly guess the signature for a specific message $m$. Specifically, the attacker could sample a random $s$ from the signature space $\mathcal{S}$ and then checks whether $\mathsf{Verify}(\mathsf{pk}, m, s) = 1$. Notice that there may exists multiple valid signatures for the same message $m$. Suppose the probability of finding one of these is $N \cdot |\mathcal{S}|^{-1}$. To protect ourselves against this attack, it is necessary that $\mathcal{S}$ is much larger than $N$. For instance, $|\mathcal{S}| = 2^{128} \cdot N$.

Many signature schemes used today use a choice of parameters that is roughly double of what we described. For instance, elliptic curve-based signatures such as ECDSA and EdDSA typically have random 256-bit secret keys (there exists stronger variants that have 512-bit keys). Moreover, there are $N \approx 2^{256}$ different signatures per message, and the signature space contains $|\mathcal{S}| \approx 2^{512}$ elements (in the stronger variants, $N$ and $\mathcal{S}$ are larger and satisfy $N \approx |\mathcal{K}|$ and $|\mathcal{S}| \approx N^2$).

Due to the existence of more sophisticated attacks, other signature schemes such as RSA signatures and DSA, or the recently introduced post-quantum constructions ML-DSA (Dilithium), SLH-DSA (SPHINKS+) and FN-DSA (Falcon) require even larger secret keys and signature spaces.

# Chapter 2

# The Consensus Problem

In 2026, most people's understanding of the subject of this course begins and ends with the word "Bitcoin" (and, perhaps, "Ethereum"). Even the few who have heard the word "blockchain" tend to reduce it to a matter of money (and, unfortunately, of unrealistic promises of easy riches). By the end of the course, I hope to have convinced you that a blockchain is much more than a financial instrument, and that it is best thought of as a *distributed ledger*.

What do I mean by these two words? The best real-world analogy for a blockchain is a public book: anybody can read it, and anybody can add a page to it without fear of censorship. In a sense, the book represents a reality that everybody agrees on and that nobody can tamper with, however hard they try.

So far, blockchains have been deployed mainly for economic and financial purposes, which is why the general public associates the word "crypto" with "cryptocurrency" rather than the more appropriate "cryptography". In these applications, the public book simply records every transfer of money and goods, making it possible to keep track of who owns what (and whether somebody can afford to buy something). Because anybody can write in the book, nobody can be barred from taking part in the economy.

## 2.1 Byzantine Agreement and Secure Broadcast

Before talking about blockchains and all their fancy machinery, it is important that we introduce the basic problems that they are trying to solve: Byzantine agreement and secure broadcast.

### 2.1.1 Byzantine Agreement

This problem is often introduced in textbooks through a story.

> *The army of Byzantium is besieging a city. Its control is split between several generals, who spread along with their divisions to occupy the strategic points surrounding the city. After months, the generals face a decision: shall they jointly attack the city from all directions to bring the war to an early end, or shall they hold on and wait for the city to surrender peacefully? Doesn't matter what choice they make, it is fundamental that they reach consensus. If the generals do not coordinate and only a subset of them were to attack the city, they would be more likely to be defeated, weakening the army as a whole with the consequent risk of losing the war. Unfortunately, the generals cannot meet in person to take this decision as they need to hold their positions, so they communicate by relying on messengers (these are the old days where Internet and telephones are yet to be invented). How can they solve their problem?*

Of course, a simple solution is to use the messengers to communicate their preference and then take a majority vote (in case of a tie, the generals decide not to attack). However, what if some of the generals are actually traitors that are secretly allied with the besieged city? In that case, we can assume that they will

try to prevent agreement among the honest generals by sending inconsistent communications, so that only a small subgroup of them attacks the city. The defending forces will have an easier time defeating them. If they succeed at that, they will have weakened the rival army and, perhaps, strip it of the strength necessary to control all the strategic points surrounding the city.

Notice that if we try to run an economy on the blockchain (as Bitcoin and Ethereum and many others are trying to do), we need to solve a problem very similar to that of the generals: if the ledger stores a list of all movements of money and goods, it is important that all honest users agree on the content of this list *all the time*, even if a subset of participants misbehaves and potentially colludes to perform an attack. To further motivate this, imagine that an attacker Eve bought a holiday house for 1000000ℵ from Alice, but at the same time managed to convince Bob that this transaction never occurred. Then, Eve has the opportunity to *double spend* her money, and buy a villa for 1000000ℵ from Bob. In Alice's world, everything is fine; her version of the ledger registered the transaction and now she owns 1000000ℵ, no transaction between Eve and Bob ever occurred. In Bob's world, instead, Alice's house was never sold to Eve, and his version of the ledger says that the 1000000ℵ were exchanged with him. As long as Alice and Bob are not aware of the inconsistency between their realities, Eve is able to enjoy both the villa and the holiday house.

This problem is called *the Byzantine agreement problem*, or, from the name of the army in the story, *the Byzantine agreement problem*.

**A formal definition.** Next, we will give a proper mathematical definition. Below (and in the rest of these notes), we use the term *protocol* to denote a series of instructions that protocol participants are expected to follow. A typical instruction looks like this:

*When receiving a message of a certain form from a certain party, update your state using some algorithm and send some other message to a set of other users.*

We will see plenty of examples during the course.

**Definition 17** (Byzantine agreement protocol)**.** *A Byzantine agreement protocol is an interactive protocols between $n$ participants $P_1, \dots, P_n$. The input of each party $P_i$ consists of their preference $x_i \in \{0,1\}^*$, $P_i$'s output is a value $y_i \in \{0,1\}^* \cup \{\perp\}$. We use the special symbol $\perp$ to denote failure, perhaps because from $P_i$'s perspective the protocol never terminates.*

It is useful to imagine that protocol participants as stateful, probabilistic polynomial-time Turing machines that are connected by point-to-point channels, whose exact behaviour is uniquely specified by the protocol. In particular, each party $P_i$ is a machine with $2n + 1$ tapes: a work tape, an input tape, an output tape, and two message tapes for the communication with every other party $P_j$ (one for incoming messages and one for outgoing messages). These machines are dormant most of the time, but activate when some message is written on any incoming message tape, or on the input tape. In this event, the machine performs some computation on the work tape based on the content of its tapes and terminates by writing some messages on the outgoing tapes or on the output tape. The outgoing messages are then sent over the right point-to-point channel and copied on the incoming message tape of the correct recipient. The content of the work tape will not be erased and will be updated activation after activation. At the beginning of the protocol, the input tape of the honest parties contains their input, while all other tapes are empty. The outputs consists of the content of the output tapes at the end of the protocol.

We say that a party is *honest* if its machine is exactly the one specified by the protocol. We call the other parties *corrupted*. We denote the set of honest parties by $H$ and the set of corrupted parties by $C$. Notice that we cannot assume that the honest parties know who else is honest. On the contrary, we can assume that the corrupted parties do. As a matter of fact, the corrupted parties may be colluding against everybody else.

In the definition of Byzantine agreement protocol, we have not specified what security and correctness properties this protocol needs to satisfy. We are going to list them below. However, as we are going to explain soon, we can guarantee these properties to hold only when the corrupted parties are not too many.

The first property requires that the honest parties event reach a conclusion and this holds even if a subset of corrupted players tries to prevent that.

**Definition 18** (Termination)**.** *We say that a Byzantine agreement protocol satisfies $\epsilon$-termination against $t$ corruptions if, for every protocol execution where $|C| \leq t$, every honest party eventually writes a value on the output tape, except with probability at most $\epsilon$.*

The second property guarantees that the honest parties will (almost) always agree on the output. Notice that we cannot prevent corrupted parties from disagreeing as they may behave arbitrarily.

**Definition 19** (Agreement)**.** *We say that a Byzantine agreement protocol satisfies $\epsilon$-agreement against $t$ corruptions if, for every protocol execution where $|C| \leq t$, we have that $y_i = y_j$ for every $i, j \in H$, except with probability at most $\epsilon$.*

The last property we require is that if all honest parties have the same preference, then the final output coincides with it. Essentially, this means that if the honest parties already agree on something, no attacker can convince them about the opposite.

**Definition 20** (Validity)**.** *We say that a Byzantine agreement protocol satisfies $\epsilon$-validity against $t$ corruptions if, for every protocol execution where $|C| \leq t$ and where there exists $x$ such that $x_i = x$ for every $i \in H$, we have that $y_i = x$ for every $i \in H$, except with probability at most $\epsilon$.*

We say that a Byzantine agreement protocol is $\epsilon$-secure against $t$ corruptions, if it satisfies $\epsilon$-termination, $\epsilon$-agreement and $\epsilon$-validity against $t$ corruptions. Since $\epsilon$ represents the probability that the protocol fails at ensuring security and correctness, we would like $\epsilon$ to be as small as possible. The best case is $\epsilon = 0$. In this case, we say that the protocol is perfectly secure. Sometimes, we are content with very small values of $\epsilon$, for instance $\epsilon \approx 2^{-128}$. Big values like $\epsilon = 1/4$ or even $\epsilon = 1/100000$ are instead unacceptable for applications.

**Exercise 1.** *There exists a stronger form of validity which is called* strong validity*: we say that a Byzantine agreement protocol satisfies $\epsilon$-strong validity against $t$ corruptions if, for every protocol execution where $|C| \leq t$ and for every honest party $P_i$, there exists another (possibly the same) honest party $P_j$ such that $y_i = x_j$, except with probability at most $\epsilon$. Prove that an $\epsilon$-strongly valid Byzantine agreement protocol is also $\epsilon$-valid.*

**Some broken attempts.** To better understand Byzantine agreement protocols, let us try to build some insecure examples and let us examine where they fail. Possibly, the most straightforward attempt is to ask all participants to reveal their preference to everybody else, and then take a majority vote.

One first issue with this approach is that it is unclear what happens if any participant does not send their preference to somebody else. We cannot ask the recipient to wait indefinitely; otherwise, the protocol will never terminate.

Another issue is validity. Suppose that more than half of the participants are corrupted. Then, even if all honest parties agree on their preference (let us say this is 0), the corrupted parties can force the output to be 1 by choosing 1 as their preference.

Finally, even agreement does not hold. Suppose that the honest parties are split: half of them prefer 0 and the other half prefers 1. Then, a single corrupted party Eve can bring disagreement in the group by claiming 0 as a preference to half of the group and 1 to everybody else. One of course can try to prevent this issue by asking the honest parties to compare what they received, trying to detect and discard any inconsistent communication from the corrupted parties. However, this is complicated by the fact that the honest parties do not know who else is honest. This gives an opportunity to the corrupted parties to disrupt this detection attempt by sending further inconsistent communication. For example, suppose that Bob told you that his preference is 1. However, later, Alice claims that Bob told her that he prefers 0, while Bob denies. Who is right? Well, without further information, there is no way we can learn it. It could be that Alice is right and Bob is lying, or it could be viceversa.

**Secure Byzantine agreement against $n/2$ corruptions is impossible.** Before focusing on building a Byzantine agreement protocol, we start with bad news: as we have anticipated, if the number of corrupted parties is too high, Byzantine agreement protocols are inherently insecure. In particular, a protocol can be

secure only if the corrupted parties are strictly less than half of all participants. This is formalised in the following theorem.

**Theorem 2** (Secure Byzantine agreement needs an honest majority)**.** *Suppose that $\epsilon < 1/3$. Then, there exists no $\epsilon$-secure $n$-party Byzantine agreement protocol that withstands $t \geq n/2$ corruptions.*

*Proof.* We prove this by contradiction. Suppose that the theorem is false and there exists a Byzantine agreement protocol that achieves perfect termination, perfect agreement and perfect validity for some $t \geq n/2$.

Now, partition the $n$ parties into two arbitrary subgroups $S_0$ and $S_1$ such that $|S_0|, |S_1| \leq t$ (this is possible because $t \geq n/2$). We consider the following protocol executions:

**Execution 1:** In this execution, there are no corrupted parties, but all parties in $S_0$ have input 0 and all parties in $S_1$ have input 1. By $\epsilon$-agreement, we know that the parties should output the same value $v$ (except with probability at most $\epsilon$).

**Execution 2:** In this execution, the parties in $S_0$ are honest and those in $S_1$ are corrupted. All parties in $S_0$ have input 0. The corrupted parties in $S_1$ behave as honest parties on input 1 (yes, a corrupted party can behave as they please, so they can also behave honestly). By $\epsilon$-validity, all parties in $S_0$ must output 0 (except with probability at most $\epsilon$).

**Execution 3:** This is the opposite scenario of Execution 2: the parties in $S_1$ are honest and those in $S_0$ are corrupted. All parties in $S_1$ have input 1. The corrupted parties in $S_0$ behave as honest parties on input 0. By $\epsilon$-validity, all parties in $S_1$ must output 1 (except with probability at most $\epsilon$).

To conclude the proof, we observe that the three scenarios are contradicting each other. This is because *they are actually all the same!* In all three executions, the Turing machines used by the parties compute the same operations. From validity in Execution 2, we have that the probability that the parties in $S_0$ do not output 0 is at most $\epsilon$. From validity in Execution 3, we have that the probability that the parties in $S_1$ do not output 1 is at most $\epsilon$. So, by the union bound, the probability that the parties in $S_0$ output 0 *and* the parties in $S_1$ output 1 is at least $1 - 2\epsilon > \epsilon$. However, this contradicts $\epsilon$-agreement in Execution 1 (because $S_0$ and $S_1$ output different values.

In conclusion, there is something that does not work. What? Well, it must be that we began from a wrong premise: there is no $\epsilon$-secure Byzantine agreement protocol that withstands $t \geq n/2$ corruptions! □

**Exercise 2.** *Design a Byzantine agreement protocol that satisfies perfect termination and perfect agreement against any number of corrupted parties, but not validity.*

**Exercise 3.** *Design a Byzantine agreement protocol that satisfies perfect termination and perfect (strong) validity against any number of corrupted parties, but not agreement.*

### 2.1.2 Secure Broadcast

Our first example of secure Byzantine agreement will be based on the related notion of *secure broadcast*. This is a protocol in which one party (called the sender) delivers the same message $m$ to all other participants (called the receivers). Similarly to Byzantine agreement, we want the recipients to reach agreement on $m$ and termination, which means that, even if the sender and some of the recipients are corrupted, the honest participants should eventually output the same message. Moreover, we would like that if the sender is honest, the honest receivers exactly output the message chosen by the sender.

**A broken attempt.** As usual, let us try to build a secure broadcast and see what goes wrong. The trivial approach is to consider the protocol in which the sender sends the message $m$ to all recipients, who directly output it. While this satisfies validity, it does not satisfy agreement when the sender is corrupted. Indeed, nothing prevents a corrupted sender to deliver $m$ to a subset of recipients and another message $m'$ to

everybody else. One attempt to solve this issue is by making the recipients compare what they have received, however, we quickly run in the same kind of issue as in Byzantine agreement: the corrupted recipients may disrupt the operation by sending inconsistent information to the other honest participants.

**A formal definition.** Next, we provide formal mathematical definitions for secure broadcast. This follows a similar blueprint of that for secure Byzantine agreement.

**Definition 21** (Broadcast protocol)**.** *A broadcast protocol is an interactive protocols between a sender $P_1$ and $n-1$ receivers $P_1, \dots, P_{n-1}$. The input of the sender is a message $m \in \{0,1\}^*$, while all other parties have input equal to the constant 0. On the other hand, every party $P_i$ outputs a message $m_i \in \{0,1\}^* \cup \{\perp\}$. We use the special symbol $\perp$ to denote failure, for instance because $P_i$ never terminates its execution.*

The above definition tells nothing about the security and correctness requirements for such a protocol. We define these below. As for Byzantine agreement, these are called *termination*, *agreement* and *validity*. The exact meaning is slightly different given the different objective of the protocol.

**Definition 22** (Termination)**.** *We say that a broadcast protocol satisfies $\epsilon$-termination against $t$ corruptions if, for every protocol execution where $|C| \leq t$ (potentially including the sender), every honest party eventually writes a value on the output tape, except with probability at most $\epsilon$.*

**Definition 23** (Agreement)**.** *We say that a broadcast protocol satisfies $\epsilon$-agreement against $t$ corruptions if, for every protocol execution where $|C| \leq t$ (potentially including the sender), we have that $m_i = m_j$ for every $i, j \in H$, except with probability at most $\epsilon$.*

**Definition 24** (Validity)**.** *We say that a broadcast protocol satisfies $\epsilon$-validity against $t$ corruptions if, for every protocol execution where $|C| \leq t$ and where the sender is honest, we have that $m_i = m$ for every $i \in H$, except with probability at most $\epsilon$.*

We say that a broadcast protocol is $\epsilon$-secure against $t$ corruptions, if it satisfies $\epsilon$-termination, $\epsilon$-agreement and $\epsilon$-validity against $t$ corruptions. Since $\epsilon$ represents the probability that the protocol fails at ensuring security and correctness, we would like $\epsilon$ to be as small as possible. The best case is $\epsilon = 0$. In this case, we say that the protocol is perfectly secure. Sometimes, we are content with very small values of $\epsilon$, for instance $\epsilon \approx 2^{-128}$. Big values like $\epsilon = 1/4$ or even $\epsilon = 1/100000$ are instead unacceptable for applications.

**The connection between secure broadcast and secure Byzantine agreement.** Secure broadcast is closely connected to secure Byzantine agreement. Indeed, we can easily build one from the other

**From Byzantine Agreement to Broadcast:** Suppose we have a secure Byzantine agreement protocol. We can build a secure broadcast protocol as follows: the sender delivers their message $m$ to all receivers. After some time, all receivers run the Byzantine agreement protocol using the received message as input and output the result (if any recipient does not receive any message from the sender in time for the Byzantine agreement protocol, they input 0). This ensures agreement. Moreover, if the sender is honest, all honest recipients run the broadcast protocol on the same input, so validity of the Byzantine agreement protocol guarantees the validity of the secure broadcast.

**From Broadcast to Byzantine Agreement:** Suppose we have a secure broadcast protocol. We can build a secure Byzantine agreement protocol against $t < n/2$ corruptions as follows: for every $i$, the participants run an instance of the secure broadcast protocol where $P_i$ acts as the sender and delivers their preference to everybody else. After these $n$ executions, the parties output the most popular preference (if two or more preferences are equally popular, the parties output the first one in lexicographical order). Agreement follows from the fact that all honest player will agree on the message broadcast by each other participant, so they all compute the output in the same way. Validity holds because, for every honest party $P_i$, the other honest parties know what's $P_i$'s preference. If the honest parties have all the same preference, that must be the most popular preference (we recall that the honest parties are in majority).

**Exercise 4.** *Formally prove that if we start from an $\epsilon$-secure Byzantine agreement protocol, then the first construction we described gives an $\epsilon$-secure broadcast protocol. Easier case: prove this for $\epsilon = 0$.*

**Exercise 5.** *Consider the variant of the construction of secure broadcast from secure Byzantine agreement where the receivers start running the Byzantine agreement protocol only after they have received a message from the sender. Why would this protocol not be secure?*

**Exercise 6.** *Formally prove that if we start from an $\epsilon$-secure broadcast protocol, then the second construction we described gives a $(n \cdot \epsilon)$-secure Byzantine agreement protocol. Easier case: prove this for $\epsilon = 0$.*

**Exercise 7.** *Consider the construction of secure Byzantine agreement from secure broadcast. Suppose that the broadcast protocol is secure against $t \geq n/2$ corruptions. Why is the Byzantine agreement protocol still not secure against $t \geq n/2$ corruptions? Specifically, which property between termination, agreement and validity is violated? (This makes sense, given that we have proven that no Byzantine agreement protocol can be secure against $t \geq n/2$ corruptions)*

### 2.1.3 The Dolev-Strong Broadcast Protocol

We present the first example of a secure broadcast protocol that withstands any $t$ corruptions. This is due to Dolev and Strong. Together with the observation in the last section, this gives a secure Byzantine agreement protocol.

This construction relies on a public-key infrastructure (PKI). This is a technical way of saying that the protocol works in a scenario in which each participant $P_i$ holds a signature key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$, where $\mathsf{pk}_i$ is already known to everybody, while $\mathsf{sk}_i$ is secret.

The protocol proceeds in $t+2$ rounds. Each of these has a fixed duration. We assume that all participants have a synchronised clock which the use to move on a timely manner to the next round.

- In round 0, every receiver $P_j$ defines a set $S_j = \emptyset$. The sender $P_1$ generates a signature on their message $m$ using their secret key
$$s_1 \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_i, m).$$
Then, they send $(m, (1, s_1))$ to all recipients.

- In round $r$, each party $P_j$ receives tuples of the form $(\hat{m}, (i_1, \hat{s}_1), \ldots, (i_\ell, \hat{s}_\ell))$. These were sent in the previous round. Party $P_j$ sanitises each of them. That is:
  - $P_j$ removes any $(i_h, \hat{s}_h)$ such that $\mathsf{Verify}(\mathsf{pk}_{i_h}, \hat{m}, \hat{s}_h) = 0$.
  - After that, for any $h < k$ such that $i_h = i_k$, party $P_j$ removes $(i_k, \hat{s}_k)$.

  If there are at least $r$ signatures left, one of them is from the sender (i.e., there exists $h$ such that $i_h = 1$) and none of them is from $P_j$ themselves, party $P_j$ adds $\hat{m}$ to $S_j$. If $r \leq t$ and $\hat{m}$ was not already in $S_j$, party $P_j$ also computes
$$s_j \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_j, \hat{m}),$$
appends $(j, s_j)$ to the sanitised list and sends it to all other participants. If instead we are in the final round $r = t+1$ and $S_j$ contains a single element, party $P_j$ outputs that value. Otherwise, they output 0.

**Theorem 3.** *Suppose that $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is an $(T, \epsilon)$-unforgeable signature scheme. If the sum of the running time of all parties (honest and corrupted) is smaller than $T$, the Dolev-Strong protocol is a $(n \cdot \epsilon)$-secure broadcast protocol against $t$ corruptions.*

*Proof.* We show that the protocol satisfies all three properties of a secure broadcast protocol.

**Claim 1.** *The Dolev-Strong protocol satisfies perfect termination.*

*Proof.* This is easy to see: all honest parties move in a timely way from one round to the next, relying on their clock. They always terminate the last round by outputting some value. □

**Claim 2.** *The Dolev-Strong protocol satisfies $\epsilon$-validity.*

*Proof.* Suppose that the sender is honest and their input is $m$. In round 1, each honest recipient $P_j$ will receive $(m, (1, s_1))$ from the honest sender. Since the signature $s_1$ always verifies, each honest recipient $P_j$ will add $m$ to the set $S_j$. Notice that no element is removed from $S_j$ during the protocol, so in round $t+1$, the set $S_j$ will still contain $m$. This will be also the only element, indeed, $P_1$ never signs any element other than $m$, and any honest recipient $P_j$ adds a message to their set $S_j$ only when this is signed by $P_1$. In other words, to prevent an honest recipient from outputting $m$, the corrupted parties would need to forge $P_1$'s signature. By the security of the signature scheme that occurs only with probability at most $\epsilon$. □

**Claim 3.** *Suppose that an honest recipient $P_j$ adds the item $\hat{m}$ to $S_j$ at round $r \leq t$. Then, at the end of round $r+1$, every other honest parties $P_i$ will have $\hat{m}$ in their set $S_i$, except with probability at most $\epsilon$.*

*Proof.* Take any honest party $P_i \neq P_j$ and suppose that $S_i$ does not contain $\hat{m}$ at the end of round $r$. After adding $\hat{m}$ to $S_j$, party $P_j$ will send $\hat{m}$ to $P_i$ along with a list of valid signatures on $\hat{m}$ from at least $r+1$ distinct parties. This list includes the sender $P_1$ (otherwise, $P_j$ would have not added $\hat{m}$ to $S_j$) and $P_j$ (who appended their signature in round $r$). Notice that when $P_j$ received this list of signatures, their signature was not yet present (otherwise, they would have not forwarded $\hat{m}$ to $P_i$). Notice also that $P_i$'s signature is not included in the list: an honest party $P_i$ signs a message only after adding that message to their set $S_i$. Since $\hat{m} \notin S_i$ by the end of round $r$, there is no signature from $P_i$ in the list received by $P_j$ (unless the corrupted parties managed to forge a signature, which occurs with probability at most $\epsilon$). Putting all of these observations together, we conclude that, at the end of round $r+1$, party $P_i$ will add $\hat{m}$ to $S_i$. □

**Claim 4.** *The Dolev-Strong protocol satisfies $(n \cdot \epsilon)$-agreement.*

*Proof.* We show that if an honest party $P_j$ holds some message $\hat{m}$ in their set $S_j$ at the end of the protocol, then every other honest parties $P_i$ will also hold $\hat{m}$ in their set $S_i$. To see why, suppose that $P_j$ added $\hat{m}$ to $S_j$ in round $r \leq t$. Then, as we argued in Claim 3, in round $r+1$, party $P_i$ added $\hat{m}$ to $S_i$ (except with probability at most $\epsilon$). However, what if $P_j$ added $\hat{m}$ to $S_j$ only in the last round? Well, in that case it must have been that, in round $r+1$, they have received $t+1$ valid signatures on $\hat{m}$ from $t+1$ different parties. But there are at most $t$ corrupted parties, so at least one of these $t+1$ signers must have been an honest party $P_k$. This means that, unless this was a forged signature (which occurs only with probability at most $\epsilon$), the honest party $P_k$ must have added $\hat{m}$ to their set $S_k$ in some round $r \leq t$. Then, by Claim 3, in round $r+1$, party $P_i$ has added $\hat{m}$ to their set $S_i$ (except with probability $\epsilon$).

This proves that, by the end of the protocol, all honest parties have the same set except with probability at most $2n \cdot \epsilon$ (here, we use the union bound). Therefore, they all output the same value. □

We have now proven that all three properties of a secure broadcast protocol hold. This terminates our proof of Theorem 3. □

**Strengths and limitations of Dolev-Strong.** Dolev-Strong has some important advantages. First of all, it is a very elegant and simple to describe construction. Secondly, unlike secure Byzantine agreement, it can support any number of corruptions, but on the downside, the number of rounds grows proportionally to it. (It turns out that this is actually mathematically unavoidable for protocols of this kind). The good news is that if all parties are honest, nobody speaks past the second round (nevertheless, the parties must wait until the last round before they output what they have received).

There are also some less explicit disadvantages. The first is that Dolev-Strong relies on a PKI. This raises the question of how the parties get to know (and agree) on the public keys of the other participants. The simplest solution is to rely on some trusted authority to distribute the public keys to all participants (this is more or less what the Internet does today: the entire system relies on a few authorities that issue certificates that bind the identity of clients and servers to their public keys. These certificates are signed

under the authority's key, which everybody can verify). However in some practical scenario, it is difficult for the parties to agree on who to trust for this operation. Moreover, this introduces a single point of failure in the construction: if the trusted party misbehaves (and potentially colludes with the corrupted participants), the protocol may no longer be secure. The use of the PKI is also problematic because it assumes that the procotol participants already know who else is joining the protocol (or at least, they already need to know their public key). In large systems, this can be problematic: we cannot assume that we know the public keys of all other participants (these would simply be too many), but we nevertheless may want to receive messages broadcast by somebody we do not know yet, or vice-versa, we may want our message to reach everybody, even people we do not know. To motivate this, consider again the classical problem of running an economy on a blockchain. In order to purchase something, I need to broadcast the information to the whole economy, so that it can be added to the ledger. Moreover, all user should receive the exact same message even if I misbehave, otherwise, they may be vulnerable to double spending attacks. In this scenario, the number of user is extremely large and dynamic (there may be a continuous flow of new users joining the system). It is infeasible for me to hold at all times an update list of all participants public keys.

The second hidden disadvantage of Dolev-Strong is the fact that it relies on clocks. That is why we say that Dolev-Strong is a *synchronous* protocol. First of all, we cannot really assume that the clocks are all perfectly synchronised (unless all parties have atomic clocks). In reality, all clocks have a perhaps small drift. Secondly, the protocol assumes that the parties communicate over an ideal network: when a message is sent, the receiver obtains it with no delay! This is an unrealistic assumption, we all know that Internet communications can have unpredictably large delays. The corrupted parties may even try to perform an attack by taking down the network! From the perspective of our protocol, any party that fails to deliver their message by the beginning of the next round is *a corrupted party* (does not matter what their original intention were). If there are too many of them, the protocol is no longer secure. To summarise, if too many party are late, they may never reach agreement. That is why it is important to choose the duration of the rounds so that we can be confident that all messages will be delivered in time. In other words, the more we wait, the more secure the protocol is. On the other hand, we do not really want to wait forever, because efficiency and usability are also important.

**Exercise 8.** *Write an implementation of Dolev-Strong using any computer language of your choice. You can try to have fun attacking the protocol.*

**Exercise 9.** *Show that if in any execution of Dolev-Strong the corrupted parties manage to forge any honest party signature with probability $\delta$, then given the code of the corrupted parties, we can design an explicit algorithm $\mathcal{A}$ running in time at most $T$ such that*

$$\Pr\left[\begin{array}{c|l} \mathsf{Verify}(\mathsf{pk}, m, s) = 1 & (\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen} \\ m \notin \mathcal{Q} & (m, s) \xleftarrow{\$} \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(\mathsf{pk}) \end{array}\right] \leq \delta/n$$

*where $\mathcal{Q}$ denotes the set of messages queried by $\mathcal{A}$ to the oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$ (check Definition 16).*

### 2.1.4 Bracha's Asynchronous Secure Broadcast Protocol

Next, we present another example of secure broadcast protocol. Compared to Dolev-Strong, it has some important advantages: it does not rely on a PKI, does not need any clocks, and is secure even if the network suffers from arbitrary delays. On the negative side, it only withstands $t < n/3$ corruptions.

The protocol also only satisfies a weaker form of termination which we call *asynchronous termination*. This says that if one honest party terminates, then all other honest parties must terminate too. Unfortunately, this form of termination is the best we can achieve in asynchronous broadcast protocols. Indeed, if the honest recipients receive no message from the sender, how can they know whether they are dealing with a corrupted sender that refuses to communicate, or an honest sender that has an extremely bad network with a lot of delay? To ensure validity, the second case would require the honest participants to wait until the bad network delivers the honest sender message, however, if we are in the first case, the honest parties will end up waiting

forever and never terminate. In synchronous protocols, the parties could simply rely on a clock and classify slow senders as corrupted senders. Without clocks, we can no longer do this. Moreover, in the asynchronous setting, slow parties are no longer necessarily corrupted. On the upside, asynchronous protocols guarantees security even to parties that are cursed by extremely slow networks.

**Definition 25** (Asynchronous termination)**.** *We say that a broadcast protocol satisfies $\epsilon$-asynchronous termination against $t$ corruptions if, for every protocol execution where $|C| \leq t$ (potentially including the sender), if any honest party eventually writes a value on the output tape, then all other honest parties will eventually write some value on their output tape, except with probability at most $\epsilon$.*

We say that a protocol is an $\epsilon$-secure asynchronous broadcast if it satisfies $\epsilon$-validity, $\epsilon$-agreement and $\epsilon$-asynchronous termination. If $\epsilon = 0$, we say that the asynchronous broadcast is perfectly secure.

**Exercise 10.** *Prove that a broadcast protocol that satisfies $\epsilon$-termination satisfies also $\epsilon$-asynchronous termination.*

**Bracha's asynchronous broadcast.** Next, we are going to present Bracha's protocol.

- The sender $P_1$ sends their message $(\mathsf{send}, m)$ to all other participants (including themselves).
- When a party $P_j$ receives a message $(\mathsf{send}, \hat{m})$ from $P_1$, they send $(\mathsf{echo}, \hat{m})$ to all other parties (including themselves). Any following message from $P_1$ of this form is ignored.
- When a party $P_j$ has received $(\mathsf{echo}, \hat{m})$ for the same message $\hat{m}$ from strictly more than $2n/3$ distinct parties, they send $(\mathsf{ready}, \hat{m})$ to everybody (including themselves), unless they have already sent a $\mathsf{ready}$ message.
- When a party $P_j$ has received $(\mathsf{ready}, \hat{m})$ for the same message $\hat{m}$ from strictly more than $n/3$ distinct parties, they send $(\mathsf{ready}, \hat{m})$ to everybody (including themselves), unless they have already sent a $\mathsf{ready}$ message.
- When a party $P_j$ has received $(\mathsf{ready}, \hat{m})$ for the same message $\hat{m}$ from strictly more than $2n/3$ distinct parties, they output $\hat{m}$ and terminate the protocol.

**Theorem 4.** *Suppose that every pair of parties is connected by a point-to-point channel and no corrupted party can send a message on the channel that connects two honest parties. Bracha's protocol is a perfectly secure asynchronous broadcast against $t < n/3$ corruptions.*

*Proof.* We proof that Bracha's protocol satisfies perfect validity, perfect agreement and perfect asynchronous termination.

**Claim 5.** *An honest party $P_i$ will never receive $(\mathsf{echo}, \hat{m}_1)$ for the same message $\hat{m}_1$ from strictly more than $2n/3$ distinct parties, along with $(\mathsf{echo}, \hat{m}_2)$ for the same message $\hat{m}_2 \neq \hat{m}_1$ from from strictly more than $2n/3$ distinct parties.*

*Proof.* We prove this result by contradiction. Suppose our claim is false. Call $S_1$ the set of parties that sent $(\mathsf{echo}, \hat{m}_1)$ to $P_i$, and call $S_2$ the set of parties that sent $(\mathsf{echo}, \hat{m}_2)$. Observe that

$$n \geq |S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2| > 4n/3 - |S_1 \cap S_2|.$$

All honest participants send at most one $\mathsf{echo}$ message, so the parties in $S_1 \cap S_2$ must all be corrupted. We conclude that

$$n > 4n/3 - |S_1 \cap S_2| \geq 4n/3 - t > 4n/3 - n/3 = n.$$

We have just proven that $n > n$, which is false. This means that we started from a wrong hypothesis, therefore our claim must be true. □

**Claim 6.** *An honest party $P_i$ will never receive $(\mathsf{ready}, \hat{m}_1)$ for the same message $\hat{m}_1$ from strictly more than $2n/3$ distinct parties, along with $(\mathsf{ready}, \hat{m}_2)$ for the same message $\hat{m}_2 \neq \hat{m}_1$ from from strictly more than $2n/3$ distinct parties.*

*Proof.* The argument is the same as in Claim 5. □

**Claim 7.** *Suppose that there exists an honest party that sends the message $(\mathsf{ready}, \hat{m})$. Then, there must exist an honest party (possibly a different one) that sent $(\mathsf{ready}, \hat{m})$ because they have received $(\mathsf{echo}, \hat{m})$ from strictly more than $2n/3$ distinct parties.*

*Proof.* Consider the first honest party $P_k$ that sent the message $(\mathsf{ready}, \hat{m})$. This party could not have sent their $\mathsf{ready}$ message because they had received $(\mathsf{ready}, \hat{m})$ from at least $n/3$ distinct parties. Indeed, there are strictly less than $n/3$ corrupted parties that could have sent this message and by the time they sent $(\mathsf{ready}, \hat{m})$, no other honest party had yet sent $(\mathsf{ready}, \hat{m})$. This means that $P_k$ must have sent $(\mathsf{ready}, \hat{m})$ because they had received $(\mathsf{echo}, \hat{m})$ from strictly more than $2n/3$ distinct parties. □

**Claim 8.** *Bracha's protocol satisfies perfect validity.*

*Proof.* Suppose that the sender $P_1$ is honest and wants to broadcast a message $m$. Once they have received $m$ from $P_1$, every honest party $P_j$ sends $(\mathsf{echo}, m)$ to everybody. This means that all honest parties receive $(\mathsf{echo}, m)$ from all other honest parties. In this way, they receive messages of this kind from strictly more than $2n/3$ distinct parties.

No honest party sends $(\mathsf{ready}, \hat{m})$ for $\hat{m} \neq m$. Indeed, by Claim 7, if that was not the case, there would exist an honest party that did this because they received $(\mathsf{echo}, \hat{m})$ from strictly more than $2n/3$ distinct parties. However, in that case, Claim 5 would require that $\hat{m} = m$.

We conclude that all honest parties send $(\mathsf{ready}, m)$ to each other (otherwise, there was the possibility that some of the parties sent $(\mathsf{ready}, \hat{m})$, instead of $(\mathsf{ready}, m)$).

Once again every honest party will receive $\mathsf{ready}$ messages of this kind from strictly more than $2n/3$ distinct parties. Given that, by Claim 6, they cannot receive $\mathsf{ready}$ messages for another message $\hat{m} \neq m$ from more than $2n/3$ distinct parties, the honest parties will all output $m$. □

**Claim 9.** *If an honest party outputs a message $m$, then all other honest parties output $m$ too.*

*Proof.* If an honest party outputs $m$, it means that they have received the message $(\mathsf{ready}, m)$ from strictly more than $2n/3$ distinct parties. Strictly more than $n/3$ of these must be honest. However, when honest parties send a message $(\mathsf{ready}, m)$, they deliver it to everybody. Therefore, all other honest parties must have also received $(\mathsf{ready}, m)$ from strictly more than $n/3$ distinct parties. Unless any honest party has already sent their $\mathsf{ready}$ message, every honest party will send $(\mathsf{ready}, m)$ to everybody. This means that every honest party will receive $(\mathsf{ready}, m)$ from strictly more than $2n/3$ distinct parties. By Claim 6, that ensures that every honest party outputs $m$.

What's left to show is that, indeed, no honest party has already sent $(\mathsf{ready}, \hat{m})$ for $\hat{m} \neq m$ before they have received $(\mathsf{ready}, m)$ from strictly more than $n/3$ distinct parties. Suppose this is false and there exists an honest party that has already sent $(\mathsf{ready}, \hat{m})$ for $\hat{m} \neq m$. By Claim 7, there exists an honest party that did this because they received $(\mathsf{echo}, \hat{m})$ from strictly more than $2n/3$ distinct parties. However, as we argued, there is also an honest party that sent $(\mathsf{ready}, m)$, and so, again by Claim 7, there exists another honest party that sent $(\mathsf{ready}, m)$ because they received $(\mathsf{echo}, m)$ from strictly more than $2n/3$ distinct parties. Claim 5 requires that $\hat{m} = m$. □

Notice that Claim 9 implies both perfect agreement and perfect asynchronous termination. This ends the proof of Theorem 4. □

**Advantages and disadvantages of Bracha's protocol.** Bracha's protocol has some very interesting quality. First of all it does not need a PKI and is secure even if the network over which the participants communicate is cursed with large, unpredictable delays. In other words, honest parties do not need to worry about attacks caused by network break-downs. The fact that we do not rely on clocks makes the protocol also optimally fast: in Dolev-Strong, the parties had to waste a lot of time waiting for timers to set off even if the network was extremely fast. In Bracha's protocol, there are no timers, and the protocol is as fast as the underlying network. Finally, unlike Dolev-Strong, the protocol does not require as many rounds as the number of corruptions we want to withstand. If the sender is honest, we only have 3 rounds: a round of send messages, followed by a round of echo messages and a final round of ready messages.

On the negative side, Bracha's protocol withstands a smaller number of corrupted parties (as we will see in the next section, this is unavoidable in the asynchronous setting). Moreover, it still requires authenticated channels. This means that the protocol assumes that corrupted parties cannot send messages on any channel that connects two honest parties. Or, rephrased in a different way, corrupted parties cannot impersonate honest parties in the eyes of other honest parties (without getting caught). If this premise does not hold, Bracha's protocol becomes insecure. Unfortunately, the Internet isn't likely to provide us with direct connections with all other participants, so how can we solve this problem? Well, one way is to ask every party to sign their communication using their secret key. By discarding any message where the signature does not verify, the recipient can be confident that the corrupted parties cannot impersonate other honest parties. However, to implement this solution we would need to rely again on a PKI that provides every participant with the public keys of everybody else. Notice also that Bracha's protocol fixes the set of participants and the parties need to know how many other parties are participating in the protocol.

### 2.1.5 Impossibility of Asynchronous Broadcast without Honest Supermajority

In this section, we show that Bracha's protocol is in some sense optimal, in the sense that no secure asynchronous broadcast can withstand more than $n/3$ parties. For security, at least $2/3$ of the parties must be honest, i.e., we want an *honest supermajority*.

**Theorem 5** (Impossibility of asynchronous broadcast without honest supermajority)**.** *Suppose that $\epsilon < 1/3$. Then, there exists no $\epsilon$-secure asynchronous broadcast protocol that withstands $t \geq n/3$ corrupted parties.*

*Proof.* We prove the theorem by contradiction. Suppose we are wrong and there exists indeed an asynchronous broadcast protocol that is secure even when $t \geq n/3$ parties are corrupted. We show that this leads to a logical contradiction.

As for the impossibility for Byzantine agreement (see Theorem 2), we partition the $n$ parties into three arbitrary subgroups $S_0$, $S_1$ and $S_2$ such that $|S_0|, |S_1|, |S_2| \leq t$ (this is possible because $t \geq n/3$) and $P_1$ is in $S_2$. We consider the following protocol executions:

**Execution 1:** In this execution, the sender is honest and sends the message $m = 0$. The parties in $S_0$ and $S_2$ are honest, while the parties in $S_1$ are corrupted and simply do not speak. By $\epsilon$-validity, we know that the parties in $S_0$ and $S_2$ eventually all output 0 (except with probability at most $\epsilon$).

**Execution 2:** In this execution, the sender is honest and sends the message $m = 1$. The parties in $S_1$ and $S_2$ are honest, while the parties in $S_0$ are corrupted and simply do not speak. By $\epsilon$-validity, we know that the parties in $S_1$ and $S_2$ eventually all output 1 (except with probability at most $\epsilon$).

**Execution 3:** In this execution, the sender is corrupted. The parties in $S_0$ and $S_1$ are honest, but the channels that connect them are down. The parties in $S_2$ are corrupted. Each corrupted parties runs two versions of the protocols. In particular, it is as if every corrupted party $P_j$ is made up of two virtual parties $P_j^{(0)}$ and $P_j^{(1)}$, both of which behave honestly. In the case of the corrupted sender $P_1$, we have that $P_1^{(0)}$ behaves as the honest sender that wants to broadcast $m = 0$, whereas $P_1^{(1)}$ behaves as the honest sender that wants to broadcast $m = 1$. Specifically, the corrupted party $P_j$ behaves as follows:

- Incoming communications from a party in $S_0$ are sent by $P_j^{(0)}$;
- Incoming communications from a party in $S_1$ are sent by $P_j^{(1)}$;
- Outgoing communications from $P_j^{(0)}$ to a party in $S_1$ are dropped;
- Outgoing communications from $P_j^{(1)}$ to a party in $S_0$ are dropped;
- Outgoing communications from $P_j^{(0)}$ to a corrupted party $P_k$ are sent to $P_k^{(0)}$;
- Outgoing communications from $P_j^{(1)}$ to a corrupted party $P_k$ are sent to $P_k^{(1)}$;

By $\epsilon$-agreement and $\epsilon$-asynchronous termination, we know that, if at least one party in $S_0 \cup S_1$ terminates and outputs some value $v$, then all other parties in $S_0 \cup S_1$ eventually terminate and output $v$ (except with probability at most $2\epsilon$).

In Execution 3 all parties in $S_0$ behave as in Execution 1. From their perspective everything is as if the sender is honest and wants to broadcast 0, the parties in $S_1$ are totally inactive, and everybody else is honest (they only interact with the other parties in $S_0$ and virtual parties $P_j^{(0)}$, who, after all, behave honestly and interact only with other honest (perhaps virtual) parties, including the sender $P_1^{(0)}$ who broadcasts 0). As a consequence, all parties in $S_0$ output 0 (except with probability $\epsilon$).

On the other hand, all parties in $S_1$ behave as in Execution 2. From their perspective everything is as if the sender is honest and wants to broadcast 1, the parties in $S_0$ are totally inactive, and everybody else is honest (they only interact with the other parties in $S_1$ and virtual parties $P_j^{(1)}$, who, after all, behave honestly and interact only with other honest (perhaps virtual) parties, including the sender $P_1^{(1)}$ who broadcasts 1). As a consequence, all parties in $S_1$ output 1 (except with probability $\epsilon$).

The issue is that $\epsilon$-agreement requires that, except with probability $\epsilon$, the parties in $S_0 \cup S_1$ all output the same value. However, with probability $1-2\epsilon > \epsilon$, the parties in $S_0$ output 0 and the parties in $S_1$ output 1. We have reached a contradiction, so it must be that we started from a wrong assumption: Theorem 5 is indeed true. □

The above impossibility can be generalised to show that, in the asynchronous setting (i.e., when the parties have no clock and the network admits arbitrary delays), if $\epsilon < 1/3$, there exists no $\epsilon$-secure Byzantine agreement protocol that withstands $t \geq n/3$ corruptions.

**Exercise 11.** *Prove the above claim by modifying the proof of Theorem 5.*

**Exercise 12.** *Prove the same claim by showing that if there existed a $\epsilon$-secure Byzantine agreement protocol that withstands $t \geq n/3$ corruptions in the asynchronous setting, then we could construct an $\epsilon$-secure asynchronous broadcast protocol that withstands $t \geq n/3$ corruptions in the asynchronous setting. Easier case: prove the result for $\epsilon = 0$.*

# Chapter 3

# Chain-Based Consensus

The consensus problems we have analysed so far study atomic executions, where the participants need to agree on a single value (this being the value broadcast by the sender, or a common decision). This structure does not reflect the level of complexity required, for instance, for today's most common application of the theory of this course: running a decentralised economy. Unlike Byzantine agreement protocols, an economy requires constant decision making about how to spend resources, and agreement is required at all times. Chain-based consensus tries to address these problems.

**Definition 26** (Chain-based consensus)**.** *A chain-based consensus protocol involves $\ell$ users $U_1, \ldots, U_\ell$ and $n$ validators $P_1, \ldots, P_n$. Each validator $P_j$ holds an ever-growing chain of messages $\{m_i^{(j)}\}_{i \in [s_j]}$, which we call the* history*, the* chain *or the* ledger*. At each point in time, any user can broadcast a new message to be added to the history.*

As for Byzantine agreement and broadcast, we split the validators into two groups: the honest ones $H$ and the corrupted ones $C$. Next, we define the security properties of these protocols: *agreement* and *liveness*. The first property guarantees that all honest validators agree on the history of messages, the second property guarantees that when a user broadcasts a message, this will be eventually added to the chain of the honest validators. Essentially, liveness is the property that guarantees no censorship.

**Definition 27** (Agreement)**.** *A chain-based consensus protocol satisfies $\epsilon$-agreement against $t$-corruptions if, for every protocol execution where $|C| \leq t$ and every $i, j \in H$ and $k \in \mathbb{N}$, we have that $m_k^{(i)} = m_k^{(j)}$, except with probability at most $\epsilon$.*

**Definition 28** (Liveness)**.** *A chain-based consensus protocol satisfies $\epsilon$-liveness against $t$-corruptions if, for every protocol execution where $|C| \leq t$, if a user broadcasts a message $\tilde{m}$, there eventually exists an honest validator $P_i$ and an index $k \in [s_i]$ such that $m_k^{(i)} = \tilde{m}$, except with probability at most $\epsilon$.*

**Exercise 13.** *Build a simple chain-based consensus protocol that satisfies agreement but not liveness. Next, build a simple chain-based consensus protocol that satisfies liveness but not agreement.*

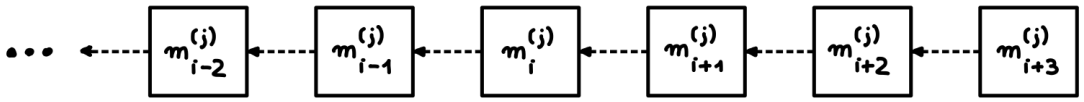


Figure 3.1: The chain of party $P_j$.

## 3.1 A Simple Construction in the Synchronous Setting

In the following protocol, we assume that all validators hold a synchronised clock. We split the execution into rounds of a given duration $T$. We say that a validator $P_i$ is the leader of the $j$-th round if $(j \mod n) = i - 1$. The protocol proceeds as follows:

1. Any user that wants to add a message $\tilde{m}$ to the chain, simply sends it to all validators.
2. At each round, the validators run a secure broadcast protocol where the sender is the leader of that round. The sender broadcasts a message consisting of all messages received from the users that have not yet been added to their chain. We call this a *block*. All validators append the output of the secure broadcast execution of to the history (removing any messages that are already on the chain).

**Theorem 6.** *If the above protocol uses an $\epsilon$-secure broadcast protocol against $t < n$ corruptions, then the construction is a chain-based consensus protocol that satisfies $\epsilon$-agreement and $\epsilon$-liveness against $t$-corruptions.*

*Proof.* It is easy to see why $\epsilon$-agreement is satisfied. This follows from the $\epsilon$-agreement of the secure broadcast protocol. Liveness is also easy to prove. Suppose that a user broadcasts a message $\tilde{m}$ to all validators. Since at least one of them is honest, there exists a future round where that particular honest validator $P_i$ is a leader. If $\tilde{m}$ is not yet on $P_i$'s chain, $P_i$ will broadcast $\tilde{m}$ as part of its message. Then, by $\epsilon$-validity of the secure broadcast, all honest validators will add $\tilde{m}$ to their chain (except with probability at most $\epsilon$). □

**What can be added to the chain?** Liveness guarantees that every user that wants to add a message to the chain will eventually succeed in doing so. This does not mean that these messages are necessarily to be trusted. Some of them may even be generated by malicious users! To counteract this issue, real world blockchains often instruct the validators to ignore any messages that do no satisfy particular properties. Honest validators will not include them as part of the block they propose when they are leaders, and will not add them to their history when they are not.

Nevertheless, some issues remain. For instance, it is not guaranteed that the messages are added to the history of the honest validators in the same order in which the users requested. For instance, if a user $U_i$ broadcasts a message $\tilde{m}$ and the next leader $P_j$ is corrupted, they may decide to temporarily censor $\tilde{m}$ and instead validate the messages added by the competitors of $U_i$. Some of these messages may have been generated after $U_i$ declared $\tilde{m}$ (potentially because $P_j$ leaked it). This is called a *front running attack*. Notice also that we cannot reject the block proposed by $P_j$ just because they are corrupted. How can we tell whether a validator is honest or corrupted? Maybe $P_j$ simply never received $\tilde{m}$, or $U_i$ is claiming the false and never sent their message $\tilde{m}$ to $P_j$! *Blocks proposed by corrupted leaders are as valid as those proposed by honest leaders.* If we try to be smart and discard blocks proposed by suspicious leaders, we may only end up hurting agreement and liveness: how would the honest parties agree on which blocks to discard? How do we ensure that we do not end up wrongly censoring honest validators?

**How can the users learn the history?** Validators are the custodians of the history of the chain. How can users have access it? They can simply ask all the validators to send their copy of the chain (or the last few blocks, if they already know the rest) and they take the answer that is proposed by the majority of validators. It is nevertheless important that this communication occurs over an authenticated channel. Assuming that there are at most $t < n/2$ corrupted validators, the users can be confident that the most popular answer coincides with the true history. Notice that if user queried the history to a single validator, they would run the risk of having chosen a corrupted one, which may lie to them.

## 3.2 Proof of Work

The simple chain-based consensus protocol we described above suffers from some important drawbacks. First of all, it requires the validators to have access to a synchronised clock. Secondly, it restricts to a fixed set

of validators that needs to be active all the time. Finally, the secure broadcast protocol typically requires that all parties speak in each round. When $n$ is big (e.g. $\approx 1000000$ as in Ethereum), this is simply too expensive. Proof-of-work consensus solves all of these issues at once. These ideas were introduced for the first time in 2008 in the Bitcoin whitepaper, published under the pseudonym of Satoshi Nakamoto (maybe a single individual, maybe a team of computer scientists).

The basic idea is the same as before: the users broadcast their messages to be added to the ledger, and the selected new validator leader gathers them in a block and adds it to the chain. The main innovation is how the new leader is selected and how agreement is established.

### 3.2.1 Permissionless Consensus

Proof-of-work protocols (PoW) are typically *permissionless*. This means that the set of validators is not fixed in advanced: everybody can join and leave the protocol as they please. In particular, at any given point in time, the validators are not necessarily even aware of the full list of active validators in the protocol. This rises of course some challenges. First of all, how do validators communicate with each other if they don't even know who to speak to?

Permissionless protocol typically rely on a Peer-to-Peer (P2P) network or gossip network. Instead of assuming that all protocol participants are connected by point-to-point channels, we assume that each active validator is aware of a small number of other active validators. Whenever a validator receives a message that they haven't received before, they relay it to a few other validators they know (at most 8 in the case of Bitcoin). Typically, these networks ensure a certain degree of connectivity by allowing participants to share information about the other active participants (for instance, by forwarding their IP address). From now on, we will assume that this network behaves well enough: whenever an honest validator receives a message $m$, all other honest validators will receive $m$ within some time bound $T$. We will not delve in the details of how to construct this infrastructure (this topic is better suited to a network security course). However, it is important to remember that if this assumption were to break, the security guaranties of the permissionless consensus protocol would no longer hold.

### 3.2.2 Mining

Permissionless consensus protocols come with plenty of other challenges. How can the validators elect a leader if they are not even aware about who stands for election? Ideally, even if a party had just joint the protocol and is known by nobody, they should have the same opportunities of being elected as everybody else. How can we ensure this? How can we ensure that this does not stand in the way of security? In particular, if all validators had the same chance of being chosen and there is no restriction who could join, an attacker could introduce a set of virtual parties to increase the chance that any of them is selected as the next leader. In such situation, the attacker can impersonate the virtual party and propose a new block that favours them. This is called a *Sybil attack*.

Proof-of-work consensus protocols solve these issues by requiring the validators to run some expensive computation in order to stand for election, with the probability of being elected proportional to the amount of computation performed. An attacker can therefore still perform the Sybil attack, but the probability that any of their puppets is elected will only grow with their computational power and not with the number of the virtual parties they impersonate. As long as the majority of the computational power in the system is controlled by honest participants, there is a good chance that the next leader will be honest. If not, we will be luckier at the next election. This is enough to ensure liveness.

Proof-of-work consensus protocols rely on hash functions modeled as random oracles (see 1.3.1). We recall that such hash functions have outputs that behave somewhat random and unpredictably. Specifically, if we run any "meaningful" statistics on their output, the results will be essentially the same as if they were uniformly random strings. This means that typically, we do not know any smart way to compute the output of the hash function other than directly evaluating it. In particular, knowing a list of input-digest pairs does not help in evaluating the hash function on a new input. This inspired the following election mechanism.

Suppose that $\mathsf{Hash} : \{0,1\}^* \to [2^k]$ is a hash function modeled as a random oracle,[1] let $\ell$ be a constant and let $W \in [2^k]$ be a hardness parameter (typically, this is the same for every validator).

We structure each block $B$ in the chain as tuple $(c, z, x, y, h)$ where $c$ is a counter, $x$ is the content of the block, $y$ is some metadata, $h$ is a digest of the previous block and $z = \mathsf{Hash}(x, y, h)$. We call $h$ the *pointer* to the previous block and $z$ the *puzzle* (it will soon become clear why). To participate to an election, the validators perform the following operations:

1. Each of them gathers all messages received from the users that are not yet on the chain into a list $x$. They also compute metadata $y$, which typically includes information about the validator themselves and/or randomness.

2. Next, they pick the freshest block $B' = (c', z', x', y', h')$ on their chain and compute the pointer $h = \mathsf{Hash}(c', z')$ and the puzzle $z = \mathsf{Hash}(x, y, h)$.[2]

3. Finally, they search for a binary string $c \in \{0,1\}^\ell$ such that $\mathsf{Hash}(c, z) < W$. We call $c$ emphthe solution of the puzzle. If they succeed in doing so before everybody else, the validator broadcasts the new candidate block $B := (c, z, x, y, h)$ over the gossip network. If someone beats them on time (or there is no such $c$), they stop the search.

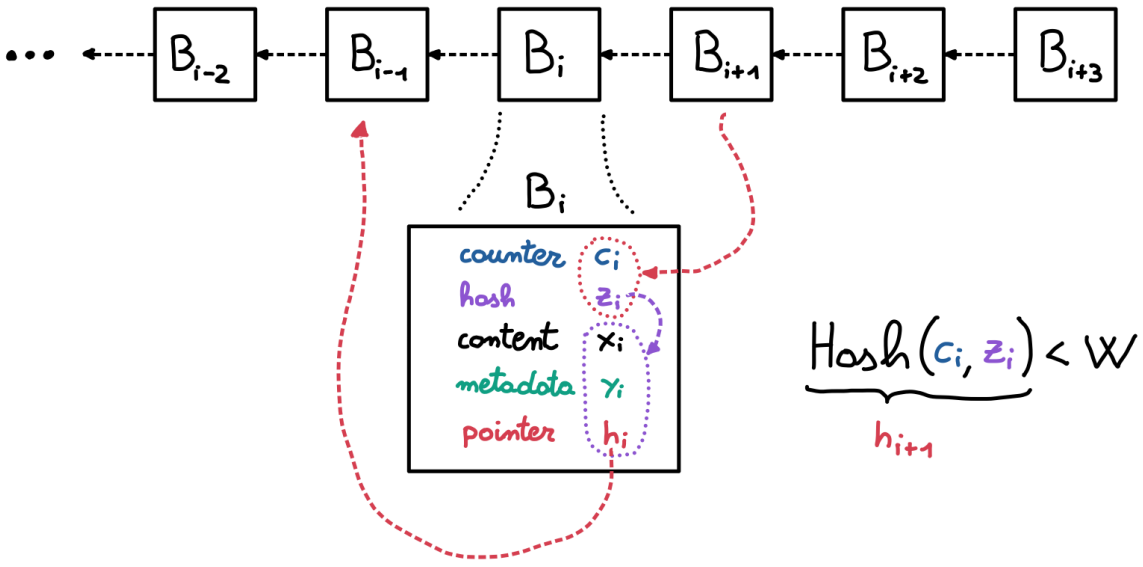


Figure 3.2: The structure of the blocks. Each colored dotted arrow indicates hashing, where the input of the hash is at the side of the head and the digest is at the tail. Notice that the chain of blocks is linked by the alternating sequence of red and purple arrows.

With this solution, any newcomer has the same opportunity to win the election as anybody else (proportionally to their computational power). Finding the counter $c$ such that $\mathsf{Hash}(c, z) < W$ requires a significant amount of computation, time and good luck depending on the value of $W$. This operation is called *mining*, and for this reason, in proof-of-work consensus, validators are often called *miners*. The lower the value of $W$, the harder winning the election becomes. In particular, when $W = 0$, the validator will never have a chance

[1] For instance, we could use SHA2 where we interpret the 256-bit output as an integer between 0 and $2^{256} - 1$.
[2] We use $x, y, h$ to denote the concatenation of $x, y$ and $h$.

to propose the next block (actually, nobody will). If instead $W = 2^k$, the validator will always manage to propose a new block (but everybody else will succeed too). In the first case, we seriously hurt liveness, in the second case, we seriously hurt agreement. This is why choosing the value of the hardness parameter $W$ requires a lot of care. We will discuss this at the end of this section.

The typical strategy of validators is to try all $c$ in lexicographical order starting from the string of zeros. Notice, however, that each validator will usually work with a different value of $z$, this is because $z$ is derived by hashing, among other things, the metadata $y$, which depends on the identity of the miner. Of course, not all $z$ are the same, some are luckier than others because they have a smaller value of $c$ such that $\mathsf{Hash}(c, z) < W$. The miners have however no way to check whether their $z$ is lucky or not other than trying many counters $c$ as fast as they can. It is worth noticing, anyway, that nothing prevents two or more miners to ally and pool their resources to mine with the same metadata $y$, if they want. In fact, this is a very common strategy. In these cases, we say that the parties formed a *mining pool*.

**What happens after the winner has been found?** Once the new leader has been established and a new block has been added to the chain, all miners move on to the next election. It is worth noticing that previous winning leader have only limited advantages in winning the next election: once their block $B = (c, z, x, y, h)$ was added to the chain, they need to derive the new content $\overline{x}$ and metadata $\overline{y}$ for their next block. Importantly, they also need to update the pointer to the previous block, which now becomes $\overline{h} = \mathsf{Hash}(c, z)$. This enforces a change in their value of $\overline{z} = \mathsf{Hash}(\overline{x}, \overline{y}, \overline{h})$ (if the new $\overline{z}$ coincides with some previous value, there musts have occurred a hash collision). In other words, once a new block is published, all miners start solving new puzzles they have never seen before, so the search for $\overline{c} \in \{0,1\}^\ell$ such that $\mathsf{Hash}(\overline{c}, \overline{z}) < W$ is reset for everyone. The advantage of the last winner reduces to being able to start working on their new puzzle $\overline{z}$ a little bit earlier than everybody else due to the time required by the network to deliver $B$ to the other participants. In principle, a corrupted miner could also decide to not reveal immediately the new block $B$ in order to gain a greater time advantage at solving $\overline{z}$, however, this is a risky move: while $B$ is kept secret, other miners may manage to solve their own puzzle and broadcast a different new block $\tilde{B}$. Even if the corrupted miner solved the puzzle first, by not immediately declaring the win, they lost the election to somebody else. On top of that, the new puzzle will become some value $\tilde{z}$ that depends on $\tilde{B}$, so for all this time, the corrupted miner wasted energy trying to solve the wrong $\overline{z}$.

**Exercise 14** (Easy)**.** *In order to optimise the communication and memory complexity of our protocol, it would be desirable to have blocks with minimal size. In this exercise, you are asked to think about some minor optimisations.*

- *We explained that each block consists of 5 terms: the counter $c$, the digest $z$, the content $x$, the metadata $y$ and the pointer $h$. It turns out that one of these is redundant and can be removed from the block. Which one?*
- *Suppose that the hash function outputs $k$-bit digests. One would expect that $h$ is represented by a string of length $k$, however, it can be shorter. Why?*

**Exercise 15.** *Consider the following election mechanism in which instead of looking for a $c \in \{0,1\}^\ell$ such that $\mathsf{Hash}(c, z) < W$, we look for one where $\mathsf{Hash}(c, x, y, h) < W$. This design has an important drawback. Can you spot it?*

### 3.2.3 The Longest Chain Principle

Suppose that we are a validator that, unfortunately, could not win the election: some other party broadcast a new candidate block $B = (c, z, x, y, h)$ before we managed to solve the puzzle. What do we do with $B$? First of all, we should check whether it satisfy some basic consistency checks:

$$\mathsf{Hash}(c, z) \stackrel{?}{<} W, \qquad\qquad z \stackrel{?}{=} \mathsf{Hash}(x, y, h).$$

Depending on the application, we could also perform some additional checks on $x$ and $y$. Anyway, if any of these conditions are not satisfied, we can discard $B$ and resume solving the puzzle.

When $h = \mathsf{Hash}(c', z')$ where $B' = (c', z', x', y', h')$ is the most recent block, $B$ is added to the chain after $B'$ and we move on to the next election:

- we update the list $\overline{x}$ of information that has not been published on the chain, the metadata $\overline{y}$ and the hash $\overline{h} = \mathsf{Hash}(c, z)$.
- We compute $\overline{z} = \mathsf{Hash}(\overline{x}, \overline{y}, \overline{h})$ and we mine for a $\overline{c} \in \{0,1\}^\ell$ such that $\mathsf{Hash}(\overline{c}, \overline{z}) < W$ (or at least, we try to).

What happens however if $B$ points to a block $B'$ that is not the most recent, but it is actually older? This may happen for multiple reasons. One, of course, is that who generated $B$ is corrupted. However, it can also be that they are an honest party that is working with a non-updated version of the chain. This kind of situation typically arises when there are multiple miners that solve the puzzle almost at the same time. Given that network communications are not instantaneous, it may be that the second winner broadcasts their new block before they receive the new block from the first winner. In these situations, all validators receive two different new blocks, which may include contradictory information: for instance, the first one says that Alice spent all her money on a jacket, but the second one says that she spent it all on a pair of shoes.

This requires careful dealing. *The validators cannot simply discard the second block they receive!* Indeed, it may be that everybody receives the blocks in a different order due to delays in the network. Nodes that sit closer to the first winner are more likely to receive their block first, whereas nodes that are closer to the second winner are more likely to receive the block of the second winner first. If the validators adopt the first block they receive, the system may split: half of it will live in a reality where Alice bought a jacket, the other half will live in a reality where Alice got new shoes.

To solve these issues, when $B$ points to older blocks on the chain (this is the case for instance when $B$ is the second new candidate block we receive), we don't discard it, but instead we add it to a new branch departing from $B'$. Even when $B$ points to a block that is not in the chain, we do not throw it away: it may that $B$ links to our chain through a block that is currently traveling through the network. In other words, the blocks will be arranged on a data structure that resembles a forest more than an actual chain (despite that, we will often keep calling it chain). One block and one branch of the forest will however be more important than the others: we call them *the genesis block* and *the main chain*.

- The genesis block is the first block generated by the system. In order for the protocol to be secure, it will be fundamental that the genesis block is generated by an honest miner, which broadcast it to everybody else immediately[3].
- The main chain is the branch that we will try to extend by mining new blocks and the one that determines the true course of history (e.g. it will tell us whether Alice bought the jacket or the shoes). We will define it as the branch that required the most computational work to be generated among those that departed from the genesis block. Typically (but not always), this is the longest branch in the forest. When there are multiple branches departing from the genesis block that required the highest amount of work, the main chain can be chosen among them arbitrarily.

In the next section, we will explain how to measure the amount of work invested on a branch.

Before going ahead, however, we stress an important aspect: it is possible for side branches to overtake the main chain as those that required the most work. *In these situations, the side branch becomes the new main chain.* This is called *the longest chain principle* (even if the main chain is not necessarily the longest). This means also that history can be (partially) rewritten: it could be that the old main chain stated that Alice bought a jacket, but after the switch, the new main chain states that she actually bought shoes.

[3]For instance, as a proof that the genesis block of Bitcoin was fresh, Satoshi Nakamoto set the content to that day's Times headline.

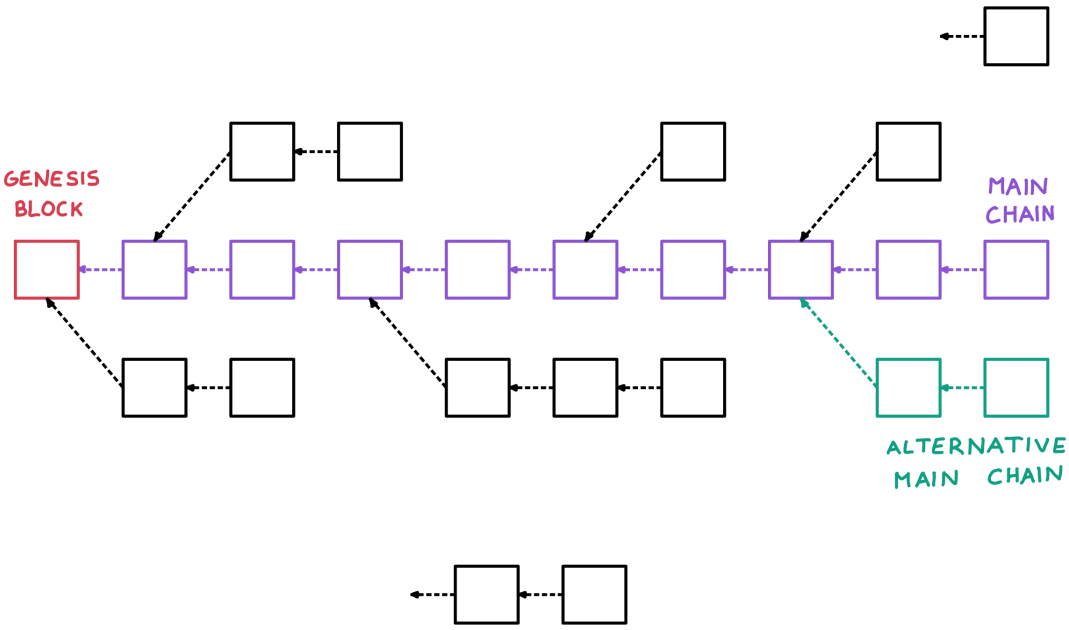


Figure 3.3: The forest of blocks of a party (notice the few disconnected components). The genesis block is indicated in red, while the main chain is purple. Observe that the main chain originates from the genesis block. The green branch (including also the purple section that connects it to the genesis block) is an alternative longest chain. In this example, the party selected the purple one as their main chain, however, the green chain was an equally valid choice, which other parties may have preferred. Either case, the two options differ only in the few most recent blocks. We highlight that all side branches are typically short: initially, these may have been part of the main chain of some party, however, with time, they were dropped in favour of the current main chain. They are what remains of a dead course of history. Eventually, the last two blocks of the green branch, or the last two blocks of the purple branch will end up in the same way.

Luckily, this kind of jumps usually affect only the most recent history. As we will point out in the next section, if a block has been part of the main chain for sufficiently many election cycles (let's say there are a dozen of fresher blocks that were added on the main chain since its generation), it is unlikely to be ever removed from the main chain and from the main course of history. If instead a block is recent, (let's say it is only two or three blocks away from the head of the main chain), there is a good chance that it will drop out of the main chain onto a forgotten course of history. The intuition behind this is that, assuming that the majority of computation power in the system is controlled by the honest miners, the majority of blocks will be produced by honest miners, which all contribute to extending the longest (or hardest) chain. In order for an adversary to extend a side branch enough to overtake the main chain, for a given interval of time, they would need to outperform the honest parties. This is feasible for short amount of time, but becomes very unlikely for large time intervals. To conclude, if you ever sell a fancy car using Bitcoin, do not give the keys as soon as a block with the transaction appears on the main chain. If that blocks drops off the main chain, you will have lost the car and the money. The chain switched to a course of history where you were rich to one where you are a fool.

**Exercise 16** (Important)**.** *It is extremely unlikely that the blocks form a graph like the one described in*

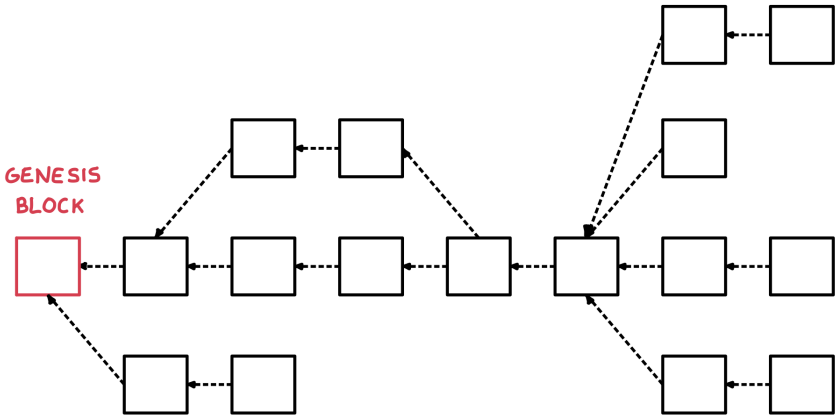


Figure 3.4: This graph structure is unlikely to ever occur. Why?

*Fig. 3.4. Why? (Remember that the construction is based on a cryptographic hash function)*

### 3.2.4 Quantifying the Computational Cost

Observe that if we model the outputs of the hash function as uniformly random elements in $[2^k]$ the probability that $\mathsf{Hash}(c, z) < W$ for a fixed $c, x, y$ is

$$\Pr[\mathsf{Hash}(c, z) < W] = \frac{W}{2^k}$$

We can expect the miners to compute $\approx 2^k/W$ different hashes before somebody manages to find one that satisfies $\mathsf{Hash}(c, z) < W$. More formally, the total number of attempts required for the next block is described by a geometric distribution of parameter $W/2^k$ (see Definition 7).

Given an estimate of the amount of computational power in the system $p$ (measured in number of hashes per second), we can also estimate the duration of an election in seconds. This is described by an exponential distribution (see Definition 9) of mean

$$L = \frac{2^k}{p \cdot W}$$

Through this analysis, we can estimate the computational cost required to generate a branch of the tree: if it includes $n$ blocks all generated with the same hardness parameter $W$, their generation required computations proportional to $\mathsf{cost} = n2^k/W$ and a time proportional to $\mathsf{cost}/p$. In other words, the longer the branch is, the more work and time was required for its generation.

So far, we have assumed that the hardness parameter $W$ remains the same for all blocks. Proof-of-work blockchains such as Bitcoin, however, continuously adapt the value of $W$ to reflect the total amount of computational power in the system and maintain the mining rate constant (we will discuss this more in detail in the following sections). In this case, the computational cost for the generation of $n$ blocks with different hardness parameters $W_1, \ldots, W_n$ becomes

$$\mathsf{cost} = \sum_{i=1}^{n} \frac{2^k}{W_i}.$$

If we denote the total amount of power in the system at the time the $i$-th block was generated by $p_i$, we can

expect the generation of the $n$ block to take time

$$L = \sum_{i=1}^{n} \frac{2^k}{p_i \cdot W_i}$$

There are multiple reasons why we can expect the computational power in the system to change. For instance, there may be new validators that join the system, or the existing miners might have bought more fancy hardware to increase their chance of winning the election. If the hardness parameter $W$ is kept unvaried, the increased computational power increments the speed at which new blocks are mined. This may be undesirable as it increases the likelihood that multiple parties win the elections at the same time, creating new branches in the chain. Conversely, if the computational power in the system decreases (typically, because validators leave) and $W$ is not adjusted, the speed at which new blocks are generated decreases, hurting liveness. The solution to these problems is to keep the mining rate constant by adjusting $W$. We can estimate the total amount of power in the system in a given period of time $T$ by estimating the computation required to generate the blocks added to the chain during this period and dividing the result by $T$. Specifically, if $n$ blocks of hardness parameter $W_1, \ldots, W_n$ were added to the chain over the time $T$, we can estimate the total power in the system by

$$p = \sum_{i=1}^{n} \frac{2^k}{T \cdot W_i}$$

The mining speed grows proportionally to $p \cdot W/2^k$, so to keep this quantity equal to some desired constant $\gamma$, we need to set the hardness parameter to

$$W = \frac{2^k \cdot \gamma}{p}$$

**Exercise 17** (Important)**.** *Consider the graph in Fig. 3.5. In each block, we have indicated the value of the hardness parameter $W$ used for its generation. What is the block we can expect to have required the least amount of computation to be generated? Which branch is the longest? Which branch was the one that required the most work? Which branch is the main chain?*

**Exercise 18.** *The structure of the blocks in Fig. 3.5 is unlikely to occur in our proof of work consensus protocol. Why? Compare this to the graph in Fig. 3.3.*

**From theory to practice.** In 2026, values of $p$ go from $10^6$ Hash/s on a normal laptop to $10^{15}$ Hash/s for top-tier specialised hardware units called ASICs. To determine the total computational power in the system, we need to add up the power of all units that contribute to mining new blocks. For big blockchains (such as Bitcoin), these number in the millions. The values of $k$ and of the hardness parameter $W$ are set so that we can expect on average a new block every few minutes (e.g. Bitcoin generates a new block every 10 minutes or so).

De facto, every proof-of-work blockchain adjusts the hardness parameter in slightly different ways. For instance, Bitcoin tries to avoid big variations in the magnitude of $W$ in short intervals of time. It splits time into slots called *epochs*. The hardness parameter is kept constant throughout every epoch and recalculated as discussed above when we enter a new one. However, if the new hardness parameter $W_{\mathsf{new}}$ is more than 4 times the previous $W_{\mathsf{old}}$, the new value for $W$ will be $4W_{\mathsf{old}}$. Conversely, if the new hardness parameter $W_{\mathsf{new}}$ is less than $1/4$ of the previous $W_{\mathsf{old}}$, the new value for $W$ will be $W_{\mathsf{old}}/4$.

**Exercise 19** (Important)**.** *Suppose that a blockchain is populated by $10^6$ validators, each equipped with one $10^{15}$ Hash/s ASIC unit. Suppose that $k = 256$ and $W = 9$ as for the easiest block in Fig. 3.5. How long can we expect the system to take to generate the next block? What value of $W$ should we choose to mine a new block every 10 minutes in expectation? Suppose that your computer performs $10^6$ Hash/s. How long would you need to wait on average until you manage to mine a block with the value of $W$ you have just found.*

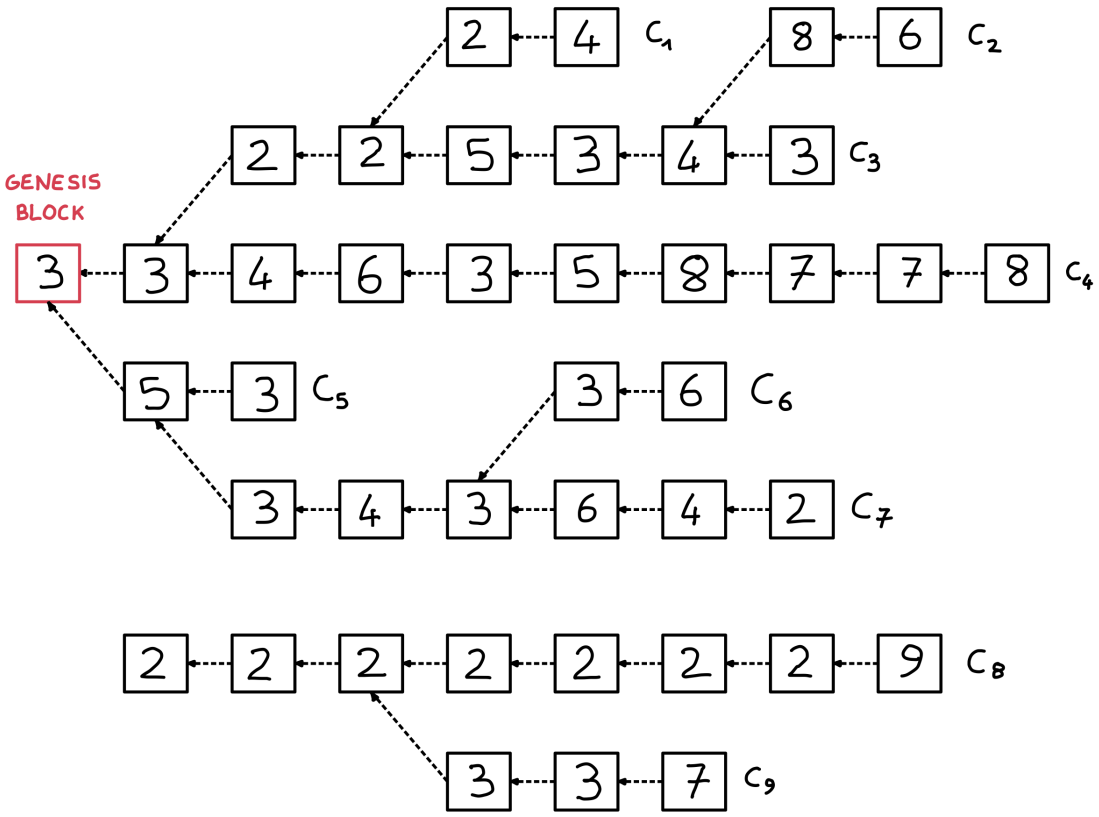


Figure 3.5: Suppose that the blocks are arranged as the above graph, where the genesis block is indicated in red. Each block contains the value of the hardness parameter $W$ that was used for its generation (these are much smaller values compared to those used in the real world). What is the longest branch? What is the hardest one to generate? What is the main chain?

**Exercise 20.** *Suppose that $k = 256$ and $W = 2^{195}$. Suppose that your friend has a new ASIC unit with hash power $10^{15}$ Hash/s, while you have a slightly older ASIC unit that is 5% less powerful. What is the probability that you manage to mine the next block before your friend?*

### 3.2.5 The Security Analysis

Next, we will formally analyse the security of our proof-of-work consensus protocol by restricting to the simplified case in which the computational power in the system remains constant and the hardness parameter $W$ is kept unvaried. This is a reasonable assumption as changes of the value of $W$ tend to occur rarely. Notice that in this case, the hardness chain is also always the longest.

**Safe blocks.** Let us introduce some notation: consider the block tree of a party, we say that a block $B$ is an ancestor of another block $B'$ if $B'$ points to $B$ or to another block that has $B$ as an ancestor. In these cases, we say that $B'$ is a descendant of $B$. Next, we introduce the concept of safe block.

**Definition 29** (Safe block)**.** *Suppose that the gossip network delivers each message with delay at most $\Delta$.*

*We say that a block is safe if it is mined by an honest party and no honest party has generated any new block in the previous time interval of length $\Delta$, neither on the main chain or on any side branch.*

In the next theorem, we show that if the adversary controls only a constant fraction $\alpha$ of all computational power and the delay $\Delta$ in the network is only a constant fraction of the expected mining time $L$, in every sequence of $\ell$ consecutive blocks, with high probability, at least a fraction $(1-\Delta/L)(1-\alpha)$ of them are safe. Moreover, the probability that thios does not hold, degrades exponentially in $\ell$.

**Theorem 7.** *Suppose that the corrupted parties control a fraction $\alpha$ of all computational power and that the gossip network has at most delay $\Delta$. Consider the sequence $S$ of the first $\ell$ blocks that are mined by the system starting from a certain time $t$ (these do not need to be all on the same branch of the tree). Then, it holds that, for every $s \geq 0$,*

$$\Pr\left[\#\ \textit{safe blocks} \leq \left(1-\frac{\Delta}{L}\right)(1-\alpha)\ell - s\right] \leq \exp\left(-\frac{s^2}{2(1-\alpha)\ell}\right).$$

*In particular, if $s = \sqrt{2\kappa \cdot \ell(1-\alpha)}$, we obtain that*

$$\Pr\left[\#\ \textit{safe blocks} \leq \left(1-\frac{\Delta}{L}\right)(1-\alpha)\ell - s\right] \leq \exp\left(-\kappa\right).$$

*Proof.* For any block, there is a probability $\alpha$ that who mined it was a corrupted party. This probability is independently of everything else. The number of blocks generated by honest parties in a given interval of time of length $\Delta$ is described by a Poisson distribution (see Definition 8) of parameter

$$(1-\alpha)\cdot p \cdot \frac{W}{2^k}\cdot\Delta = (1-\alpha)\frac{\Delta}{L}$$

Therefore, the probability that no honest party produces a new block in a time interval of $\Delta$ is

$$\exp\left(-(1-\alpha)\cdot\frac{\Delta}{L}\right)$$

independently of everything else. We conclude that the probability that a given block is safe is

$$(1-\alpha)\cdot\exp\left(-(1-\alpha)\cdot\frac{\Delta}{L}\right)$$

independently of its position in the sequence. In conclusion, the number of safe blocks is described by a binomial distribution with $\ell$ trials and success probability

$$(1-\alpha)\cdot\exp\left(-(1-\alpha)\cdot\frac{\Delta}{L}\right) \geq (1-\alpha)\cdot\left(1-(1-\alpha)\cdot\frac{\Delta}{L}\right).$$

Now, let $\beta := 1-\alpha$ and $\gamma = \Delta/L$ By the Chernoff bound (see Theorem 1) with $\mu = \beta\exp(-\beta\gamma)\,\ell$ and $\delta = 1 - \frac{(1-\gamma)\beta\ell - s}{\mu}$, we obtain that

$$\Pr\left[\#\text{ safe blocks} \leq \left(1-\frac{\Delta}{L}\right)(1-\alpha)\ell - s\right] \leq \exp\left(-\frac{(\delta\mu)^2}{2\mu}\right) = \exp\left(-\frac{((\exp(-\beta\gamma)-1+\gamma)\beta\ell+s)^2}{2\exp(-\beta\gamma)\,\beta\ell}\right)$$

Notice that $\beta \leq 1$ and $\gamma \geq 0$, so $\exp(-\beta\gamma) \geq \exp(-\gamma) \geq 1-\gamma$, while $\exp(-\beta\gamma) \leq 1$ (see Lemma 2). We derive that

$$\Pr\left[\#\text{ safe blocks} \leq \left(1-\frac{\Delta}{L}\right)(1-\alpha)\ell - s\right] \leq \exp\left(-\frac{s^2}{2(1-\alpha)\ell}\right)$$

In particular, if $s = \sqrt{2\kappa\cdot\ell(1-\alpha)}$, we obtain that

$$\Pr\left[\#\text{ safe blocks} \leq \left(1-\frac{\Delta}{L}\right)(1-\alpha)\ell - s\right] \leq \exp\left(-\kappa\right).$$

□

Let us define the history of a block in the tree of an honest party as the sequence of blocks from the root of the tree to the block itself. Let us define the height of the block as the length of its history. Safe blocks will be fundamental in our security analysis for two reasons, we are going to summarise below.

**Lemma 5.** *Consider a protocol execution. In the block chain of any honest party*

- No two safe blocks have at the same height.
- The length of the main chain is at least as big as the number of safe blocks generated until that point in time.

*Proof.* Suppose that there are two safe blocks $B$ and $B'$ that lie on different branches of the tree but at the same height. Suppose that $B'$ was generated after $B$. Since both blocks are safe, they are both generated by honest parties. Moreover, $B'$ was generated at an interval of time that is at least $\Delta$ away from when $B$ was generated. Since the network has delay at most $\Delta$, the party that generated $B'$ had already received $B$ as well as its history before it produced $B'$. This means that the length of that party's main chain was at least equal to the height of $B$. Since the party appended $B'$ to their main chain, it must be that the height of $B'$ is strictly greater than that of $B$. More in detail, we have that

$$\mathsf{height}(B') \geq \mathsf{height}(B) + 1.$$

Now, let $B_0, \ldots, B_n$ be the safe blocks order in chronological order of generation. By recursively applying the above formula, we obtain that

$$\mathsf{height}(B_n) \geq n.$$

Since the length of the main chain is always at least as large as the height of any safe block, we conclude that the length of the main chain is at least $n$. □

**Common prefix.** The following theorem proves that, if the adversary only controls a minority of all computational power in the system and the delay in the network is sufficiently smaller than the expected mining time, the longest chains of different parties can only differ in the most recent blocks.

**Theorem 8.** *Consider a protocol execution where the adversary controls a constant fraction $\alpha < 1$ of all computational power in the system, the network has delay at most $\Delta$ and the expected mining time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Suppose that*

$$(1 - \alpha)(1 - \gamma) > \frac{1}{2}$$

*and pick two honest parties and a time $t$. Suppose that their genesis blocks coincide and are safe, and let $C_1$ and $C_2$ be their main chain at time $t$. Let $\ell \geq 1$ be a positive integer and let $\lfloor C_1 \rfloor_\ell$ and $\lfloor C_2 \rfloor_\ell$ denote the chains obtained by stripping the $\ell$ most recent blocks in $C_1$ and $C_2$, respectively. Then,*

$$\Pr\left[\lfloor C_1 \rfloor_\ell \not\subseteq \lfloor C_2 \rfloor_\ell \text{ and } \lfloor C_1 \rfloor_\ell \not\supseteq \lfloor C_2 \rfloor_\ell\right] \leq 2^{-\Omega(\ell)}.$$

*Proof.* Suppose that $\lfloor C_1 \rfloor_\ell \not\subseteq \lfloor C_2 \rfloor_\ell$ and $\lfloor C_1 \rfloor_\ell \not\supseteq \lfloor C_2 \rfloor_\ell$. Consider the union of the trees of the two parties, let us call this the *joint tree*. Let $B^*$ be the most recent block that belongs to both $C_1$ and $C_2$ ($C_1$ and $C_2$ have at least the genesis block in common). Notice that this may have been generated by the adversary. Therefore, let $B_0$ be the most recent safe block of height smaller or equal to that of $B^*$. This block exists because the genesis block is safe. Let $h_0$ be the height of $B_0$, let $h^*$ be the height of $B^*$ and let $h_1$ be the height of the most recent block on $C_1$ and $h_2$ the height of the most recent block on $C_2$. Without loss of generality, we assume that $h_2 \geq h_1$. Since $B_0$ was generated, the tree was extended by $n \geq 2(\ell + 1)$ new blocks, where $h_1 - h^* \geq \ell + 1$ went on $C_1$ but not on $C_2$, and $h_2 - h^* \geq \ell + 1$ went on $C_2$ but not on $C_1$.

Notice that, at time $t$, there can exist at most one safe block $\hat{B}$ of height strictly greater than $h_1$. This is because safe blocks are generated at times that are at least $\Delta$ apart, so by the time the second safe block of height strictly greater than $h_1$ was added to the tree of one party, the previous one (and all its ancestors) were added to the tree of the other party. At that point, both parties would have had a chain strictly longer

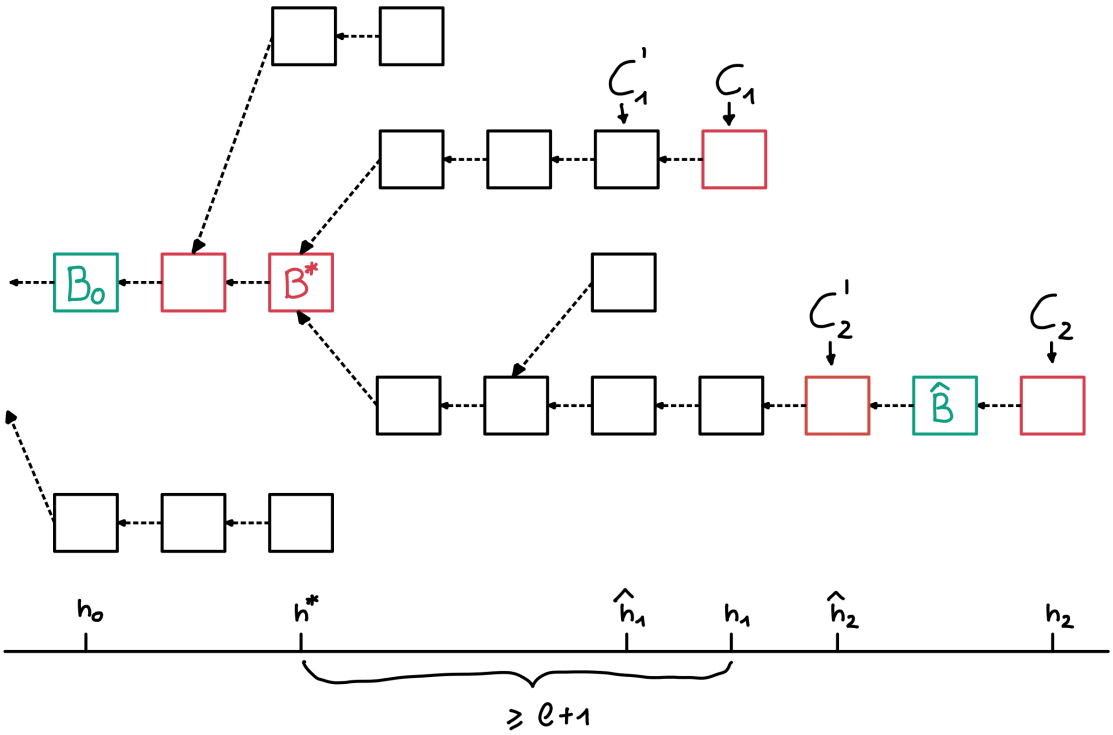


Figure 3.6: Conformation of the joint tree in the proof of Theorem 8 (in this example $\ell \leq 3$). Heights are indicated on the line at the bottom. Green blocks are blocks that are certainly safe (these are the blocks between $B_0$ and $B^*$, and the blocks of $C_1$ and $C_2$ that are generated after $\hat{B}$, i.e. those of height strictly greater than $\hat{h}_1$ on $C_1$ and those with height strictly greater than $h_1$ in $C_2$, except $\hat{B}$). Red blocks are blocks that are certainly not safe. Black blocks can be either safe or not safe, however, by Lemma 5, only one block per height can be safe. Notice that there exist blocks that were generated after $B_0$ but that do not belong to either $C_1$ or $C_2$. Some of them are not even descendants of $B_0$. In this example, we have $h^* = h_0 + 2$, $\hat{h}_1 = h_1 - 1$, $\hat{h}_2 = h_1 + 1$, $h_2 = h_1 + 3$ and $k = 2$. In general, any combination of heights that satisfies $h_0 \leq h^*, h^* + \ell < h_1, \hat{h}_1 \leq h_1 \leq \hat{h}_2 \leq h_2, k = h_1 + h_2 - \hat{h}_1 - \hat{h}_2 - 1$ is possible.

than $h_1$. Given that all of this happens before time $t$, we contradict the fact that $C_1$ is the longest chain at time $t$ for one of the parties.

Let $\hat{t}$ be the time at which $\hat{B}$ was generated (if $\hat{B}$ does not exist, we set $\hat{t} = t$). Let $k$ be the number of blocks in the joint tree that were generated after $\hat{B}$ and before $t$. Let $C_1'$ and $C_2'$ be the subchain of $C_1$ and $C_2$ with all the blocks generated strictly before time $\hat{t}$. Notice that at time $\hat{t}$, the parties had received all safe blocks generated before $\hat{B}$. Let $\hat{h}_1$ and $\hat{h}_2$ be the length of $C_1'$ and $C_2'$. Notice that $\hat{h}_2 \geq h_1 \geq \hat{h}_1$. All $h^* - h_0$ blocks between $B_0$ (excluded) and $B^*$ (included) cannot be safe. Of the remaining $\hat{h}_1 + \hat{h}_2 - 2h^*$ blocks on $(C_1' \cup C_2') \setminus (C_1' \cap C_2')$ at most $\hat{h}_1 - h^*$ can be safe, because, by Lemma 5, all safe blocks have different heights and no safe block generated before time $\hat{t}$ has height strictly larger than $\hat{h}_1$. This proves that, since the generation of $B_0$ till $\hat{t}$, there have been at least $\hat{h}_2 - h_0$ new blocks that are not safe. By Lemma 5, the number of safe blocks is upper bounded by $\hat{h}_1 - h_0$. We conclude that since the generation of $B_0$ till time $\hat{t}$, the number of non-safe blocks has been at least as big as the number of safe blocks. If $k \geq 1$, this holds also at time $t$. If instead $k = 0$, then the number of blocks since the generation of $B_0$ till time $\hat{t}$ is $n - 1 \geq 2\ell + 1$. Either way, between the generation of $B_0$ and some time $t' \in \{\hat{t}, t\}$, the system has generated $n' \geq 2\ell + 1$

new blocks and at least half of them are not safe. Let $\epsilon := (1-\alpha)(1-\gamma) - 1/2 > 0$ and notice that this is a constant. By Theorem 7, the probability of this event is

$$\Pr\left[\#\text{ safe blocks} \leq \ell\right] \leq \exp\left(-\frac{((1-\alpha)(1-\gamma)n' - n'/2)^2}{2(1-\alpha)\cdot n'}\right) = \exp\left(-\frac{\epsilon^2}{2(1-\alpha)}\cdot n'\right) \leq 2^{-\Omega(\ell)}.$$

□

We are ready to state and prove the second security property of the proof-of-work consensus protocol. This states that once a block has been on the chain for long enough, it will be never removed from it.

**Theorem 9.** *Consider a protocol execution where the adversary controls a constant fraction $\alpha < 1$ of all computational power in the system, the network has delay at most $\Delta$ and the expected mining time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Suppose that*

$$(1-\alpha)\,(1-\gamma) > \frac{1}{2}$$

*and pick any honest party and any two time instants $t_1 \leq t_2$. Let the main chain of the party at time $t_1$ be $C_1$ and let that at time $t_2$ be $C_2$ and suppose that their genesis block was a safe block. Let $\ell$ be a positive integer, then*

$$\Pr\left[\lfloor C_1\rfloor_\ell \not\subseteq \lfloor C_2\rfloor_\ell\right] \leq 2^{-\Omega(\ell)}.$$

*Proof.* Suppose that $\lfloor C_1\rfloor_\ell \not\subseteq \lfloor C_2\rfloor_\ell$ and consider the blocks that have been added to the block tree of the party between time $t_1$ and time $t_2$. Some of these blocks have the property that, before being added, the main chain contained $\lfloor C_1\rfloor_\ell$, while after they were added, this no longer held. We consider the first of such blocks, let it be $B$. Let $C_1'$ be the longest chain before adding $B$ and let $C_2'$ be the longest chain after adding $B$. Let $C_2''$ be $C_2'$ with $B$ removed. Notice that $\lfloor C_1\rfloor_\ell \subseteq \lfloor C_1'\rfloor_\ell$ as $C_1'$ is at least as long as $C_1$.

By the maximality of $C_2'$ after adding $B$, it must be that the length of $C_1'$ is smaller or equal to that of $C_2'$. Similarly, by the maximality of $C_1'$ before adding $B$, the length of $C_2''$ is smaller or equal to that of $C_1'$. Since the lengths of $C_2'$ and $C_2''$ differ at most by 1, it must be that either the length of $C_1'$ matches that of $C_2'$ or that of $C_2''$. By Theorem 8, we have that, except with probability $2^{-\Omega(\ell)}$, either $\lfloor C_1'\rfloor_\ell \subseteq \lfloor C_2'\rfloor_\ell$ or $\lfloor C_1'\rfloor_\ell \subseteq \lfloor C_2''\rfloor_\ell \subseteq \lfloor C_2'\rfloor_\ell$. Either way, we obtain that

$$\lfloor C_1\rfloor_\ell \subseteq \lfloor C_1'\rfloor_\ell \subseteq \lfloor C_2'\rfloor_\ell \subseteq C_2'.$$

This contradicts the fact that we chose $C_2'$ so that $\lfloor C_1\rfloor_\ell \not\subseteq C_2'$. □

**Chain growth.** We have seen that the expected time required to mine a new block is $L = \frac{2^k}{pW}$. This holds only when the consensus protocol is not under attack. The next theorem estimates the minimum speed at which the main chain grows when the adversary tries to slow down the system, for instance, by not contributing with their computational power to mining.

**Theorem 10.** *Consider a protocol execution where the adversary controls a constant fraction $\alpha < 1$ of all computational power in the system, the network has delay at most $\Delta$ and the expected mining time is $L$. Consider the main chain of an honest party at time $t$. Then, the honest party will extend their main chain by a new block by time*

$$t + 2\Delta + \kappa\cdot\frac{L}{1-\alpha}$$

*except with probability* $\exp(-\kappa)$.

*Proof.* Let $B$ be the last block added to the main chain of the the honest party, let $h$ be its height and let $t$ be the instant in which such block was received. Now, suppose that no honest party in the system extended any branch in their tree to a length greater or equal to $h+1$ by time $t+\Delta$. By that time, we can be sure that every honest party has a longest chain of length $h$ (which may or may not be the same as that of the party we selected). This means that at time $t+\Delta$, all honest parties are working to generate a block of

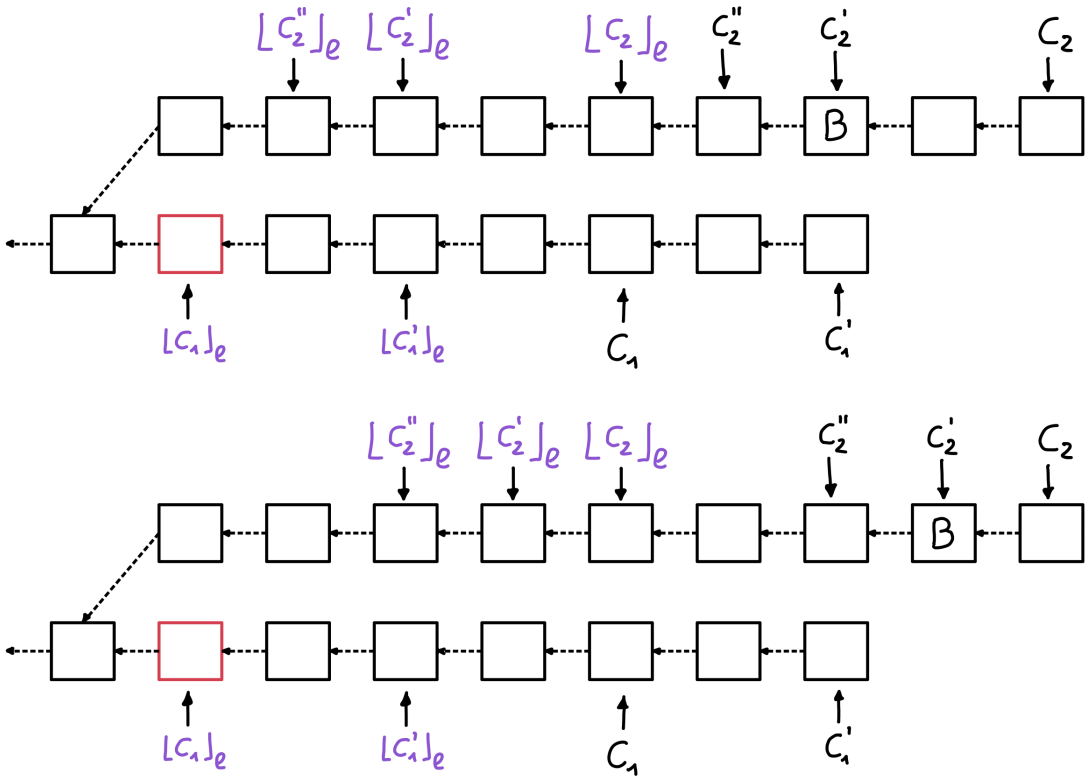


Figure 3.7: The two possible conformation of the tree in the proof of Theorem 9 (in this example $\ell = 4$). On top is the case in which $C_1'$ and $C_2'$ have the same length. At the bottom is the case in which $C_2'$ is one block longer than $C_1'$. We denote the chains with the last $\ell$ blocks removed in purple. The red block consists of a block that belongs to $\lfloor C_1 \rfloor_\ell$, but not to $\lfloor C_2 \rfloor_\ell$.

height $h + 1$. Now, the time required by the honest parties to generate any such block is described by an exponential distribution (see Definition 9) of parameter

$$(1 - \alpha)p \cdot \frac{W}{2^k} = \frac{1 - \alpha}{L}$$

This means, that the first honest party to succeed will take time at most $\frac{\kappa \cdot L}{(1-\alpha)}$, except with probability $\exp(-\kappa)$. Our honest party will receive the new block at most with delay $\Delta$. □

The above theorem establishes that the main chain keeps growing. The result however doesn't explain what is the quality of this growth. For instance, it would be quite bad if only the adversary were to add blocks. In the following, we provide a lower bound for the fraction of honest blocks that ends up on the main chain. We call this ratio the *chain quality*.

**Theorem 11.** *Consider a protocol execution where the adversary controls a constant fraction $\alpha < 1$ of all computational power in the system, the network has delay at most $\Delta$ and the expected mining time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Let $\epsilon > 0$ be a constant and suppose that*

$$(1 - \epsilon)(1 - \alpha)\,(1 - \gamma) > \frac{1}{2}$$

*Pick an honest party and suppose that their genesis block is safe. Consider any sequence $S$ of $\ell$ consecutive blocks on the main chain of the honest party. Then, we have*

$$\Pr\left[\#\textit{ honest blocks in } S \geq \frac{2(1-\epsilon)(1-\gamma)(1-\alpha)-1}{(1-\epsilon)(1-\gamma)(1-\alpha)} \cdot \ell\right] \geq 1 - \exp(-\Omega(\ell)).$$

*We call*

$$\frac{2(1-\gamma)(1-\alpha)-1}{(1-\gamma)(1-\alpha)}$$

*the* chain quality ratio.

*Proof.* Pick any safe block $B$ on the main chain of the party (at least one of these blocks exists because the genesis block is safe). Consider any sequence $S$ of $\ell$ consecutive blocks that are generated by the system after $B$ (not all of these are necessarily descendants of $B$). By Theorem 7, for every $s \geq 0$,

$$\Pr\left[\#\text{ safe blocks in } S \leq (1-\gamma)\,(1-\alpha)\ell - s\right] \leq \exp\left(-\frac{s^2}{2(1-\alpha)\ell}\right).$$

In particular, except with probability $\exp\left(-\frac{s^2}{2(1-\alpha)\ell}\right)$, at most $s+(\gamma+\alpha-\alpha\gamma)\ell$ blocks in $S$ could be non-safe descendants of $B$.

By Lemma 5, we know that, as the blocks in $S$ are added to the tree of the honest party, their main chain grows at least by the number of safe blocks in $S$. We conclude that, except with probability $\exp\left(-\frac{s^2}{2(1-\alpha)\ell}\right)$, as the blocks in $S$ are added to the tree of the honest party, their main chain grows at least by $(1-\gamma)\,(1-\alpha)\ell - s$ blocks, and at least

$$2(1-\gamma)(1-\alpha)\ell - \ell - 2s$$

of them is safe. Observe that, setting $s = \epsilon(1-\gamma)(1-\alpha)\ell$, we obtain

$$\begin{aligned}\frac{2(1-\gamma)(1-\alpha)\ell - \ell - 2s}{(1-\gamma)\,(1-\alpha)\ell - s} &= 2 - \frac{\ell}{(1-\gamma)\,(1-\alpha)\ell - s} \\ &= 2 - \frac{1}{(1-\epsilon)\,(1-\gamma)\,(1-\alpha)} \\ &= \frac{2(1-\epsilon)(1-\gamma)(1-\alpha)-1}{(1-\epsilon)(1-\gamma)(1-\alpha)}\end{aligned}$$

We conclude that

$$\Pr\left[\frac{\#\text{ new safe blocks in the main chain}}{\#\text{ new blocks in the main chain}} \geq \frac{2(1-\epsilon)(1-\gamma)(1-\alpha)-1}{(1-\epsilon)(1-\gamma)(1-\alpha)}\right] = 1 - \exp(-\Omega(\ell)).$$

Now, consider any sequence $\hat{S}$ of $\ell$ blocks on the main chain of the honest party. Let $\hat{B}$ be the earliest block in this sequence and let $B^*$ be the most recent safe ancestor of $\hat{B}$. We, know that, except with probability $\exp(-\Omega(\ell))$, at least a fraction

$$\frac{2(1-\epsilon)(1-\gamma)(1-\alpha)-1}{(1-\epsilon)(1-\gamma)(1-\alpha)}$$

of the $\ell$ immediate descendants of $B^*$ on the main chain must be safe. However, none of the blocks between $B^*$ and $\hat{B}$ is safe. This means that at least

$$\frac{2(1-\epsilon)(1-\gamma)(1-\alpha)-1}{(1-\epsilon)(1-\gamma)(1-\alpha)}\ell$$

safe blocks appear after $\hat{B}$. All of these blocks must belong to $\hat{S}$ too. □

Now, suppose for simplicity that $L \gg \Delta$ so that $\gamma \approx 0$. Imagine that $1 - \alpha = 2/3$. By Theorem 7, one would imagine that roughly $2/3$ of blocks on the main chain are generated by honest participants. Theorem 11, however, tell us that these may be as low as roughly $1/2$ of all blocks. As we are going to explain, there is a good reason for this.

Suppose that when the honest parties have multiple longest chain in their tree, they set their main chain to the first branch that reached that particular length (this is what typically happens in Bitcoin). Consider the adversary that, even if it controls only a constant fraction $\alpha < 1$ of all computational power in the system, occupies a favourable position in the gossip network. In particular, the adversary is connected to all parties with connections that are must faster than all connections between two honest parties. Let us say that their delay is at most $\delta \ll \Delta'/2$, where $\Delta'$ is the minimum delay of the fastest connection between two honest parties.

The adversary performs the following attack: at every election, it uses all its computational power to mine the next block, however, when it succeeds, they do not immediately reveal their block $B$ to the honest parties. Instead, they withhold it and they listen on the channels that connect them to the honest parties. As soon as the adversary receives a new block $B'$ from an honest party, they send $B$ to everybody else. Given that their channels are much faster than the honest parties' channels, $B$ will reach everybody else before $B'$ does. In other words, all honest parties (except the one that mined $B'$) will include $B$ in their main chain instead of $B'$. After that, all parties except one will start mining for a new block that extends $B$. Therefore, it becomes extremely unlikely that $B$ will ever drop out of the main chain.

We can see that with this attack the chain quality roughly matches the lower bound of Theorem 11. The chain grows at the rate at which honest parties generate new blocks: roughly $(1 - \alpha)\ell$ blocks every $\ell$ blocks generated by the system as a whole. Moreover, each block generated by the adversary pushes a honestly generated block out of the main chain. Given that there are roughly $\alpha\ell$ adversarially generated blocks every $\ell$ blocks produced by the system, we conclude that the chain growth becomes roughly

$$\frac{(1-\alpha)\ell - \alpha\ell}{(1-\alpha)\ell} = \frac{1-2\alpha}{1-\alpha}$$

Substituting $1 - \alpha = 2/3$, we obtain a chain quality of roughly $1/2$ as predicted by Theorem 11.

### 3.2.6 Joining the Protocol

The proof-of-work consensus protocol we have presented is permissionless, meaning that it allows any validator to join or leave the protocol without requiring the protocol to restart. While leaving the protocol is easy (one just stop mining for new blocks), we need to explain how a validator can join. The issue here is that the newcomer needs to learn the current state of the chain in order to know which block to mine for at the current election. This requires interaction between the new party and the current set of validators.

The operation requires some care. For instance, the newcomer should not query the most recent block to a single preexisting validator: this may be a corrupted party which may try to trick them. It is fundamental that the newcomer queries the whole system (the gossip network should provide a good enough level of connectivity to ensure that the newcomer can connect with at least one honest validator). However, they may get a lot of different answers, so which one do they choose? They cannot simply take the most popular answer because nothing prevents an attacker to perform a Sybil attack and spam their incoming communication with the same wrong answer, while impersonating a large number of different parties. That is why the party needs to query the whole system for the whole chain, not just the most recent block. After that, they combine all answers into a single graph and pick the hardest branch that departs from the genesis block as their main chain (we can assume that the newcomer already knew the genesis block, as typically, this is information that is easy to retrieve online or via other sources[4]).

[4]If the genesis block is not known, it is advisable that the newcomer sets the genesis block to the root of the hardest connected component in the graph.

### 3.2.7 Advantages and Disadvantages of Proof-of-Work Consensus

In this final subsection, we summarise the advantages and disadvantages of our proof-of-work consensus protocol.

**Communication complexity.** One important but often forgotten advantage of this protocol is the low communication complexity. Compare this to the simple chain-based consensus protocol of Section 3.1: in such construction, the generation of any block required all validators to take part in a secure broadcast protocol. In particular, this step required each honest party to communicate with every other party to establish agreement. In other words, the total communication per block scaled quadratically with the number of validators. The proof-of-work based consensus protocol instead simply requires the winner of the election to distribute their newly created block: the total communication per block now scales linearly in the number of validators. For large scale systems, this is a very significant improvement.

**Permissionlessness and fluidity.** The most influential feature of the proof-of-work consensus protocol is its permissionlessness and the fluidity with which validators can join and leave the system. The fact that nobody can prevent anybody else from joining the protocol and, in order to censor any content, one would need to control the majority of computational power in the system is what inspired Satoshi Nakamoto, in the aftermath of the 2008 financial crash, to use this protocol to implement a decentralised economy. In theory, in such system no government, central bank, or major financial institution could exert enough influence on the economy to steer it against the popular will. This is the kind of unregulated and maximally fluid economy that the cypherpunk and anarcho-capitalist movements have always dreamed about, following the ideas of influential economists of the Austrian School such as Friedrich Hayek, and monetarists such as Milton Friedman.

**Inequalities.** Unfortunately, these ideas need to come to terms with reality. A deregulated economy is not necessarily more fair or equal than a regulated one (depending on the definition of fairness and equality that one adopts). The participants of a proof-of-work protocol are not equal in any way: they have influence proportional to their computational power, and who is richer can afford fancier and more efficient equipment to increase their chances of mining the next block (as pointed out in Exercise 19 and Exercise 20). We should also not forget that political power and economic power are two sides of the same coin, and one can be traded for the other. Essentially, these decentralised economies often inherit the large inequalities (economical and political) of today's world, but unlike traditional economies, they often lack a regulatory body that can mitigate their effects (or even to protect users from scams or other illegal activities). In fact, decentralised economies such as Bitcoin and Ethereum display concentrations of capital unmatched even by the most unequal countries in today's extremely unequal world. If Hayek and Friedman were alive today, they may have argued that the advantages would outweight the disadvantages. Other influential economists, some dead like Keynes, some alive like Stiglitz, Piketty and Zucman (2026), who see the effects of Friedman and Hayek ideas on today's society, would most likely disagree. Regardless of one political perspective, we need to keep in mind that today, blockchains are often the only tool that provides oppressed minorities (and sometimes majorities) in authoritarian countries with access to the benefits of the global economy. Moreover, nothing prevents us from taking inspiration from proof-of-work consensus to try to build a more equal, fairer and safer economic system.

**Energy costs.** There is however an undebatable negative aspect in proof-of-work consensus: to ensure security, the honest parties need to invest all their computational power to mine the next block. This holds even when the system is not under attack. Moreover, only few among all computed hashes end up being winners, all the rest is wasted computations to just get a chance of becoming a winner. Essentially, the protocol forces the system to be paranoid and behave as if it is under attack at all times. The most popular blockchains such as Bitcoin may be populated by millions of validators with fancy, quickly obsolete, energy-hungry mining equipment, and even more energy-hungry and water-hungry cooling systems. This drives up

the energy consumption of the system to match that of a country like Argentina, Norway or Sweden (2026). This is particularly problematic, not only for the costs, but because of the global climatic and biodiversity crisis that is increasingly affecting all of us. For this reason, the current trend is to move away from proof-of-work protocols towards proof-of-stake protocols, which offer similar low communication, permissionlessness and fluidity, while requiring virtually no energy to ensure security and correct functioning. The most famous example is Ethereum, which was originally based on proof-of-work and switched to proof-of-stake in 2022. The energy consumption of the cryptocurrency went overnight from $\approx 84$ TWh/year (roughly the same as Finland) to virtually zero, a decrease of $\approx 99.95\%$! We will discuss Proof-of Stake Consensus in detail in future lectures.

# Chapter 4

# Decentralised Economies and Smart Contracts

Most applications of blockchains in today's world are connected to economic and financial aspects. In this section, we will explain how these are typically implemented. As we mentioned before in these notes, the blockchain can be viewed as a public book, where everybody can read the content (and get consistent information) or add a new entry (nothing can be erased though). Essentially, the blockchain provides an agreed reality to which everybody can contribute. Then, why not to use it to keep track of ownership of wealth and goods? This may allow us to run the economy without having to rely on centralised institutions such as banks and other financial providers. These often hold monopolies and oligopolies on particular services, and consequently increase the cost of financial products. In this context, the promise of blockchains is a more efficient economy where everybody can participate.

How is this done in practice? Consider the following toy example: at a given point in time, the blockchain contains an entry saying that Alice owns 100ℵ. A second entry instead states that Bob owns 70ℵ. If Alice wants to buy shoes for 80ℵ from Bob, she will just add an entry to the ledger (in a new block) stating that now she only owns 20ℵ and Bob now owns 150ℵ (after that, Alice and Bob will need to organise off-chain the delivery of the shoes). This is called a *transaction* and everybody in the system will be able to see it. Of course, we typically want Alice to have control on how she spends her money, that is why we usually require her to sign the transaction under her public key. This proves that the 80ℵ were spent with Alice's approval. Transactions where the signature does not verify under the payer's public key, where the amount of transferred money does not match the amount that is received (i.e. non-balanced transactions), or where the payers spend more funding than they own are usually invalid and typically ignored. In fact, in most systems, validators are instructed to not accept any block that contains invalid transactions (and therefore, these will not be included in any block they propose).

As I said, this is just a toy example, nothing prevents us from allowing more complex transactions. For instance, a system may allow for *joint transactions* where multiple payers transfer different amounts of money to multiple payees all in one go (e.g. Alice spends 20ℵ, Bob contributes with 30ℵ, Charlie receives 10ℵ and David receives 40ℵ). Such a transaction is usually valid if all the payers hold enough funds, if the sum of paid amounts matches the sum of received amounts (in our example, $20ℵ + 30ℵ = 10ℵ + 40ℵ$), and if it is signed by each payer (in our example, Alice and Bob). The blockchain can also keep track of different types of ownership: it can record multiple currencies (e.g. an entry may say that Alice owns 20ℵ and 1000£) or even physical or digital goods (this are the so called *non-fungible tokens* (NFTs) e.g. another entry may say that Bob owns the house on 749 Leith Walk, Edinburgh), and these can be exchanged in arbitrary ways, perhaps, using joint transactions. Now, every time I talked about what is valid and what is not, I used the words "typically" or "usually". This is because there is no general rule written in stone, every system is free to come up with their own set of rules defining what is allowed and what is not. However, from an economic perspective, some systems make more sense than others. Next, we will talk about some common designs:

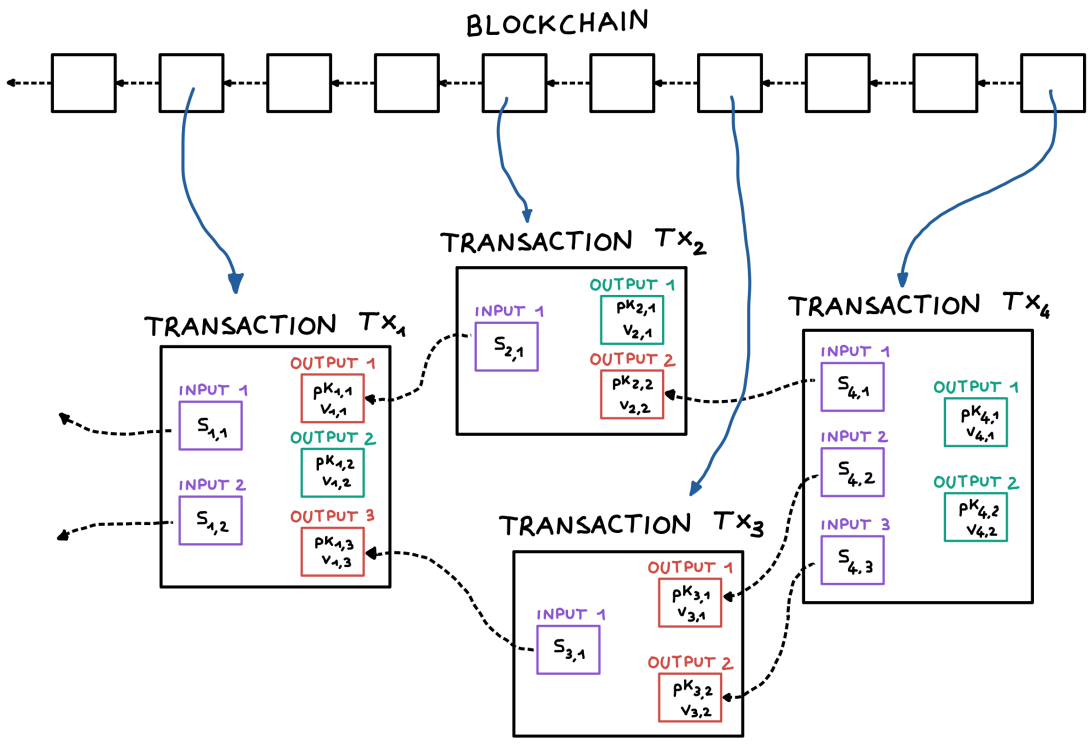


Figure 4.1: An example of UTXO transactions. Inputs are denoted in purple. Spent output are denoted in red and spendable outputs are denoted in green. Each output specifies its value and the public key of the owner. Each input references a previous output (the one it spends) and specifies a signature. Specifically, this is a signature where the message is the transaction that contains the input, and the public key is that specified by the spent output. Notice that an output is spent if and only if there is a input that references it (therefore, if and only if there is an arrow that points to it). Notice that no output can be spent more than once.

the UTXO system used in Bitcoin and the account-based system adopted by Ethereum.

## 4.1 The UTXO System

UTXO stands for Unspent Transaction Output. In this model, the blockchain only stores the list of valid transactions requested by the users throughout the history of the system.

Assume that each user is identified by their signature public key. Each transaction $\mathsf{tx}$ specifies a set of inputs and outputs, roughly corresponding of the paid amounts and the received amounts respectively: each input $\mathsf{tx.in}_i$ consists of an output of a previous transaction $\mathsf{tx}'_i$ (this is the currency that is about to be spent); every output $\mathsf{tx.out}_j$ is described instead by a non-negative amount of currency $v_j \geq 0$ and the public key of the corresponding recipient $\mathsf{pk}_j$ (which is often called the *address*). Essentially, the transaction *spends* the inputs $\{\mathsf{tx.in}_i\}_i$ and creates new *spendable* outputs $\{\mathsf{tx.out}_j\}_j$. Now, suppose that $\mathsf{tx}$ has $n$ inputs and $m$ outputs, and the $i$-th input points towards the output of a previous transaction $\mathsf{tx}_i$ that has value $v'_i$ and public key $\mathsf{pk}'_i$. In order to be valid $\mathsf{tx}$ needs to satisfy the following properties:

- It must be balanced: the sum of the values of the inputs should be equal to the sum of the values of

the outputs. In other words,

$$\sum_{j=1}^{m} v_j \stackrel{?}{=} \sum_{i=1}^{n} v'_i.$$

- It must be signed under the public keys associated with all inputs, i.e., $\mathsf{pk}'_1, \ldots, \mathsf{pk}'_n$.
- All inputs must not be spent by any preexisting transaction on the same branch of the chain (or in the same block that is about to be proposed).

Now, it is important to notice that when we spend an output, we must spend all the currency associated with it. This may at first be confusing. Suppose that there is an unspent output of 100א under Alice's public key, but Alice wants to only spend 20א. How can she proceed? If she issues a transaction pointing towards the 100א output, she will spend all 100א!

Well, there is an easy way she can do this: nothing prevent Alice to be both a payer and a payee, so she can issue a transaction with a single input (the unspent 100א), and two outputs, 20א under the payee public key and 80א under her own public key. Essentially, the transaction is giving 20א to the actual recipient and 80א of change back to Alice. The 100א output will then become spent, but the new 80א output is instead spendable.

**Exercise 21.** *In the UTXO system, every transaction consists of a list of inputs and outputs and inputs are outputs of previous transactions. Now, one needs to pay attention about how this is implemented in practice: if each transaction specifies their inputs by listing the whole transactions where these were generated, the size of the transaction grows as the cryptocurrency is exchanged from hand to hand. Specifically, if* $\mathsf{tx}'$ *spends an output created by* $\mathsf{tx}$*, it would be a bad idea to include* $\mathsf{tx}$ *as part of* $\mathsf{tx}'$*. Indeed, think about what would happen when we spend an output of* $\mathsf{tx}'$*: the new transaction* $\mathsf{tx}''$ *would need to contain* $\mathsf{tx}'$*, which contains* $\mathsf{tx}$*. This would force the size of the transactions to grow by a constant amount at each step. Essentially, the transaction size grows with the length of the "history" they carry. Try to find a solution to this problem using a cryptographic hash function (we want the transactions to have a constant size).*

**Exercise 22.** *In the UTXO system, every transaction is signed under the public keys of the payers. However, what's exactly the message that is signed? Is it just the list of inputs and outputs? Do we also sign the signatures of the other payers? Or do we sign all signatures including our own?*

### 4.1.1 The UTXO System in Bitcoin

We analyse how the UTXO model is implemented in Bitcoin. Actually, Bitcoin does not offer a completely faithful implementation of UTXO, but a more flexible version. In Bitcoin, each transaction $\mathsf{tx}$ is identified by its cryptographic hash $\mathsf{Hash}(\mathsf{tx})$(why can we assume that the hash uniquely identifies the transaction?) and is composed of a list of inputs and outputs (plus additional metadata). Each input $\mathsf{in}_i$ is composed as follows:

- The hash of the previous transaction $\mathsf{tx}'_i$ that created the output that is going to be spent.
- The index $k_i$ of the output of $\mathsf{tx}'_i$ which is going to be spent in $\mathsf{in}_i$ (in other words, $\mathsf{tx}.\mathsf{in}_i$ is going to spend $\mathsf{tx}'_i.\mathsf{out}_{k_i}$).
- A witness $w_i$ (usually a signature)

Each output $\mathsf{out}_j$ is instead composed as follows:

- A non-negative amount of cryptocurrency $v_j \geq 0$, expressed in Satoshi (1 Bitcoin = $10^8$ Satoshi)
- A script $\mathsf{script}_j$, which specifies the conditions that need to be satisfied in order to spend this output in the future. Typically, this just states that, to spend this output, the new transaction needs to be signed under the public key of the owner of this output.

To verify a transaction $\mathsf{tx}$, validators consider all inputs $\mathsf{in}_i$ and fetch the previous transactions $\mathsf{tx}'_i$ they are pointing towards. From each of these, they fetch the output $\mathsf{tx}'_i.\mathsf{out}_{k_i}$ that $\mathsf{tx}.\mathsf{in}_i$ is trying to spend and perform the following checks:

- they check that all these previous outputs $\mathsf{tx}'_i.\mathsf{out}_{k_i}$ exist and have not been spent yet by any transaction on the same branch of the chain (not even by other transactions in the same block).
- they run the script $\mathsf{script}'_i$ in each of these unspent outputs $\mathsf{tx}'_i.\mathsf{out}_{k_i}$ on the witness $w_i$ of $\mathsf{tx}.\mathsf{in}_i$. They check that all of these output $\mathsf{true}$.
- they check that the sum of the values $v'_i$ in $\mathsf{tx}'_i.\mathsf{out}_{k_i}$ is greater than the sum of all values $v_j$ in the new outputs $\mathsf{tx}.\mathsf{out}_j$ (i.e., they check that $\mathsf{tx}$ does not create money out of thin air)

If all these checks succeed, the transaction is considered valid. Notice that in Bitcoin the sum of the inputs may be strictly greater than the sum of the outputs. In fact, this is the case for almost all transactions! This does not mean that the transaction isn't balanced: the difference between the inputs and outputs is claimed by the validator that proposes the block in which the transaction is included. This is a transaction fee that the user pays for the service of the miner. Since no money was created or destroyed, the transaction is still balanced. Essentially, the fees incentivise the validators to do their work. We will discuss this aspect in detail in Section 4.4.

**What can the script check?** The script specified in each output described which conditions need to be satisfied by the transaction that is going to spend the received cryptocurrency. In order to have a faithful implementation of the UTXO, this can simply be a signature check under the public key of the owner. However, Bitcoin allows specifying more general conditions. These are expressed using a stack-based programming language supporting a vast range of operations:

- arithmetic operations (add, multiply, absolute value, ...)
- logic operation (equality, less than, AND, OR, NOT, ...)
- stack operations (drop, swap, duplicate, ...)
- flow control (if and else, ...)
- cryptographic operations (hashing, verifying signatures, ...)

For instance, a user can generate a transaction where, in order to spend an output, we require signatures from at least $t$ distinct parties in a specified committee. Alternatively, one can generate an transaction that generates an output can be spent only if somebody provides the solution of some hard mathematical problem (in this case the witness consists of the solution and script simply verifies that the solution is correct).

There are, however, limits of what this programming language can express. For instance, the script is not Turing complete as it does not support loops. Moreover, the script can only be evaluated on the witness provided by the transaction that spends that particular output. This means that it cannot express conditions about the other items in the spending transaction and about the way this is added to the chain (e.g. it cannot set conditions on the value of the outputs and of the other inputs, on the identities of the recipients and the other recipients, or about the height of the block containing the transaction). This design flaw has significantly hindered the use of Bitcoin, in favour of more flexible systems like Ethereum.

Then, why hasn't Bitcoin over these years modified its code? Well, Bitcoin is a decentralised system and so there is no central authority or influential figure that can efficiently coordinate all validators towards changing their software (Satoshi Nakamoto disappeared from the scene around 2011). Any change in the code needs to be agreed by the community as a whole (otherwise, the system risk forking), but for some reason a good fraction is rather adverse to change. For this reason the community is rather paralised. On the other hand, even if it has no central authority, Ethereum benefits from active influential figures in its community, e.g., its founder Vitalik Buterin and the Ethereum Foundation. Over the year, these figures

managed to bring a lot of change in Ethereum (for instance, they managed to move away from Proof-of-Work in favour of the more environmentally friendly Proof-of-Stake). However, for these and other reasons, fringes of the blockchain community accuse Ethereum to not be fully decentralised.

## 4.2 The Account-Based System

In account-based systems, the blockchain does not only store the transactions but also a database indicating the amount of cryptocurrency held in every account and the amount of transaction issued by each of them. Each account is identified with a signature public key $\mathsf{pk}$, its address is usually derived from $\mathsf{pk}$, for instance as $\mathsf{Hash}(\mathsf{pk})$ (this is the case for instance in Ethereum). To transfer cryptocurrency, any user can broadcast a transaction consisting of the following information:

- The address of the sender (there is only one sender).
- The address of the recipient (there is only one recipient).
- The amount of cryptocurrency $v \geq 0$ that is sent (the system may support multiple types of currencies and also NFTs, in that case, this field should also specify the type of currency that is sent).
- The number $n$ of transactions issued by the sender till this point in time (this is usually called *nonce*).
- A signature $s$ on the transaction under the sender public key.

Validators gather transactions, check their validity, and update the accounts accordingly. Formally, the validator performs the following checks:

- It retrieves the amount of cryptocurrency $w$ in the sender's account and the number of transactions $m$ they have issued so far.
- It checks that $s$ verifies under the public key of the sender public key.
- It checks that $v \leq w$.
- It checks that $n = m + 1$.

If all these checks succeed, the validator updates its version of the database: the amount of cryptocurrency on the sender's account is decreased by $v$, the number of transactions issued by the sender's account is increased by 1, and the amount of cryptocurrency on the receiver's account is increased by $v$. The block that the validator will try to propose will consist of the updated state along with all the transactions that justify these changes.

**Exercise 23** (Replay Attacks)**.** *The check on the nonce is fundamental for security. Indeed, without that check, if Alice issues a transaction of any extremely low non-zero amount (let us say* $1\aleph$*) to Bob, then Bob is able to drain all money from Alice's account. How?*

**UTXO vs Account-Based: Advantages and Disadvantages.** Both the UTXO and the account-based design have their advantages and disadvantages. The account-based model for instance provides a conceptually simple design with efficient storage (storing the balance of all accounts is easier than storing the list of all transactions). On the other hand, UTXO is better for parallelism because, unlike the account-based system, the transactions can be processed by the system (and added to the chain) in an arbitrary order without compromising their validity. Moreover, as we will see in future lectures, the UTXO model can allow a higher degree of anonymity.

## 4.3 Smart Contracts

If we compare the Bitcoin instantiation of the UTXO to the account-based model, it seems that the first offers a higher degree of flexibility on how to spend outputs. Then, why did we say that Ethereum, which is an account-based system, is actually much more expressive than Bitcoin?

The reason is that Ethereum allows for the creation of special accounts called *smart contracts*. Like all other accounts, these are associated with an amount of cryptocurrency they own and we keep track of the number of transactions they have issued. Differently from normal accounts, however, they are not controlled by any user. In fact, they are not associated with any signature key (their address is simply a string that uniquely identifies them) and are instead linked to some programming code and a contract state that describes their functioning. All of the information concerning the contract (code, contract state, amount of cryptocurrency owned and number of issued transactions) is stored on the chain at all time and continuously updated.

Any account in the system (either one owned by a user or another smart contract) can call for the execution of a specific smart contract by issuing a transaction that has the contract address as recipient. Such transactions usually specify an additional *data field*, containing information about which methods of the smart contract are called and the inputs that need to be passed to them. Validators that receive valid transactions of this kind react by transferring the spent cryptocurrency on the contract account, and then evaluate the smart contract methods on the inputs provided in the data field, updating the contract state accordingly. Sometimes, this triggers the smart contract to issue new transactions, which may simply transfer some of the cryptocurrency owned by the contract to other accounts, or may be calls to other smart contracts (in this case the transaction is called a *message*). Notice that transactions issued by smart contract do not need to be signed. First of all, smart contracts are not even associated with signature keys. Secondly, there is no need for signatures: the goal of signatures is to guarantee that a particular transaction is approved by the sender account; however, everybody can verify that the execution of the smart contract on the given inputs requires it to send that particular transaction. To summarise, the smart contract is a machine with a public state with which everybody can interact, updating its state by calling any of its methods on any input. This is called a *replicated state machine*.

**A smart contract example.** Let us make an example of a smart contract that implements an auction. This can be equipped with the following methods:

- An initialisation method. This receives as input the auction deadline (expressed e.g. as the block height past which no offer is considered) and the minimum bid (which must be non-negative). The transaction that calls this method passes ownership of the prize to the contract (for instance, this can be a non-fungible token). The contract reacts by augmenting its state with a new object storing a unique identifier $\mathsf{id}$ for the auction, the address of the highest bidder (initially set to the address of the seller), the highest bid (initially set to 0), and a list of refundable bids $L$ (initially this is empty).

- An offer method. This receives as input the unique identifier $\mathsf{id}$ for the auction. The value of the offer is equal to the amount of cryptocurrency paid in the transaction. If this is not strictly greater than the highest offer, or is lower than the minimum bid, or the auction deadline has passed, the contract updates the list $L$ by appending the pair consisting of the received amount and the account of the caller. Otherwise, the contract updates the address of the highest bidder and the value of the highest bid. It also updates $L$ by adding the pair consisting of the previous highest bid and the address of the previous highest bidder.

- A refund method. This receives as input the unique identifier $\mathsf{id}$ for the auction. The contract checks whether the list $L$ contains an entry associated with the address of the caller. If that is the case, the contract removes such entry from $L$ and transfers the indicated amount of cryptocurrency back to the caller.

- A winner method. This receives as input the unique identifier id for the auction. The contract checks that the deadline for the auction has passed, that the caller is the highest bidder, and that the prize has not been transferred yet. Then, it transfers the prize to the winner.
- A payback method. This receives as input the unique identifier id for the auction. The contract checks that the deadline for the auction has passed, that the caller is the seller, and that the highest bid has not been sent to the seller yet. Then, the contract transfers the highest bid to the seller.

So, what is the advantage of implementing an auction using a smart contract instead of a traditional one? Well, there are two main advantages. First of all, with our solution, there is no need for the participants to rely on a trustee to ensure the honest execution of the auction; they simply need to trust the security of the blockchain (in PoW this means trusting that the majority of computational power is in the hand of participants that follow the protocol). This is a fundamental security advantage! What if the trustee is not so trustful as we think and runs away with the money? Or what if the trustee rigs the auction favouring a participant over another? Smart contracts allow us to avoid these problems by decentralising trust. We do not need anymore to put all eggs in one basket!

The second advantage is instead related to costs: since there is no trustee, we do not need to pay one. Now, as we are going to discuss in a minute, smart contracts charge a service fee which covers for the cost of electricity and the hardware necessary to run the auction, however, this is likely to be much lower than the cost required to hire a trustee. Moreover, even if a particular auction smart contract were to charge very high fees, nothing prevents anybody to create a another auction smart contract that charges more reasonable fees. Since nobody can exert a monopoly on auctions, the prices are likely to drop significantly to match the real costs of the service (i.e. the cost of electricity, hardware, cooling systems etc. required to run the service). Overall, we can get a cheaper solution.

**How to Create a Smart Contract?** As we discussed above, blockchains typically allow anybody to create a new smart contract. This is definitely the case on the most important blockchains that support this technology, first among them Ethereum. For instance, to create a smart contract in Ethereum, one just needs to create a transaction where the recipient field is empty. The code of the smart contract and its initial state will be included in the data field of the transaction. Moreover, the amount of cryptocurrency transferred in this transaction is credited directly to the new created smart contract. The address of the smart contract will become $\mathsf{Hash}(\mathsf{addr}, n)$ where addr is the address of the creator and $n$ is the value of its nonce at the moment of the smart contract creation. Observe that smart contracts can be also created by other smart contracts.

## 4.4 Transaction Fees and Rewards

Computations are not free: they require specialised hardware, they use electricity, they require complex cooling systems, etc. We cannot expect a blockchain to run at no cost, especially in PoW based consensus where validators are continuously using all their power to solve complex puzzles. That is why every blockchain has its own fee and reward mechanisms which establish how users pay validators for their service and incentivise their work. This is particularly sophisticated in blockchains that offer smart contracts such as Ethereum: updating the state of a smart contract may require extremely expensive computations (depending on the smart contract). All these operations need to be performed by the validator who spends resources to derive the updated state to propose in their next block. The user that calls for the smart contract computation is therefore charged proportionally to the cost of these operations. Next, we will see how the fee mechanisms of Bitcoin and Ethereum work.

**Fees and Rewards in Bitcoin.** As we discussed in Section 4.1.1, in Bitcoin transactions, the sum of (the values of) the inputs is strictly greater than the sum of (the value of) the outputs. It is understood that the difference between these amounts are transaction fees that will be claimed by the validator that mines the

block including this transaction. The user is free to choose the amount of cryptocurrency that they offer as a fee, however, the higher the value the faster the transaction is added to the blockchain.

All fees of all transactions included in a block are claimed by the miner through a special transaction called *coinbase* (there is only one per block). This is the first transaction included in the block and is the only Bitcoin transaction that is not balanced. The coinbase transaction has no input and can have multiple outputs. In order for it to be valid, the sum of (the value of) the output should equal the sum of all transaction fees in the block plus a fixed reward that is determined by an algorithm. (If the coinbase transaction is not valid, the validators will reject the block) The reward is an amount of cryptocurrency that is freshly minted every time a block is added to the chain. The reward amount is dictated by the algorithm and depends on the height of the block: the first Bitcoin block had a reward of 50 Bitcoins ($\approx 4.3$ million dollars as of 2026), this halves every 210000 blocks (roughly every 4 years). As of 2026, the current reward is 3.125 ($\approx 270k$ dollars in 2026 prices). The next halving of the reward is expected for April 2028. The average transaction fee in 2026 is instead 2 satoshis (approximately 0.40$).

Notice that every fraction of Bitcoin in the system was minted at some point in a coinbase transaction of some block. We can even trace the history of any fraction of cryptocurrency back to the transaction that minted it by backward parsing the chain of UTXO transactions that described its movements (this is publicly available on the blockchain). The fact that the amount of minted cryptocurrency halves every fixed number of blocks means that the total amount of Bitcoin that will ever be minted is capped to

$$210000 \cdot \sum_{i=0}^{\infty} 50 \cdot 2^{-i} = 210000 \cdot 50 \cdot \underbrace{\sum_{i=0}^{\infty} 2^{-i}}_{2} = 21000000.$$

This is why Bitcoin is often referred to as the digital gold (the amount of gold on earth is also capped by a finite amount). In 2026, more than 20 million Bitcoins have been minted out of the maximum of 21 million.

It is worth explaining why this particular reward mechanism was adopted (or at least, we can only try to do so, given that Satoshi Nakamoto disappeared from the scenes around 2010). The reason why the reward was higher in the first blocks is that at the time the system was not well established, the value of Bitcoin was much lower (in 2010, somebody bought 2 pizzas for 10000 Bitcoins, which corresponds to $\approx 860$ million dollars in 2026 prices) and therefore higher rewards were needed to incentivise the work of the validators. The hope of who designed Bitcoin was that, over time, the system would become more and more established, to the point that transaction fees could suffice alone to encourage validators to do their work. What Satoshi Nakamoto probably did not expect is that the value of Bitcoin grew so much so quickly. This resulted in an extremely unequal distribution of wealth: most of the wealth in the system is still controlled by the miners of the first few blocks. For instance, Satoshi Nakamoto themselves is believed to own 1.1 million Bitcoins (approximately 94 billion dollars in 2026 prices). This is roughly 5% of all Bitcoin that will ever be mined (the richest person in the world owns 0.22% of total wealth as of 2026, and we still live in an extremely unequal world). This made Satoshi extremely rich, but also undermined the functioning of its creation and any of its claims of being "decentralised" (the economical pèower in the system is very centralised). Satoshi is probably aware of that, given that none of Satoshi's known accounts have moved any money since their disappearance from the scenes.

Anyway, so far, we have not explained why the total amount of Bitcoin that will be ever mined is capped. Well, this probably comes from the love of the cypher-punk movement (which influenced Satoshi Nakamoto thinking) for the gold standard, following the controversial ideas of the Austrian School. The gold standard has some fundamental disadvantages such as an inherent incentivation for deflation and the lack of recovery mechanisms after economic crises, e.g., quantitative easing (though supporters of the Austrian School would argue this is something positive).

**Fees in Ethereum.** In addition to the fields we have already presented (sender, amount, recipient, nonce, signature, data), each Ethereum transaction includes also a *start gas* field, a *max priority fee* field, and a *max gas price* field which are used to calculate the transaction fees.

*Gas* is the unit of measurement for the complexity of the smart contract operations. Every smart contract execution can be split up as a sequence of basic operations (e.g. additions, multiplications, Boolean operations, etc.). Each of them is associated with a fixed gas amount (e.g. an addition requires 3 gas units). The start gas field of the transaction specifies the initial gas budget for the smart contract execution. Each operation performed by the smart contract reduces the corresponding amount of gas from the budget. If this reaches 0 before the termination the smart contract execution, the validator stops and the smart contract is reverted to its started before the call. Each Ethereum transaction has also a baseline cost of 21000 gas, which de fact lower bounds the amount of gas used to process it (for instance, standard transactions have a cost of 21000 gas).

The fees will be calculated on the basis of the amount of gas that was used during the execution (this also holds for executions that fail because the gas budget reached 0, i.e., failed executions are not for free). The amount of cryptocurrency that is deducted by the caller account is equal to

$$\mathsf{usedgas} \cdot \mathsf{gasprice}.$$

The value of gasprice is chosen by the validator but cannot be greater than the *max gas price* specified in the transaction by the caller. Before executing the smart contract, the validator needs also to make sure that the caller owns an amount of cryptocurrency that is greater than the sum between the transferred amount and the product between *start gas* and *max gas price* (otherwise, the transaction is not considered valid).

Now, unlike Bitcoin, the validator only receives a part of the transaction fee: an amount equal to

$$\mathsf{usedgas} \cdot \mathsf{basefee},$$

where basefee is an algorithmically determined amount, is burnt. The rest of the transaction fee can instead be claimed by the validator. This amount is equal to

$$\mathsf{usedgas} \cdot \mathsf{priorityfee}$$

where $\mathsf{priorityfee} = \mathsf{gasprice} - \mathsf{basefee}$ is required to be smaller than the *max priority fee* specified in the transaction by the caller. Clearly, the higher the priority fee offered by the caller, the faster the transaction is processed by the validators. In addition to transaction fees, Ethereum validators receive also a per block reward similarly to Bitcoin. This is however calculated in a more complex way, so we will discuss this topic in another lecture.

Anyway, we still need to explain how to compute the base fee. This is derived from the amount of gas used by the transactions in the previous block and its basic fee. Specifically, an Ethereum block has a limit on the amount of gas used by the transactions it contains. This is set to 60 million gas. Every time a block is added to the chain, the validators compute the total amount of gas used to process its transactions. Let this be totalgasused. Then, they check whether this is above or below the target of 30 million gas (equal to half of the gas limit of the block). Then, they adjust the value of the basic fees according to it: if the total gas used is above target, the fee is increased proportionally; if instead is below target, the fee is decreased proportionally. The exact base fee is derived by the following formula

$$\mathsf{newbasefee} = \mathsf{oldbasefee} \cdot \left(1 + \frac{\mathsf{totalusedgas} - \mathsf{target}}{8 \cdot \mathsf{target}}\right)$$

Notice that since totalusedgas is capped by 2target, the base fee can increase (or decrease) at most by 12.5% at every block.

As of October 2026, the base fee is 0.217 Gwei (1 Ether=$10^{18}$ wei, 1 Gwei=$10^9$ wei). Priority fees range between 0.1 and 2 Gwei depending on the amount of traffic in the system. This brings the average cost of an Ethereum transaction to 0.25$.

Finally, let us discuss the reason behind this particular design. First of all, the decoupling between gas and fees is fundamental to account for the fluctuations of prices. Gas allows us to measure the complexity of the smart contract computation in an absolute way: it does not matter when the program is run, its

| FIELDS | STANDARD TRANSACTION FROM | | SMART CONTRACT CALL FROM | | SMART CONTRACT CREATION FROM | |
|---|---|---|---|---|---|---|
| | USERS | SMART CONTRACTS | USERS | SMART CONTRACTS | USERS | SMART CONTRACTS |
| SENDER | | | | | | |
| RECIPIENT | | | | | EMPTY | EMPTY |
| AMOUNT | | | | | | |
| SIGNATURE | | EMPTY | | EMPTY | | EMPTY |
| NONCE | | | | | | |
| DATA | EMPTY | EMPTY | | | | |
| START GAS | | | | | | |
| MAX GAS PRICE | | | | | | |
| MAX PRIORITY FEE | | | | | | |

Figure 4.2: A summary of the fields in Ethereum transactions and which fields are empty depending of the transaction type.

gas cost will be the same at all times. This makes sense given that the cost of basic operations remains roughly constant relative to each other (e.g., the ratio between the cost of a multiplication and that of an addition remains roughly constant). Prices of electricty, land, and the other resources needed to keep the system going instead fluctuate at all times based on the geopolitical situation, technological progress, and the moods of the markets. The fee mechanism in Ethereum offers on one hand an absolute and fixed way to determine the complexity of the computation, and, on the other hand, a way to adapt the price per unit of computation based on the market situation.

Another fundamental aspect of the Ethereum fee mechanism is that the basic fee is burnt instead of given to the validator. This solution makes sure that nobody profits by artificially rising gas demand. If all transaction fees were transferred to the validator, there would be an incentive for them to fill their blocks with dummy transactions that drive the gas cost up in a way that does no longer reflect real world expenses (electricity, land, etc.). In such system, dummy transactions would come at (almost) no costs for validators as they would receive all the transaction fees that they spend on them. On the other hand, it would increase the costs for normal users driving up the profits of the validators. The goal of the base fee is to estimate real world costs of the system. By burning this amount, the system ensures that nobody can speculate over it. The work of the validators is instead incentivised via the priority fees and the reward mechanisms. As an additional advantage, by burning cryptocurrency, the system abandons the gold standard and introduces a way to fight deflation (which is particularly problematic as it brings the economy to a halt).

Finally, it is worth discussing how the base price is adjusted over time. The idea is that the system wants to keep the amount of gas used in every block roughly equal to the target value. Now, if the most recent block is above target, it means that there is a lot of demand for computations (probably because they are cheap), so their price should be increased proportionally. On the other hand, if the amount of gas used in the block is below target, it must be that the cost of computations is too high. Consequently, the price needs to be lowered down.

**Exercise 24.** *In Ethereum, smart contract calls that fail due to insufficient gas budget are still charged a fee equal to* $\mathsf{startgas} \cdot \mathsf{gasprice}$ *(we recall that these failed calls have no effect on the state of the contract). This may sound unfair but actually is fundamental for the security of the system. Can you see why? Suppose that no fee was charged for failed executions, how could you attack this system?*

**Exercise 25** (Front Running)**.** *Suppose that Alice issues a transaction where she buys 1000000$ of a particular good. A few seconds later, Bob issues another transaction where he buys 1000000$ of the same good. It turns out that even if Alice's issued her transaction first, it could be that Bob's transaction is processed first. Why? How can Bob increase the chances that this happens?*

*Notice that when the demand for a particular good goes up, prices go up too. When the offer increases, prices instead go down. Try to devise a way with which Bob can extract value from Alice's purchase by relying on these market dynamics and what we discussed above. At the end of the attack, Bob must own strictly more cryptocurrency than before Alice issued her transaction. If Bob became richer, who became poorer and by how much? How does Bob discover that Alice is about to make a big purchase?*

*It turns out that this is a very common attack on blockchains and something that researchers are trying to contrast, as it artificially drives up costs. This is called* front running.