Blockchains and Distributed Ledgers
Lecture Notes

Damiano Abram

February 2026

Contents

How to Read these Notes **3**

1 Basic Mathematical and Cryptographic Notions** **4**
1.1 Notation . . . 4
1.2 A Basic Recap of Probability Theory** . . . 4
1.2.1 Some Fundamental Distributions** . . . 5
1.3 Turing Machines** . . . 7
1.4 Hash Functions** . . . 8
1.4.1 Random Oracles** . . . 10
1.5 Digital Signatures** . . . 10

2 The Consensus Problem **13**
2.1 Byzantine Agreement and Secure Broadcast . . . 13
2.1.1 Byzantine Agreement . . . 13
2.1.2 Secure Broadcast . . . 16
2.1.3 The Dolev–Strong Broadcast Protocol . . . 18
2.1.4 Bracha's Asynchronous Secure Broadcast Protocol . . . 20
2.1.5 Impossibility of Asynchronous Broadcast without Honest Supermajority . . . 23

3 Chain-Based Consensus **25**
3.1 A Simple Construction in the Synchronous Setting . . . 26
3.2 Proof of Work . . . 26
3.2.1 Permissionless Consensus . . . 27
3.2.2 Mining . . . 27
3.2.3 The Longest Chain Principle . . . 29
3.2.4 Quantifying the Computational Cost . . . 32
3.2.5 The Security Analysis . . . 34
3.2.6 Joining the Protocol . . . 42
3.2.7 Advantages and Disadvantages of Proof-of-Work Consensus . . . 42

4 Decentralised Economies and Smart Contracts **44**
4.1 The UTXO System . . . 45
4.1.1 The UTXO System in Bitcoin . . . 46
4.2 The Account-Based System . . . 48
4.3 Smart Contracts . . . 48
4.4 Transaction Fees and Rewards . . . 50
4.5 Coins and Tokens . . . 54

5 Proof of Stake **56**
5.1 A New Election Mechanism . . . 56
5.2 Verifiable Random Functions . . . 57
5.3 Updatable Signatures . . . 59
5.4 The Proof-of-Stake Election Using a Randomness Beacon . . . 60
5.4.1 An Informal Explanation of the Design . . . 62
5.5 A Security Analysis of the Proof-of-Stake Blockchain . . . 63
5.6 Implementing the Randomness Beacon . . . 71
5.7 Joining the Protocol . . . 74
5.8 Advantages and Disadvantages of Proof-of-Stake Consensus . . . 76

6 Scalability **79**
6.1 Light and Heavy Nodes . . . 79
6.2 Merkle Trees . . . 82
6.2.1 The Construction . . . 82
6.2.2 Updating a Merkle Tree . . . 86
6.2.3 Accumulators . . . 87
6.2.4 Other Applications of Merkle Trees . . . 91
6.3 Pools of Validators . . . 91
6.3.1 Pools in Proof of Work . . . 92
6.3.2 Pools in Proof of Stake . . . 93
6.3.3 Pools Without a Trusted Leader . . . 93
6.4 Succinct Arguments . . . 94
6.5 Multisignatures and Aggregatable Signatures . . . 96

7 Game Theory on Blockchains **98**
7.1 Basic Notions of Game Theory . . . 98
7.2 Game Theory in Blockchains . . . 100
7.2.1 Swapping Secrets with Deposits . . . 101
7.2.2 Selfish Mining . . . 102
7.2.3 The Proof-of-Stake Protocol of Ethereum* . . . 104
7.3 The Limits of Game Theory . . . 108

8 Anonymity and Privacy **111**
8.1 Anonymous Communication . . . 111
8.2 Pseudonymity . . . 112
8.3 Mixing Coins . . . 113
8.4 Commitments and Zero-Knowledge Proofs . . . 114
8.4.1 Commitments . . . 114
8.4.2 Zero-Knowledge Proofs . . . 115
8.4.3 Confidential Transactions in Mimblewimble . . . 116
8.4.4 Zerocash . . . 117
8.5 Signatures with Anonymity Properties . . . 119
8.5.1 Ring Signatures and Monero . . . 119
8.5.2 Blind Signatures, Electronic Cash and CoinJoin . . . 120
8.6 Should Money Be Anonymous? . . . 122

How to Read these Notes

Students of Blockchains and Distributed Ledgers should assume that, by default, every section of these notes is included in the exam. Exceptions are marked by appending the symbol * or ** to the section title. A single asterisk * denotes optional material for motivated and curious students. A double asterisk ** indicates preliminary material that is not the main focus of the course; students are nevertheless expected to have some basic familiarity with these concepts from courses they have previously taken.

Students are expected to have an informal understanding of all the definitions and theorems in these notes, i.e., to know what they state and why they matter. However, students are not expected to know all the details of the proofs, nor to understand them completely. Students are nevertheless encouraged to read the proofs, as they often explain why the definitions and the constructions are the way they are.

Chapter 1

Basic Mathematical and Cryptographic Notions**

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

1.1 Notation

For any positive integer n, we use $[n]$ to denote the set $\{1, \ldots, n\}$ and $\mathbb{Z}_n$ to denote the set $\{0, 1, \ldots, n-1\}$. For any set $\mathcal{X}$, we use $\mathcal{X}^*$ to denote $\bigcup_{i=0}^{\infty} \mathcal{X}^i$. In other words, $\mathcal{X}^*$ is the set of all finite-length vectors with entries in $\mathcal{X}$. We are particularly interested in two cases: $\mathcal{X} = \{0, 1\}$ and $\mathcal{X} = \mathbb{Z}_n$.

1.2 A Basic Recap of Probability Theory**

In this section, we recall some basic notions and results of probability theory. We start by recalling that discrete random variables are random variables that can take 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.

For a discrete random variable X, its distribution is often identified with the list $\{\Pr[X = x]\}_x$ of the probabilities of the elements in its support. For continuous random variables, this is not possible (typically, $\Pr[X = x] = 0$ for every x). Their 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 all sets $A_1, \ldots, A_n$, we have*

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

Next, we recall 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 of a sum of *independent* random variables is equal to the sum of their averages. This is, however, not true when the variables are not independent.

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

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

We also recall the union bound, which upper-bounds the probability that at least one among n events occurs.

Lemma 2 (Union bound)**.** *Let $E_1, \ldots, E_n$ be n (not necessarily independent) events. Then*

$$\Pr[\textit{at least one } E_i \textit{ occurs}] \leq \sum_{i=1}^{n} \Pr[E_i].$$

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

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

$$\Pr[S \leq (1 - \delta)\mu] \leq \exp(-\delta^2 \mu / 2),$$
$$\Pr[S \geq (1 + \delta)\mu] \leq \exp(-\delta^2 \mu / 3).$$

We also recall the following bound.

Lemma 3. *For every $x \geq 0$, it holds that*

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

1.2.1 Some Fundamental Distributions**

We start by recalling the simplest distribution: the Bernoulli distribution. This is a discrete distribution with support $\{0, 1\}$, which can be used, for instance, to model a coin toss.

Definition 5 (Bernoulli distribution)**.** *We say that a random variable X is distributed according to a Bernoulli distribution with 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 distribution, 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 $k \in \{0,1,\dots,n\}$, we have*

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

In particular, the mean of X is np.

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

Lemma 4. *Let $p \in [0,1]$ be a parameter and suppose that $X_1, \dots, X_n$ are independent random variables, all distributed according to a Bernoulli distribution with 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 describes how many times we need to sample from a Bernoulli distribution until we obtain the first success.

Lemma 5. *Let $p \in [0,1]$ be a parameter and suppose that $(X_i)_{i \in \mathbb{N} \setminus \{0\}}$ are independent random variables, all distributed according to a Bernoulli distribution with parameter p. Define $S := \min\{i : X_i = 1\}$. Then, S is distributed according to a geometric distribution with parameter p.*

We recall another fundamental discrete distribution, called the *Poisson distribution*. For instance, this can be used to model the number of customers that enter a shop in a time interval of length Δ (assuming that the probability that a new customer arrives is the same at every instant of the day and is independent of the number of previous and current customers). In this case, the parameter λ of the Poisson distribution is $\lambda_0 \cdot \Delta$, where λ_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 with 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 λ.

We conclude with the only continuous probability distribution we will use in this course: the *exponential distribution*. This can be used, for instance, to model the waiting time between one customer and the next, where the parameter λ is 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 with 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 λ^{-1} and

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

1.3 Turing Machines**

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

We start by recalling the notion of a Turing machine, which we sometimes (improperly) call an *algorithm*. Looking ahead, the reason we are interested in this definition is the Church–Turing thesis, which conjectures that every computable function 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_{init}. We call F the transition function*.*

Essentially, Q is 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 bit of the input x, and the state is set to q_{init}. At each step, the machine reads the bit b under the head and uses the transition function F to update both this bit and the state. Then, the head moves by one position in the direction (L or R) indicated by F. Notice that the head does not always need to point to entries of x: by repeatedly moving to the right, it can move past the end of x; by moving to the left, it can move before its beginning. In the first case, the character read will be $\bot$; in the second case, it will be a random bit. Either way, this value is immediately overwritten with the output of the transition function F. The execution terminates when the updated state is q_{halt}. The output is 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 $i \in \{0, \dots, |x| - 1\}$, 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 $d = \mathsf{L}$, and $s \leftarrow s + 1$ otherwise. If $t[s]$ is not yet defined, 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 the tape*.*

How many steps does a Turing machine need to produce its output? In general, no algorithm can tell. Some Turing machines terminate after a number of steps that is independent of the input length $|x|$, some require time that grows like a power of $|x|$ (e.g., $|x|^3$), some require exponential time (e.g., $2^{|x|}$), and some never stop! In blockchains and distributed ledgers, we focus on machines whose running time is upper-bounded by a power of the input length. We informally call these machines *efficient*. Of course, it is debatable whether an algorithm that runs in time $|x|^{100}$ is actually efficient; however, when $|x|$ becomes large, it will be significantly faster 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 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 to define Turing machines. In fact, the definition above will most likely differ from those in any other resource you may compare these notes with. What matters, however, is that all definitions of polynomial-time machines are equivalent: if an algorithm runs in polynomial time according

to the definition in these notes, it also runs in polynomial time according to any other valid definition of Turing machine (and vice versa).

Next, we define deterministic Turing machines. Essentially, these are the machines that never read any position to 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 s of the head to a negative value. If 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.

1.4 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 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 there are none!

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 finding any of them to require either a disproportionate amount of computation or a disproportionate amount of luck.

A formal definition. We now give a formal mathematical definition of a hash function. Unlike in the informal description above, the hash function also takes as input a *hash key* hk, sampled at random by an algorithm KeyGen. We explain below why the key is needed, and why we never see it in practice.

Definition 14 (Hash Function)**.** *Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets such that $|\mathcal{X}| > |\mathcal{Y}|$. A hash function with domain $\mathcal{X}$ and range $\mathcal{Y}$ consists of a pair of polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{Hash})$ *with the following syntax:*

KeyGen**:** *This algorithm is probabilistic and takes no input. It outputs a hash key* hk*.*

Hash**:** *This algorithm is deterministic, and takes as input a hash key* hk *and an input $x \in \mathcal{X}$. It outputs a digest* $\mathsf{Hash}(\mathsf{hk}, x) \in \mathcal{Y}$*.*

Let $T \in \mathbb{N}$ and $\epsilon \in [0,1]$. The hash function is (T, ϵ)-secure *(or* (T, ϵ)-collision resistant*) if, for every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

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

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

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: it receives a random hash key hk and, if $\mathcal{A}$ runs in time at most T, the probability that it finds two different inputs with the same digest under hk is at most ϵ. The higher the value of T and the lower the value of ϵ, the more secure the hash function. Typically, we are interested in hash functions where T and ϵ guarantee that, even if we took the most powerful computer on Earth and ran it for a billion times the age of the universe, the probability of finding a collision would be as low as the probability that a meteorite hits you in the next second: essentially, it is (almost) impossible!

Why do we need a key? Since $|\mathcal{X}| > |\mathcal{Y}|$, every function from $\mathcal{X}$ to $\mathcal{Y}$ has collisions. This means that, if the hash function had no key, there would always exist an extremely fast algorithm that finds a collision with probability 1: the algorithm that simply outputs a fixed collision (x_0, x_1), written in its code. We may not know this algorithm, but it exists, so no keyless hash function could satisfy a definition that requires every fast algorithm to fail. With a key, instead, the adversary is fixed before the key is sampled: an algorithm that always outputs the same pair (x_0, x_1) only succeeds for the keys hk such that $\mathsf{Hash}(\mathsf{hk}, x_0) = \mathsf{Hash}(\mathsf{hk}, x_1)$, which, for a good hash function, are very few.

Where are the keys? When cryptographic hash functions such as SHA2 and SHA3 are used in practice, we never see any key. The reason is that the key is essentially hard-coded in the parameters of the construction: for instance, SHA2 uses a fixed list of constants, derived from the square and cube roots of the first prime numbers. We can think of these parameters as a key sampled once and for all by the designers of the hash function. Strictly speaking, once the key is fixed, an algorithm that outputs a collision exists again. Nobody, however, knows such an algorithm: finding one would require finding a collision of SHA2, and none has ever been found. For this reason, in the rest of these notes, we always assume that we are working with such a construction, in which the key generation is implicit: we omit the key, we write $\mathsf{Hash} : \mathcal{X} \to \mathcal{Y}$ and $\mathsf{Hash}(x)$ instead of $(\mathsf{KeyGen}, \mathsf{Hash})$ and $\mathsf{Hash}(\mathsf{hk}, x)$, and we say that Hash is (T, ϵ)-secure if the corresponding keyed hash function is.

Theoretical attacks. To understand what a secure hash function looks like, let us 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 (for many hash functions, e.g., SHA2 or SHA3, this 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 is to sample $x_0 \xleftarrow{\$} \mathcal{X}$ and $x_1 \xleftarrow{\$} \mathcal{X}$, hoping that $\mathsf{Hash}(x_0) = \mathsf{Hash}(x_1)$. Regardless of how Hash is constructed, it is not too hard to prove that this attack succeeds with probability $p \geq |\mathcal{Y}|^{-1} - |\mathcal{X}|^{-1}$. When $|\mathcal{X}| \gg |\mathcal{Y}|$, this is roughly $|\mathcal{Y}|^{-1}$.

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

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

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

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

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

in order to be secure, a hash function needs to have a large 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 digest is conjectured to be extremely small, namely $p \approx 1/2^{256}$. To have a good chance of finding a collision, an attacker would need to sample $N \approx 2^{128}$ inputs (this is why we say that the standard versions of SHA2 and SHA3 provide 128 bits of security). Since the attacker would need to evaluate Hash on each of these values, this is an astronomical amount of computation and memory, far beyond what today's best computers could ever achieve! Even with enormous luck (say, winning the lottery twice in a row), the computation would require several geological eras.

1.4.1 Random Oracles**

The outputs of many hash functions look *random* and *unpredictable*. Such constructions are often used as *random oracles*. It is important to keep in mind 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 compute any "interesting" statistic (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 computed it on uniformly random strings.

Unfortunately, there is no formal definition of what a good random oracle is beyond this hand-wavy description (which is why random oracles are a somewhat contentious topic in cryptography). Nevertheless, the concept has been extremely influential in practice and today lies at the foundation of Internet security. Although not all hash functions are suitable for use as random oracles, the most popular ones, such as SHA2 and SHA3, behave sufficiently well for this purpose.

1.5 Digital Signatures**

Digital signatures are a fundamental tool for message authentication. In a nutshell, a signature is a tag s that links a message m to the signer, who is identified by a public key pk: anybody holding m, s and pk can verify that s is indeed consistent with m and pk. Deriving such a tag s is, in general, extremely hard without knowing a secret key sk, which the signer generates together with pk. To summarise, a signature scheme guarantees that, if we receive a valid signature s on a message m under the public key pk, then s was generated by somebody who knows the secret counterpart sk of pk. In other words, as long as the signer keeps their secret key sk private, nobody other than the signer can issue signatures that verify under pk.

This suggests the following use case. Imagine that a group of users $P_1, \ldots, P_n$ want to communicate over the Internet. They start by locally generating signature key pairs: each party P_i generates a random key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$, discloses pk_i to everybody else and (crucially) keeps sk_i secret. From then on, every time P_i wants to send a message m to somebody, they append a signature s on m generated using sk_i. In this way, s proves to any recipient that m was indeed sent by P_i, and not by somebody else trying to impersonate them. For example, this prevents the following attacks:

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

Change the sender: Imagine again that the attacker controls the router closest to P_i. As soon as P_i sends (m, s) to everybody else, the attacker intercepts it and sends (m, s) to the other parties, claiming that it was 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 now ready to formally define this primitive. We start by defining its 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:*

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

Sign**:** *This algorithm is often probabilistic, and takes as input the secret key* sk *and a message* m*. It outputs a signature* s*.*

Verify**:** *This algorithm is deterministic, and takes as input a public key* pk*, a message* m *and a signature* s*. It outputs a bit: 0 stands for "*reject*", 1 stands for "*accept*".*

We additionally require that, for every message m,

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

Essentially, the above property (correctness) states that the verification algorithm Verify always accepts the signatures that Sign produces using the secret key corresponding to pk.

Notice that we have not yet discussed the security requirements, namely, that without knowing the secret key, generating valid signatures that verify under a given public key pk requires either an extreme amount of computation or an extreme amount of luck. Moreover, we want this to hold even if the attacker receives signatures generated by the honest signer. Indeed, while the attacker tries to forge, the honest signer will keep issuing valid signatures to authenticate their communication, and we should expect some of these (if not all) to become known to the attacker. These must not make the attacker's task any easier: forging new signatures must remain intractable! We formalise these properties as follows.

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

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

Above, $\mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}$ means that $\mathcal{A}$ has unrestricted access to the oracle that, on input any message m', responds with a signature $s' \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}, m')$. In other words, $\mathcal{A}$ can query the oracle for signatures on messages of its 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 $\mathcal{Q}$: they were provided by the oracle! This is why the above definition only requires the probability that $\mathcal{A}$ outputs a valid signature on a message *not in* $\mathcal{Q}$ to be at most ϵ.

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

Guess the secret key: The attacker samples a random secret key from the key space $\mathcal{K}$, hoping to be lucky enough to obtain the secret counterpart of pk. This would allow them to efficiently generate as many forgeries 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 attempt is 2^{-128}, which is extremely low. Of course, we should expect the attacker to repeat this attack many times. However, to make the success probability even remotely high (say, $1/2$), the attacker would need $\approx 2^{128}$ attempts. This is an astronomical amount of computation! Observe that, if the attacker guessed the key correctly, they could easily check it: they just need to use the candidate secret key to sign a fixed message m and see whether the signature verifies under pk.

Guess a signature: The other straightforward attack is to directly guess a signature for a specific message m. Specifically, the attacker samples a random s from the signature space $\mathcal{S}$ and checks whether $\mathsf{Verify}(\mathsf{pk}, m, s) = 1$. Notice that there may exist multiple valid signatures for the same message m. Suppose that there are N of them. Then, the probability of guessing one of them is $N \cdot |\mathcal{S}|^{-1}$. To protect ourselves against this attack, $\mathcal{S}$ must be much larger than N, for instance, $|\mathcal{S}| = 2^{128} \cdot N$.

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

Many signature schemes used today choose parameters that are roughly double what we described. For instance, elliptic-curve-based signatures such as ECDSA and EdDSA typically have random 256-bit secret keys (there exist stronger variants with 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 $|\mathcal{S}| \approx N^2$).

Because 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 (SPHINCS+) 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 with 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, we need to 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 command is split among several generals, who, together with their divisions, have spread out to occupy the strategic points surrounding the city. After months of siege, the generals face a decision: should they jointly attack the city from all directions to bring the war to an early end, or should they hold their positions and wait for the city to surrender peacefully? Whatever they choose, it is fundamental that they reach consensus. If the generals do not coordinate and only some of them attack the city, they are more likely to be defeated, weakening the army as a whole and risking losing the war. Unfortunately, the generals cannot meet in person to take this decision, as they need to hold their positions, so they communicate through messengers (these are the old days, before the Internet and telephones were invented). How can they solve their problem?*

Of course, a simple solution is to use the messengers to communicate each general's 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, 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 messages, so that only a small subgroup of them attacks the city. The defending forces will then have an easier time defeating them. If they succeed, they will have weakened the rival army and, perhaps, stripped it of the strength necessary to control all the strategic points surrounding the city.

Notice that, if we try to run an economy on a blockchain (as Bitcoin, 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 *at all times*, 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 from Alice for 1000000ℵ, 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 from Bob for 1000000ℵ. In Alice's world, everything is fine: her version of the ledger has registered the transaction, she now owns the 1000000ℵ, and 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 paid to him. As long as Alice and Bob are not aware of the inconsistency between their realities, Eve can enjoy both the villa and the holiday house.

This problem is called *the consensus problem* or, after the army in the story, *the Byzantine agreement problem*.

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

When you receive 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 protocol among n participants $P_1, \ldots, P_n$. The input of each party P_i is their preference $x_i \in \{0,1\}^*$; the output of P_i is a value $y_i \in \{0,1\}^* \cup \{\perp\}$. We use the special symbol $\perp$ to denote failure, for instance because, from P_i's perspective, the protocol never terminates.*

It is useful to think of the protocol participants as stateful, probabilistic polynomial-time Turing machines 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, for every other party P_j, two message tapes (one for incoming messages and one for outgoing messages). These machines are dormant most of the time, but they activate when a message is written on one of their incoming-message tapes or on their input tape. When this happens, the machine performs some computation on its work tape based on the content of its tapes, and ends its activation by writing some messages on its outgoing-message tapes or on its output tape. Each outgoing message is then sent over the appropriate point-to-point channel and copied onto the incoming-message tape of the intended recipient. The content of the work tape is never erased and is updated from one activation to the next. At the beginning of the protocol, the input tape of each honest party contains its input, while all other tapes are empty. The outputs consist 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: in fact, the corrupted parties may be colluding against everybody else.

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

The first property requires that the honest parties eventually reach a conclusion, even if a subset of corrupted players tries to prevent it.

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

The second property guarantees that the honest parties (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 ϵ-agreement against t corruptions if, for every protocol execution where $|C| \leq t$, we have $y_i = y_j$ for every $i, j \in H$, except with probability at most ϵ.*

The last property requires 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 of the opposite.

Definition 20 (Validity)**.** *We say that a Byzantine agreement protocol satisfies ϵ-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 $y_i = x$ for every $i \in H$, except with probability at most ϵ.*

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

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

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

A first issue with this approach is that it is unclear what happens if some 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 have the same preference (say, 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 create disagreement by announcing 0 as her preference to half of the group and 1 to everybody else. One can, of course, try to prevent this issue by asking the honest parties to compare what they received, so as 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, which gives the corrupted parties an opportunity to disrupt the detection attempt by sending further inconsistent messages. For example, suppose that Bob told you that his preference is 1. Later, however, Alice claims that Bob told her that he prefers 0, while Bob denies it. Who is right? Without further information, there is no way to tell: it could be that Alice is right and Bob is lying, or vice versa.

Secure Byzantine agreement against $n/2$ corruptions is impossible. Before trying to build a Byzantine agreement protocol, we start with some bad news: as 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 strictly fewer than half of the participants are corrupted. 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 ϵ-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, i.e., that there exists a Byzantine agreement protocol that achieves ϵ-termination, ϵ-agreement and ϵ-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: There are no corrupted parties, but all parties in S_0 have input 0 and all parties in S_1 have input 1. By ϵ-agreement, we know that all parties output the same value v (except with probability at most ϵ).

Execution 2: 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 with input 1 (yes, corrupted parties can behave as they please, so they can also behave honestly). By ϵ-validity, all parties in S_0 must output 0 (except with probability at most ϵ).

Execution 3: This is the mirror image 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 with input 0. By ϵ-validity, all parties in S_1 must output 1 (except with probability at most ϵ).

To conclude the proof, we observe that the three executions contradict each other. This is because *they are actually all the same!* In all three executions, the Turing machines of the parties perform exactly the same operations. By validity in Execution 2, the probability that the parties in S_0 do not output 0 is at most ϵ. By validity in Execution 3, the probability that the parties in S_1 do not output 1 is at most ϵ. 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 ϵ-agreement in Execution 1 (because S_0 and S_1 output different values).

In conclusion, something does not work. What? It must be that we started from a wrong premise: there is no ϵ-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). As in Byzantine agreement, we want termination and agreement: even if the sender and some of the receivers are corrupted, the honest participants should eventually output the same message. Moreover, we want that, if the sender is honest, the honest receivers output exactly the message chosen by the sender.

A broken attempt. As usual, let us try to build a secure broadcast protocol and see what goes wrong. The trivial approach is the protocol in which the sender sends the message m to all receivers, who directly output it. While this satisfies validity, it does not satisfy agreement when the sender is corrupted. Indeed, nothing prevents a corrupted sender from delivering m to a subset of the receivers and another message m' to everybody else. One attempt to solve this issue is to make the receivers compare what they have received; however, we quickly run into the same kind of issue as in Byzantine agreement: the corrupted receivers may disrupt this operation by sending inconsistent information to the honest participants.

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

Definition 21 (Broadcast protocol)**.** *A broadcast protocol is an interactive protocol between a sender P_1 and $n-1$ receivers $P_2, \dots, P_n$. The input of the sender is a message $m \in \{0,1\}^*$, while all other parties have the constant 0 as input. 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 says nothing about the security and correctness requirements of such a protocol. We define them below. As for Byzantine agreement, they are called *termination*, *agreement* and *validity*, although their exact meaning is slightly different, given the different goal of the protocol.

Definition 22 (Termination)**.** *We say that a broadcast protocol satisfies ϵ-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 their output tape, except with probability at most ϵ.*

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

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

We say that a broadcast protocol is ϵ-secure against t corruptions if it satisfies ϵ-termination, ϵ-agreement and ϵ-validity against t corruptions. Since ϵ represents the probability that the protocol fails to ensure security and correctness, we would like ϵ to be as small as possible. The best case is $\epsilon = 0$, in which case we say that the protocol is perfectly secure. Sometimes, we are content with very small values of ϵ, for instance $\epsilon \approx 2^{-128}$. Large values, such as $\epsilon = 1/4$ or even $\epsilon = 1/100000$, are instead unacceptable in 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 (any receiver that does not receive a message from the sender in time for the Byzantine agreement protocol uses 0 as input). This ensures agreement. Moreover, if the sender is honest, all honest receivers run the Byzantine agreement protocol on the same input, so the validity of the Byzantine agreement protocol guarantees the validity of the 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 in which 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, they output the first one in lexicographic order). Agreement follows from the fact that all honest players agree on the message broadcast by each participant, so they all compute the output in the same way. Validity holds because, for every honest party P_i, the other honest parties learn P_i's preference. If the honest parties all have the same preference, this must be the most popular one (recall that the honest parties are in the majority).

Exercise 4. *Formally prove that, if we start from an ϵ-secure Byzantine agreement protocol, the first construction we described gives an ϵ-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 in which 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 ϵ-secure broadcast protocol, the second construction we described gives an $(n \cdot \epsilon)$-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 resulting Byzantine agreement protocol still not secure against $t \geq n/2$ corruptions? Specifically, which of 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, which withstands any number t of corruptions. It is due to Dolev and Strong. Together with the observation in the previous section, it 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 pk_i is already known to everybody, while sk_i is secret.

The protocol proceeds in $t + 2$ rounds, each of fixed duration. We assume that all participants have synchronised clocks, which they use to move to the next round in a timely manner.

- In round 0, every receiver P_j initialises a set $S_j = \emptyset$. The sender P_1 generates a signature on their message m using their secret key,
$$s_1 \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_1, m),$$
and sends $(m, (1, s_1))$ to all receivers.
- In each round $r \in \{1, \ldots, t + 1\}$, each party P_j receives tuples of the form $(\hat{m}, (i_1, \hat{s}_1), \ldots, (i_\ell, \hat{s}_\ell))$, which were sent in the previous round. Party P_j sanitises each of them, that is:
  - P_j removes every $(i_h, \hat{s}_h)$ such that $\mathsf{Verify}(\mathsf{pk}_{i_h}, \hat{m}, \hat{s}_h) = 0$.
  - After that, for every $h < k$ such that $i_h = i_k$, party P_j removes $(i_k, \hat{s}_k)$.

  If at least r signatures are left, one of them is from the sender (i.e., $i_h = 1$ for some h), none of them is from P_j themselves, and $\hat{m}$ is not already in S_j, then party P_j adds $\hat{m}$ to S_j. In this case, if $r \leq t$, 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. At the end of the final round $r = t + 1$, if S_j contains a single element, party P_j outputs it; otherwise, they output 0.

Theorem 3. *Suppose that $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ is a (T, ϵ)-unforgeable signature scheme. If the sum of the running times of all parties (honest and corrupted) is smaller than T, the Dolev–Strong protocol is an $(n \cdot \epsilon)$-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 from one round to the next in a timely way, relying on their clocks, and they always end the last round by outputting some value. □

Claim 2. *The Dolev–Strong protocol satisfies ϵ-validity.*

Proof. Suppose that the sender is honest and their input is m. In round 1, each honest receiver P_j receives $(m, (1, s_1))$ from the honest sender. Since the signature s_1 always verifies, each honest receiver P_j adds m to the set S_j. Notice that no element is ever removed from S_j, so in round $t+1$ the set S_j still contains m. Moreover, m is its only element: indeed, P_1 never signs any message other than m, and an honest receiver P_j adds a message to S_j only if it is signed by P_1. In other words, to prevent an honest receiver from outputting m, the corrupted parties would need to forge P_1's signature. By the security of the signature scheme, this occurs with probability at most ϵ. □

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

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 sends $\hat{m}$ to P_i, together 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 not have added $\hat{m}$ to S_j) and P_j (who appended their signature in round r). Notice that, when P_j received this list, their own signature was not yet present (otherwise, they would not have forwarded $\hat{m}$ to P_i). Notice also that P_i's signature is not in the list: an honest party P_i signs a message only after adding it to their set S_i. Since $\hat{m} \notin S_i$ at the end of round r, the list received by P_j contains no signature from P_i (unless the corrupted parties managed to forge one, which occurs with probability at most ϵ). Putting all these observations together, we conclude that, in round $r+1$, party P_i adds $\hat{m}$ to S_i. □

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

Proof. We show that, if an honest party P_j has some message $\hat{m}$ in their set S_j at the end of the protocol, then every other honest party P_i also has $\hat{m}$ 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, party P_i added $\hat{m}$ to S_i in round $r+1$ (except with probability at most ϵ). But what if P_j added $\hat{m}$ to S_j only in the last round? In that case, in round $t+1$, P_j must have received $t+1$ valid signatures on $\hat{m}$ from $t+1$ different parties. Since there are at most t corrupted parties, at least one of these $t+1$ signers must be an honest party P_k. This means that, unless this signature was forged (which occurs with probability at most ϵ), the honest party P_k added $\hat{m}$ to their set S_k in some round $r \leq t$. Then, by Claim 3, party P_i added $\hat{m}$ to their set S_i in round $r+1$ (except with probability at most ϵ).

This proves that, at 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 concludes the proof of Theorem 3. □

Strengths and limitations of Dolev–Strong. Dolev–Strong has some important advantages. First of all, it is a very elegant construction that is simple to describe. Secondly, unlike secure Byzantine agreement, it can withstand any number of corruptions; on the downside, however, its number of rounds grows proportionally with the number of corruptions. (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 after the second round (nevertheless, the parties must wait until the last round before they output the message they received).

There are also some less obvious 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 a trusted authority that distributes 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 binding the identities 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 scenarios, it is difficult for the parties to agree on whom to trust for this task. 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 PKI is also problematic because it assumes that the protocol

participants already know who else is joining the protocol (or at least that they already know their public keys). In large systems, this can be a problem: we cannot assume that we know the public keys of all other participants (they would simply be too many), but we may nevertheless want to receive messages broadcast by somebody we do not know yet or, vice versa, we may want our message to reach everybody, including people we do not know. To motivate this, consider again the classical problem of running an economy on a blockchain. To purchase something, I need to broadcast the information to the whole economy, so that it can be added to the ledger. Moreover, all users should receive exactly the same message even if I misbehave; otherwise, they may be vulnerable to double-spending attacks. In this scenario, the number of users is extremely large and dynamic (there may be a continuous flow of new users joining the system), and it is infeasible for me to keep an up-to-date list of all participants' public keys at all times.

The second hidden disadvantage of Dolev–Strong is that it relies on clocks. This is why we say that Dolev–Strong is a *synchronous* protocol. First of all, we cannot really assume that all clocks are perfectly synchronised (unless all parties have atomic clocks): in reality, every clock has 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 suffer 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 messages by the beginning of the next round is *a corrupted party* (regardless of their original intentions). If there are too many such parties, the protocol is no longer secure. To summarise, if too many parties are late, the honest parties may never reach agreement. This 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 longer we wait, the more secure the protocol is. On the other hand, we do not want to wait forever, because efficiency and usability are also important.

Exercise 8. *Write an implementation of Dolev–Strong in any programming language of your choice. You can have fun trying to attack the protocol.*

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

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

where $\mathcal{Q}$ denotes the set of messages that $\mathcal{A}$ queries to the oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$ (see Definition 16).

2.1.4 Bracha's Asynchronous Secure Broadcast Protocol

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

The protocol also satisfies only a weaker form of termination, which we call *asynchronous termination*: 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 receivers receive no message from the sender, how can they know whether they are dealing with a corrupted sender that refuses to communicate, or with an honest sender suffering from an extremely bad network with a lot of delay? To ensure validity in the second case, the honest participants would need to wait until the bad network delivers the honest sender's message; however, in the first case, they would end up waiting forever and never terminate. In synchronous protocols, the parties could simply rely on a clock and classify slow senders as corrupted. 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 guarantee security even to parties cursed with extremely slow networks.

Definition 25 (Asynchronous termination)**.** *We say that a broadcast protocol satisfies ϵ-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 their output tape, then all other honest parties eventually write some value on their output tape, except with probability at most ϵ.*

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

Exercise 10. *Prove that a broadcast protocol that satisfies ϵ-termination also satisfies ϵ-asynchronous termination.*

Bracha's asynchronous broadcast. We now present Bracha's protocol.

- The sender P_1 sends the message (send, m) to all 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 parties (including themselves). Any subsequent message of this form from P_1 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 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 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.

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

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

Claim 5. *An honest party P_i never receives $(\mathsf{echo}, \hat{m}_1)$ from strictly more than $2n/3$ distinct parties and, at the same time, $(\mathsf{echo}, \hat{m}_2)$ from strictly more than $2n/3$ distinct parties, for some $\hat{m}_2 \neq \hat{m}_1$.*

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

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

Every honest participant sends at most one 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, so our claim must be true. □

Claim 6. *An honest party P_i never receives $(\mathsf{ready}, \hat{m}_1)$ from strictly more than $2n/3$ distinct parties and, at the same time, $(\mathsf{ready}, \hat{m}_2)$ from strictly more than $2n/3$ distinct parties, for some $\hat{m}_2 \neq \hat{m}_1$.*

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

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

Proof. Consider the first honest party P_k that sent $(\mathsf{ready}, \hat{m})$. This party could not have sent their ready message because they had received $(\mathsf{ready}, \hat{m})$ from strictly more than $n/3$ distinct parties. Indeed, strictly fewer than $n/3$ corrupted parties could have sent this message, and, by the time P_k 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. After receiving m from P_1, every honest party P_j sends (echo, m) to everybody. This means that all honest parties receive (echo, m) from all honest parties, i.e., from strictly more than $2n/3$ distinct parties.

No honest party sends $(\mathsf{ready}, \hat{m})$ for some $\hat{m} \neq m$. Indeed, if this were the case, by Claim 7, there would exist an honest party that did so because they received $(\mathsf{echo}, \hat{m})$ from strictly more than $2n/3$ distinct parties. However, in that case, Claim 5 would imply that $\hat{m} = m$.

We conclude that all honest parties send (ready, m) to each other (the only alternative would be that some of them sent $(\mathsf{ready}, \hat{m})$ instead of (ready, m), which we have just excluded).

Once again, every honest party receives ready messages of this kind from strictly more than $2n/3$ distinct parties. Since, by Claim 6, they cannot also receive ready messages for another message $\hat{m} \neq m$ from more than $2n/3$ distinct parties, all honest parties 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, they must have received (ready, m) from strictly more than $2n/3$ distinct parties, strictly more than $n/3$ of which are honest. When honest parties send (ready, m), they deliver it to everybody. Therefore, all other honest parties also receive (ready, m) from strictly more than $n/3$ distinct parties. Unless they have already sent their ready message, all honest parties then send (ready, m) to everybody. This means that every honest party receives (ready, m) from strictly more than $2n/3$ distinct parties. By Claim 6, this ensures that every honest party outputs m.

It remains to show that no honest party has already sent $(\mathsf{ready}, \hat{m})$, for some $\hat{m} \neq m$, before receiving (ready, m) from strictly more than $n/3$ distinct parties. Suppose that this is false, i.e., that some honest party has already sent $(\mathsf{ready}, \hat{m})$ for some $\hat{m} \neq m$. By Claim 7, there exists an honest party that did so 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 (ready, m), so, again by Claim 7, there exists an honest party that sent (ready, m) because they received (echo, m) from strictly more than $2n/3$ distinct parties. Claim 5 then implies that $\hat{m} = m$. □

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

Advantages and disadvantages of Bracha's protocol. Bracha's protocol has some very interesting qualities. First of all, it does not need a PKI, and it 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 exploiting network breakdowns. Not relying on clocks also makes the protocol optimally fast: in Dolev–Strong, the parties had to waste a lot of time waiting for timers to go 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, there are only 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 fewer 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 connecting two honest parties or, put differently, that 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 is unlikely to provide us with direct connections to all other participants, so how can we solve this problem? One way is to ask every party to sign their communication using their secret key. By discarding any message whose signature does not verify, the recipient can be confident that the corrupted parties cannot impersonate honest parties. However, to implement this solution, we would again need a PKI providing 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 optimal, in the sense that no secure asynchronous broadcast protocol can withstand $n/3$ or more corrupted parties. For security, strictly more than $2/3$ of the parties must be honest, i.e., we need an *honest supermajority*.

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

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

As in the impossibility result 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 $P_1 \in S_2$. We consider the following protocol executions:

Execution 1: 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 remain silent. By ϵ-validity, we know that the parties in S_0 and S_2 eventually all output 0 (except with probability at most ϵ).

Execution 2: 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 remain silent. By ϵ-validity, we know that the parties in S_1 and S_2 eventually all output 1 (except with probability at most ϵ).

Execution 3: The sender is corrupted. The parties in S_0 and S_1 are honest, but the channels connecting them are down. The parties in S_2 are corrupted. Each corrupted party runs two copies of the protocol. In particular, it is as if every corrupted party P_j consisted 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, $P_1^{(0)}$ behaves as an honest sender that wants to broadcast $m = 0$, whereas $P_1^{(1)}$ behaves as an honest sender that wants to broadcast $m = 1$. Specifically, the corrupted party P_j behaves as follows:

- Incoming messages from a party in S_0 are forwarded to $P_j^{(0)}$;
- Incoming messages from a party in S_1 are forwarded to $P_j^{(1)}$;
- Outgoing messages from $P_j^{(0)}$ to a party in S_1 are dropped;
- Outgoing messages from $P_j^{(1)}$ to a party in S_0 are dropped;
- Outgoing messages from $P_j^{(0)}$ to a corrupted party P_k are delivered to $P_k^{(0)}$;
- Outgoing messages from $P_j^{(1)}$ to a corrupted party P_k are delivered to $P_k^{(1)}$.

By ϵ-agreement and ϵ-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 2ϵ).

In Execution 3, all parties in S_0 behave as in Execution 1. From their perspective, everything looks as if the sender were honest and wanted to broadcast 0, the parties in S_1 were totally inactive, and everybody else were honest (the parties in S_0 only interact with each other and with the virtual parties $P_j^{(0)}$, which, after all, behave honestly and only interact with other honest (possibly virtual) parties, including the sender $P_1^{(0)}$, who broadcasts 0). As a consequence, all parties in S_0 output 0 (except with probability ϵ).

On the other hand, all parties in S_1 behave as in Execution 2. From their perspective, everything looks as if the sender were honest and wanted to broadcast 1, the parties in S_0 were totally inactive, and everybody else were honest (the parties in S_1 only interact with each other and with the virtual parties $P_j^{(1)}$, which, after all, behave honestly and only interact with other honest (possibly virtual) parties, including the sender $P_1^{(1)}$, who broadcasts 1). As a consequence, all parties in S_1 output 1 (except with probability ϵ).

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

The above impossibility result can be generalised to show that, in the asynchronous setting (i.e., when the parties have no clocks and the network admits arbitrary delays), if $\epsilon < 1/3$, there exists no ϵ-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 an ϵ-secure Byzantine agreement protocol withstanding $t \geq n/3$ corruptions in the asynchronous setting, then we could construct an ϵ-secure asynchronous broadcast protocol withstanding $t \geq n/3$ corruptions. Easier case: prove the result for $\epsilon = 0$.*

Chapter 3

Chain-Based Consensus

The consensus problems we have analysed so far concern single-shot executions, in which the participants need to agree on a single value (the value broadcast by the sender, or a common decision). This structure does not capture the complexity required, for instance, by today's most common application of the theory of this course: running a decentralised economy. Unlike in Byzantine agreement, 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 a set of users $U_1, U_2, \ldots$ 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 any 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 guarantees that, when a user broadcasts a message, it will eventually be added to the chain of the honest validators. Essentially, liveness is the property that guarantees the absence of censorship.

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

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

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.*

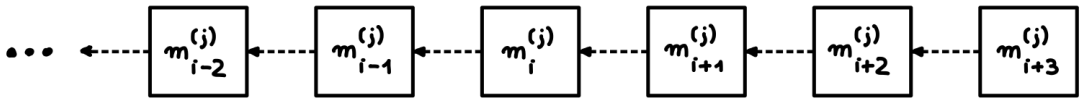

Figure 3.1: The chain of party P_j.

3.1 A Simple Construction in the Synchronous Setting

In the following protocol, we assume that all validators have synchronised clocks. 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 $(j \bmod n) = i - 1$. The protocol proceeds as follows:

1. Any user who wants to add a message $\tilde{m}$ to the chain simply sends it to all validators.
2. In each round, the validators run a secure broadcast protocol in which 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 message a *block*. All validators append the output of the secure broadcast execution to their history (removing any messages that are already on the chain).

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

Proof. It is easy to see why ϵ-agreement is satisfied: it follows from the ϵ-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 validator is honest, there is a future round in which this honest validator P_i is the leader. If $\tilde{m}$ is not yet on P_i's chain, P_i will broadcast $\tilde{m}$ as part of its block. Then, by the ϵ-validity of the secure broadcast protocol, all honest validators will add $\tilde{m}$ to their chain (except with probability at most ϵ). □

What can be added to the chain? Liveness guarantees that every user who wants to add a message to the chain will eventually succeed. This does not mean that these messages can necessarily be trusted. Some of them may even be generated by malicious users! To counter this issue, real-world blockchains often instruct the validators to ignore any message that does not satisfy certain properties. Honest validators will not include such messages in the blocks they propose when they are leaders, and will not add them to their history when they are not.

Nevertheless, some issues remain. For instance, there is no guarantee that messages are added to the history of the honest validators in the same order in which the users submitted them. If a user U_i broadcasts a message $\tilde{m}$ and the next leader P_j is corrupted, P_j may decide to temporarily censor $\tilde{m}$ and instead include the messages of U_i's competitors. Some of these messages may have been generated after U_i announced $\tilde{m}$ (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 P_j is corrupted: how could we tell whether a validator is honest or corrupted? Maybe P_j simply never received $\tilde{m}$, or U_i is lying and never sent $\tilde{m}$ 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 could 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 access it? They can simply ask all validators to send their copy of the chain (or just the last few blocks, if they already know the rest) and take the answer given by the majority of validators. It is nevertheless important that this communication occurs over authenticated channels. Assuming that at most $t < n/2$ validators are corrupted, the users can be confident that the most popular answer coincides with the true history. Notice that, if a user asked a single validator for the history, they would run the risk of picking a corrupted one, which may lie to them.

3.2 Proof of Work

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

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

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

3.2.1 Permissionless Consensus

Proof-of-work (PoW) protocols are typically *permissionless*. This means that the set of validators is not fixed in advance: anybody 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. Of course, this raises some challenges. First of all, how do validators communicate with each other if they do not even know whom to speak to?

Permissionless protocols typically rely on a peer-to-peer (P2P), or gossip, network. Instead of assuming that all protocol participants are connected by point-to-point channels, we assume that each active validator knows a small number of other active validators. Whenever a validator receives a message that they have not 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 other active participants (for instance, by forwarding their IP addresses). From now on, we assume that this network behaves well enough: whenever an honest validator receives a message m, all other honest validators receive m within some time bound Δ. We will not delve into the details of how to build this infrastructure (this topic is better suited to a network security course). However, it is important to remember that, if this assumption were to fail, the security guarantees 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 of who stands for election? Ideally, even a party that has just joined the protocol and is known by nobody should have the same chance of being elected as everybody else. How can we ensure this? And 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 were no restriction on who can join, an attacker could introduce many virtual parties to increase the chance that one of them is selected as the next leader. The attacker could then impersonate the selected 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 perform 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 still perform the Sybil attack, but the probability that one of their puppets is elected grows only with their computational power, and not with the number of 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 modelled as random oracles (see Section 1.4.1). We recall that the outputs of such hash functions look random and unpredictable. Specifically, if we compute any "meaningful" statistic on their outputs, the results are essentially the same as if the outputs were uniformly random strings. This means that, typically, we do not know any smarter way to compute the output of the hash function 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 $\mathsf{Hash} : \{0,1\}^* \to \mathbb{Z}_{2^k}$ is a hash function modelled as a random oracle,[1] let b be a constant, and let $W \in \mathbb{Z}_{2^k}$ be a hardness parameter (typically, this is the same for every validator).[2] Let $\mathsf{Hash}' : \{0,1\}^* \to \{0,1\}^k$ be another collision-resistant hash function.

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

1. Each validator gathers all the messages received from the users that are not yet on the chain into a list x. They also compute some metadata y, which typically includes information about the validator themselves and/or randomness.
2. Next, they pick the most recent block $B' = (c', z', x', y', h')$ on their chain and compute the pointer $h = \mathsf{Hash}(c', z', h')$ and the digest $z = \mathsf{Hash}'(x, y)$.[3]
3. Finally, they search for a binary string $c \in \{0,1\}^b$ such that $\mathsf{Hash}(c, z, h) < W$. We call c *the solution of the puzzle*. If they succeed before everybody else, the validator broadcasts the new candidate block $B := (c, z, x, y, h)$ over the gossip network. If someone beats them to it (or if no such c exists), they stop the search.

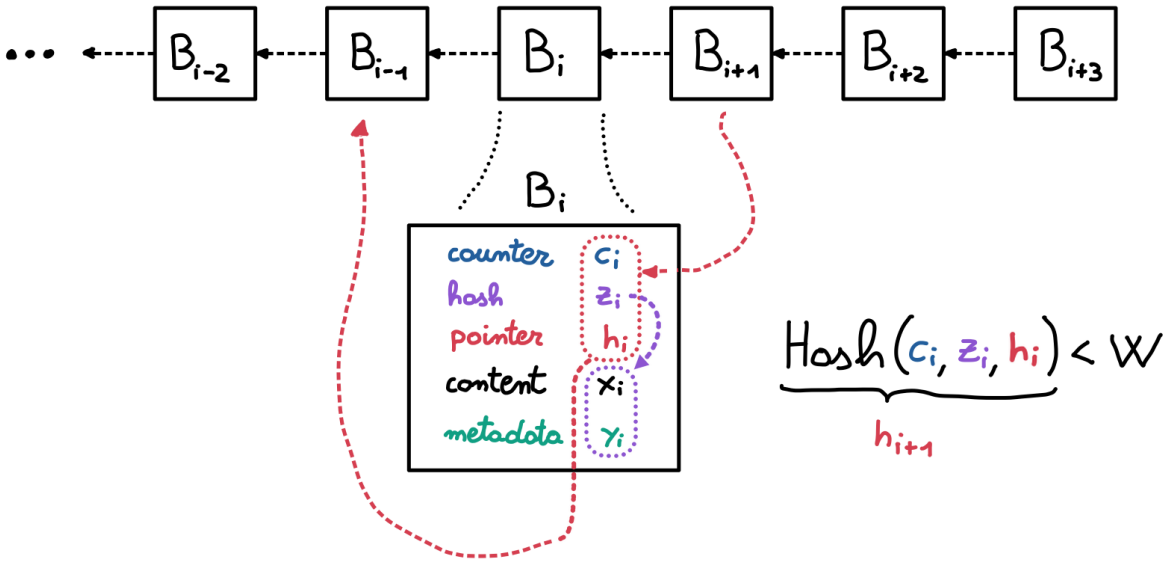

Figure 3.2: The structure of the blocks (in the figure, the digest z_i is labelled "hash"). Each coloured dotted arrow indicates hashing: the input of the hash is on the side of the arrowhead and the digest is at the tail. The purple arrow represents Hash', i.e., $z_i = \mathsf{Hash}'(x_i, y_i)$, while the red arrows represent Hash, i.e., $h_{i+1} = \mathsf{Hash}(c_i, z_i, h_i)$. Notice that the chain of blocks is linked by the red arrows alone: the pointer of each block is the hash of the counter, the digest and the pointer of the previous block.

With this solution, any newcomer has the same chance of winning the election as anybody else (proportionally to their computational power). Finding a counter c such that $\mathsf{Hash}(c, z, h) < W$ requires a significant

[1] For instance, we could use SHA2, interpreting its 256-bit output as an integer between 0 and $2^{256} - 1$.
[2] We recall that $\mathbb{Z}_{2^k}$ denotes the set $\{0, 1, \ldots, 2^k - 1\}$.
[3] We use c', z', h' to denote the concatenation of c', z' and h' (and similarly for x, y).

amount of computation, time and luck, depending on the value of W. This operation is called *mining* and, for this reason, validators in proof-of-work consensus are often called *miners*. The lower the value of W, the harder it is to win the election. In particular, when $W = 0$, the validator never has a chance to propose the next block (in fact, nobody does). If instead $W = 2^k$, the validator always manages to propose a new block (but so does everybody else). In the first case, we seriously hurt liveness; in the second case, we seriously hurt agreement. This is why the value of the hardness parameter W must be chosen with a lot of care. We will discuss this at the end of this section.

The typical strategy of a validator is to try all values of c in lexicographic order, starting from the all-zero string. Notice, however, that each validator usually works with a different value of z, because z is derived by hashing, among other things, the metadata y, which depends on the identity of the miner. Of course, not all puzzles are equally good: some are luckier than others because they admit a smaller c such that $\mathsf{Hash}(c, z, h) < W$. However, miners have no way to check whether their puzzle is lucky other than trying as many counters c as fast as they can. It is worth noticing that nothing prevents two or more miners from joining forces and pooling their resources to mine with the same metadata y. In fact, this is a very common strategy. In these cases, we say that the parties have formed a *mining pool* (see Section 6.3).

What happens after the winner has been found? Once the new leader has been established and the new block has been added to the chain, all miners move on to the next election. It is worth noticing that the winner of the previous election has only a limited advantage in the next one: once their block $B = (c, z, x, y, h)$ has been added to the chain, they need to derive 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 $\overline{h} = \mathsf{Hash}(c, z, h)$. This forces their puzzle $(\overline{z}, \overline{h})$ to change (if the new pointer $\overline{h}$ coincided with some previous value, a hash collision would have occurred). In other words, once a new block is published, all miners start solving new puzzles they have never seen before, so the search for $\overline{c} \in \{0,1\}^b$ such that $\mathsf{Hash}(\overline{c}, \overline{z}, \overline{h}) < W$ is reset for everyone. The advantage of the last winner reduces to being able to start working on their new puzzle $(\overline{z}, \overline{h})$ slightly earlier than everybody else, due to the time the network needs to deliver B to the other participants. In principle, a corrupted miner could decide not to reveal the new block B immediately, in order to gain a larger head start on solving $\overline{z}$. However, this is a risky move: while B is kept secret, other miners may solve their own puzzles and broadcast a different new block $\tilde{B}$. Even though the corrupted miner solved the puzzle first, by not declaring the win immediately, they lost the election to somebody else. On top of that, the new puzzles will contain a pointer to $\tilde{B}$, so the corrupted miner wasted all this time and energy trying to solve the wrong puzzle $(\overline{z}, \overline{h})$.

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

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

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

3.2.3 The Longest Chain Principle

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

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

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

When $h = \mathsf{Hash}(c', z', h')$, 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 yet been published on the chain, the metadata $\overline{y}$ and the pointer $\overline{h} = \mathsf{Hash}(c, z, h)$.
- We compute $\overline{z} = \mathsf{Hash}'(\overline{x}, \overline{y})$ and mine for a $\overline{c} \in \{0,1\}^b$ such that $\mathsf{Hash}(\overline{c}, \overline{z}, \overline{h}) < W$ (or, at least, we try to).

What happens, however, if B points to a block B' that is not the most recent one, but an older one? This may happen for several reasons. One, of course, is that whoever generated B is corrupted. However, it may also be that they are an honest party working with an outdated version of the chain. This kind of situation typically arises when multiple miners solve their puzzles almost at the same time. Since network communication is not instantaneous, the second winner may broadcast their new block before receiving the new block of the first winner. In these situations, all validators receive two different new blocks, which may contain contradictory information: for instance, the first one says that Alice spent all her money on a jacket, while the second one says that she spent it all on a pair of shoes.

This requires careful handling. *The validators cannot simply discard the second block they receive!* Indeed, different validators may receive the blocks in different orders due to network delays. Nodes closer to the first winner are more likely to receive the first winner's block first, whereas nodes closer to the second winner are more likely to receive the second winner's block first. If the validators adopted the first block they received, the system could split: half of it would live in a reality where Alice bought a jacket, the other half in a reality where Alice got new shoes.

To solve these issues, when B points to an older block on the chain (this is the case, for instance, when B is the second new candidate block we receive), we do not discard it; 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: B may link to our chain through a block that is still travelling through the network. In other words, the blocks are arranged in a data structure that resembles a forest more than an actual chain (despite this, we will often keep calling it a chain). One block and one branch of the forest are, however, 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. For the protocol to be secure, it is fundamental that the genesis block is generated by an honest miner, who broadcasts it to everybody else immediately.[4]
- The main chain is the branch that we try to extend by mining new blocks, and the one that determines the true course of history (e.g., it tells us whether Alice bought the jacket or the shoes). We define it as the branch that required the most computational work to be generated among those departing from the genesis block. Typically (but not always), this is the longest branch in the forest. When several branches departing from the genesis block required the same, maximal amount of work, the main chain can be chosen arbitrarily among them.

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

Before going ahead, however, we stress an important aspect: a side branch may overtake the main chain as the branch that required the most work. *In this situation, the side branch becomes the new main chain.* This is called *the longest chain principle* (even though the main chain is not necessarily the longest). This also means that history can be (partially) rewritten: the old main chain may have stated that Alice bought a jacket, while, after the switch, the new main chain states that she actually bought shoes. Luckily, such jumps usually affect only the most recent history. As we will point out in the next section, if a block has

[4]For instance, as proof that the genesis block of Bitcoin was fresh, Satoshi Nakamoto included that day's headline from The Times in its content.

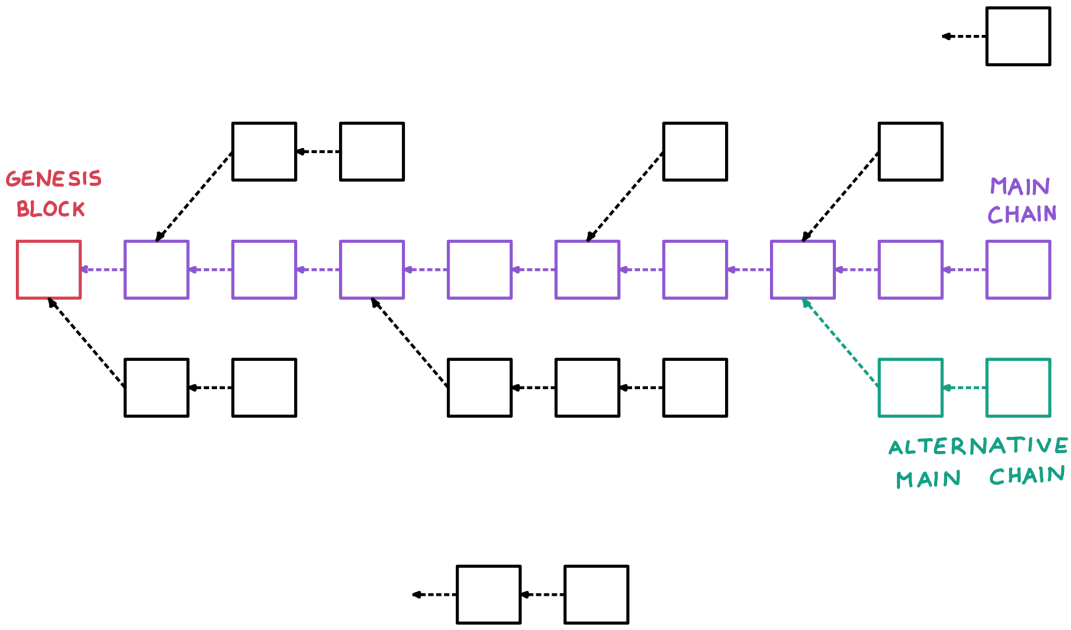

Figure 3.3: The forest of blocks of a party (notice the few disconnected components). The genesis block is shown in red, while the main chain is shown in purple. Observe that the main chain originates from the genesis block. The green branch (together with the purple section that connects it to the genesis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

been part of the main chain for sufficiently many election cycles (say, a dozen more recent blocks have been added to the main chain since its generation), it is unlikely to ever be removed from the main chain and from the main course of history. If instead a block is recent (say, 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 and end up in a forgotten course of history. The intuition behind this is that, assuming that the majority of the computational power in the system is controlled by honest miners, the majority of blocks will be produced by honest miners, who all contribute to extending the longest (or hardest) chain. For an adversary to extend a side branch enough to overtake the main chain, they would need to outperform the honest parties over a certain interval of time. This is feasible over short intervals, but becomes very unlikely over long ones. To conclude, if you ever sell a fancy car for Bitcoin, do not hand over the keys as soon as a block with the transaction appears on the main chain. If that block drops off the main chain, you will have lost both the car and the money: the chain will have switched from a course of history in which you are rich to one in which you are a fool.

Exercise 16 (Important)**.** *It is extremely unlikely that the blocks form a graph like the one shown in Fig. 3.4. Why? (Remember that the construction is based on a cryptographic hash function.)*

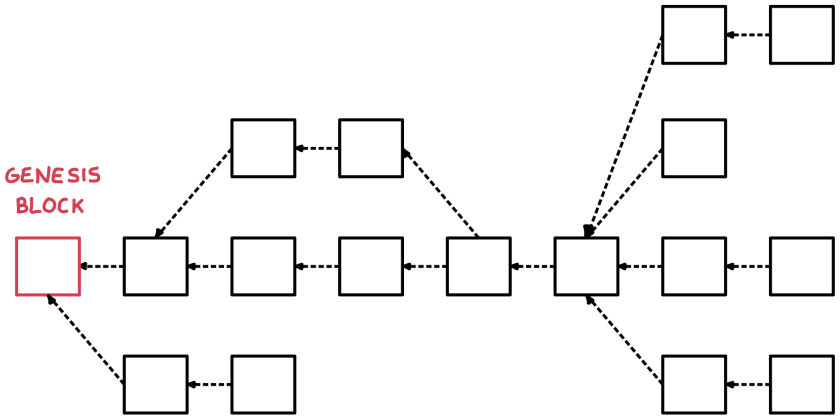

Figure 3.4: This graph structure is unlikely to ever occur. Why?

3.2.4 Quantifying the Computational Cost

Observe that, if we model the outputs of the hash function as uniformly random elements of $\mathbb{Z}_{2^k}$, then, for fixed c, z and h,

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

We can expect the miners to compute $\approx 2^k/W$ hashes before somebody finds one satisfying $\mathsf{Hash}(c, z, h) < W$. More formally, the total number of attempts required for the next block follows a geometric distribution with parameter $W/2^k$ (see Definition 7).

Given an estimate p of the amount of computational power in the system (measured in hashes per second), we can also estimate the duration of an election in seconds. This is described by an exponential distribution (see Definition 9) with 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 an amount of computation proportional to $\mathsf{cost} = n2^k/W$ and an amount of time proportional to cost/p. In other words, the longer the branch, the more work and time its generation required.

So far, we have assumed that the hardness parameter W is 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 keep the mining rate constant (we discuss this in more detail below). In this case, the computational cost of generating n blocks with hardness parameters $W_1, \ldots, W_n$ becomes

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

If we denote by p_i the total amount of computational power in the system at the time the i-th block was generated, we can expect the generation of the n blocks to take time

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

There are multiple reasons why the computational power in the system may change. For instance, new validators may join the system, or existing miners may buy fancier hardware to increase their chance of winning elections. If the hardness parameter W is kept unchanged, the increased computational power increases the speed at which new blocks are mined. This may be undesirable, as it increases the likelihood that multiple parties win an election 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 computational power in the system over 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 with hardness parameters $W_1, \ldots, W_n$ were added to the chain over time T, we can estimate the total computational power in the system as

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

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

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

Exercise 17 (Important)**.** *Consider the graph in Fig. 3.5. Each block shows the value of the hardness parameter W used for its generation. Which block can we expect to have required the least amount of computation to be generated? Which branch is the longest? Which branch 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 it with the graph in Fig. 3.3.*

From theory to practice. In 2026, values of p range 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 units number in the millions. The values of k and of the hardness parameter W are set so that, on average, we can expect a new block every few minutes (e.g., Bitcoin generates a new block roughly every 10 minutes).

In practice, every proof-of-work blockchain adjusts the hardness parameter in a slightly different way. For instance, Bitcoin tries to avoid big variations in the magnitude of W over short intervals of time. It splits time into periods called *epochs*. The hardness parameter is kept constant throughout each epoch and is recomputed, as discussed above, at the beginning of the next one. However, if the new hardness parameter W_{new} is more than 4 times the previous one, W_{old}, the new value of W is set to $4W_{\mathsf{old}}$. Conversely, if W_{new} is less than $1/4$ of W_{old}, the new value of W is set to $W_{\mathsf{old}}/4$.

Exercise 19 (Important)**.** *Suppose that a blockchain is populated by 10^6 validators, each equipped with one ASIC unit computing 10^{15} Hash/s. 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 computes 10^6 Hash/s. How long would you need to wait, on average, to mine a block with the value of W you have just found?*

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

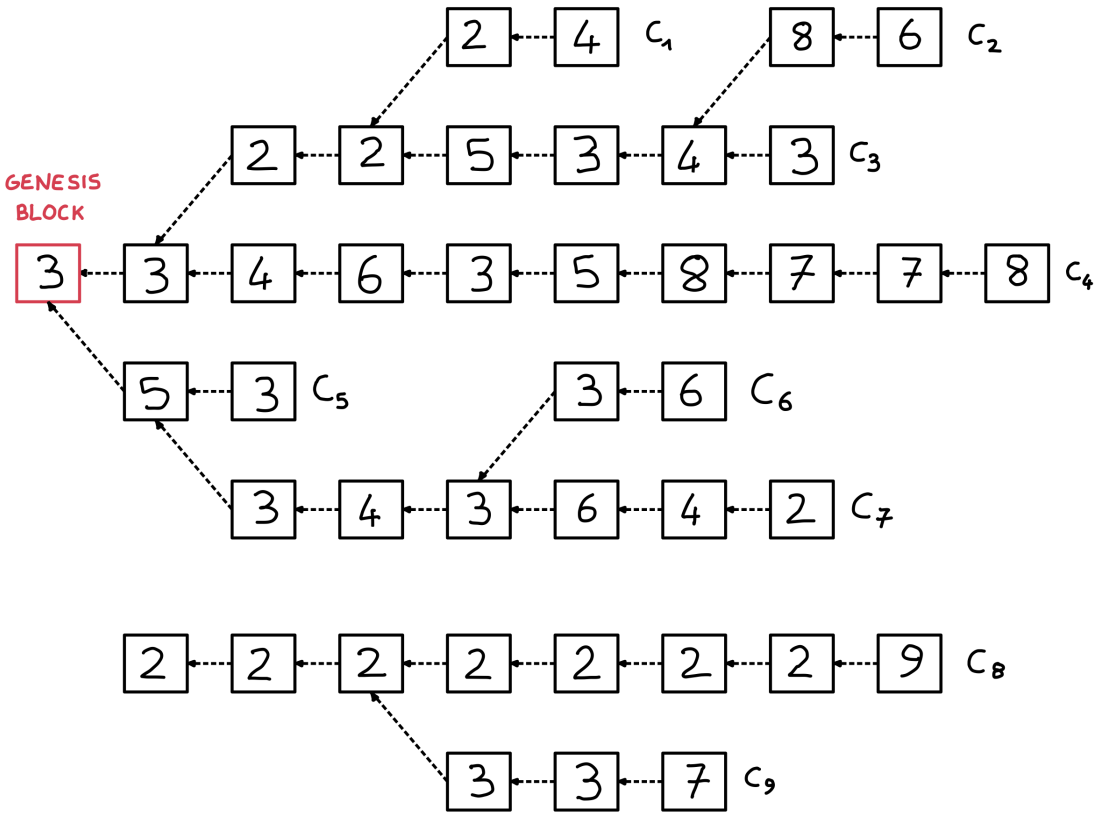

Figure 3.5: Suppose that the blocks are arranged as in the above graph, where the genesis block is shown in red and the branches are labelled $C_1, \ldots, C_9$. Each block contains the value of the hardness parameter W used for its generation (these values are much smaller than those used in the real world). What is the longest branch? What is the hardest one to generate? What is the main chain?

3.2.5 The Security Analysis

Next, we formally analyse the security of our proof-of-work consensus protocol, restricting ourselves to the simplified case in which the computational power in the system remains constant and the hardness parameter W is kept unchanged. This is a reasonable assumption, as changes in the value of W tend to be rare. Notice that, in this case, the hardest 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 this case, we say that B' is a *descendant* of B. Next, we introduce the concept of a safe block.

Definition 29 (Safe block)**.** *Suppose that the gossip network delivers each message with delay at most Δ. We say that a block is* safe *if it is mined by an honest party and no honest party generated any other block, either on the main chain or on any side branch, in the time interval of length Δ preceding it.*

In the next theorem, we show that, if the network delay Δ is only a constant fraction of the expected mining time L, then, with high probability, at least a fraction $1 - \Delta/L$ of any given sequence of ℓ consecutive

honest blocks are safe. Moreover, the probability that this does not hold decreases exponentially in ℓ.

Theorem 7. *Suppose that the corrupted parties control a fraction α of all computational power and that the gossip network has delay at most Δ. Number the honest blocks $1, 2, 3, \ldots$ in the order in which they are mined (these do not need to all lie on the same branch of the tree), and consider the sequence S of the ℓ consecutive honest blocks $i+1, \ldots, i+\ell$, where $i \geq 0$ and $\ell \geq 1$ are fixed. Then, for every $s \geq 0$,*

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

In particular, if $s = \sqrt{2\kappa \cdot \ell}$, we obtain

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

Proof. Let τ_j be the time at which the j-th honest block is mined, and let τ_0 be the time at which the execution starts (when the genesis block is created). By definition, the j-th honest block is safe if and only if no other honest block was mined in the time interval of length Δ preceding it, i.e., if and only if $\tau_j - \tau_{j-1} \geq \Delta$.

The honest parties control a fraction $1 - \alpha$ of the computational power p, and each hash they compute satisfies the puzzle with probability $W/2^k$, independently of everything else (in particular, of the behaviour of the adversary and of the block that the honest parties are trying to extend). Therefore, the time $\tau_j - \tau_{j-1}$ between two consecutive honest blocks follows an exponential distribution (see Definition 9) with parameter

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

independently of $\tau_1, \ldots, \tau_{j-1}$. Hence, the events "the j-th honest block is safe", for $j = i+1, \ldots, i+\ell$, are independent, and each of them occurs with probability

$$\Pr[\tau_j - \tau_{j-1} \geq \Delta] = \exp\left(-(1-\alpha)\cdot\frac{\Delta}{L}\right) \geq 1 - (1-\alpha)\cdot\frac{\Delta}{L} \geq 1 - \frac{\Delta}{L}$$

(see Lemma 3). In other words, the number of safe blocks in S follows a binomial distribution with ℓ trials and success probability $\exp(-(1-\alpha)\Delta/L)$. Notice that it is important to number only the honest blocks: if we numbered all blocks, the events "the j-th block is safe" would not be independent, because whether a block is safe depends on whether the block mined just before it is honest.

Let $\mu := \exp(-(1-\alpha)\Delta/L)\cdot\ell$, and notice that $(1-\Delta/L)\ell \leq \mu \leq \ell$. If $s > \mu$, the event in the statement is impossible. Otherwise, by the Chernoff bound (see Theorem 1) with $\delta = s/\mu$, we obtain

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

The second claim follows by setting $s = \sqrt{2\kappa\cdot\ell}$. □

Theorem 7 concerns a fixed sequence of honest blocks. In the proofs below, however, we need to look at sequences of blocks, honest or not, that depend on what happens during the execution, and in particular on the behaviour of the adversary: for instance, all the blocks mined after a certain honest block. To deal with this, we show that, with high probability, all sufficiently long sequences of consecutive blocks contain many safe blocks at the same time.

Corollary 1. *Suppose that the corrupted parties control a constant fraction $\alpha < 1$ of all computational power, that the gossip network has delay at most Δ, and that the expected mining time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Consider a protocol execution that ends when the system has mined N blocks, and let $\eta \in (0,1)$ be a constant. Then, for every integer $\ell \geq 1$, except with probability $N \cdot 2^{-\Omega(\ell)}$, every sequence of $m \geq \ell$ consecutive blocks contains more than $(1-\eta)(1-\gamma)(1-\alpha)\cdot m$ safe blocks.*

Proof. Number all blocks $1, 2, 3, \ldots$ in the order in which they are mined, and let $\ell_H := (1-\eta/2)(1-\alpha)\ell$. We proceed in three steps.

Honest blocks. Fix a sequence of m consecutive blocks $i+1, \ldots, i+m$. Each of them is mined by an honest party with probability $1-\alpha$, independently of the others. By the Chernoff bound (see Theorem 1) with $\mu = (1-\alpha)m$ and $\delta = \eta/2$, the probability that at most $(1-\eta/2)(1-\alpha)m$ of them are honest is at most $\exp(-\eta^2(1-\alpha)m/8)$. There are at most N choices for the first block of the sequence so, by the union bound, the probability that some sequence of $m \geq \ell$ consecutive blocks contains at most $(1-\eta/2)(1-\alpha)m$ honest blocks is at most

$$N \cdot \sum_{m=\ell}^{\infty} \exp\left(-\frac{\eta^2(1-\alpha)}{8} \cdot m\right) = N \cdot \frac{\exp\left(-\eta^2(1-\alpha)\ell/8\right)}{1-\exp\left(-\eta^2(1-\alpha)/8\right)} = N \cdot 2^{-\Omega(\ell)}.$$

Safe blocks. Similarly, fix a sequence of K consecutive honest blocks. By Theorem 7 with $s = (\eta/2)(1-\gamma)K$, the probability that at most $(1-\eta/2)(1-\gamma)K$ of them are safe is at most $\exp(-\eta^2(1-\gamma)^2K/8)$. By the union bound over the at most N choices of the first honest block of the sequence and over all $K \geq \ell_H$, the probability that some sequence of $K \geq \ell_H$ consecutive honest blocks contains at most $(1-\eta/2)(1-\gamma)K$ safe blocks is at most

$$N \cdot \sum_{K \geq \ell_H} \exp\left(-\frac{\eta^2(1-\gamma)^2}{8} \cdot K\right) = N \cdot 2^{-\Omega(\ell_H)} = N \cdot 2^{-\Omega(\ell)},$$

where the last step uses the fact that ℓ_H is a constant fraction of ℓ.

Putting things together. Except with probability $N \cdot 2^{-\Omega(\ell)}$, neither of the two bad events above occurs. In this case, consider any sequence of $m \geq \ell$ consecutive blocks. The honest blocks it contains are consecutive among the honest blocks, and their number K is greater than $(1-\eta/2)(1-\alpha)m \geq \ell_H$. Hence, the number of safe blocks in the sequence is greater than

$$\left(1-\frac{\eta}{2}\right)(1-\gamma) \cdot K > \left(1-\frac{\eta}{2}\right)^2 (1-\gamma)(1-\alpha) \cdot m \geq (1-\eta)(1-\gamma)(1-\alpha) \cdot m.$$

□

The factor N in the above bound is the price we pay for considering all sequences at the same time: the adversary can try to attack at any point of the execution. The bound is meaningful as long as ℓ is large compared to $\log N$.

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, and the *height* of the block as the length of its history. Safe blocks are fundamental to our security analysis for the two reasons summarised below.

Lemma 6. *Consider a protocol execution. Then:*

- No two safe blocks have the same height (even if they belong to the trees of different parties).
- At any time t, the main chain of any honest party is at least as long as the number of safe blocks generated up to time $t - \Delta$.

Proof. Suppose that there are two distinct safe blocks B and B' at the same height, and suppose that B' was generated after B. Since both blocks are safe, they were both generated by honest parties. Moreover, B' was generated at least Δ time units after B. Since the network has delay at most Δ, the party that generated B' had already received B, together with its history, before producing B'. This means that the length of that party's main chain was at least the height of B. Since the party appended B' to their main chain, the height of B' must be strictly greater than that of B. More precisely, we have

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

Now, let $B_1, \ldots, B_n$ be the safe blocks generated up to time $t - \Delta$, ordered by time of generation. Since $\mathsf{height}(B_1) \geq 1$, by recursively applying the above inequality, we obtain

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

Since B_n was generated by time $t - \Delta$, every honest party has received it, together with its history, by time t. Hence, at time t, the main chain of every honest party is at least as long as the height of B_n, i.e., at least n. □

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

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

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

and that all honest parties start from the same genesis block, which is safe. Let $\ell \geq 1$ be an integer, and, for every chain C, let $\lfloor C \rfloor_\ell$ denote the chain obtained by removing the ℓ most recent blocks from C. Then, except with probability $N \cdot 2^{-\Omega(\ell)}$, the following holds at every point in time t: if C_1 and C_2 are longest chains in the block trees of two honest parties at time t (the two parties may coincide), then

$$\lfloor C_1 \rfloor_\ell \subseteq \lfloor C_2 \rfloor_\ell \qquad \text{or} \qquad \lfloor C_1 \rfloor_\ell \supseteq \lfloor C_2 \rfloor_\ell.$$

Proof. Since the bound in the statement is asymptotic, we can assume that ℓ is larger than any given constant. Let $\eta > 0$ be a constant such that $q := (1 - \eta)(1 - \alpha)(1 - \gamma) > 1/2$ (it exists because $(1 - \alpha)(1 - \gamma) > 1/2$). By Corollary 1, except with probability $N \cdot 2^{-\Omega(\ell)}$, every sequence of $m \geq \ell$ consecutive blocks contains more than $q \cdot m$ safe blocks. Since $q > 1/2$ and ℓ is large enough, this is more than $(m + 1)/2$ safe blocks (indeed, $q \cdot m \geq (m + 1)/2$ whenever $m \geq 1/(2q - 1)$). We show that, whenever this is the case, the property in the statement holds at every point in time.

Suppose, towards a contradiction, that, at some time t, the chains C_1 and C_2 are longest chains in the trees of two honest parties P_1 and P_2, but neither $\lfloor C_1 \rfloor_\ell \subseteq \lfloor C_2 \rfloor_\ell$ nor $\lfloor C_1 \rfloor_\ell \supseteq \lfloor C_2 \rfloor_\ell$. Let h_1 and h_2 be the lengths of C_1 and C_2. Without loss of generality, we assume that $h_1 \leq h_2$. Let B^* be the most recent block belonging to both C_1 and C_2 (the two chains have at least the genesis block in common), and let h^* be its height. Observe that $h_1 - h^* \geq \ell + 1$. Otherwise, $\lfloor C_1 \rfloor_\ell$, which consists of the first $h_1 - \ell \leq h^*$ blocks of C_1, would be part of the common prefix of the two chains, and would therefore be contained in $\lfloor C_2 \rfloor_\ell$, which consists of the first $h_2 - \ell \geq h_1 - \ell$ blocks of C_2.

Now, let B_0 be the most recent *honest* block belonging to both C_1 and C_2 (this block exists because the genesis block is safe, and hence honest). Let $h_0 \leq h^*$ be its height, and let t_0 be the time at which it was mined. All blocks of C_1 and C_2 with height greater than h_0 are descendants of B_0, so they were mined in the time interval $(t_0, t]$. We now count the safe and the non-safe blocks mined in this interval (see Fig. 3.6 for an example).

Safe blocks. Consider any safe block B mined in the time interval $(t_0, t - \Delta]$. Since B_0 is honest and B is safe, B was mined at least Δ time units after B_0. By then, the party that mined B had already received B_0, so its main chain had length at least h_0, and the height of B is greater than h_0. Moreover, P_1 received B, together with its history, by time t. Since C_1 is a longest chain in the tree of P_1, the height of B is at most h_1. By Lemma 6, no two safe blocks have the same height, so at most $h_1 - h_0$ safe blocks were mined in $(t_0, t - \Delta]$. Moreover, safe blocks are mined at least Δ apart, so at most one safe block was mined in $(t - \Delta, t]$. In total, at most $h_1 - h_0 + 1$ safe blocks were mined in $(t_0, t]$.

Non-safe blocks. Consider any height h such that $h_0 < h \leq h_1$. If $h \leq h^*$, the block of height h on C_1, which also lies on C_2, is not honest (by the choice of B_0), so it is not safe. If $h > h^*$, the two chains contain

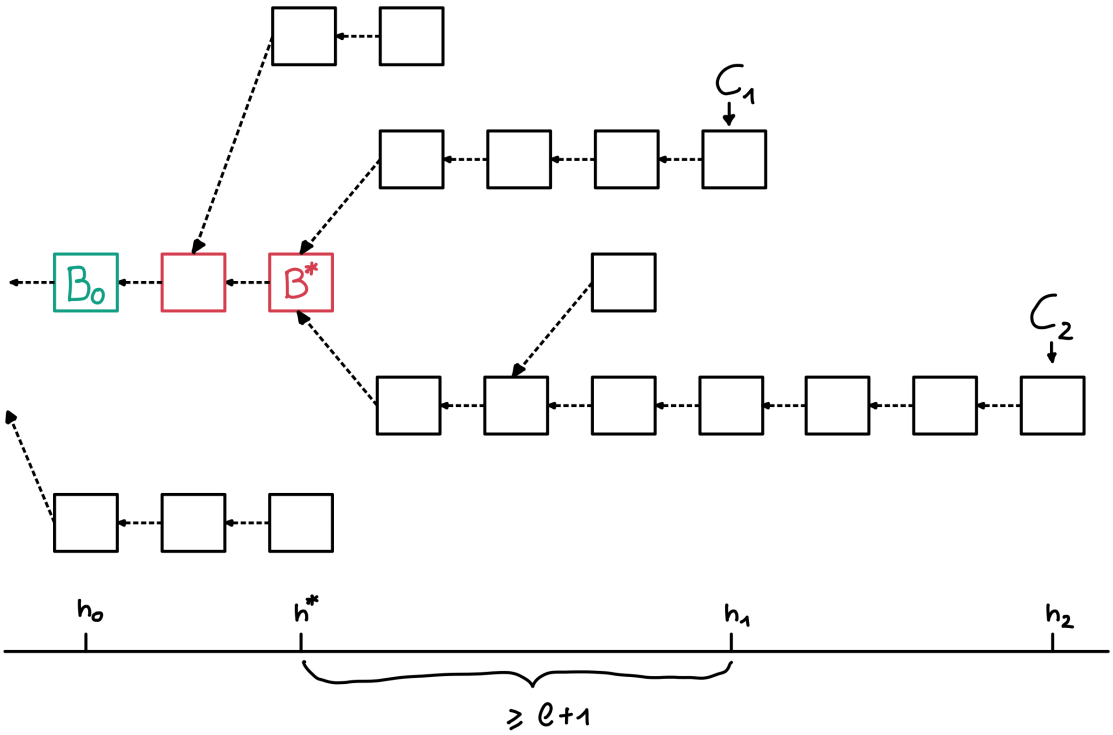

Figure 3.6: An example of the blocks considered in the proof of Theorem 8 (in this example, $\ell \leq 3$). Heights are indicated on the line at the bottom. The green block B_0 is the most recent honest block belonging to both C_1 and C_2, so the red blocks between B_0 (excluded) and B^* (included) are not honest, and hence not safe. For the black blocks, the proof only uses the fact that, by Lemma 6, at most one block per height can be safe. Notice that some blocks mined after B_0 belong neither to C_1 nor to C_2, and some of them are not even descendants of B_0. In this example, $h^* = h_0 + 2$ and $h_2 = h_1 + 3$: C_2 is longer than C_1 because, at time t, P_1 has not yet received the last three blocks of C_2. In general, any heights with $h_0 \leq h^*$ and $h^* + \ell < h_1 \leq h_2$ are possible.

two distinct blocks of height h and, by Lemma 6, at most one of them is safe. Either way, at least one block of height h on C_1 or C_2 is not safe. Since all these blocks were mined in $(t_0, t]$, at least $h_1 - h_0$ non-safe blocks were mined in $(t_0, t]$.

Let m be the number of blocks mined in $(t_0, t]$. These blocks form a sequence of consecutive blocks, and $m \geq h_1 - h_0 \geq h_1 - h^* \geq \ell + 1$. By the above, the number of safe blocks in this sequence is at most $h_1 - h_0 + 1$, i.e., at most the number of non-safe blocks plus one. Hence, the sequence contains at most $(m+1)/2$ safe blocks. This contradicts our assumption on the execution. □

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

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

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

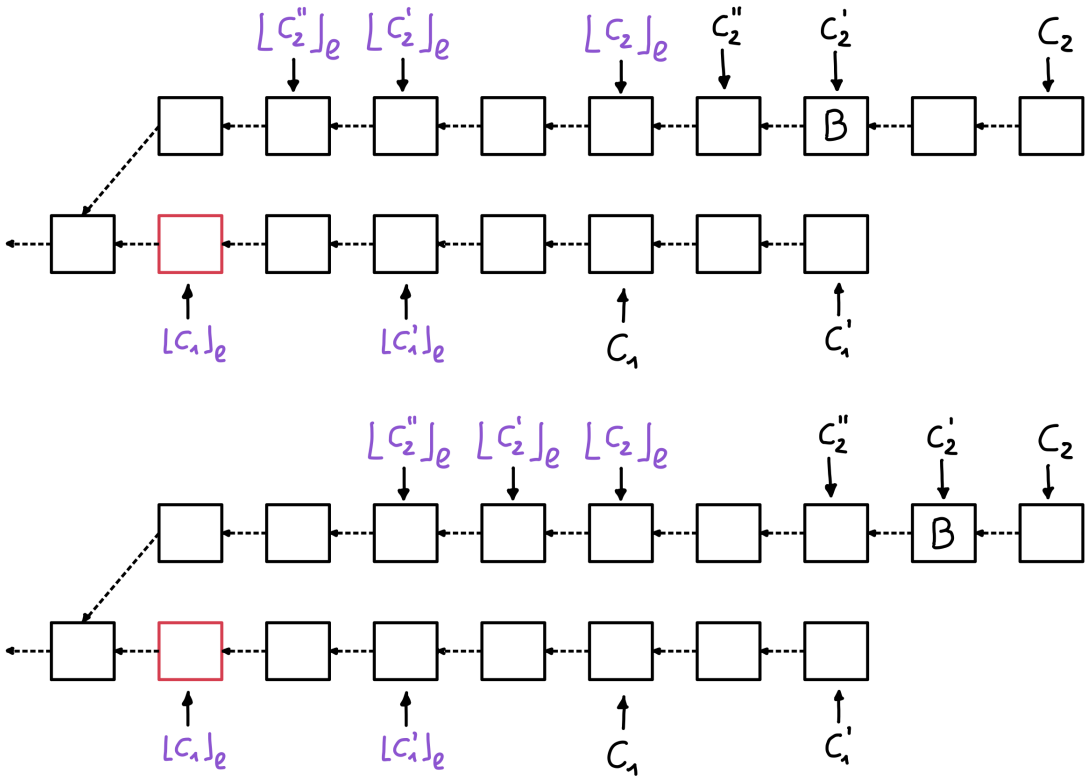

Figure 3.7: The two possible shapes of the tree in the proof of Theorem 9 (in this example, $\ell = 4$). At the top is the case in which C_1' and C_2' have the same length. At the bottom is the case in which C_2' is one block longer than C_1'. The chains with the last ℓ blocks removed are labelled in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

and pick any honest party and any two time instants $t_1 \leq t_2$. Let C_1 and C_2 be the main chains of the party at times t_1 and t_2, respectively, and suppose that their genesis block is safe. Let ℓ be a positive integer. Then,

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

Proof. Suppose that $\lfloor C_1 \rfloor_\ell \not\subseteq \lfloor C_2 \rfloor_\ell$ and consider the blocks added to the party's block tree between time t_1 and time t_2. Some of these blocks have the following property: before they were added, the main chain contained $\lfloor C_1 \rfloor_\ell$, whereas, after they were added, this was no longer the case. Let B be the first such block. Let C_1' be the longest chain before B was added, let C_2' be the longest chain after B was added, and 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 B is added, the length of C_1' is smaller than or equal to that of C_2'. Similarly, by the maximality of C_1' before B is added, the length of C_2'' is smaller than or equal to that of C_1'. Since the lengths of C_2' and C_2'' differ by 1, the length of C_1' must match either that of C_2' or that of C_2''. In the first case, C_1' and C_2' are both longest chains in the party's tree right after B is added. In the second case, C_1' and C_2'' are both longest chains in the party's tree right before B is added. Except with probability $N \cdot 2^{-\Omega(\ell)}$, the property of Theorem 8 holds at every point in time, including the (random) time at which B is added. Since the two chains in each pair have the same length, their truncations are then equal: either $\lfloor C_1' \rfloor_\ell = \lfloor C_2' \rfloor_\ell$, or $\lfloor C_1' \rfloor_\ell = \lfloor C_2'' \rfloor_\ell \subseteq \lfloor C_2' \rfloor_\ell$. Either way, we obtain

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

This contradicts the fact that B was chosen 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 gives a lower bound on the speed at which the main chain grows when the adversary tries to slow the system down, for instance by not contributing their computational power to mining.

Theorem 10. *Consider a protocol execution in which the adversary controls a constant fraction $\alpha < 1$ of all computational power in the system, the network has delay at most Δ, and the expected mining time is L. Consider the main chain of an honest party at time t. Then, except with probability $\exp(-\kappa)$, the honest party extends their main chain by a new block by time*

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

Proof. Let B be the last block added to the main chain of the honest party, let h be its height, and let $t' \leq t$ be the time at which this block was received. Now, suppose that no honest party in the system has extended any branch of their tree to length $h+1$ or more by time $t' + \Delta$. By that time, we can be sure that the longest chain of every honest party has length h (it may or may not coincide with that of the party we selected). This means that, at time $t' + \Delta$, all honest parties are working to generate a block of height $h+1$. Now, the time required by the honest parties to generate such a block follows an exponential distribution (see Definition 9) with 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 with delay at most Δ. □

The above theorem establishes that the main chain keeps growing. However, it says nothing about the quality of this growth. For instance, it would be quite bad if only the adversary were adding blocks. In the following, we provide a lower bound on the fraction of honest blocks that end up on the main chain. We call this ratio the *chain quality*.

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

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

Pick an honest party and suppose that their genesis block is safe. Consider any sequence S of ℓ consecutive blocks on the main chain of the honest party at some point in time. Then,

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

We call

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

the chain quality ratio.

Proof. Let $q := (1-\eta)(1-\gamma)(1-\alpha)$, so that the ratio in the statement is $(2q-1)/q$, and let $q' := (1-\eta/2)(1-\gamma)(1-\alpha)$. Notice that $1/2 < q < q'$. Since the bound in the statement is asymptotic, we can assume that ℓ is larger than any given constant. By Corollary 1 with $\eta/2$ in place of η, except with probability $N \cdot \exp(-\Omega(\ell))$, every sequence of $m \geq \ell$ consecutive blocks contains more than $q' \cdot m$ safe blocks. We show that, whenever this is the case, S contains at least $\frac{2q-1}{q} \cdot \ell$ honest blocks.

Let t be the time at which we look at the main chain C of the honest party, and suppose that S consists of the blocks of C with heights $u+1, \ldots, u+\ell$. Let B_0 be the most recent honest block of C with height at most u (it exists because the genesis block is safe), let h_0 be its height, and let t_0 be the time at which it was mined. Similarly, let B_1 be the earliest honest block of C with height greater than $u+\ell$, let h_1 be its height, and let t_1 be the time at which it was mined. If there is no such block, we set $h_1 := |C|+1$ and $t_1 := t$. Let D be the sequence of blocks of C with heights strictly between h_0 and h_1. By construction, D contains S, so $|D| \geq \ell$, and all the honest blocks of D belong to S. Moreover, all blocks of D are descendants of B_0, and they were mined before time t_1 (they are ancestors of B_1 or, if B_1 does not exist, they belong to the tree of the party at time t). Hence, all blocks of D were mined in the time interval $(t_0, t_1]$.

Let m be the number of blocks mined in $(t_0, t_1]$. These form a sequence of consecutive blocks with $m \geq |D| \geq \ell$, so more than $q' \cdot m$ of them are safe. On the other hand, as in the proof of Theorem 8, every safe block mined in $(t_0, t_1 - \Delta]$ has height greater than h_0. Its height is also smaller than h_1: if B_1 exists, the party that mined B_1 had already received the safe block and its history; otherwise, the honest party received it by time t, so its height is at most $|C|$. Since no two safe blocks have the same height, and at most one safe block was mined in $(t_1 - \Delta, t_1]$, we obtain

$$q' \cdot m < \#\text{ safe blocks mined in } (t_0, t_1] \leq |D| + 1.$$

Now, the blocks of D that are not honest are not safe, and they were mined in $(t_0, t_1]$. Therefore, their number is at most the number of non-safe blocks mined in $(t_0, t_1]$, which is smaller than $m - q' \cdot m$. Putting everything together, and using $m < (|D|+1)/q'$, we obtain

$$\begin{aligned}\#\text{ honest blocks in } S &= |D| - \#\text{ non-honest blocks in } D > |D| - (1-q') \cdot m > |D| - (1-q') \cdot \frac{|D|+1}{q'} \\ &= \frac{2q'-1}{q'} \cdot |D| - \frac{1-q'}{q'} > \frac{2q'-1}{q'} \cdot \ell - 1,\end{aligned}$$

where the last step uses $|D| \geq \ell$, $2q'-1 > 0$ and $1-q' < q'$ (all because $q' > 1/2$). Finally, the difference

$$\frac{2q'-1}{q'} - \frac{2q-1}{q} = \frac{1}{q} - \frac{1}{q'} = \frac{q'-q}{q \cdot q'}$$

is a positive constant. Hence, as soon as $\ell \geq q \cdot q'/(q'-q)$, we have $\frac{2q'-1}{q'} \cdot \ell - 1 \geq \frac{2q-1}{q} \cdot \ell$, which concludes the proof. □

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

Suppose that, when an honest party has multiple longest chains in their tree, they set their main chain to the branch that first reached that length (this is what typically happens in Bitcoin). Consider an adversary that, even though 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 through connections that are much faster than all connections between two honest parties. Let us say that their delay is at most $\delta \ll \Delta'/2$, where Δ' is the 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, it does not immediately reveal its block B to the honest parties. Instead, it withholds the block and listens on the channels connecting it to the honest parties. As soon as the adversary receives a new block B' from an honest party, it sends B to everybody else. Since its channels are much faster than those of the honest parties, B reaches everybody else before B' does. In other words, all honest parties (except the one that mined B') include B in their main chain instead of B'. After that, all parties except one start mining 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 for every ℓ blocks generated by the system as a whole. Moreover, each block generated by the adversary pushes an honestly generated block out of the main chain. Since there are roughly $\alpha\ell$ adversarially generated blocks for every ℓ blocks produced by the system, we conclude that the chain quality 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 restarting it. While leaving the protocol is easy (one just stops mining 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 on in the current election. This requires interaction between the new party and the current set of validators.

This operation requires some care. For instance, the newcomer should not ask a single pre-existing validator for the most recent block: this validator may be a corrupted party trying 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 to at least one honest validator). However, they may receive many different answers, so which one should they choose? They cannot simply take the most popular answer, because nothing prevents an attacker from performing a Sybil attack and flooding the newcomer with the same wrong answer, while impersonating a large number of different parties. This is why the newcomer needs to ask 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 departing from the genesis block as their main chain (we can assume that the newcomer already knows the genesis block since, typically, this information is easy to retrieve online or from other sources[5]).

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 its low communication complexity. Compare it with the simple chain-based consensus protocol of Section 3.1: in that construction, the generation of each 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 grew quadratically with the number of validators. The proof-of-work consensus protocol, instead, only requires the winner of the election to distribute their newly created block: the total communication per block now grows linearly with 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, together with the fluidity with which validators can join and leave the system. The fact that nobody can prevent anybody else from joining the protocol, and that censoring any content would require controlling the majority of the 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 a system, no government, central bank or major financial institution could

[5] If the genesis block is not known, the newcomer should set the genesis block to the root of the hardest connected component in the graph.

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 of, following the ideas of influential economists of the Austrian School, such as Friedrich Hayek, and of monetarists, such as Milton Friedman.

Inequalities. Unfortunately, these ideas need to come to terms with reality. A deregulated economy is not necessarily fairer or more equal than a regulated one (depending on the definitions of fairness and equality one adopts). The participants of a proof-of-work protocol are by no means equal: their influence is proportional to their computational power, and those who are 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 (economic and political) inequalities of today's world but, unlike traditional economies, they often lack a regulatory body that can mitigate their effects (or even protect users from scams and 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 might argue that the advantages outweigh the disadvantages. Other influential economists would most likely disagree: some no longer alive, like Keynes, and others still active, like Stiglitz, Piketty and Zucman (2026), who have seen the effects of Friedman's and Hayek's ideas on today's society. Regardless of one's political perspective, we need to keep in mind that, today, blockchains are often the only tool that gives oppressed minorities (and sometimes majorities) in authoritarian countries 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 undeniably negative aspect of proof-of-work consensus: to ensure security, the honest parties need to invest all their computational power in mining the next block. This holds even when the system is not under attack. Moreover, only a few of all computed hashes end up being winners; all the rest are computations wasted just to get a chance of winning. Essentially, the protocol forces the system to be paranoid and to behave as if it were 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 the energy consumption of the system up to that of a country like Argentina, Norway or Sweden (2026). This is particularly problematic, not only because of the costs, but also because of the global climate 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 similarly 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 dropped overnight from ≈ 84 TWh/year (roughly the same as Finland) to virtually zero, a decrease of $\approx 99.95\%$! We discuss proof-of-stake consensus in detail in Chapter 5.

Chapter 4

Decentralised Economies and Smart Contracts

Most applications of blockchains in today's world are related to economics and finance. In this chapter, we explain how these applications are typically implemented. As mentioned earlier in these notes, the blockchain can be viewed as a public book, in which everybody can read the content (and get consistent information) or add a new entry (but nothing can be erased). Essentially, the blockchain provides an agreed reality to which everybody can contribute. So why not use it to keep track of the ownership of wealth and goods? This may allow us to run the economy without relying on centralised institutions, such as banks and other financial providers, which often hold monopolies or oligopolies on particular services and consequently increase the cost of financial products. In this context, the promise of blockchains is a more efficient economy in which 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 stating that Alice owns 100ℵ, and a second entry stating that Bob owns 70ℵ. If Alice wants to buy shoes from Bob for 80ℵ, she just adds an entry to the ledger (in a new block) stating that she now owns only 20ℵ and that Bob now owns 150ℵ (after that, Alice and Bob need to arrange the delivery of the shoes off-chain). This is called a *transaction*, and everybody in the system can see it. Of course, we typically want Alice to have control over how she spends her money, which 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 whose signature does not verify under the payer's public key, in which the amount of money transferred does not match the amount received (i.e., unbalanced transactions), or in which the payers spend more funds than they own, are usually invalid and ignored. In fact, in most systems, validators are instructed not to accept any block containing invalid transactions (and, therefore, not to include such transactions in the blocks 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 *joint transactions*, in which multiple payers transfer different amounts of money to multiple payees all in one go (e.g., Alice pays 20ℵ, Bob contributes 30ℵ, Charlie receives 10ℵ and David receives 40ℵ). Such a transaction is usually valid if all payers hold enough funds, if the sum of the amounts paid matches the sum of the amounts received (in our example, $20ℵ + 30ℵ = 10ℵ + 40ℵ$), and if it is signed by every 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 state that Alice owns 20ℵ and £1000) or even physical or digital goods (these are the so-called *non-fungible tokens* (NFTs); e.g., another entry may state that Bob owns the house at 749 Leith Walk, Edinburgh), and all of 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 its 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 discuss two common designs:

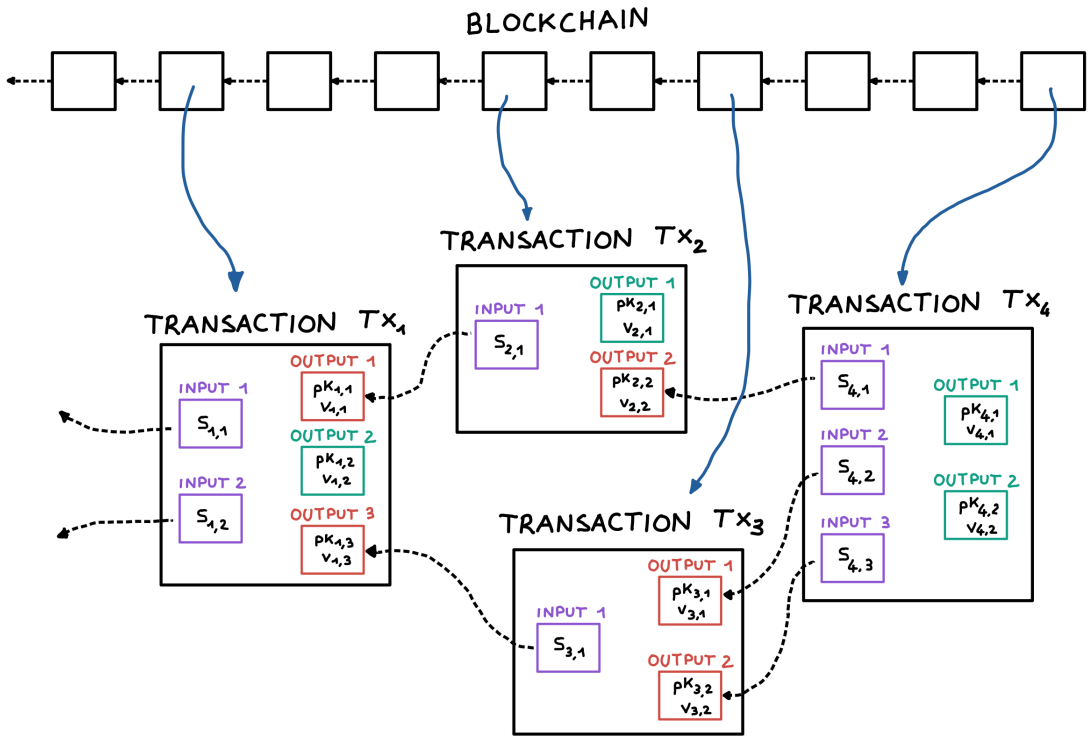

Figure 4.1: An example of UTXO transactions. In the figure, $s_{i,j}$ is the signature in the j-th input of transaction Tx_i, while $\mathsf{pk}_{i,j}$ and $v_{i,j}$ are the public key and the value of its j-th output. Inputs are shown in purple, spent outputs in red and spendable outputs in green. Each output specifies its value and the public key of its owner. Each input references a previous output (the one it spends) and contains a signature, whose message is the transaction containing the input and whose public key is the one specified by the spent output. Notice that an output is spent if and only if there is an input referencing it (i.e., if and only if an arrow points to it), and that no output can be spent more than once.

the UTXO system used by 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 tx specifies a set of inputs and a set of outputs, corresponding roughly to the amounts paid and the amounts received, respectively: each input $\mathsf{tx.in}_i$ refers to an output of a previous transaction tx'_i (this is the currency about to be spent), whereas each output $\mathsf{tx.out}_j$ is described by a non-negative amount of currency $v_j \geq 0$ and the public key pk_j of the corresponding recipient (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 tx has n inputs and m outputs, and that its i-th input points to an output of a previous transaction tx'_i with value v'_i and public key pk'_i. To be valid, tx must satisfy the following properties:

- It must be balanced: the sum of the values of the inputs must equal 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., $\mathsf{pk}'_1, \ldots, \mathsf{pk}'_n$.
- None of its inputs may have been spent by a preexisting transaction on the same branch of the chain (or by another transaction in the block 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 be confusing at first. Suppose that there is an unspent output of 100א under Alice's public key, but Alice only wants to spend 20א. How can she proceed? If she issues a transaction pointing to the 100א output, she will spend all 100א!

Well, there is an easy way out: nothing prevents Alice from being 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's public key and 80א under her own public key. Essentially, the transaction gives 20א to the actual recipient and 80א of change back to Alice. The 100א output becomes spent, but the new 80א output is spendable.

Exercise 21. *In the UTXO system, every transaction consists of a list of inputs and outputs, where the inputs are outputs of previous transactions. One needs to pay attention to how this is implemented in practice: if each transaction specified its inputs by including the whole transactions in which they were generated, transaction sizes would grow as the cryptocurrency changes hands. Specifically, if* tx' *spends an output created by* tx*, it would be a bad idea to include* tx *as part of* tx'*. Indeed, think about what would happen when we spend an output of* tx'*: the new transaction* tx'' *would need to contain* tx'*, which contains* tx*. This would force the size of the transactions to grow by a constant amount at each step. Essentially, the size of a transaction would grow with the length of the "history" it carries. Try to find a solution to this problem using a cryptographic hash function (we want transactions to have constant size).*

Exercise 22. *In the UTXO system, every transaction is signed under the public keys of the payers. However, what exactly is 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 now analyse how the UTXO model is implemented in Bitcoin. Bitcoin does not actually offer a completely faithful implementation of UTXO, but rather a more flexible version. In Bitcoin, each transaction tx is identified by its cryptographic hash $\mathsf{Hash}(\mathsf{tx})$ (why can we assume that the hash uniquely identifies the transaction?) and consists of a list of inputs and outputs (plus additional metadata). Each input in_i consists of the following:

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

Each output out_j consists of the following:

- A non-negative amount of cryptocurrency $v_j \geq 0$, expressed in satoshis (1 Bitcoin = 10^8 satoshis).
- A script script_j, which specifies the conditions that must be satisfied to spend this output in the future. Typically, it simply states that, to spend this output, the new transaction must be signed under the public key of the owner of this output.

To verify a transaction tx, validators consider all its inputs in_i and fetch the previous transactions tx'_i they point to. From each of these, they retrieve 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 yet been spent by any transaction on the same branch of the chain (not even by other transactions in the same block).
- They run the script script'_i of each of these unspent outputs $\mathsf{tx}'_i.\mathsf{out}_{k_i}$ on the witness w_i of $\mathsf{tx}.\mathsf{in}_i$, and check that all of them output true.
- They check that the sum of the values v'_i of the outputs $\mathsf{tx}'_i.\mathsf{out}_{k_i}$ is at least the sum of the values v_j of the new outputs $\mathsf{tx}.\mathsf{out}_j$ (i.e., they check that 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 is unbalanced: the difference between 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 miner's service. Since no money is created or destroyed, the transaction is still balanced. Essentially, fees incentivise the validators to do their work. We discuss this aspect in detail in Section 4.4.

What can the script check? The script specified in each output describes the conditions that must be satisfied by the transaction that spends the received cryptocurrency. For a faithful implementation of UTXO, this can simply be a signature check under the public key of the owner. However, Bitcoin allows more general conditions to be specified. These are expressed in a stack-based programming language supporting a wide range of operations:

- arithmetic operations (addition, multiplication, absolute value, ...)
- logical operations (equality, less than, AND, OR, NOT, ...)
- stack operations (drop, swap, duplicate, ...)
- flow control (if and else, ...)
- cryptographic operations (hashing, signature verification, ...)

For instance, a user can generate a transaction with an output that can be spent only with signatures from at least t distinct parties in a specified committee. Alternatively, one can generate a transaction with an output that can be spent only by somebody who provides the solution to some hard mathematical problem (in this case, the witness is the solution, and the script simply verifies that the solution is correct).

There are, however, limits to what this programming language can express. For instance, it 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 of the spending transaction or about the way this transaction is added to the chain (e.g., it cannot set conditions on the values of the outputs or of the other inputs, on the identities of the recipients, or on 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.

So, why hasn't Bitcoin modified its code over all these years? Well, Bitcoin is a decentralised system, 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 upon by the community as a whole (otherwise, the system risks forking), but, for some reason, a good fraction of the community is rather averse to change. As a result, the community is rather paralysed. On the other hand, even though it has no central authority, Ethereum benefits from active and influential figures in its community, e.g., its founder Vitalik Buterin and the Ethereum Foundation. Over the years, these figures have managed to bring about a lot of change in Ethereum (for instance, they moved 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 of not being fully decentralised.

4.2 The Account-Based System

In account-based systems, the blockchain stores not only the transactions but also a database recording the amount of cryptocurrency held by every account and the number of transactions issued by each of them. Each account is identified by a signature public key pk, and its address is usually derived from 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 up to this point, including this one (this is usually called the *nonce*).
- A signature s on the transaction under the sender's public key.

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

- It retrieves the amount of cryptocurrency w in the sender's account and the number m of transactions the sender has issued so far.
- It checks that s verifies under the sender's 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 in 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 in the recipient's account is increased by v. The block that the validator will try to propose consists of the updated state together with all the transactions that justify these changes.

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

UTXO vs Account-Based: Advantages and Disadvantages. Both the UTXO and the account-based designs have advantages and disadvantages. The account-based model, for instance, provides a conceptually simple design with efficient storage (storing the balances of all accounts is easier than storing the list of all transactions). On the other hand, UTXO is better suited to parallelism because, unlike in the account-based system, 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 later in the course, the UTXO model can allow a higher degree of anonymity.

4.3 Smart Contracts

If we compare Bitcoin's instantiation of UTXO with the account-based model, the former seems to offer more flexibility in how outputs can be spent. Why, then, did we say that Ethereum, which is an account-based system, is actually much more expressive than Bitcoin?

The reason is that Ethereum allows the creation of special accounts called *smart contracts*. Like all other accounts, smart contracts hold an amount of cryptocurrency, and the system keeps track of the number of transactions they have issued. Unlike 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); instead, they are linked to some program code and to a contract state that describes their behaviour. All the information about the contract (code, contract state, amount of cryptocurrency owned and number of transactions issued) is stored on the chain at all times and continuously updated.

Any account in the system (either one owned by a user or another smart contract) can trigger the execution of a specific smart contract by issuing a transaction with the contract's address as the recipient. Such transactions usually include an additional *data field*, which specifies which methods of the smart contract are called and the inputs to pass to them. Validators that receive valid transactions of this kind react by transferring the cryptocurrency being spent to the contract's account, and then evaluating the smart contract's 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 call other smart contracts (in this case, the transaction is called a *message*). Notice that transactions issued by smart contracts do not need to be signed. First of all, smart contracts are not even associated with signature keys. Secondly, signatures are not needed: their purpose is to guarantee that a particular transaction was approved by the sender's account, whereas here everybody can verify that executing the smart contract on the given inputs requires it to send that particular transaction. To summarise, a smart contract is a machine with a public state that everybody can interact with, 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 give an example of a smart contract implementing an auction. It can be equipped with the following methods:

- An initialisation method. This takes as input the auction deadline (expressed, e.g., as the block height after which no offer is considered) and the minimum bid (which must be non-negative). The transaction calling this method transfers ownership of the prize to the contract (for instance, the prize can be a non-fungible token). The contract reacts by adding to its state a new object storing a unique identifier 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 L of refundable bids (initially empty).
- An offer method. This takes as input the unique identifier id of the auction. The value of the offer is the amount of cryptocurrency paid in the transaction. If this is not strictly greater than the highest bid, is lower than the minimum bid, or the auction deadline has passed, the contract appends to L the pair consisting of the received amount and the caller's address. Otherwise, the contract updates the address of the highest bidder and the value of the highest bid, and appends to L the pair consisting of the previous highest bid and the address of the previous highest bidder.
- A refund method. This takes as input the unique identifier id of the auction. The contract checks whether L contains an entry associated with the caller's address. If so, the contract removes this entry from L and transfers the corresponding amount of cryptocurrency back to the caller.
- A winner method. This takes as input the unique identifier id of the auction. The contract checks that the auction deadline 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 takes as input the unique identifier id of the auction. The contract checks that the auction deadline has passed, that the caller is the seller, and that the highest bid has not been sent to the seller yet. Then, it transfers the highest bid to the seller.

So, what is the advantage of implementing an auction using a smart contract rather than in the traditional way? There are two main advantages. First of all, with our solution, the participants do not need to rely on

a trustee to ensure the honest execution of the auction; they only need to trust the security of the blockchain (in proof of work, this means trusting that the majority of the computational power is in the hands of participants who follow the protocol). This is a fundamental security advantage! What if the trustee is not as trustworthy as we think and runs away with the money? Or what if the trustee rigs the auction, favouring one participant over another? Smart contracts allow us to avoid these problems by decentralising trust. We no longer need to put all our eggs in one basket!

The second advantage is related to costs: since there is no trustee, we do not need to pay one. Now, as we will discuss shortly, smart contracts charge a service fee that covers the cost of the electricity and hardware needed to run the auction; however, this is likely to be much lower than the cost of hiring a trustee. Moreover, even if a particular auction smart contract charged very high fees, nothing would prevent anybody from creating another auction smart contract with more reasonable fees. Since nobody can hold a monopoly on auctions, prices are likely to drop significantly, to match the real costs of the service (i.e., the cost of electricity, hardware, cooling systems, etc.). Overall, we get a cheaper solution.

How to Create a Smart Contract? Blockchains typically allow anybody to create a new smart contract. This is certainly the case for the most important blockchains supporting this technology, first among them Ethereum. For instance, to create a smart contract in Ethereum, one just needs to create a transaction whose recipient field is empty. The code of the smart contract and its initial state are included in the data field of the transaction. Moreover, the amount of cryptocurrency transferred by this transaction is credited directly to the newly created smart contract. The address of the smart contract is $\mathsf{Hash}(\mathsf{addr}, n)$, where addr is the address of the creator and n is the value of the creator's nonce at the time of creation. Observe that smart contracts can also be created by other smart contracts.

4.4 Transaction Fees and Rewards

Computation is not free: it requires specialised hardware, it uses electricity, it requires complex cooling systems, etc. We cannot expect a blockchain to run at no cost, especially with proof-of-work consensus, where validators continuously use all their computational power to solve complex puzzles. This is why every blockchain has its own fee and reward mechanisms, which establish how users pay validators for their service and incentivise their work. These mechanisms are particularly sophisticated in blockchains that support 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 calling for the smart contract computation is therefore charged proportionally to the cost of these operations. Next, we see how the fee mechanisms of Bitcoin and Ethereum work.

Fees and Rewards in Bitcoin. As discussed in Section 4.1.1, in Bitcoin transactions, the sum of (the values of) the inputs is typically strictly greater than the sum of (the values of) the outputs. The difference between these amounts is understood to be a transaction fee, which will be claimed by the validator that mines the block including the transaction. Users are free to choose the amount of cryptocurrency they offer as a fee; however, the higher the fee, the sooner 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 in the block, and it is the only Bitcoin transaction that is not balanced. The coinbase transaction has no inputs and can have multiple outputs. For it to be valid, the sum of (the values of) its outputs must equal the sum of all transaction fees in the block plus a fixed reward determined by an algorithm. (If the coinbase transaction is not valid, the validators 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 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), and the reward halves every 210000 blocks (roughly every 4 years). As of 2026, the reward is 3.125 Bitcoins ($\approx$ 270k dollars

at 2026 prices). The next halving is expected in April 2028. The average transaction fee in 2026 is instead 2 satoshis (approximately 0.40$).

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

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

This is why Bitcoin is often referred to as digital gold (the amount of gold on Earth is also finite). 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 trying to, given that Satoshi Nakamoto disappeared from the scene around 2010). The reward was higher in the first blocks because, at the time, the system was not well established and the value of Bitcoin was much lower (in 2010, somebody bought 2 pizzas for 10000 Bitcoins, which corresponds to $\approx$ 860 million dollars at 2026 prices), so higher rewards were needed to incentivise the work of the validators. The hope of Bitcoin's designer was that, over time, the system would become more and more established, to the point that transaction fees alone would suffice to encourage validators to do their work. What Satoshi Nakamoto probably did not expect is that the value of Bitcoin would grow 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 at 2026 prices). This is roughly 5% of all Bitcoins that will ever be mined (for comparison, the richest person in the world owns 0.22% of the world's total wealth as of 2026, and we still live in an extremely unequal world). This made Satoshi extremely rich, but it also undermined the functioning of their creation and any of its claims of being "decentralised" (the economic power in the system is very centralised). Satoshi is probably aware of this, given that none of Satoshi's known accounts have moved any money since their disappearance from the scene.

So far, however, we have not explained why the total amount of Bitcoin that will ever be minted is capped. This probably comes from the love of the cypherpunk movement (which influenced Satoshi Nakamoto's thinking) for the gold standard, following the controversial ideas of the Austrian School. The gold standard has some fundamental disadvantages, such as an inherent tendency towards deflation and the lack of recovery mechanisms after economic crises, e.g., quantitative easing (though supporters of the Austrian School would argue that this is a positive feature).

Fees in Ethereum. In addition to the fields we have already presented (sender, amount, recipient, nonce, signature, data), each Ethereum transaction also includes 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 smart contract operations. Every smart contract execution can be split into a sequence of basic operations (e.g., additions, multiplications, Boolean operations, etc.), each associated with a fixed amount of gas (e.g., an addition costs 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 deducts the corresponding amount of gas from the budget. If the budget reaches 0 before the smart contract execution terminates, the validator stops the execution and the smart contract is reverted to its state before the call. Each Ethereum transaction also has a baseline cost of 21000 gas, which de facto lower-bounds the amount of gas used to process it (for instance, standard transactions cost 21000 gas).

The fees are calculated on the basis of the amount of gas used during the execution (this also holds for executions that fail because the gas budget reached 0, i.e., failed executions are not free). The amount of

cryptocurrency deducted from the caller's account is

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

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 also needs to make sure that the caller owns an amount of cryptocurrency at least equal to the transferred amount plus the product of *start gas* and *max gas price* (otherwise, the transaction is not considered valid).

Now, unlike in Bitcoin, the validator receives only 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 be claimed by the validator. This amount is equal to

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

where $\mathsf{priorityfee} = \mathsf{gasprice} - \mathsf{basefee}$ must be at most the *max priority fee* specified in the transaction by the caller. Clearly, the higher the priority fee offered by the caller, the sooner validators process the transaction. In addition to transaction fees, Ethereum validators also receive a per-block reward, similarly to Bitcoin. This reward is, however, calculated in a more complex way, so we will discuss it later in the course.

We still need to explain how the base fee is computed. It is derived from the amount of gas used by the transactions in the previous block and from that block's base fee. Specifically, an Ethereum block has a limit on the amount of gas used by the transactions it contains, which 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 (half of the block gas limit), and adjust the base fee accordingly: if the total gas used is above target, the fee is increased proportionally; if it is below target, the fee is decreased proportionally. The exact base fee is given by the following formula:

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

Notice that, since totalgasused is capped at $2 \cdot \mathsf{target}$, the base fee can increase (or decrease) by at most 12.5% per 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$.

Let us now discuss the reasons behind this particular design. First of all, decoupling gas from fees is fundamental to account for price fluctuations. Gas allows us to measure the complexity of a smart contract computation in absolute terms: no matter when the program is run, its gas cost is always the same. This makes sense, given that the costs of basic operations remain roughly constant relative to each other (e.g., the ratio between the cost of a multiplication and that of an addition remains roughly constant). Instead, the prices of electricity, land and the other resources needed to keep the system running fluctuate all the time, depending on the geopolitical situation, technological progress and the mood of the markets. The Ethereum fee mechanism offers, on the one hand, an absolute and fixed way to measure the complexity of a computation and, on the other hand, a way to adapt the price per unit of computation to the market situation.

Another fundamental aspect of the Ethereum fee mechanism is that the base fee is burnt instead of being given to the validator. This makes sure that nobody profits from artificially raising gas demand. If all transaction fees were transferred to the validator, validators would have an incentive to fill their blocks with dummy transactions that drive the gas price up to levels that no longer reflect real-world expenses (electricity, land, etc.). In such a system, dummy transactions would come at (almost) no cost to validators,

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 of Ethereum transactions, showing which fields are empty depending on the transaction type.

as they would receive back all the transaction fees they pay for them. On the other hand, this would increase the costs for normal users, driving up the profits of the validators. The goal of the base fee is to estimate the real-world costs of the system. By burning this amount, the system ensures that nobody can speculate on it. The work of the validators is instead incentivised through priority fees and rewards. 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 fee is adjusted over time. The idea is that the system wants to keep the amount of gas used in each block roughly equal to the target value. If the most recent block is above target, there is a lot of demand for computation (probably because it is cheap), so its price should be increased proportionally. On the other hand, if the amount of gas used in the block is below target, the cost of computation must be too high, so the price needs to be lowered.

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

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

Notice that, when the demand for a particular good goes up, its price goes up too. When the supply increases, prices instead go down. Try to devise a way for Bob to extract value from Alice's purchase by relying on these market dynamics and on 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 find out 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 counter, as it artificially drives up costs. This is called front running.

4.5 Coins and Tokens

Every blockchain that we have discussed so far has its own cryptocurrency, for instance, bitcoins in Bitcoin and ether in Ethereum. This is called the *native coin* of the blockchain, and it is managed by the protocol itself: the validators check that every transfer follows the rules of the UTXO or of the account-based system (see Sections 4.1 and 4.2), new coins are only created according to the rules of the protocol, typically as rewards for the validators (see Section 4.4), and transaction fees are paid in the native coin. These rules, for instance, the maximum number of coins that will ever exist, can only change if the validators adopt a new version of the protocol.

A *token*, instead, is a digital asset created on top of an existing blockchain by a smart contract (see Section 4.3). The state of the contract contains a table that records how many tokens each address owns, and the contract offers methods to transfer them. For instance, to send tokens to Bob, Alice calls a method of the contract, which checks that Alice owns enough tokens and updates the table. The validators do not know that the contract implements a token: they simply run its code, as for any other smart contract, and Alice pays the transaction fee in the native coin. Since anybody can create a smart contract, anybody can create a new token. Its rules, for instance, who can mint new tokens and whether their total number is capped, are decided by the code of the contract, that is, by whoever wrote it.

Token standards. To allow wallets, exchanges and other smart contracts to handle all tokens in the same way, token contracts follow standard interfaces. On Ethereum, the most common standard for *fungible* tokens, whose units are all interchangeable (like coins or banknotes), is ERC-20. An ERC-20 contract offers, among others, methods to read the number of tokens owned by an address (balanceOf), to transfer tokens from the caller to another address (transfer), and to authorise another address to transfer up to a given number of the caller's tokens (approve), which the authorised address can then do with transferFrom. Authorisations are needed because a smart contract that receives tokens through transfer is not notified: only the token contract runs, so the recipient cannot react to the payment. To pay a smart contract in tokens, for instance, to bid in an auction like the one of Section 4.3 with tokens instead of the native coin, a user first calls approve to authorise the auction contract, and then calls the auction contract, which collects the tokens with transferFrom. Another standard, ERC-721, describes *non-fungible tokens* (NFTs): every token has a unique identifier, and the contract records the owner of each identifier. NFTs are used to represent unique items, such as digital artworks, tickets or domain names. Notice that ether itself is not an ERC-20 token, since it is managed by the protocol and not by a contract. For this reason, many applications use *wrapped ether*, an ERC-20 token issued by a smart contract that holds the same amount of ether and gives it back to whoever returns the tokens.

Native tokens. Some blockchains support tokens in the protocol itself, without smart contracts. For instance, in Cardano, which uses a variant of the UTXO system, every transaction output can hold, besides an amount of the native coin, amounts of other tokens, and the validators check that every transaction is balanced for every type of token. New tokens of a given type can only be created (or destroyed) according to rules fixed when the type is defined. This is how a blockchain can record multiple currencies, as we mentioned at the beginning of this chapter.

What is a token worth? A native coin is not a claim on anything: its value depends on what people are willing to pay for it, which partly comes from the need to pay transaction fees (and, in proof-of-stake blockchains, to become a validator). Many tokens, instead, represent a promise made by their issuer, and the blockchain only guarantees that the records of the token cannot be tampered with, not that the promise will be kept. For instance, since Ethereum contracts cannot hold bitcoins, most wrapped bitcoins on Ethereum

are backed by bitcoins held by a custodian, and they are only worth as much as the custodian's promise. Similarly, the blockchain only records who owns an NFT, and the link between the token and a physical good (such as the house in the example at the beginning of this chapter) or a file stored elsewhere only holds if the people involved, or the law, recognise it. Finally, other tokens give their owners the right to vote on changes to a protocol (these are called *governance tokens*), while many tokens, such as the so-called *meme coins*, are not backed by anything and are not needed to use any service.

Stablecoins. The value of cryptocurrencies is prone to huge fluctuations: for instance, bitcoin has repeatedly lost more than half of its value within a few months. This makes them risky to hold and hard to use as money. A shop that accepts payments in a cryptocurrency does not know how much its revenue will be worth when it pays its suppliers and its employees, and a loan in a cryptocurrency may cost the borrower far more, or far less, than expected. To protect themselves, people avoid such transactions, convert their coins as soon as possible, or pay for some form of insurance against price changes: all of this makes the economy riskier and less efficient. *Stablecoins* (which, despite their name, are tokens) try to solve this problem: they are designed to always be worth one unit of a traditional currency, typically one US dollar, while living on a blockchain, so that they can be transferred and used in smart contracts like any other token.

The most common way to keep this promise is to back every token with one dollar held by the issuer. The largest stablecoins, USDT and USDC, are issued by two companies, Tether and Circle, which create new tokens when customers deposit dollars and destroy them when customers redeem them. These stablecoins are inherently centralised: dollars live in the traditional banking system, outside the blockchain, so somebody has to hold them, and the holders of the tokens must trust the issuer to keep enough reserves, to invest them safely and to honour redemptions. The issuer also controls the token contract, which allows it to freeze the tokens of any address, for instance, at the request of the authorities. To avoid trusted issuers, some stablecoins, such as DAI, are backed by cryptocurrencies locked in smart contracts and worth more than the stablecoins they back, so that the backing survives moderate price drops. Others, called *algorithmic stablecoins*, have no real reserves and try to keep their price stable through automatic rules that create and destroy tokens.

In all these designs, the one-to-one promise can break. When holders start doubting that a stablecoin is worth one dollar, they all try to sell it at once, which pushes its price further down, as in a run on a bank. In May 2022, the algorithmic stablecoin TerraUSD, at the time one of the largest stablecoins, lost its peg and, within a few days, almost all of its value, together with LUNA, the token used to stabilise it: tens of billions of dollars were lost. In March 2023, USDC fell below 90 cents when Silicon Valley Bank, which held part of its reserves, failed, and it recovered only after the US authorities guaranteed all the deposits of the bank. DAI lost its peg at the same time, because a large part of its collateral consisted of USDC: even stablecoins designed to be decentralised may end up depending on centralised ones. Finally, in 2021, the US Commodity Futures Trading Commission fined Tether for claiming that USDT was fully backed by dollars, while, in a period of more than two years, its reserves had been sufficient on only about a quarter of the days.

Chapter 5

Proof of Stake

In this chapter, our goal is to build a permissionless consensus protocol that does not require the participants to invest a lot of computational power to guarantee its security and correct functioning. This, however, raises an issue: how do we prevent Sybil attacks? Being permissionless prevents us from relying on an honest majority of participants, because an attacker could always introduce a large number of virtual parties under their control. In proof-of-work consensus, we relied on computational power to prevent this attack: if influence over the protocol is assigned proportionally to the computational power of the various actors, the Sybil attack fails because introducing virtual parties does not increase the computational power of the adversary. Unfortunately, this approach requires all participants to continuously perform a lot of computation just to prove that they indeed possess enough computational power to legitimise their influence. If we want to avoid this, we need another mechanism to prevent Sybil attacks.

Proof of stake (PoS) proposes to grant influence proportionally to the amount of capital owned by the various actors: this is the so-called *stake*. Specifically, in these protocols, all participants are identified by their signature verification keys and are linked to a stake deposit on the chain, which is visible to everybody. The protocol relies on a leader election mechanism in which each validator's probability of winning is proportional to the fraction of all stake in the system held in their deposit. Each leader can propose a new block. Blocks are arranged in a graph, as in proof-of-work protocols, and security and liveness are enforced via the *longest chain principle*. Essentially, Sybil attacks are unsuccessful because introducing virtual parties does not increase the amount of stake an attacker controls.

5.1 A New Election Mechanism

We now describe an election mechanism that can be used in proof of stake. Combined with the longest chain principle, it will give a secure consensus protocol. Let us first point out some of the challenges we need to overcome.

First of all, no party should be able to increase their chance of winning by performing more computation. Otherwise, the validators would be incentivised to waste a lot of energy to maximise their likelihood of becoming the next leader. In particular, the winning probability should be independent of the content of the block the validator wants to propose; otherwise, the validators could start a brute-force search for content that makes them win. Secondly, it is fundamental that nobody can predict who the next leaders are. If this were not the case, an attacker wishing to maximise their influence over the protocol could try to hack into the next winner's machine (this is called *adaptive corruption*). If the attacker does not know the identity of the next leader, this strategy fails: the attacker is forced either to concentrate their efforts on somebody picked at random, or to spread them over a much larger attack surface. On the other hand, the winner must be able to detect that they won, and must also be able to prove to everybody else that they are allowed to produce a new block.

5.2 Verifiable Random Functions

To solve these issues, we need to introduce a new primitive called a *verifiable random function* (VRF). This is essentially a pseudorandom function whose outputs can be verified to be correct using a verification key.

Definition 30 (Verifiable Random Function)**.** *A verifiable random function with output in $\mathbb{Z}_{2^k}$ consists of a triple of efficient algorithms* $(\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Verify})$ *with the following syntax:*

KeyGen : *This algorithm is probabilistic, takes no input and outputs a pair (K, vk), where K is a secret key and* vk *is a verification key.*

Eval**:** *This algorithm is deterministic and takes as input a secret key K and a binary string c. It outputs an integer $r \in \mathbb{Z}_{2^k}$ and a proof π.*[1]

Verify**:** *This algorithm is deterministic and takes as input a verification key* vk*, a binary string c, an integer $r \in \mathbb{Z}_{2^k}$, and a proof π. It outputs a bit: 0 stands for 'reject'; 1 stands for 'accept'.*

Additionally, we require that, for every value of c,

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, c, r, \pi) = 1 \middle| \begin{array}{l}(K, \mathsf{vk}) \xleftarrow{\$} \mathsf{KeyGen} \\ (r, \pi) \leftarrow \mathsf{Eval}(K, c)\end{array}\right] = 1.$$

VRFs need to satisfy additional security properties. The first one states that, as for a PRF, the output r of the VRF looks random.

Definition 31 (Pseudorandomness)**.** *We say that* $\mathsf{VRF} = (\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Verify})$ *is (T, ϵ)-pseudorandom if, for every pair of algorithms $(\mathcal{A}_0, \mathcal{A}_1)$ whose combined running time is at most T, it holds that*

$$\left|\Pr\left[\begin{array}{c}\mathcal{A}_1(r, \psi) = 1 \\ c \notin \mathcal{Q}\end{array}\middle| \begin{array}{l}(K, \mathsf{vk}) \xleftarrow{\$} \mathsf{KeyGen} \\ (c, \psi) \xleftarrow{\$} \mathcal{A}_0^{\mathsf{Eval}(K,\cdot)}(\mathsf{vk}) \\ (r, \pi) \leftarrow \mathsf{Eval}(K, c)\end{array}\right] - \Pr\left[\begin{array}{c}\mathcal{A}_1(r, \psi) = 1 \\ c \notin \mathcal{Q}\end{array}\middle| \begin{array}{l}(K, \mathsf{vk}) \xleftarrow{\$} \mathsf{KeyGen} \\ (c, \psi) \xleftarrow{\$} \mathcal{A}_0^{\mathsf{Eval}(K,\cdot)}(\mathsf{vk}) \\ r \xleftarrow{\$} \mathbb{Z}_{2^k}\end{array}\right]\right| \leq \epsilon,$$

where $\mathcal{Q}$ denotes the set of queries that $\mathcal{A}_0$ issues to the oracle $\mathsf{Eval}(K, \cdot)$.

Let us discuss the above definition, as it may look confusing at first. We are asking that, even with the help of $\mathcal{A}_0$, the attacker $\mathcal{A}_1$ cannot distinguish the value r produced by $\mathsf{Eval}(K, c)$ from a truly random value $r \in \mathbb{Z}_{2^k}$, unless $\mathcal{A}_0$ has queried the oracle $\mathsf{Eval}(K, \cdot)$ on c. We can interpret the output 1 of $\mathcal{A}_1$ as saying "yes, I believe that the r I have received comes from $\mathsf{Eval}(K, c)$". Then, the probability on the left (where $\mathcal{A}_1$ indeed receives an r generated by $\mathsf{Eval}(K, c)$) is the probability of a true positive, while the probability on the right (where $\mathcal{A}_1$ receives a random r) is the probability of a false positive. By requiring these two probabilities to be at most ϵ apart, we are saying that $\mathcal{A}_1$ is not really good at distinguishing an r coming from the VRF from a random one. Notice that $\mathcal{A}_1$ is helped in its distinguishing task by $\mathcal{A}_0$, which can encode in ψ all the information it gathered about vk and by querying the oracle $\mathsf{Eval}(K, \cdot)$, as well as all information about its choice of c. This means that there exist choices of $\mathcal{A}_0$ and $\mathcal{A}_1$ such that $\mathcal{A}_1$ knows vk, c, and all the queries that $\mathcal{A}_0$ made to $\mathsf{Eval}(K, \cdot)$, together with the corresponding answers. However, two things would make $\mathcal{A}_1$'s task trivial. The first is if $\mathcal{A}_0$ queried the oracle on c: by sharing the answer with $\mathcal{A}_1$, the attackers could immediately tell whether r is truly random. The second is if $\mathcal{A}_1$ also received the proof π produced together with r: in that case, $\mathcal{A}_1$ could simply check whether $\mathsf{Verify}(\mathsf{vk}, c, r, \pi) = 1$. For these reasons, we do not provide π to $\mathcal{A}_1$, and we only consider the probabilities of true and false positives when $c \notin \mathcal{Q}$.

The second property that a VRF needs to satisfy is called soundness. It essentially states that no attacker can generate a valid VRF proof for a wrong value.

[1] We recall that $\mathbb{Z}_{2^k}$ denotes the set $\{0, 1, \ldots, 2^k - 1\}$.

Definition 32 (Soundness)**.** *We say that* $\mathsf{VRF} = (\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Verify})$ *is* (T, ϵ)*-sound if, for every algorithm* $\mathcal{A}$ *running in time at most* T*, it holds that*

$$\Pr\left[\begin{array}{c|l} \mathsf{Verify}(\mathsf{vk}, c, r, \pi) = 1 & (K, \mathsf{vk}) \xleftarrow{\$} \mathsf{KeyGen} \\ r \neq r' & (c, r, \pi) \xleftarrow{\$} \mathcal{A}(K, \mathsf{vk}) \\ & (r', \pi') \leftarrow \mathsf{Eval}(K, c) \end{array}\right] \leq \epsilon.$$

The last property that a VRF needs to satisfy is called unbiasability. It essentially states that, even if we let the adversary choose multiple VRF keys (potentially in a malicious way), their evaluations on random inputs look random.

Before stating it, we need to introduce the definition of a monotone statistic.

Definition 33 (Monotone statistic)**.** *Let* $p : (\mathbb{Z}_{2^k} \cup \{\perp\})^* \to \{0,1\}$ *be a probabilistic function. We say that* p *is a* monotone statistic *if, for every* $y \in (\mathbb{Z}_{2^k} \cup \{\perp\})^*$ *and every* $j \in [|y|]$*, we have*

$$\Pr[p(y) = 1] \geq \Pr[p(y') = 1],$$

where y' *is the string obtained by replacing the* j*-th entry of* y *with* $\perp$*.*

Essentially, the above definition says that replacing any entry of the input of p with $\perp$ can only decrease the probability that p outputs 1.

Definition 34 (Unbiasability)**.** *Let* $p : (\mathbb{Z}_{2^k} \cup \{\perp\})^* \to \{0,1\}$ *be a monotone statistic. We say that* $\mathsf{VRF} = (\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Verify})$ *is* $(T, \lambda, \epsilon, p)$*-unbiasable if, for every pair of algorithms* $(\mathcal{A}_0, \mathcal{A}_1)$ *whose combined running time is at most* T*, it holds that*

$$\mathsf{Exp}_0^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1) \leq \mathsf{Exp}_1^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1) + \epsilon,$$

where

$$\mathsf{Exp}_0^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1) := \Pr\left[\begin{array}{c|l} \forall j \neq j' : \mathsf{vk}_j \neq \mathsf{vk}_{j'} & (\mathsf{vk}_1, \ldots, \mathsf{vk}_n, m, \psi) \xleftarrow{\$} \mathcal{A}_0 \\ p((r_{i,j})_{i \in [m], j \in [n]}) = 1 & c_1, \ldots, c_m \xleftarrow{\$} \{0,1\}^\lambda \\ & (r_{i,j}, \pi_{i,j})_{i \in [m], j \in [n]} \xleftarrow{\$} \mathcal{A}_1(c_1, \ldots, c_m, \psi) \\ & \textit{if } \mathsf{Verify}(\mathsf{vk}_j, c_i, r_{i,j}, \pi_{i,j}) = 0 : r_{i,j} \leftarrow \perp \end{array}\right],$$

$$\mathsf{Exp}_1^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1) := \Pr\left[\begin{array}{c|l} \forall j \neq j' : \mathsf{vk}_j \neq \mathsf{vk}_{j'} & (\mathsf{vk}_1, \ldots, \mathsf{vk}_n, m, \psi) \xleftarrow{\$} \mathcal{A}_0 \\ p((r_{i,j})_{i \in [m], j \in [n]}) = 1 & r_{i,j} \xleftarrow{\$} \mathbb{Z}_{2^k} \end{array}\right].$$

Let us discuss the above definition. As in the pseudorandomness property, we consider the probabilities of two different events, $\mathsf{Exp}_0^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1)$ and $\mathsf{Exp}_1^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1)$, and we require that the first exceeds the second by at most a small amount ϵ, regardless of what $\mathcal{A}_0$ and $\mathcal{A}_1$ do. View p as a statistical test: for instance, it can compute the average or the variance of the values it receives as input and output 1 if this is within a certain interval. In general, it can perform an arbitrary computation. We can interpret p as follows: when its output is 1, the inputs have passed the test; when its output is 0, the inputs have failed the test.

Let us look at the event described by $\mathsf{Exp}_0^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1)$. This is associated with an experiment in which $\mathcal{A}_0$ provides us with an arbitrary number of VRF public keys $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$, which may be generated in arbitrary ways, an integer m, and an internal state ψ, which is passed on to $\mathcal{A}_1$. Next, the experiment samples m random strings $c_1, \ldots, c_m$ from $\{0,1\}^\lambda$ and gives them, along with ψ, to $\mathcal{A}_1$. Notice that ψ may encode the secret counterparts of any of $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$. The goal of $\mathcal{A}_1$ is to provide us with an evaluation $r_{i,j}$ of each VRF key vk_j on each random string c_i, together with a proof of correctness $\pi_{i,j}$. These can be computed in arbitrary ways, so, as a next step, the experiment checks the proofs: if the verification of some $\pi_{i,j}$ fails,

the corresponding value $r_{i,j}$ is set to the special symbol $\perp$; otherwise, $r_{i,j}$ is kept unchanged. The term $\mathsf{Exp}_0^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1)$ is the probability that $\mathcal{A}_0$ outputs n distinct VRF keys and that the test p succeeds on $(r_{i,j})_{i,j}$.

Next, let us look at the event associated with $\mathsf{Exp}_1^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1)$. The experiment initially proceeds as before: $\mathcal{A}_0$ provides us with VRF public keys $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$. However, the elements $r_{i,j}$ are now sampled uniformly at random from $\mathbb{Z}_{2^k}$. Once again, the term $\mathsf{Exp}_1^{\mathsf{bias}}(\mathcal{A}_0, \mathcal{A}_1)$ is the probability that $\mathcal{A}_0$ outputs n distinct VRF keys and that the test p succeeds on $(r_{i,j})_{i,j}$.

To summarise, the above definition states that, even if the VRF keys are generated maliciously, their evaluations on random inputs have the same statistical properties as random strings. Notice, however, that this security property crucially relies on the fact that the inputs $c_1, \ldots, c_m$ cannot be predicted at the time $\mathcal{A}_0$ chooses the VRF keys. If $\mathcal{A}_0$ could predict the inputs before they are revealed, it could select $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$ so that the evaluations do not have the statistical properties of random strings. For instance, it could select $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$ so that the evaluations are lower than average.

VRFs in Practice. We will not see how VRFs can be built. However, what is important to know is that the constructions currently in use are (T, ϵ)-pseudorandom, (T, ϵ)-sound and $(T, \lambda, \epsilon, p)$-unbiasable for extreme values of the parameters T and ϵ. In particular, ϵ, which measures the probability that the attackers $(\mathcal{A}_0, \mathcal{A}_1)$ succeed, is typically smaller than 2^{-128}, a value smaller than the probability of being hit by a meteorite in the next second. The value of T is instead extremely high, typically 2^{128}. This means that, for the attackers $(\mathcal{A}_0, \mathcal{A}_1)$ to succeed with probability greater than ϵ, they would need to run their machines for several geological eras. The parameter λ is usually 128 or 256, while p can be any monotone statistic computable within a certain time T'. Typically, T' is also huge, e.g., 2^{128}.

5.3 Updatable Signatures

In our proof-of-stake protocol, validators will sign the blocks they propose. Recall, however, that an attacker may try to corrupt validators adaptively, for instance, by hacking into their machines. Once the attacker controls a validator's machine, they learn the validator's secret signing key and can sign anything on their behalf, including alternative versions of the blocks that the validator proposed in the past. Ideally, each validator would use a different secret key in every slot and erase it as soon as they no longer need it: an attacker who corrupts a validator would then only learn the keys for the current and future slots, which are useless for signing blocks of past slots. However, generating a fresh key pair for every slot would be impractical, as validators would need to announce a new public key for every slot. *Updatable signatures* solve this problem: the signer has a single public key, but their secret key evolves over time.

More precisely, time is split into *periods* $1, 2, \ldots, M$ (in our protocol, the periods will be the slots), and the signer has a different secret key sk_t for each period t. When the signer no longer needs sk_t, they derive the secret key sk_{t+1} for the next period from sk_t, and then erase sk_t. Updatable signatures are also known as *key-evolving* or *forward-secure* signatures.

Definition 35 (Updatable Signature)**.** *Let M be a positive integer. An updatable signature scheme with M periods consists of a quadruple of polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ *with the following syntax:*

KeyGen**:** *This algorithm is probabilistic and takes no input. It outputs a public key* pk *and a secret key* sk_1 *for the first period.*

Update**:** *This algorithm may be probabilistic, and takes as input a secret key* sk_t *for a period* $t \in [M-1]$*. It outputs a secret key* sk_{t+1} *for period* $t+1$*.*

Sign**:** *This algorithm is often probabilistic, and takes as input a secret key* sk_t *for a period* $t \in [M]$ *and a message m. It outputs a signature s.*

Verify: *This algorithm is deterministic, and takes as input a public key* pk*, a period* $t \in [M]$*, a message* m *and a signature* s*. It outputs a bit: 0 stands for "*reject*", 1 stands for "*accept*".*

We additionally require that, for every period $t \in [M]$ *and every message* m*,*

$$\Pr\left[\mathsf{Verify}(\mathsf{pk}, t, m, s) = 1 \;\middle|\; \begin{array}{l}(\mathsf{pk}, \mathsf{sk}_1) \xleftarrow{\$} \mathsf{KeyGen} \\ \textit{for } j = 1, \ldots, t-1 : \mathsf{sk}_{j+1} \xleftarrow{\$} \mathsf{Update}(\mathsf{sk}_j) \\ s \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_t, m)\end{array}\right] = 1.$$

We assume that each secret key sk_t also encodes the index t of its period, so Update and Sign do not need to receive t as a separate input. The above property (correctness) states that, if we update the initial secret key $t-1$ times and use the result to sign a message m, the verification algorithm accepts the signature for period t. Notice that, for $M = 1$, an updatable signature scheme is just a digital signature scheme.

As for digital signatures, we require that it is infeasible to forge signatures, even after seeing signatures on messages of the attacker's choice. In addition, signatures for a period t must remain unforgeable even if the attacker later breaks into the signer's machine, learning the secret key stored there at that moment. This property is called *forward security*.

Definition 36 (Forward Security)**.** *Let* $T \in \mathbb{N}$ *and* $\epsilon \in [0, 1]$*. An updatable signature scheme with* M *periods is* (T, ϵ)*-forward-secure if, for every pair of probabilistic algorithms* $(\mathcal{A}_0, \mathcal{A}_1)$ *whose combined running time is at most* T*, it holds that*

$$\Pr\left[\begin{array}{c}\mathsf{Verify}(\mathsf{pk}, t, m, s) = 1 \\ t < t^* \\ (t, m) \notin \mathcal{Q}\end{array} \;\middle|\; \begin{array}{l}(\mathsf{pk}, \mathsf{sk}_1) \xleftarrow{\$} \mathsf{KeyGen} \\ \textit{for } j = 1, \ldots, M-1 : \mathsf{sk}_{j+1} \xleftarrow{\$} \mathsf{Update}(\mathsf{sk}_j) \\ (t^*, \psi) \xleftarrow{\$} \mathcal{A}_0^{\mathsf{Sign}(\mathsf{sk}_\cdot, \cdot)}(\mathsf{pk}) \\ (t, m, s) \xleftarrow{\$} \mathcal{A}_1(\mathsf{sk}_{t^*}, \psi)\end{array}\right] \leq \epsilon.$$

Above, $\mathcal{A}_0^{\mathsf{Sign}(\mathsf{sk}_\cdot, \cdot)}$ *means that* $\mathcal{A}_0$ *has unrestricted access to the oracle that, on input a period* $t' \in [M]$ *and a message* m'*, responds with a signature* $s' \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_{t'}, m')$*. We denote the set of pairs* (t', m') *queried by* $\mathcal{A}_0$ *by* $\mathcal{Q}$*. The output* t^* *of* $\mathcal{A}_0$ *belongs to* $[M+1]$*, and we set* $\mathsf{sk}_{M+1} := \bot$*.*

Let us discuss the above definition. The attack proceeds in two phases. In the first phase, $\mathcal{A}_0$ receives the public key and observes the signer at work: it can obtain signatures on messages of its choice for any period. At the end of this phase, $\mathcal{A}_0$ chooses the period t^* in which it breaks into the signer's machine, and passes its internal state ψ to $\mathcal{A}_1$. In the second phase, $\mathcal{A}_1$ receives sk_{t^*}, which is what the attacker finds on the signer's machine: by then, the signer has erased all the keys for the periods before t^* (if $t^* = M+1$, there is no break-in and $\mathcal{A}_1$ receives nothing). The attacker wins if $\mathcal{A}_1$ outputs a valid signature s on a message m for a period t preceding the break-in, and $\mathcal{A}_0$ never asked the oracle for a signature on m for period t. We cannot hope to protect the periods $t \geq t^*$: using sk_{t^*} and the algorithm Update, the attacker can sign any message for period t^* and for all subsequent periods. Notice also that the guarantee is meaningful only if the signer erases sk_t as soon as they compute sk_{t+1}: if the old keys were still stored on the signer's machine, the attacker would learn them too when breaking in. Finally, observe that, for $M = 1$, forward security boils down to the unforgeability of digital signatures (Definition 16).

We will see how to build updatable signatures when we introduce Merkle trees (see Exercise 31).

5.4 The Proof-of-Stake Election Using a Randomness Beacon

We are now ready to present our first construction, which is a simplified version of Ouroboros Praos, a proof-of-stake protocol of the Ouroboros family that is used by the Cardano blockchain. We assume that

all validators are equipped with a clock. We split time into slots of fixed duration, and we organise them into groups of N consecutive slots, which we call *epochs*. Our construction assumes that everybody has access to a randomness beacon: this is a mechanism that, at the beginning of every epoch, reveals a random value $a \in \{0,1\}^\lambda$ to all participants. In particular, everybody receives the same value. We will explain in Section 5.6 how to construct such a tool.

We assume that, before the beginning of the previous epoch, every participant P_i has committed some amount of stake V_i on the blockchain, linking it to their signature verification key pk_i and to their VRF key vk_i (we will see in Section 5.6 why the stake must be committed one epoch in advance). In particular, all this information is public and available to everybody. The signature scheme is an updatable signature scheme with M periods (see Section 5.3), where M is larger than the total number of slots in the execution, and we identify period t with slot t. To distinguish the two primitives, we denote the algorithms of the updatable signature scheme by $\mathsf{SIG.KeyGen}$, $\mathsf{SIG.Update}$, $\mathsf{SIG.Sign}$ and $\mathsf{SIG.Verify}$, and those of the VRF by $\mathsf{VRF.KeyGen}$, $\mathsf{VRF.Eval}$ and $\mathsf{VRF.Verify}$. Let $\mathsf{sk}_{i,1}$ be the initial secret signing key of P_i and let K_i be their secret VRF key. We also assume that the party published a signature $s_i \xleftarrow{\$} \mathsf{SIG.Sign}(\mathsf{sk}_{i,1}, \mathsf{vk}_i)$ on their VRF key for period 1, and a pair $(r_i, \pi_i) = \mathsf{VRF.Eval}(K_i, \mathsf{pk}_i)$. If $\mathsf{SIG.Verify}(\mathsf{pk}_i, 1, \mathsf{vk}_i, s_i) = 0$ (i.e., the signature on vk_i does not verify), or if $\mathsf{VRF.Verify}(\mathsf{vk}_i, \mathsf{pk}_i, r_i, \pi_i) = 0$ (i.e., the VRF proof does not verify), the validator P_i is excluded from the election. If multiple validators have the same VRF key vk, only the first one that announced vk on the blockchain is considered for the elections. At the beginning of every slot t, each validator P_i stores the secret key $\mathsf{sk}_{i,t}$ for period t and has erased the secret keys for all previous periods (a validator who registers after the first slot obtains $\mathsf{sk}_{i,t}$ by applying $\mathsf{SIG.Update}$ to $\mathsf{sk}_{i,1}$ repeatedly, erasing all intermediate keys). Let $W \in \mathbb{Z}_{2^k}$ be a hardness parameter and let V_{tot} be the total amount of stake in the system.

Each block consists of a tuple

$$(h, t, z, x, y, \mathsf{pk}, r, \pi, s),$$

where h is a pointer to the previous block, t is the index of the slot, z is a hash, x is the content of the block, y is metadata, pk is the public key of the block proposer, r is a VRF output, π is a VRF proof, and s is a signature. A valid block satisfies the following conditions:

- $z = \mathsf{Hash}(x, y)$;
- (signature verification) $\mathsf{SIG.Verify}(\mathsf{pk}, t, (h, z), s) = 1$, i.e., s is a valid signature on (h, z) under pk for period t;
- (VRF verification) $\mathsf{VRF.Verify}(\mathsf{vk}, \mathsf{Hash}(t, a), r, \pi) = 1$, where vk is the VRF key registered together with pk, i.e., π proves that r is obtained by evaluating the block proposer's VRF key on $\mathsf{Hash}(t, a)$, where a is the output of the randomness beacon in the epoch of the block;
- (election) $r < \frac{V}{V_{\mathsf{tot}}} \cdot W$, where V is the amount of stake of the block proposer.

Notice that t does not need to be part of the signed message: since s is a signature for period t, it cannot be used in a block for any other slot.

At the beginning of every slot t, each validator P_i performs the following operations:

1. P_i picks the most recent block $B' = (h', t', z', x', y', \mathsf{pk}', r', \pi', s')$ on their main chain and computes the pointer $h = \mathsf{Hash}(h', t', z', \mathsf{pk}')$.
2. P_i sets $c = \mathsf{Hash}(t, a)$, where a is the value produced by the randomness beacon for the current epoch. Then, P_i computes $(r, \pi) \leftarrow \mathsf{VRF.Eval}(K_i, c)$ and checks whether $r < \frac{V_i}{V_{\mathsf{tot}}} \cdot W$. If this is the case, they gather 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. Then, they derive $z = \mathsf{Hash}(x, y)$ and $s \xleftarrow{\$} \mathsf{SIG.Sign}(\mathsf{sk}_{i,t}, (h, z))$.
3. Whether or not they won the election, P_i computes $\mathsf{sk}_{i,t+1} \xleftarrow{\$} \mathsf{SIG.Update}(\mathsf{sk}_{i,t})$, the secret key for the next slot, and erases $\mathsf{sk}_{i,t}$ from their machine. If they won, they then send the new block

$B = (h, t, z, x, y, \mathsf{pk}_i, r, \pi, s)$ over the gossip network (it is important that the erasure occurs before B is released). Finally, they wait for the beginning of the next slot.

When the validators receive a new block $B = (h, t, z, x, y, \mathsf{pk}, r, \pi, s)$, they check whether slot t has begun. If not, they put the processing of B on hold until slot t begins, and process other blocks in the meantime. When slot t begins, they retrieve the output a of the randomness beacon for the current epoch. They also retrieve the verification key vk and the amount of stake V that were committed on the chain by the validator with signature key pk before the beginning of the previous epoch. If these do not exist, the block is discarded. Next, the parties verify the following properties:

- $z \stackrel{?}{=} \mathsf{Hash}(x, y)$, i.e., z is the hash of the content and of the metadata;
- $\mathsf{SIG.Verify}(\mathsf{pk}, t, (h, z), s) \stackrel{?}{=} 1$ (signature verification), i.e., the block was signed with the secret key of pk for period t;
- $\mathsf{VRF.Verify}(\mathsf{vk}, \mathsf{Hash}(t, a), r, \pi) \stackrel{?}{=} 1$ (VRF verification), i.e., r is the output of the VRF of the block proposer on input $\mathsf{Hash}(t, a)$;
- $r \stackrel{?}{<} \frac{V}{V_{\mathsf{tot}}} \cdot W$, i.e., the block proposer won the election in slot t.

They also check that the slot of the block that B points to is strictly smaller than t. If all these conditions are satisfied, they add B to their graph. Otherwise, they discard it.

5.4.1 An Informal Explanation of the Design

Before analysing the security of our construction, let us informally explain the role of its main ingredients. Recall the requirements that we listed in Section 5.1: nobody should be able to increase their chance of winning the election by performing more computation, nobody should be able to predict who will win the next elections, and the winners must be able to prove that they won.

The election does not depend on the block. In slot t, a validator finds out whether they won the election by evaluating their VRF on $\mathsf{Hash}(t, a)$, which only depends on the slot t and on the output a of the randomness beacon. In particular, the outcome of the election is independent of the content x and of the metadata y of the block that the validator is going to propose, as well as of the block that the new block is going to be appended to (i.e., of the pointer h). Suppose, for instance, that the validators evaluated their VRF on $\mathsf{Hash}(t, a, h, z)$ instead. A corrupted validator who lost the election could then try many different contents, metadata and previous blocks, until finding a combination that makes them win. This is called a *grinding attack*: by performing more computation, the corrupted validator would increase their chance of winning the election, which is exactly what we wanted to avoid. In our construction, instead, every validator has a single chance of winning in every slot: the input of the VRF is fixed, and the security of the VRF prevents the validator from choosing among several outputs with valid proofs. This is also why we use a VRF rather than, for instance, the hash of a signature on $\mathsf{Hash}(t, a)$: since signing is often probabilistic, a corrupted validator could produce many different signatures on the same message and pick the one with the smallest hash.

The role of the randomness beacon. The outcome of the elections of an epoch is determined by the VRF keys of the validators, by their stake, and by the output a of the randomness beacon. It is crucial that a is unpredictable at the time the VRF keys and the stake are committed. Otherwise, the adversary could grind over the keys and over the distribution of its stake: for instance, it could generate many VRF key pairs, evaluate each of them on the inputs $\mathsf{Hash}(t, a)$ of the slots of the epoch, register the keys that win the largest number of elections, and split its stake among these keys in the most convenient way. Indeed, recall that the unbiasability of the VRF (Definition 34) only holds for inputs that cannot be predicted at the time the keys are chosen. For the same reason, we cannot simply replace a by a constant: the adversary could

then grind over its keys before registering them. In our construction, the VRF keys and the stake used in an epoch must therefore be committed before the output of the beacon for that epoch is determined (we will see in Section 5.6 that, with our implementation of the beacon, this means one epoch in advance).

Protection against adaptive corruption. VRFs and updatable signatures are fundamental to protect the protocol against adaptive corruption. Every validator evaluates the VRF with their own secret key and, by the pseudorandomness of the VRF, the outputs look random to anybody who does not know the key. Hence, nobody can tell who will win the election in a slot until the winner publishes their block, together with the VRF proof showing that they won. If we used a public function to determine the winners (for instance, if a validator won whenever $\mathsf{Hash}(t, a, \mathsf{pk}) < \frac{V}{V_{\mathsf{tot}}} \cdot W$), everybody could compute the winners of all the slots of an epoch as soon as a is revealed, and the attacker could try to corrupt them (or, more simply, take them offline) before their slots. With VRFs, instead, by the time the identity of a winner is revealed, their block has already been published. An attacker who corrupts the winner right after the publication of the block could still try to use the secret signing key of the winner to sign a different block for the same slot, or to rewrite the blocks that the winner proposed in the past. Updatable signatures prevent this: the winner erases the secret key of the current slot before publishing the block, so the attacker only learns secret keys for future slots which, by forward security, are useless for signing blocks for the current slot or for past slots.

The role of the signature. Finally, the signature s binds the block to the winner of the election. The VRF output r and the proof π only show that the validator who registered the keys pk and vk won the election in slot t: as we saw above, they do not depend on the rest of the block. Without the signature, anybody who saw the block of a winner could replace its content, its metadata or its pointer, re-use r and π, and obtain a different valid block for the same slot. The signature prevents this. The signed message (h, z) contains the pointer h to the previous block and the digest $z = \mathsf{Hash}(x, y)$, which, by the collision resistance of Hash, determines the content x and the metadata y. Moreover, s is a signature for period t, so it is bound to slot t as well. Hence, the signature binds the content, the metadata, the slot and the pointer to the previous block to the winner of the election and, by forward security, nobody else can produce it.

5.5 A Security Analysis of the Proof-of-Stake Blockchain

In this section, we provide a mathematical proof of security for the proof-of-stake design we have just introduced. It follows the same approach as for proof of work (see Section 3.2.5). We begin by introducing the notions of *non-empty slot* and *safe slot*.

Definition 37 (Non-empty slot)**.** *Consider the graph of blocks of an honest party. We say that a slot is* non-empty *if the graph contains at least one valid block generated in that slot.*

Definition 38 (Equivalent blocks)**.** *Two blocks*

$$B_0 = (h_0, t_0, z_0, x_0, y_0, \mathsf{pk}_0, r_0, \pi_0, s_0) \qquad \textit{and} \qquad B_1 = (h_1, t_1, z_1, x_1, y_1, \mathsf{pk}_1, r_1, \pi_1, s_1)$$

are equivalent if

$$h_0 = h_1, \qquad t_0 = t_1, \qquad z_0 = z_1, \qquad \mathsf{pk}_0 = \mathsf{pk}_1.$$

Definition 39 (Safe slot and safe blocks)**.** *Consider the graph of blocks of an honest party. We say that a slot is* safe *if all the blocks generated in it are equivalent and one of them was generated by an honest validator. We say that a block is* safe *if it was generated in a safe slot.*

Observe that, if a party with stake V_i is honest, by the pseudorandomness of the VRF, the probability that they win the election in a given slot is

$$\approx \frac{V_i}{V_{\mathsf{tot}}} \cdot \frac{W}{2^k}.$$

Suppose that all parties are honest. Using the union bound, we can estimate the probability that at least one party wins the election as

$$\approx \sum_i \frac{V_i}{V_{\mathsf{tot}}} \cdot \frac{W}{2^k} = \frac{W}{2^k}.$$

This means that, if everybody follows the protocol, we should expect roughly one new block every $\frac{2^k}{W}$ slots. Throughout this section, we assume that each slot lasts exactly Δ, i.e., as long as the maximum network delay, and that a block that reaches a validator exactly at the beginning of a slot is processed before the operations of that slot. In particular, a block that an honest validator sends at the beginning of a slot reaches all honest parties before they perform the operations of the next slot. Hence, when the blockchain is not under attack, producing a new block requires time

$$L := \frac{2^k}{W} \cdot \Delta.$$

The next theorem shows that, if an adversary controls only a fraction α of all stake, the fraction of non-empty slots that are safe is roughly

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

where Δ is the maximum delay in the gossip network and L is the expected time required to generate a new block. This is the proof-of-stake analogue of Theorem 7.

Theorem 12. *Suppose that*

- *the VRF scheme is (T,ϵ)-pseudorandom, (T,ϵ)-sound and (T,λ,ϵ,p)-unbiasable for every monotone statistic p computable in time T, where λ is the output size of* Hash*;*
- *the updatable signature scheme is (T,ϵ)-forward-secure;*
- *the hash function is (T,ϵ)-secure (i.e., collision resistant).*

Suppose that the corrupted parties control at most a fraction α of all stake in the protocol. Let Δ be an upper bound on the network delay and let L be the expected time required to generate a new block. Consider any sequence S of ℓ consecutive non-empty slots and suppose that T is an upper bound on the total amount of computation performed by the system by the end of the last slot in S. Then, for every $\delta \in [0,1]$, it holds that

$$\Pr\left[\#\textit{ safe slots in } S \le (1-\delta)(1-\alpha)\left(1-\frac{\Delta}{L}\right)\ell\right] \le \exp\left(-\frac{\delta^2}{2}(1-\alpha)\left(1-\frac{\Delta}{L}\right)\ell\right) + (4n+2)\epsilon + T\cdot 2^{-\lambda}.$$

Proof. Observe that all participants in the election have distinct VRF keys. Moreover, with extremely high probability, all honest participants are included. Indeed, the probability that two honest participants select the same key is at most ϵ (if it were greater than ϵ, we could use this to break the pseudorandomness of the VRF). An adversary cannot kick an honest party out of the election by copying their VRF key either. Indeed, the adversary would also need to provide the evaluation of this VRF on the adversary's own signature public key, together with a proof that this evaluation is correct. This would require the adversary either to guess the result of this evaluation correctly (which happens with probability at most ϵ, by the pseudorandomness of the VRF), or to generate a valid proof for an incorrect evaluation (which happens with probability at most ϵ, by the soundness of the VRF). In other words, except with probability $3n\cdot\epsilon$, all honest participants are included in the election.

It also turns out that an attacker cannot produce any valid block $B = (h,t,z,x,y,\mathsf{pk},r,\pi,s)$ in which pk is the key of a validator P that is honest at the beginning of slot t, unless P has already published a block B' equivalent to B in slot t. Indeed, for B to be valid, s must be a signature on (h,z) that verifies under pk for period t. The attacker may try to corrupt P (e.g., by taking control of their machine) to learn their secret key. However, P is honest at the beginning of slot t, so P performs the operations of slot t honestly:

in particular, P computes the secret key for period $t+1$ and erases the one for period t (if P wins slot t, this happens before their block is released). Hence, if the attacker corrupts P at a later point, they only learn the secret key for some period $t^* > t$. By the forward security of the updatable signature scheme, this key is useless: the attacker cannot produce a valid signature for period t on a message that P has not signed for period t (for each validator, this happens with probability at most ϵ, so with probability at most $n\epsilon$ overall). The only remaining strategy is to reuse a signature that P has already revealed. Unfortunately for the attacker, an honest party P signs at most one pair (h, z) for period t, and only if P wins slot t.[2] Suppose that $B' = (h', t, z', x', y', \mathsf{pk}, r', \pi', s')$ is the block that P proposes. To use s', the adversary is forced to set $h = h'$ and $z = z'$, so B is equivalent to B'. Moreover, the adversary is also forced to set $x = x'$ and $y = y'$; otherwise, $\mathsf{Hash}(x, y) \neq z$. The alternative is to try to find a collision for the hash function, but this is extremely unlikely to succeed (it happens with probability at most ϵ).

This argument shows that, if the corrupted parties want to generate a block that is not equivalent to any honestly generated block in the same slot, their only option is to win the election themselves.

Hence, if in a given slot there is only one winner and this is an honest party, then the slot is most likely safe (unless the attacker manages to break the forward security of the updatable signature scheme or the collision resistance of the hash function).

Now, observe that, if we model Hash as a random oracle, the inputs of the VRF are just random strings. Moreover, all these inputs are unpredictable at the time the VRF keys are selected, because the output a of the randomness beacon had not yet been revealed (the probability that the adversary queried the random oracle on some pair (t, a) before a was revealed is at most $T \cdot 2^{-\lambda}$, because the adversary's running time is at most T, so it can issue at most T queries). This allows us to apply the unbiasability of the VRF to argue that the statistical properties of the evaluations are essentially the same as those of random strings.

Formally, let p be the statistic that, on input $(r_{i,j})_{i,j}$, with $r_{i,j} \in \mathbb{Z}_{2^k} \cup \{\perp\}$, computes

$$Q_i := \left\{ j \middle| r_{i,j} < \frac{V_j}{V_{\mathsf{tot}}} \cdot W, r_{i,j} \neq \perp \right\}$$

for every i. Then, it sets ω_i to 1 if either Q_i has a single element, which is the index of an honest party, or there exists $r_{i,j} = \perp$ where j is the index of an honest party (otherwise, $\omega_i = 0$). Then, p computes the total number ℓ of non-empty sets Q_i and counts how many ω_i are equal to 1. If this number is smaller than

$$(1-\delta) \cdot (1-\alpha) \cdot \left(1 - \frac{W}{2^k}\right) \cdot \ell,$$

p outputs 1; otherwise, it outputs 0. It is easy to see that p is monotone (replacing any $r_{i,j}$ with $\perp$ can only decrease the probability that p outputs 1).

Consider now an execution of the proof-of-stake blockchain, and let $r_{i,j}$ be the VRF evaluation of the j-th party in the i-th slot. For the honest parties, we can derive it from their secret keys. For the corrupted parties, we can derive it only if they publish a valid block in the i-th slot; if this does not happen, we set $r_{i,j} \leftarrow \perp$. Observe that (assuming that the adversary does not break the collision resistance of Hash or the forward security of the updatable signature scheme) p computes the fraction of non-empty slots that are safe and outputs 1 only when this is smaller than

$$(1-\delta) \cdot (1-\alpha) \cdot \left(1 - \frac{W}{2^k}\right).$$

By the unbiasability of the VRF, however, the probability that p outputs 1 is roughly the same as when each $r_{i,j}$ is sampled at random (it can be greater by at most ϵ).

Next, we show that the probability that p outputs 1 when the $r_{i,j}$ are sampled at random is extremely small. Define $q_i := \frac{V_i}{V_{\mathsf{tot}}} \cdot \frac{W}{2^k}$ and let H be the set of indices of the honest parties. If we sample $r_1, \ldots, r_n$

[2] At registration, P also signed their VRF key for period 1. We implicitly assume that VRF keys and pairs (h, z) are encoded differently, so this signature cannot be used in a block.

uniformly at random from $\mathbb{Z}_{2^k}$, the probability that there exists exactly one r_i such that $r_i < \frac{V_i}{V_{\mathsf{tot}}} \cdot W$, and that $i \in H$, is

$$\sum_{i \in H} q_i \cdot \prod_{j \neq i} (1 - q_j).$$

By the union bound, we have

$$\prod_{j \neq i} (1 - q_j) \geq 1 - \sum_{j \neq i} q_j \geq 1 - \frac{W}{2^k}.$$

Therefore, the probability that there exists a single r_i such that $r_i < \frac{V_i}{V_{\mathsf{tot}}} \cdot W$ and $i \in H$ is

$$\sum_{i \in H} q_i \cdot \prod_{j \neq i} (1 - q_j) \geq \sum_{i \in H} q_i \cdot \left(1 - \frac{W}{2^k}\right) \geq (1 - \alpha) \frac{W}{2^k} \cdot \left(1 - \frac{W}{2^k}\right).$$

On the other hand, by the union bound, the probability that there exists at least one r_i such that $r_i < \frac{V_i}{V_{\mathsf{tot}}} \cdot W$ is at most $\frac{W}{2^k}$. Therefore, the probability that there exists a single r_i such that $r_i < \frac{V_i}{V_{\mathsf{tot}}} \cdot W$ and $i \in H$, conditioned on the existence of at least one r_i such that $r_i < \frac{V_i}{V_{\mathsf{tot}}} \cdot W$, is at least

$$\frac{(1 - \alpha) \frac{W}{2^k} \cdot \left(1 - \frac{W}{2^k}\right)}{\frac{W}{2^k}} \geq (1 - \alpha) \cdot \left(1 - \frac{W}{2^k}\right).$$

We conclude by applying the Chernoff bound and observing that $W/2^k = \Delta/L$: the probability that there are fewer than

$$(1 - \delta) \cdot (1 - \alpha) \cdot \left(1 - \frac{W}{2^k}\right) \ell$$

safe slots among ℓ non-empty slots is at most

$$\exp\left(-\frac{\delta^2}{2}(1 - \alpha)\left(1 - \frac{\Delta}{L}\right)\ell\right) + (4n + 2)\epsilon + T \cdot 2^{-\lambda},$$

where $T \cdot 2^{-\lambda}$ is an upper bound on the probability that the corrupted parties manage to predict the output of the randomness beacon, and $(4n + 2)\epsilon$ is an upper bound on the probability that they manage to break any of the security properties of the VRF, the hash function or the updatable signature scheme. □

From now on, we assume that the VRF, the updatable signature scheme and the hash function satisfy the assumptions of Theorem 12, where T is an upper bound on the total amount of computation performed by the system during the whole execution, and we let

$$\epsilon^* := (4n + 2)\epsilon + T \cdot 2^{-\lambda}.$$

Recall also that the execution has fewer than M slots, where M is the number of periods of the updatable signature scheme. Moreover, in the rest of this section, we apply the definitions of non-empty and safe slots to the union of the block trees of all honest parties at the end of the execution. In this way, these notions do not depend on the party we consider: for instance, all the blocks of a safe slot that appear in the tree of any honest party are equivalent. As in proof of work, we need a version of Theorem 12 that holds for all sufficiently long sequences of consecutive non-empty slots at the same time.

Corollary 2. *Suppose that the corrupted parties control a constant fraction $\alpha < 1$ of all stake, that the network has delay at most Δ, and that the expected block generation time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Let $\eta \in (0, 1)$ be a constant. Then, for every integer $\ell \geq 1$, except with probability $M \cdot 2^{-\Omega(\ell)} + \epsilon^*$, every sequence of $m \geq \ell$ consecutive non-empty slots contains more than $(1 - \eta)(1 - \gamma)(1 - \alpha) \cdot m$ safe slots.*

Proof. We proceed as in the proof of Corollary 1. Fix a sequence of m consecutive non-empty slots. By Theorem 12 with $\delta = \eta$, the probability that it contains at most $(1-\eta)(1-\gamma)(1-\alpha)\cdot m$ safe slots is at most

$$\exp\left(-\frac{\eta^2(1-\gamma)(1-\alpha)}{2}\cdot m\right) + \epsilon^*.$$

The term ϵ^* bounds the probability that the adversary predicts the output of the randomness beacon or breaks one of the cryptographic primitives. These events concern the whole execution rather than the specific sequence of slots, so we only need to account for them once.[3] There are fewer than M choices for the first slot of the sequence so, by the union bound, the probability that some sequence of $m \geq \ell$ consecutive non-empty slots contains too few safe slots is at most

$$M \cdot \sum_{m=\ell}^{\infty} \exp\left(-\frac{\eta^2(1-\gamma)(1-\alpha)}{2}\cdot m\right) + \epsilon^* = M \cdot 2^{-\Omega(\ell)} + \epsilon^*.$$

□

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, and the height of a block as the length of its history. As for proof-of-work consensus, safe blocks play a fundamental role in the security analysis of proof-of-stake consensus. The reason boils down to the following lemma, which is the proof-of-stake analogue of Lemma 6.

Lemma 7. *Consider a protocol execution (recall that slots last exactly Δ). Then:*

- No two safe blocks generated in different slots have the same height (even if they belong to the trees of different parties).
- At the beginning of every slot t, the main chain of every honest party is at least as long as the number of safe slots preceding t.

Proof. This result is proven as Lemma 6, using the fact that distinct safe blocks cannot be generated in the same slot and must therefore be generated at least Δ apart. More precisely, the honest block of a safe slot is generated at the beginning of the slot and, since slots last Δ, it reaches all honest parties before they perform the operations of the next slot. □

Common prefix. The following theorem is the proof-of-stake analogue of Theorem 8.

Theorem 13. *Consider a protocol execution in which the adversary controls a constant fraction $\alpha < 1$ of all stake in the system, the network has delay at most Δ, and the expected block generation time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Suppose that*

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

and that all honest parties start from the same genesis block, which is safe. Let $\ell \geq 1$ be an integer and, for every chain C, let $\lfloor C \rfloor_\ell$ denote the chain obtained by removing the ℓ most recent blocks from C. Then, except with probability $M \cdot 2^{-\Omega(\ell)} + \epsilon^*$, the following holds at every point in time t: if C_1 and C_2 are longest chains in the block trees of two honest parties at time t (the two parties may coincide), then

$$\lfloor C_1 \rfloor_\ell \subseteq \lfloor C_2 \rfloor_\ell \qquad \text{or} \qquad \lfloor C_1 \rfloor_\ell \supseteq \lfloor C_2 \rfloor_\ell.$$

We do not prove Theorem 13, since its proof is considerably more involved than that of Theorem 8. Let us explain, at a high level, where the difficulty comes from. Unlike in proof of work, generating a block for a slot that one has won costs nothing (this is often called the *nothing-at-stake* problem): a corrupted

[3] Formally, in the proof of Theorem 12, we can use a single monotone statistic that checks all sequences of consecutive non-empty slots at the same time.

validator who wins the election in some slot can sign as many different blocks for that slot as they wish. All these blocks, however, carry the same slot number and, since the slots of the blocks along a chain are strictly increasing, no two of them can belong to the same chain. In other words, a slot won by the adversary allows them to extend several branches at the same time, but each branch by a single block. In the proof of Theorem 8, we found a distinct non-safe block at each height at which C_1 and C_2 disagree. In proof of stake, a single non-safe slot can provide a block to both chains, so the same counting argument only gives half as many non-safe slots.

In fact, no argument that only counts the safe and the non-safe slots in a sequence of slots can establish Theorem 13. For instance, suppose that, from some point on, every third non-empty slot is won by the adversary, while all the other non-empty slots are safe. Even though two thirds of the non-empty slots are safe, the adversary can keep two branches of the same length forever (assuming that ties between longest chains are broken in its favour): it signs two blocks for each of its slots, one for each branch, and lets the honest parties extend the two branches in turn. Such regular patterns are, however, extremely unlikely, and a proof of Theorem 13 needs to exploit this fact, by looking at the order in which safe and non-safe slots occur rather than just at their number (see, e.g., the analysis of the Ouroboros protocols).

Here, we only prove the following weaker statement. It is clearly a consequence of Theorem 13 but, unlike Theorem 13, it can be proven using the same strategy as in proof of work.

Corollary 3. *Consider a protocol execution in which the adversary controls a constant fraction $\alpha < 1$ of all stake in the system, the network has delay at most Δ, and the expected block generation time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Suppose that*

$$(1-\alpha)\,(1-\gamma) > \frac{2}{3}$$

and that all honest parties start from the same genesis block, which is safe. Let $\ell \geq 1$ be an integer. Then, except with probability $M \cdot 2^{-\Omega(\ell)} + \epsilon^*$, the following holds at every point in time t: if C_1 and C_2 are longest chains in the block trees of two honest parties at time t (the two parties may coincide), then

$$\lfloor C_1 \rfloor_\ell \subseteq \lfloor C_2 \rfloor_\ell \qquad \text{or} \qquad \lfloor C_1 \rfloor_\ell \supseteq \lfloor C_2 \rfloor_\ell.$$

Proof. The proof follows that of Theorem 8, with slots playing the role of mining times. Throughout the proof, we treat equivalent blocks as the same block: the pointer to a block only depends on its components h, t, z and pk, so every block that points to one of two equivalent blocks also points to the other.

Since the bound in the statement is asymptotic, we can assume that ℓ is larger than any given constant. Let $\eta > 0$ be a constant such that $q := (1-\eta)(1-\alpha)(1-\gamma) > 2/3$ (it exists because $(1-\alpha)(1-\gamma) > 2/3$). By Corollary 2 (with $\lceil \ell/2 \rceil$ in place of ℓ), except with probability $M \cdot 2^{-\Omega(\ell)} + \epsilon^*$, every sequence of $m \geq \ell/2$ consecutive non-empty slots contains more than $q \cdot m$ safe slots. Since $q > 2/3$ and ℓ is large enough, this is more than $(2m+1)/3$ safe slots (indeed, $q \cdot m \geq (2m+1)/3$ whenever $m \geq 1/(3q-2)$). We show that, whenever this is the case, the property in the statement holds at every point in time.

Suppose, towards a contradiction, that, at some time t during slot s, the chains C_1 and C_2 are longest chains in the trees of two honest parties P_1 and P_2, but neither $\lfloor C_1 \rfloor_\ell \subseteq \lfloor C_2 \rfloor_\ell$ nor $\lfloor C_1 \rfloor_\ell \supseteq \lfloor C_2 \rfloor_\ell$. Define $h_1 \leq h_2$, B^* and h^* as in the proof of Theorem 8. As shown there, $h_1 - h^* \geq \ell + 1$. Let B_0 be the most recent honest block belonging to both C_1 and C_2, let $h_0 \leq h^*$ be its height, and let s_0 be the slot in which it was generated. All blocks of C_1 and C_2 with height greater than h_0 are descendants of B_0, so they were generated in the slots $s_0 + 1, \ldots, s$ (recall that the honest parties put the blocks of future slots on hold). We now count the safe and the non-safe non-empty slots among $s_0 + 1, \ldots, s$.

Safe slots. Consider any safe slot σ with $s_0 < \sigma < s$, and let B be its block. Since slots last Δ, the honest party that generated B at the beginning of slot σ had already received B_0, so the height of B is greater than h_0. For the same reason, P_1 received B, together with its history, by the beginning of slot s. Since C_1 is a longest chain in the tree of P_1, the height of B is at most h_1. By Lemma 7, the blocks of different safe slots have different heights, so at most $h_1 - h_0$ safe slots lie strictly between s_0 and s. Including slot s, at most $h_1 - h_0 + 1$ safe slots lie among $s_0 + 1, \ldots, s$.

Non-safe slots. Consider any height h such that $h_0 < h \leq h^*$. The block of height h on C_1, which also lies on C_2, is not honest (by the choice of B_0), so it was generated in a non-safe slot. Consider now any

height h such that $h^* < h \leq h_1$. The two chains contain two distinct blocks of height h and, by Lemma 7, at most one of them was generated in a safe slot (recall that all the blocks of a safe slot are equivalent). For $i \in \{1, 2\}$, let A_i be the set of heights h with $h^* < h \leq h_1$ such that the block of height h on C_i was generated in a non-safe slot. Every height between $h^* + 1$ and h_1 belongs to A_1 or to A_2, so one of these two sets, say A_i, contains at least $(h_1 - h^*)/2$ heights. Since the slots in which the blocks of a chain were generated form a strictly increasing sequence, the blocks of C_i with heights in $\{h_0 + 1, \ldots, h^*\} \cup A_i$ were generated in distinct non-safe slots. Hence, at least

$$(h^* - h_0) + \frac{h_1 - h^*}{2}$$

non-safe non-empty slots lie among $s_0 + 1, \ldots, s$. This is where the proof differs from that of Theorem 8: as we explained after Theorem 13, a single non-safe slot may provide blocks to both chains, so we can only rely on half of the heights above h^*.

Let m be the number of non-empty slots among $s_0 + 1, \ldots, s$. These form a sequence of consecutive non-empty slots, and $m \geq (h_1 - h^*)/2 \geq (\ell + 1)/2$. By the above, the number of safe slots in this sequence is at most

$$h_1 - h_0 + 1 = (h^* - h_0) + (h_1 - h^*) + 1 \leq 2\left((h^* - h_0) + \frac{h_1 - h^*}{2}\right) + 1,$$

i.e., at most twice the number of non-safe slots plus one. Hence, the sequence contains at most $(2m + 1)/3$ safe slots. This contradicts our assumption on the execution. □

Using Theorem 13, we obtain finality exactly as in proof of work.

Theorem 14. *Consider a protocol execution that satisfies the assumptions of Theorem 13. Pick any honest party and any two time instants $t_1 \leq t_2$, and let C_1 and C_2 be the main chains of the party at times t_1 and t_2, respectively. Let ℓ be a positive integer. Then,*

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

Proof. The proof is identical to that of Theorem 9, using Theorem 13 in place of Theorem 8. □

Chain growth. The next theorem is the proof-of-stake analogue of Theorem 10.

Theorem 15. *Consider a protocol execution in which the adversary controls a fraction $\alpha < 1$ of all stake in the system and the network has delay at most Δ. Consider the main chain of an honest party at time t, and let $\kappa > 0$. Then, except with probability $\exp(-\kappa) + \epsilon^*$, the honest party extends their main chain by a new block by time*

$$t + 3\Delta + \kappa \cdot \frac{L}{1 - \alpha}.$$

Proof. As in the proof of Theorem 12, except with probability ϵ^*, we can analyse the execution as if the VRF evaluations were truly random. Let B be the last block added to the main chain of the honest party, let h be its height, and let $t' \leq t$ be the time at which this block was received. All honest parties receive B by time $t' + \Delta \leq t + \Delta$. Let t_1 be the beginning of the first slot that begins at time $t + \Delta$ or later. Since slots last Δ, we have $t_1 < t + 2\Delta$. At the beginning of every slot that begins at time t_1 or later, the main chain of every honest party has length at least h. Hence, if an honest validator wins the election in such a slot, they generate a block of height at least $h + 1$ at the beginning of the slot, and our honest party receives it within time Δ.

Let H be the set of indices of the honest parties. The probability that no honest validator wins the election in a given slot is at most

$$\prod_{j \in H}\left(1 - \frac{V_j}{V_{\mathsf{tot}}} \cdot \frac{W}{2^k}\right) \leq \exp\left(-\sum_{j \in H} \frac{V_j}{V_{\mathsf{tot}}} \cdot \frac{W}{2^k}\right) \leq \exp\left(-(1 - \alpha) \cdot \frac{W}{2^k}\right) = \exp\left(-(1 - \alpha) \cdot \frac{\Delta}{L}\right),$$

where we used Lemma 3, the fact that the honest parties hold at least a fraction $1-\alpha$ of all stake, and $L = \frac{2^k}{W} \cdot \Delta$. Now, consider the slots that begin in the time interval from t_1 to $t_1 + \kappa \cdot L/(1-\alpha)$ (both included). Since slots last Δ, their number N is greater than $\kappa \cdot L/((1-\alpha)\cdot\Delta)$. Since the elections in different slots are independent, the probability that no honest validator wins the election in any of these slots is at most

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

In other words, as in the proof of Theorem 10, except with probability $\exp(-\kappa)$, the honest parties generate a block of height at least $h+1$ within time $\kappa \cdot L/(1-\alpha)$ after time t_1. Our honest party receives this block within time Δ, i.e., by time

$$t_1 + \kappa \cdot \frac{L}{1-\alpha} + \Delta < t + 3\Delta + \kappa \cdot \frac{L}{1-\alpha}. \qquad \square$$

Compared to Theorem 10, the above bound contains an additional term Δ. This is the price of dividing time into slots: after receiving B, the honest parties may have to wait until the beginning of the next slot before they can try to extend it.

Chain quality. Finally, we prove the proof-of-stake analogue of Theorem 11. Unlike common prefix, chain quality concerns a single chain, so the adversary does not benefit from signing several blocks for the same slot, and we obtain the same bound as in proof of work.

Theorem 16. *Consider a protocol execution in which the adversary controls a constant fraction $\alpha < 1$ of all stake in the system, the network has delay at most Δ, and the expected block generation time is $L = \Delta/\gamma$ for a constant $\gamma < 1$. Let $\eta > 0$ be a constant and suppose that*

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

Pick an honest party and suppose that their genesis block is safe. Consider any sequence S of ℓ consecutive blocks on the main chain of the honest party at the beginning of some slot. Then,

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

Proof. The proof follows that of Theorem 11, with slots playing the role of mining times, and it is actually slightly simpler. As in the proof of Corollary 3, we treat equivalent blocks as the same block. Let $q := (1-\eta)(1-\gamma)(1-\alpha)$. By Corollary 2, except with probability $M\cdot\exp(-\Omega(\ell)) + \epsilon^*$, every sequence of $m \geq \ell$ consecutive non-empty slots contains more than $q\cdot m$ safe slots. We show that, whenever this is the case, S contains at least $\frac{2q-1}{q}\cdot\ell$ honest blocks.

Let s be the slot at whose beginning we look at the main chain C of the honest party (this is the chain that the party extends if they win the election in slot s, so all its blocks were generated in earlier slots), and define u, B_0, h_0, B_1, h_1 and D as in the proof of Theorem 11 (if B_1 does not exist, $h_1 := |C| + 1$). Let s_0 and s_1 be the slots in which B_0 and B_1 were generated (if B_1 does not exist, we set $s_1 := s$). All blocks of D are descendants of B_0, and they are ancestors of B_1 or, if B_1 does not exist, they belong to the tree of the party at the beginning of slot s. Hence, they were generated in slots strictly between s_0 and s_1. Since they lie on the same chain, they were generated in distinct slots. Therefore, the number m of non-empty slots strictly between s_0 and s_1 is at least $|D| \geq \ell$, and more than $q\cdot m$ of these slots are safe.

As in the proof of Corollary 3, the block of every safe slot strictly between s_0 and s_1 has height greater than h_0. Its height is also smaller than h_1: if B_1 exists, the party that generated B_1 at the beginning of slot s_1 had already received the block of the safe slot; otherwise, the honest party received it by the beginning of slot s, so its height is at most $|C|$. Since the blocks of different safe slots have different heights, we obtain

$$q\cdot m < \#\text{ safe slots strictly between } s_0 \text{ and } s_1 \leq |D|.$$

Unlike in the proof of Theorem 11, there is no additional term +1: since slots last Δ and we look at the main chain at the beginning of a slot, the party that generated B_1 (or the honest party itself) has already received the blocks of all the previous safe slots. Now, the blocks of D that are not honest were generated in distinct non-safe slots strictly between s_0 and s_1, so their number is smaller than $m - q \cdot m$. Putting everything together, and using $m < |D|/q$, we obtain

$$\begin{aligned}\#\text{ honest blocks in } S &= |D| - \#\text{ non-honest blocks in } D > |D| - (1-q)\cdot m > |D| - (1-q)\cdot\frac{|D|}{q} \\ &= \frac{2q-1}{q}\cdot|D| \geq \frac{2q-1}{q}\cdot\ell.\end{aligned}$$

□

5.6 Implementing the Randomness Beacon

In Section 5.4, we assumed that, at the beginning of every epoch, a randomness beacon reveals a uniformly random value a to all participants. We now discuss a simple way of implementing such a beacon using the blockchain itself: the randomness for the next epoch is derived from the VRF outputs contained in the blocks of the current epoch. A similar approach is used, for instance, in the Ouroboros Praos protocol.

The construction. The output a_1 of the beacon for the first epoch is a random value contained in the genesis block. For every epoch $e \geq 1$, let $B_1, \ldots, B_m$ be the blocks of epoch e that belong to the main chain, ordered by slot, and let $r_1, \ldots, r_m$ be the VRF outputs they contain. The output of the beacon for epoch $e+1$ is

$$a_{e+1} := \mathsf{Hash}(r_1 \| r_2 \| \ldots \| r_m),$$

where $\|$ denotes concatenation and Hash is modelled as a random oracle.[4]

Two remarks are in order. First, all honest parties must obtain the same value a_{e+1}, so they must agree on the blocks of epoch e that belong to the main chain. By the common prefix property (Theorem 13), this is the case for the blocks that are buried deep enough in the main chain at the beginning of epoch $e+1$. For this reason, in practice, the beacon only uses the blocks generated in the first part of each epoch. To keep things simple, we ignore this issue and assume that, at the beginning of epoch $e+1$, all honest parties agree on the blocks of epoch e that belong to the main chain.

Second, the VRF keys used in epoch $e+1$ must be fixed before anybody can influence a_{e+1}. Otherwise, a corrupted validator could try many VRF keys and register one that wins many slots in epoch $e+1$ (this is an example of a *grinding attack*). Since the blocks that determine a_{e+1} are generated during epoch e, the stake and the keys used in the elections of epoch $e+1$ must be committed before the beginning of epoch e. This is why, in Section 5.4, we required the stake to be committed one epoch in advance. With this precaution, the VRF keys are fixed before the output of the beacon is determined, as required in the proof of Theorem 12.

How the beacon can be biased. The adversary cannot predict the VRF outputs of the honest validators before they publish their blocks. It cannot choose the VRF outputs of its own blocks either: these are determined by its VRF keys and by the slots, and the soundness of the VRF prevents it from proving any other value. However, the adversary can decide which of its blocks end up on the main chain, and this gives it some control over a_{e+1}. Suppose, for instance, that the last slot of epoch e in which some validator wins the election is won by a corrupted validator only. Before publishing its block, the adversary can compute two candidate values for a_{e+1}: one that includes the VRF output of its block, and one that does not. Since Hash is a random oracle, the two candidates are independent and uniformly random. The adversary, however, sees

[4] In practice, one also includes $e+1$ and a_e in the input of the hash function. This guarantees that the inputs used in different epochs are always different, even if, for instance, two epochs contain no blocks.

both of them, and it can pick the one it prefers (for instance, the one that maximises the number of slots it wins in epoch $e+1$) by publishing or withholding its block. More generally, if the last j slots of epoch e in which some validator wins the election are all won by corrupted validators only, the adversary can publish any subset of the corresponding blocks (each of them can point to the previous block of the subset or, for the first one, to the last block of the main chain). Hence, it can choose among 2^j candidate values for a_{e+1}. The adversary can also try to bias the beacon in other, more complex ways, which may give it even more options.

Quantifying the bias. To quantify the power of the adversary, we introduce the following model.

Definition 40 (Biasable beacon)**.** *Let $R \geq 1$ be an integer. We say that the randomness beacon is* R-biasable *if, in every epoch, its output is chosen as follows: R candidate values $a^{(1)}, \ldots, a^{(R)}$ are sampled independently and uniformly at random from $\{0,1\}^\lambda$, independently of everything else; the adversary sees all of them and chooses one, which becomes the output of the beacon for that epoch.*

The ideal beacon that we assumed so far is 1-biasable. The next lemma shows that, with an R-biasable beacon, the probability of an event that concerns j epochs increases by a factor of at most R^j.

Lemma 8. *Suppose that the randomness beacon is R-biasable, and let E be an event that only depends on the elections in j given epochs and on the blocks published in these epochs. Suppose that, for every adversary, E occurs with probability at most p when the outputs of the beacon in these j epochs are uniformly random (whatever happens in the other epochs). Then, for every adversary, E occurs with probability at most $R^j \cdot p$.*

Proof. Let $\mathcal{A}$ be any adversary. For every choice of indices $i_1, \ldots, i_j \in \{1, \ldots, R\}$, consider the adversary $\mathcal{A}_{i_1,\ldots,i_j}$ that behaves exactly like $\mathcal{A}$, except that, in the k-th of the j epochs, it always chooses the i_k-th candidate $a^{(i_k)}$ as the output of the beacon, regardless of what $\mathcal{A}$ would choose. Since the candidates are independent of everything else, the outputs of the beacon in the j epochs are uniformly random when we run $\mathcal{A}_{i_1,\ldots,i_j}$, so E occurs with probability at most p. Now, imagine running $\mathcal{A}$ and all the adversaries $\mathcal{A}_{i_1,\ldots,i_j}$ with the same randomness (the same candidates, the same keys, the same random coins, and so on). If $\mathcal{A}$ chooses the candidates with indices $i_1, \ldots, i_j$, the execution with $\mathcal{A}$ coincides with the execution with $\mathcal{A}_{i_1,\ldots,i_j}$. Hence, whenever E occurs in the execution with $\mathcal{A}$, it also occurs in the execution with one of the R^j adversaries $\mathcal{A}_{i_1,\ldots,i_j}$. By the union bound, E occurs with probability at most $R^j \cdot p$. □

The bound of Lemma 8 is essentially tight: if, for each candidate, the event E occurs independently with probability p, an adversary who picks a candidate for which E occurs (if there is one) succeeds with probability $1-(1-p)^R$, which is close to $R \cdot p$ when $R \cdot p$ is small.

For example, let μ be the expected number of slots of an epoch in which at least one corrupted validator wins the election, when the output of the beacon is uniformly random. In this case, the elections in different slots are independent so, by the Chernoff bound (see Theorem 1), the adversary wins in at least $(1+\delta)\mu$ slots of the epoch with probability at most $\exp(-\delta^2\mu/3)$, for every $\delta \in [0,1]$. By Lemma 8, with an R-biasable beacon, this probability is at most $R \cdot \exp(-\delta^2\mu/3)$. Setting $\delta := \sqrt{3(\ln R + \kappa)/\mu}$ (and assuming that this is at most 1), we conclude that, except with probability $\exp(-\kappa)$, the adversary wins in fewer than $(1+\delta)\mu$ slots of the epoch. Roughly speaking, biasing the beacon has the same effect as increasing the stake of the adversary by a factor $1+\delta$, which is close to 1 when the epochs are long (i.e., when μ is much larger than $\ln R + \kappa$).

To get a feeling for the number of options of the adversary, let us estimate it for the attack described above, in which the adversary withholds some of its blocks at the end of the epoch (for simplicity, we ignore the rare slots won by several corrupted validators). Let J be the number of consecutive slots at the end of epoch e that are won by corrupted validators only, among the slots in which some validator wins the election. As we have seen, this gives the adversary 2^J options. By the computation in the proof of Theorem 12, every slot in which some validator wins the election is won by corrupted validators only with probability at most $\beta := 1-(1-\alpha)(1-\gamma)$, independently of the other slots. Hence, if a_e is uniformly random, we have

$\Pr[J \geq j] \leq \beta^j$ for every $j \geq 0$: the adversary rarely has many options. What if the adversary also biased a_e? Suppose that it could choose a_e among 2^{j_0} candidates. By Lemma 8, we obtain

$$\Pr[J > j_0] \leq 2^{j_0} \cdot \beta^{j_0+1} \leq (2\beta)^{j_0}.$$

Hence, if $\beta < 1/2$, which is exactly the assumption $(1-\alpha)(1-\gamma) > 1/2$ of Theorem 13, the bias does not accumulate over time: if the adversary has at most 2^{j_0} options in some epoch, then it has at most 2^{j_0} options in the next epoch, except with probability $(2\beta)^{j_0}$. Keep in mind, however, that this is only one possible attack.

Security with a biased beacon. We now explain how to adapt the security analysis of Section 5.5 to an R-biasable beacon. In our proofs, the randomness of the elections is only used in Corollary 2 (through Theorem 12) and in Theorem 15, so we only need to adapt these two results. The price we pay is a factor R^2 in the failure probabilities. Moreover, we need the epochs to be long, so that every epoch contains many non-empty slots, and the bias to be small compared with the length of an epoch.

Corollary 4. *Consider the setting of Corollary 2, and suppose that the randomness beacon is R-biasable and that every epoch consists of N slots, where $\gamma N \geq C \cdot (\ell + \ln R)$ for a sufficiently large constant C that only depends on α. Then, except with probability $M \cdot R^2 \cdot 2^{-\Omega(\ell)} + \epsilon^*$, every sequence of $m \geq \ell$ consecutive non-empty slots contains more than $(1-\eta)(1-\gamma)(1-\alpha) \cdot m$ safe slots.*

Proof. Let $q := (1-\eta)(1-\gamma)(1-\alpha)$. As in the proof of Theorem 12, except with probability ϵ^*, we can analyse the execution as if the VRF evaluations were truly random.[5]

First, it suffices to consider sequences of m consecutive non-empty slots with $\ell \leq m < 2\ell$. Indeed, every longer sequence can be split into consecutive sequences of this kind and, if each of them contains more safe slots than q times its length, so does the whole sequence.

Second, every epoch contains at least 2ℓ non-empty slots, except with probability $M \cdot 2^{-\Omega(\ell)}$. Indeed, consider any epoch. As in the proof of Theorem 15, if the output of the beacon is uniformly random, in every slot some honest validator wins the election with probability at least $1 - \exp(-(1-\alpha)\gamma) \geq (1-\alpha)\gamma/2$ (since $e^{-x} \leq 1 - x/2$ for $x \in [0,1]$), independently of the other slots. Hence, the number of slots of the epoch won by some honest validator has expectation at least $(1-\alpha)\gamma N/2$, which is at least 4ℓ if C is large enough. By the Chernoff bound (with $\delta = 1/2$), fewer than 2ℓ slots of the epoch are won by honest validators with probability at most $\exp(-(1-\alpha)\gamma N/16)$. Since these slots are non-empty, by Lemma 8, with an R-biasable beacon, the epoch contains fewer than 2ℓ non-empty slots with probability at most $R \cdot \exp(-(1-\alpha)\gamma N/16)$. This is at most $e^{-\ell}$ if C is large enough, since $\gamma N \geq C \cdot (\ell + \ln R)$. We conclude by the union bound over the fewer than M epochs.

Third, if every epoch contains at least 2ℓ non-empty slots, any sequence of fewer than 2ℓ consecutive non-empty slots lies within two consecutive epochs. Fix a slot u and an integer m with $\ell \leq m < 2\ell$, and let $E_{u,m}$ be the event that the first m non-empty slots starting from slot u lie in the epoch of slot u and in the next one, and that at most $q \cdot m$ of them are safe. This event only depends on the elections in two given epochs and, by Theorem 12 (with $\delta = \eta$), it occurs with probability at most $\exp(-\eta^2(1-\gamma)(1-\alpha)m/2) = 2^{-\Omega(\ell)}$ when the outputs of the beacon are uniformly random. By Lemma 8, it occurs with probability at most $R^2 \cdot 2^{-\Omega(\ell)}$. Since every sequence of m consecutive non-empty slots consists of the first m non-empty slots starting from its first slot, we conclude by the union bound over the fewer than M choices of u and the ℓ choices of m. □

Theorem 17. *Consider the setting of Theorem 15, and suppose that the randomness beacon is R-biasable and that every epoch consists of N slots. If $\kappa \cdot L/(1-\alpha) \leq (N-1) \cdot \Delta$, then, except with probability $R^2 \cdot \exp(-\kappa) + \epsilon^*$, the honest party extends their main chain by a new block by time*

$$t + 3\Delta + \kappa \cdot \frac{L}{1-\alpha}.$$

[5] The term $T \cdot 2^{-\lambda}$ in ϵ^* bounds the probability that the adversary predicts an output of the beacon. Since the adversary now sees R candidate outputs in every epoch, this term should be multiplied by R. This does not affect our conclusions as long as $R \cdot T$ is much smaller than 2^λ.

Proof. In the proof of Theorem 15, the slots that begin in the time interval from t_1 to $t_1 + \kappa \cdot L/(1-\alpha)$ only depend on t. There are at most $\kappa \cdot L/((1-\alpha)\cdot\Delta) + 1 \leq N$ of them, so they lie in at most two consecutive epochs. The event that no honest validator wins the election in any of these slots only depends on the elections in these two epochs, and its probability is at most $\exp(-\kappa)$ when the outputs of the beacon are uniformly random. By Lemma 8, its probability is at most $R^2 \cdot \exp(-\kappa)$. The rest of the proof is unchanged. □

The other results of Section 5.5 only use the randomness of the execution through Corollary 2. Hence, with an R-biasable beacon, Corollary 3 and Theorem 16 still hold under the assumptions of Corollary 4 (applied with $\lceil \ell/2 \rceil$ in place of ℓ in the case of Corollary 3), with $M \cdot 2^{-\Omega(\ell)}$ replaced by $M \cdot R^2 \cdot 2^{-\Omega(\ell)}$. The same holds for finality (Theorem 14), if we derive it from Corollary 3 rather than from Theorem 13. We do not discuss Theorem 13, whose proof we omitted. Notice that these bounds are meaningful only if $\ln R$ is much smaller than ℓ: intuitively, every bit of bias in the beacon must be compensated by a constant number of additional blocks of confirmation depth.

5.7 Joining the Protocol

Like proof-of-work consensus, our proof-of-stake protocol is permissionless: validators can join and leave the protocol without restarting it. In this section, we explain how this works, following the discussion of Section 3.2.6. We will see that joining a proof-of-stake protocol raises a new threat, which does not arise in proof of work.

Becoming a validator. To take part in the elections, a newcomer needs some stake, which they can acquire, for instance, by buying coins from other users. Then, they commit their stake on the blockchain, together with their signature verification key, their VRF key, and the signature and the VRF evaluation described in Section 5.4. Recall that the stake used in the elections of an epoch must be committed before the beginning of the previous epoch (we saw the reason in Section 5.6). Hence, the newcomer starts taking part in the elections between one and two epochs after committing their stake (other designs require shorter or longer delays, see Section 5.8). This delay has another useful consequence: at the beginning of every epoch, the stake used in its elections was committed at least one epoch earlier, so the blocks containing it are buried deep in the main chain. Hence, if the epochs are long enough, all honest parties agree on these blocks by the common prefix property.

Leaving the protocol is also slightly more complex than in proof of work. A validator who simply goes offline keeps being elected, but the slots they win remain empty, which slows down the growth of the chain (for this reason, several protocols penalise validators who remain inactive). A validator who wishes to leave should therefore withdraw their stake with a transaction on the blockchain. They stop being elected once the withdrawal takes effect.

Learning the state of the chain. As in proof of work, the newcomer must also learn the current state of the chain, and the same precautions apply: the newcomer should query the whole system, ask for whole chains rather than just for the most recent blocks, and combine all the answers into a single graph. In proof of work, the newcomer then picks the hardest branch departing from the genesis block. The natural analogue in proof of stake would be to pick the longest branch. Unfortunately, this is not secure because of the attacks that we discuss next. Notice that the security analysis of Section 5.5 does not capture these attacks: it implicitly assumes that the stake of every validator remains the same throughout the execution, and that every honest party receives every block within time Δ from its generation, which is not the case for a party that joins the protocol later.

Long-range attacks. In proof of work, creating a block requires a lot of computation. To convince the honest parties to switch to an alternative chain, the adversary must produce a chain with more work than the main chain which, as we saw in Section 3.2.5, is infeasible if the adversary controls a small enough fraction of

the computational power. Moreover, the further back in the past the alternative chain departs from the main chain, the more work the adversary must redo. In proof of stake, instead, creating a block costs nothing: the adversary only needs to win sufficiently many elections on the alternative chain. Crucially, the outcome of these elections depends on the stake recorded on the alternative chain itself, which the adversary can influence.

In a *long-range attack*, the adversary picks a block B far in the past (for instance, many epochs ago) and secretly builds a private chain departing from B. Its goal is to make the private chain longer than the main chain, so that, for instance, newcomers adopt it. On the private chain, the adversary can rewrite history at will: for instance, it can remove the payments that it made on the main chain and spend the same coins again. At first sight, the attack seems hopeless. Let $\alpha < 1/2$ be the fraction of all stake that the adversary controls at the time of B. Right after B, the stake recorded on the private chain is the same as on the main chain. Hence, the adversary can extend the private chain only in the slots won by corrupted validators, which are roughly a fraction α of the slots in which some validator wins the election, while the honest validators extend the main chain in roughly a fraction $1 - \alpha$ of them. Since $\alpha < 1/2$, the private chain grows more slowly than the main chain. Over a long period of time, however, the adversary can increase its share of the stake on the private chain, for instance, in the following ways.

- *Stake bleeding.* The adversary is the only validator producing blocks on the private chain, so it collects the fees of all the transactions it includes there (see Section 4.4). Moreover, a transaction does not specify the chain it belongs to, so the adversary can copy the transactions of the main chain into its private chain and collect their fees. In this way, on the private chain, stake slowly flows from the honest parties to the adversary (on the main chain, instead, the fees are collected by the honest validators, assuming that the adversary does not produce blocks there). Once the share of the stake of the adversary on the private chain exceeds (roughly) the share $1 - \alpha$ of the honest validators on the main chain, the private chain starts growing faster than the main chain. If the adversary waits long enough, the private chain eventually becomes longer. This attack works for any constant fraction $\alpha > 0$.

- *Posterior corruption.* Some validators who held a large share of the stake at the time of B may have withdrawn their stake since then. Their old keys are now worthless to them, so they have little reason to protect them (or they may even be willing to sell them). On a private chain departing from B, however, these keys still control the stake that their owners held at the time of B. Hence, an adversary who obtains the keys of validators that held a large share of the stake at the time of B can make the private chain grow faster than the main chain. Notice that this does not contradict our assumption that the adversary controls at most a fraction α of all stake: at the time of the corruption, the corrupted validators no longer hold any stake.

Mitigations. Proof-of-stake protocols combine several countermeasures against long-range attacks.

- *Forward security.* Posterior corruption is one of the reasons why we use updatable signatures (see Section 5.3). Since the honest validators erase their old secret keys, an adversary who corrupts them later cannot sign blocks for past slots, so it cannot use their old stake in the slots between B and the time of the corruption. This makes posterior corruption much less effective, although it does not rule it out completely: the adversary can still use the keys it obtains to sign blocks for the current and future slots of its private chain. Moreover, nobody can check whether a validator actually erased their old keys.

- *Bounded reorganisations.* A party who has been online all the time knows the main chain, so they can simply refuse to switch to any chain that departs from their main chain more than ℓ blocks in the past, for a suitable integer ℓ. By the common prefix property (Theorems 13 and 14), if ℓ is large enough, the honest parties never need to abandon more than ℓ blocks of their main chain in the executions covered by our analysis, except with small probability. This rule protects the parties who are always online, but it does not help newcomers, nor parties who have been offline for a long time, since they do not know the main chain yet.

- *Trusted checkpoints.* A newcomer can obtain a recent block of the main chain (a *checkpoint*) from sources they trust (for instance, several independent websites, or the software they install) and discard all chains that do not contain it. This is the approach followed by Ethereum, where it is known as *weak subjectivity*: the newcomer needs to trust some information obtained outside of the protocol but, after that, they can verify everything else by themselves. The checkpoint must be recent compared with the time it takes to withdraw stake, so that the validators who held stake after the checkpoint still hold it. A party who has been offline for a long time also needs a new checkpoint.

- *Density-based chain selection.* The Ouroboros Genesis protocol modifies the longest chain principle as follows. When two chains depart from each other only in their last few blocks, the parties pick the longest one, as usual. When the two chains depart from each other further in the past, the parties instead compare the numbers of blocks that the two chains contain in a window of slots right after the point where they depart from each other, and pick the chain with more blocks. Right after that point, the stake recorded on the two chains is the same so, if the honest validators hold the majority of the stake, the main chain contains more blocks in this window than the private chain (with high probability, if the window is long enough), regardless of what happens later. Under suitable assumptions (including forward security), this allows newcomers to join the protocol knowing only the genesis block, as in proof of work.

- *Context-sensitive transactions.* If every transaction contains the hash of a recent block, and is valid only on chains that contain that block, the adversary can no longer copy the transactions of the main chain into its private chain. This limits stake bleeding.

5.8 Advantages and Disadvantages of Proof-of-Stake Consensus

We conclude this chapter by comparing proof-of-stake consensus with proof-of-work consensus, following the discussion of Section 3.2.7. Before doing so, let us put our construction into context.

Other proof-of-stake designs. The construction of Section 5.4 is only one of many proof-of-stake designs, and the randomness beacon of Section 5.6 is only one of many ways of generating randomness. Many designs address some of the issues discussed in this chapter in different ways. For instance, Ethereum uses a different design: the proposer of each slot is computed from the output of a randomness beacon called RANDAO and is publicly known in advance, so VRFs are not needed (but the proposers are exposed to denial-of-service attacks). Moreover, instead of following the longest chain principle, the parties follow the branch supported by the votes of the largest amount of stake, and blocks become final once validators holding at least two thirds of the stake have voted for them, in a way reminiscent of Bracha's protocol (see Theorem 4). The time that newcomers must wait before taking part in the elections also varies: for instance, it is much shorter in Algorand, where blocks are final as soon as they are added to the chain, while, in Ethereum, newcomers also wait in a queue, which limits the number of validators who can join at the same time.

Fees, rewards and the economy. Like proof-of-work blockchains, proof-of-stake blockchains are usually equipped with the fee and reward mechanisms discussed in Section 4.4: users pay fees for their transactions, and the validators collect (part of) these fees, together with rewards consisting of newly created coins. For instance, Ethereum kept the fee mechanism of Section 4.4 when it switched to proof of stake, and it rewards its validators not only for proposing blocks but also for voting. Moreover, proof-of-stake blockchains typically run a whole economy, based either on the UTXO system (Section 4.1) or on the account-based system (Section 4.2), and often support smart contracts (Section 4.3). For instance, Ethereum also kept its account-based economy and its smart contracts, while Cardano uses an extension of the UTXO system that supports smart contracts. In these systems, the stake is cryptocurrency recorded on the blockchain itself (for instance, coins locked in a special deposit, or delegated to a stake pool), so the fees and rewards collected by the validators increase their stake.

Energy costs. The main advantage of proof of stake is that it does not require the validators to waste computational power. In our construction, checking whether one has won the election in a slot only requires a single VRF evaluation. In general, running a validator requires roughly the same resources as running an ordinary server, and the security of the protocol does not depend on how much energy the validators consume. As discussed in Section 3.2.7, the energy consumption of Ethereum dropped to virtually zero when it switched from proof of work to proof of stake.

Communication complexity. As in proof of work, only the winners of the elections need to distribute their blocks, so the total communication per block grows linearly with the number of validators.

Permissionlessness and fluidity. Proof of stake is still permissionless: anybody who acquires stake can become a validator, and Sybil attacks do not help, since introducing virtual parties does not increase the stake of the attacker. However, the system is less fluid than in proof of work. First, newcomers take part in the elections only after a delay (between one and two epochs in our construction). Second, as we saw in Section 5.7, joining the protocol securely may require trusting a recent checkpoint while, in proof of work, anybody can identify the hardest chain starting from the genesis block alone. Third, the resource that grants influence is internal to the system: computational power can be bought from any hardware manufacturer, while stake can only be acquired from those who already hold it (or earned as fees and rewards by taking part in the protocol). As a consequence, the initial distribution of the stake, which is recorded in the genesis block, has a long-lasting influence on the system.

Inequalities. As in proof of work, the influence of each validator, as well as their rewards, is proportional to their resources, so the considerations of Section 3.2.7 apply to proof of stake too. Some differences are, however, worth noticing. In proof of work, large miners benefit from economies of scale (for instance, they can access cheaper energy and specialised hardware), so their profits may grow more than proportionally with their investment. In proof of stake, instead, the rewards are proportional to the stake, regardless of the size of the validator. On the other hand, when the rewards consist of newly created coins, the users who do not take part in the elections see their share of the stake shrink over time. For this reason, many users who do not want to run a validator themselves delegate their stake to validators run by others, called *stake pools* (see Section 6.3), in exchange for a share of the rewards. Some of the largest stake pools are run by large companies, such as cryptocurrency exchanges, which concentrates the influence over the protocol in the hands of a few operators.

Security assumptions. Proof of stake typically relies on more assumptions than proof of work. Our construction requires synchronised clocks, since time is divided into slots, and additional cryptographic primitives, namely, VRFs and updatable signatures. Moreover, the honest validators must erase their old secret keys, something that nobody can check, and the protocol needs a randomness beacon which, as we saw in Section 5.6, the adversary can bias to some extent. Finally, newcomers may need to trust a recent checkpoint. Other designs replace some of these assumptions with different ones (for instance, Ethereum needs neither VRFs nor updatable signatures, but it relies on trusted checkpoints). The security analysis is also considerably more involved: because of the nothing-at-stake problem, we did not prove Theorem 13, and we only proved the weaker Corollary 3, which requires a larger fraction of honest stake than its proof-of-work analogue (Theorem 8).

Cost of attacks and accountability. In proof of work, the resources of the adversary (hardware and energy) lie outside the system. In proof of stake, instead, to attack the system, the adversary must own a large fraction of the cryptocurrency itself, whose value is likely to collapse if the attack succeeds. Moreover, some forms of misbehaviour leave evidence that everybody can verify: for instance, two different blocks signed by the same validator for the same slot. Several proof-of-stake protocols, including Ethereum, punish such misbehaviour by destroying part of the stake of the culprit (this is called *slashing*). Slashing also mitigates the nothing-at-stake problem, since signing several blocks for the same slot is no longer free. Nothing similar

is possible in proof of work, where the miners are anonymous and their resources lie outside the system. Notice, however, that these arguments are economic rather than cryptographic, and that they do not apply to long-range attacks, in which the adversary may use stake that was withdrawn a long time ago.

Today, Bitcoin still relies on proof of work, while Ethereum and many of the blockchains created in recent years rely on proof of stake.

Chapter 6

Scalability

Over the years, the most popular blockchains have grown to a huge size. Today (2026), both Bitcoin and Ethereum are populated by hundreds of millions of users. Maintaining such systems is no easy task. Since the hope is that these systems will continue to grow, it is fundamental that their maintenance costs do not grow too quickly with the size of their population. We have already seen that one of the greatest advantages of PoW-based consensus over solutions based on Byzantine agreement is that the generation of each block does not require every validator to speak, but only the winner. This is probably the most fundamental aspect that allowed blockchains to be deployed at the scale we see today. Imagine how congested the network would become if millions of validators had to send multiple messages to everybody else for each block generated! Notice that the total amount of communication would grow at least quadratically with the population size! With PoW-based consensus (and also with proof-of-stake consensus, see Chapter 5), the communication complexity instead grows linearly with the population size (which is optimal, as everybody needs to receive the new block).

PoW-based consensus, however, only solves the communication issue. What about computation? Every time a block is generated, every validator needs to verify its consistency. This is no easy task, especially if done naively. Each block can contain hundreds, if not thousands, of transactions. Some of these may be particularly hard to process, either because the validators need to search the huge history of the system (which is several hundred gigabytes large) to find the outputs being spent and make sure that they have not been spent yet (in the UTXO model), or because the transaction is a smart contract call requiring a particularly heavy computation. Ideally, we would like a solution in which only the block proposer invests computation to derive the new block, while verifying it comes almost for free. As a whole, such a system would require less computation by a factor of hundreds of millions!

6.1 Light and Heavy Nodes

Imagine that you issue a transaction on a blockchain and you would like to check whether it has succeeded. How do you do that? An insecure solution would be to contact a validator and ask them whether the transaction succeeded. It is easy to see why this is not a good idea: what if the validator lies?

A better, but still insecure, solution is to ask the validator to send their whole chain. Then, the user can verify it and check that their transaction has been added to it. This is still a bad solution: first of all, it requires you to download the whole chain (which is hundreds of gigabytes large) and to verify it (which is an extremely heavy computation). This will be extremely slow and will drain the battery of your poor phone or laptop. Even worse, you need to repeat this every time you issue a transaction. On top of that, this is still insecure: what if the validator is trying to trick you and just provided you with a side branch of their chain? It may be that your transaction has not yet been added to the main chain. However, if you talk to only a single validator, you have no way to check whether what you received is the main chain or just a side branch.

A secure solution is to ask all validators to send you their main chain. Among the chains you receive, you discard those that do not stem from the genesis block (let us assume that you know which block this is; e.g., you can easily check it on the Internet) and those that fail verification (perhaps because they contain invalid transactions). Then, you select the main chain as explained in Sections 3.2.6 and 5.7 (for instance, in proof of work, you pick the chain that required the most computation to be generated). You can be sure that this is the real main chain! To check whether your transaction has succeeded, you just need to check whether it appears on the main chain. The issue is that, by the time you are done, several hours have passed and your phone is out of battery.

Exercise 26. *Why would it be insecure to just ask all validators whether the transaction has succeeded (without asking them to send the whole blockchain) and then take a majority vote?*

To solve this issue without compromising security, systems typically split their population into two groups: *heavy nodes* (also called *full nodes*) and *light nodes*, which correspond to everyday users like you and me. Heavy nodes are servers (or server farms) with huge amounts of computational power. They store the whole blockchain, and verify and process all transactions added to the chain. Every validator is a heavy node: to propose new blocks, validators need to know the whole chain and to check the validity of the transactions they include. However, not every heavy node is a validator: for instance, cryptocurrency exchanges, companies offering services to light nodes, and users who prefer to verify everything by themselves often run heavy nodes without taking part in the elections. Light nodes, instead, are small devices like phones and laptops. They do not take part in the creation of new blocks: in proof of work, they have too little computational power to have any chance of winning the election and, in proof of stake, their owners can delegate their stake to a validator (see Section 5.8). Light nodes do not even store the whole chain, but only *the headers* of the blocks.

The crucial property that makes light nodes possible is that, in both our consensus protocols, the validity of the *header* of a block, i.e., the block without its content x and its metadata y, can be checked without knowing x and y. Indeed, x and y only enter the validity checks through their digest z.

- In our proof-of-work protocol (see Section 3.2.2), a block is a tuple (c, z, x, y, h), where c is a counter, $z = \mathsf{Hash}'(x, y)$ is the digest of the content and of the metadata, and h is a pointer to the previous block. The header is (c, z, h), and checking it consists of verifying that $\mathsf{Hash}(c, z, h) < W$, where W is the hardness parameter.

- In our proof-of-stake protocol (see Section 5.4), a block is a tuple $(h, t, z, x, y, \mathsf{pk}, r, \pi, s)$, where $z = \mathsf{Hash}(x, y)$. The header is $(h, t, z, \mathsf{pk}, r, \pi, s)$, and checking it consists of verifying the signature s on (h, z) for period t, the VRF proof π, and the election condition $r < \frac{V}{V_{\text{tot}}} \cdot W$.

Every time a light node receives a new block, it throws away x and y and checks the header. If this check succeeds, the light node appends the header to its block graph, linking it to the header indicated by the pointer h. Notice that the light node can check whether the new header points to a pre-existing header even if it knows neither the content nor the metadata of the latter, since the pointer is computed from the header of the previous block: $h = \mathsf{Hash}(\overline{c}, \overline{z}, \overline{h})$ in proof of work, and $h = \mathsf{Hash}(\overline{h}, \overline{t}, \overline{z}, \overline{\mathsf{pk}})$ in proof of stake, where the bars denote the components of the previous block. Moreover, the pointer is bound to the rest of the header: in proof of work, it is part of the input of the puzzle and, in proof of stake, it is signed by the block proposer. Hence, a header cannot be attached to a different predecessor without solving a new puzzle or producing a new signature. The light node then selects its main chain among the chains of headers stemming from the genesis block, exactly as a heavy node would (see Sections 3.2.6 and 5.7).

In proof of stake, there is one subtlety. To check the VRF proof and the election condition, the light node also needs the output a of the randomness beacon, as well as the VRF keys and the stake of the validators. The output of the beacon only depends on the VRF outputs contained in the headers of the previous epoch (see Section 5.6), but the keys and the stake are recorded in the content of earlier blocks. Hence, the light node needs to obtain this information from the heavy nodes, together with a guarantee that it is correct. One solution is to include in the headers a digest of the stake distribution, against which the light node can check the information it receives (we will see how in Section 6.2.2).

As long as the honest validators control the majority of the computational power (in proof of work) or of the stake (in proof of stake), the light nodes can be sure that their main chain is consistent with the main chain of every honest heavy node.

This solution ensures that light nodes do not need to use too many resources. The chain of headers is much smaller than the whole chain of blocks (tens of megabytes versus hundreds of gigabytes), so it requires less storage, and less bandwidth when it is sent. Moreover, verifying new headers is much cheaper than verifying new blocks since, unlike heavy nodes, light nodes neither validate transactions nor execute smart contracts. Light nodes do not need to check these, as they simply rely on the fact that honest validators never add invalid transactions or invalid smart contract executions to the main chain. Essentially, security is delegated to the heavy nodes.

How do light nodes check that their transaction (or even somebody else's) was added to the main chain? This is easy: they simply contact a few heavy nodes and ask whether the transaction has been added. If so, the heavy node can provide the content x and the metadata y of the block containing the transaction in question. The light node can then verify that x indeed contains the requested transaction and that the header of the block sent by the heavy node is included in its main chain (there is no need to check any of the other transactions in the block, nor any smart contract execution). As soon as it receives even a single such block, the light node can be sure that the transaction ended up on the chain, and it does not need to communicate with any other heavy node. Of course, there is the usual issue: the light node needs to wait for a number of new blocks to be created before it can be sure that the transaction is final; otherwise, there is a risk that the block containing the transaction ends up on a dead branch. If, instead, the heavy node does not provide such a block, it could be either that the heavy node is honest and the transaction has not been added to the chain yet, or that it is corrupted and does not want to tell the light node that the transaction succeeded. For this reason, the light node should reach out to some other random heavy node. We will see a much more efficient solution in Section 6.2.

To summarise, the light-node/heavy-node design allows us to push all heavy computation and security responsibilities onto servers with a lot of computational power. Everyday users are offered a secure infrastructure without draining their phone battery, using up all their data or congesting their Internet connection.

Digital Wallets. In practice, all light-node operations are managed by software called *digital wallets*. Typically, a wallet is in contact with a large set of heavy nodes: every time the light node receives or issues a transaction, the heavy nodes notify the wallet as soon as the transaction is added to the chain (and provide the full block, proving the correctness of their claim). As long as one of these heavy nodes is honest, the wallet will never miss any such transaction.

In addition, digital wallets offer other services, such as secure key management (both generation and storage) and a user-friendly interface to issue new transactions, view balances and interact with smart contracts. Wallets come in two types: *hot* and *cold*. Hot wallets are purely software-based and are connected to the Internet, while cold wallets also have a specialised hardware component that stores keys securely offline (typically, these look like USB pen drives). Cold wallets usually offer a higher degree of security, while hot wallets offer better usability. A good strategy is to store the keys of accounts holding large amounts of money in cold wallets, while the keys of accounts holding small amounts of money can be stored in hot wallets and used for everyday purchases.

Exercise 27 (Password reset mechanisms)**.** *What happens if somebody loses the secret key of their account? Why can't we have a "password reset" mechanism? What would be a potential security issue with a digital wallet offering such functionality? There are notorious examples of users who lost the secret keys of their accounts. Today, some of these accounts hold 8000 Bitcoins (around 600 million dollars at 2026 prices). Some of these people keep wasting their money paying cryptographers to try to recover their lost keys, without understanding that this is a lost cause.*

6.2 Merkle Trees

Suppose that a heavy node wants to prove to a light node that a given transaction belongs to a certain block. For instance, the light node may belong to the recipient of the transaction, who wants to make sure that a payment has been added to the chain before shipping some goods. In Section 6.1, we saw a naive solution: the heavy node sends the content x and the metadata y of the whole block. The light node then needs to

1. download the whole block, most of which it does not care about;
2. hash all of it, to check that the result matches the digest z contained in the corresponding header of its chain;
3. parse the content of the block until it finds the transaction it is interested in.

This is expensive. In 2026, a Bitcoin block typically contains around four thousand transactions and occupies between one and two megabytes, while an Ethereum block typically contains a few hundred transactions and occupies less than two hundred kilobytes. A single transaction, instead, typically occupies a few hundred bytes. Hence, to check a single transaction, the light node would need to download and hash hundreds or even thousands of times more data than it actually needs, and it would need to repeat this for every transaction it is interested in.

Merkle trees solve this problem. A Merkle tree is a particular way of hashing a list of elements (for instance, the transactions in a block) into a short digest, called the *root*. As with any hash function, the root *authenticates* the list: whoever knows the root can check whether a list they receive is the right one, by hashing it and comparing the result with the root. Merkle trees, however, offer an additional feature: given the list, one can compute a short proof, called an *opening*, showing that a given element appears at a given position of the list, and whoever knows the root can check this proof without knowing the rest of the list. Notice that a Merkle tree does not store the list: the list (together with its tree) is kept by whoever needs to prove things about it (in our example, the heavy node), while the verifier (the light node) only needs to know the root. Using Merkle trees, the heavy node can prove that the block contains the transaction by sending only the transaction and an opening, whose size grows only logarithmically with the number of transactions in the block. The light node does not need to trust the heavy node: if the opening is valid, the transaction is in the block, unless the heavy node managed to find a collision of the hash function.

6.2.1 The Construction

Let $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^k$ be a hash function (see Definition 14), for some parameter k: it takes as input binary strings of any length and outputs digests of exactly k bits. Let $x_1, \ldots, x_n \in \{0,1\}^*$ be the list that we want to authenticate. For simplicity, we assume that n is a power of two, i.e., $n = 2^d$ for some integer d (otherwise, we can pad the list with dummy elements, denoted by $\perp$, until its length becomes a power of two). We also assume that d is public, i.e., known to everybody; we will see later why this is important.

The tree. The Merkle tree of the list is a complete binary tree of depth d. We identify every node of the tree with a binary string of length at most d, which describes the path from the root to the node: the root is the empty string, which we denote by ε, and the left and right children of a node b are the strings $b0$ and $b1$ obtained by appending 0 and 1 to b, respectively. Hence, the nodes of level j are the binary strings of length j, and the leaves are the $n = 2^d$ binary strings of length d. We associate the elements of the list with the leaves from left to right, and we use the leaves to denote the positions of the list: for every $b \in \{0,1\}^d$, we denote by x_b the element at position b, i.e., $x_b := x_i$, where b is the binary representation of $i - 1$ (so, $x_{0\cdots0} = x_1$ and $x_{1\cdots1} = x_n$).

Hashing. Every node b has a digest, which we denote by v_b. The digests are computed from the bottom up:

- the digest of a leaf is the hash of the element at the corresponding position, i.e., $v_b := \mathsf{Hash}(x_b)$ for every $b \in \{0,1\}^d$;
- the digest of every other node is the hash of the digests of its two children, i.e., $v_b := \mathsf{Hash}(v_{b0}, v_{b1})$ for every binary string b of length smaller than d.

The digest of the root authenticates the list. We denote it by $\mathsf{MT.Root}(x_1, \ldots, x_n) := v_\varepsilon$, and we usually call it z, or simply the root. Figure 6.1 shows how the root is computed when $n = 8$, so that $d = 3$: every trapezoid labelled H is an evaluation of Hash, which takes as input the string below it (an element of the list, at the bottom level, or the concatenation of the digests of two children, at the other levels) and outputs the digest above it.

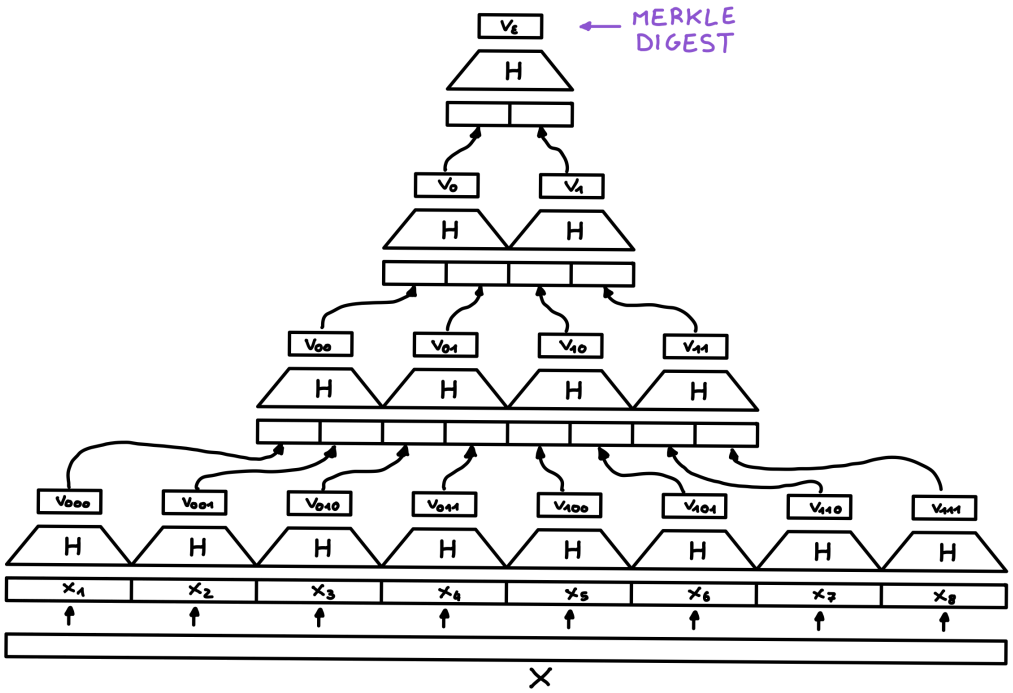

Figure 6.1: Computing the root of the Merkle tree of a list X of $n = 8$ elements $x_1, \ldots, x_8$ (so $d = 3$). Every trapezoid labelled H is an evaluation of Hash, which takes as input the string below it and outputs the digest above it. At the bottom level, the digest of every leaf b is the hash of the element at position b (for instance, $v_{000} = \mathsf{Hash}(x_1)$ and $v_{111} = \mathsf{Hash}(x_8)$). At every other level, the digest of every node b is the hash of the concatenation of the digests of its two children $b0$ and $b1$ (for instance, $v_0 = \mathsf{Hash}(v_{00}, v_{01})$). The digest v_ε of the root, i.e., the Merkle digest $\mathsf{MT.Root}(x_1, \ldots, x_8)$, authenticates the whole list.

Openings. The *sibling* of a node is the other child of its parent: the sibling of $b0$ is $b1$, and vice versa. Consider a position $b = b_1 b_2 \cdots b_d$, where $b_1, \ldots, b_d$ are bits. The path from the leaf b to the root consists of the prefixes of b, namely, $b_1 \cdots b_d$, then $b_1 \cdots b_{d-1}$, and so on, up to b_1 and ε. The opening of position b consists of the digests of the siblings of the nodes along this path (the root has no sibling):

$$\mathsf{MT.Open}(x_1, \ldots, x_n, b) := (s_d, \ldots, s_1), \qquad \text{where } s_j := v_{b_1 \cdots b_{j-1} \overline{b_j}} \text{ for every } j \in [d],$$

and $\overline{b_j} := 1 - b_j$ denotes the complement of the bit b_j. For instance, in Fig. 6.2, the opening of position 011, i.e., of the element x_4, consists of the blue digests: it is (v_{010}, v_{00}, v_1). Notice that, if the prover keeps the digests of all the nodes of the tree, computing an opening simply consists of reading d of them.

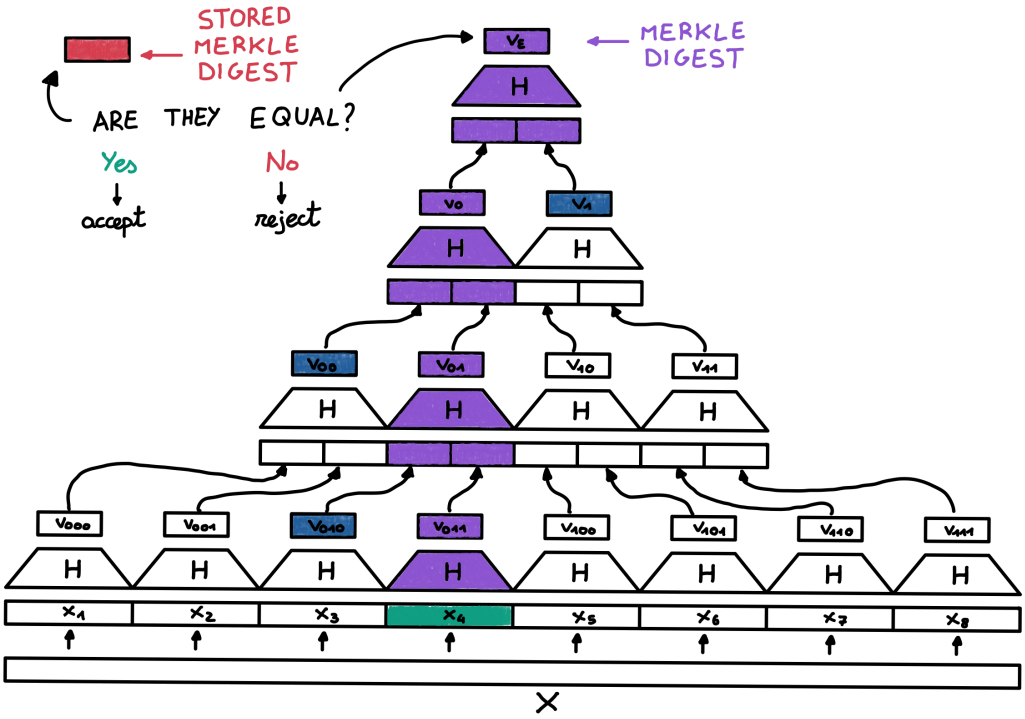

Figure 6.2: Opening position 011, i.e., the element x_4 (in green), in the Merkle tree of Fig. 6.1, and verifying the opening. The opening consists of the digests of the siblings of the nodes along the path from the leaf 011 to the root (in blue): v_{010}, v_{00} and v_1. To verify it, the verifier computes $\mathsf{Hash}(x_4)$ and then recomputes the digests of the nodes along the path from the bottom up (in purple), using the blue digests in place of the missing siblings: this requires $d + 1 = 4$ evaluations of Hash (the purple trapezoids). Finally, the verifier compares the digest obtained for the root with the Merkle digest that it stored (in red; for instance, the root contained in the header of a block): it accepts if they are equal, and rejects otherwise. The purple digests are also the only ones that change when x_4 is updated (see Section 6.2.2).

To check that a value x appears at position b of the list authenticated by the root z, given an opening $\pi = (s_d, \ldots, s_1)$, the verifier first checks that π consists of exactly d digests. Then, it recomputes the digests of the nodes along the path from the leaf b to the root, using the digests in the opening in place of the missing siblings. For $j = d, d-1, \ldots, 0$, we denote by u_j the digest that the verifier computes for the node $b_1 \cdots b_j$. The verifier starts by computing $u_d := \mathsf{Hash}(x)$. Then, for $j = d, d-1, \ldots, 1$, it computes

$$u_{j-1} := \begin{cases} \mathsf{Hash}(u_j, s_j) & \text{if } b_j = 0, \text{ i.e., if } b_1 \cdots b_j \text{ is a left child,} \\ \mathsf{Hash}(s_j, u_j) & \text{if } b_j = 1, \text{ i.e., if } b_1 \cdots b_j \text{ is a right child.} \end{cases}$$

Finally, the verifier accepts if and only if it obtains the root, i.e., $u_0 = z$. For instance, in Fig. 6.2, the verifier computes the purple digests $u_3 = \mathsf{Hash}(x_4)$, $u_2 = \mathsf{Hash}(v_{010}, u_3)$, $u_1 = \mathsf{Hash}(v_{00}, u_2)$ and $u_0 = \mathsf{Hash}(u_1, v_1)$ (labelled v_{011}, v_{01}, v_0 and v_ε in the picture, since the opening is honest), and compares u_0 with the stored root z (in red). We write $\mathsf{MT.Verify}(z, b, x, \pi) = 1$ if the verifier accepts, and $\mathsf{MT.Verify}(z, b, x, \pi) = 0$ otherwise. In the first case, we say that π is a *valid opening of position b to x*. If the opening is honest, i.e., $x = x_b$

and $\pi = \mathsf{MT.Open}(x_1, \ldots, x_n, b)$, the verifier recomputes exactly the digests of the nodes along the path, i.e., $u_j = v_{b_1 \cdots b_j}$ for every j, so it obtains $u_0 = v_\varepsilon = z$ and accepts.

The following theorem summarises the properties of Merkle trees.

Theorem 18. *Let* $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^k$ *be a hash function and let* $n = 2^d$.

1. (Efficiency.) *Computing* $\mathsf{MT.Root}(x_1, \ldots, x_n)$ *requires* $2n-1$ *evaluations of* Hash. *An opening consists of* $d = \log_2 n$ *digests, and verifying it requires* $d+1$ *evaluations of* Hash. *In particular, hashing the list takes time* $O(n)$, *while verifying an opening takes time* $O(\log n)$.

2. (Position binding.) *Given a root* z, *a position* $b \in \{0,1\}^d$, *two different values* $x \neq x'$, *a valid opening* π *of position* b *to* x, *and a valid opening* π' *of position* b *to* x', *one can find a collision of* Hash *with* $2(d+1)$ *evaluations of* Hash.

The second part of the theorem, which we call *position binding*, states that whoever manages to open the same position of a Merkle tree to two different values can also find a collision of Hash. Hence, if Hash is (T, ϵ)-secure, no algorithm running in time at most $T - t$, where t is the time needed to verify two openings, can produce two valid openings of the same position to two different values with probability greater than ϵ: otherwise, running this algorithm and then finding a collision as in the theorem would contradict the security of Hash. In particular, if z is the root of a list $x_1, \ldots, x_n$, nobody can efficiently produce a valid opening of a position b to a value $x' \neq x_b$, since, together with the honest opening of position b to x_b, this would give a collision.

Proof. The first part follows by counting. The tree contains $n = 2^d$ leaves and $1+2+\cdots+2^{d-1} = 2^d - 1 = n-1$ other nodes, and computing the digest of each node requires one evaluation of Hash. Verifying an opening requires one evaluation for the leaf and one for each of the d levels above it.

For the second part, we verify both openings, which requires $2(d+1)$ evaluations of Hash. Let $u_d, \ldots, u_0$ be the digests computed while verifying the opening $\pi = (s_d, \ldots, s_1)$ of position b to x, and let $u'_d, \ldots, u'_0$ be those computed while verifying the opening $\pi' = (s'_d, \ldots, s'_1)$ of position b to x'. Since both openings are valid, $u_0 = z = u'_0$. We distinguish two cases.

- If $u_d = u'_d$, then $\mathsf{Hash}(x) = \mathsf{Hash}(x')$. Since $x \neq x'$, the pair (x, x') is a collision.
- If $u_d \neq u'_d$, let j be the largest index such that $u_j = u'_j$. Such an index exists because $u_0 = u'_0$, and $j < d$. By the choice of j, we have $u_{j+1} \neq u'_{j+1}$. Since both openings concern the same position b, both verifications compute u_j and u'_j in the same way, depending on the bit b_{j+1}. If $b_{j+1} = 0$, then $u_j = \mathsf{Hash}(u_{j+1}, s_{j+1})$ and $u'_j = \mathsf{Hash}(u'_{j+1}, s'_{j+1})$. The inputs (u_{j+1}, s_{j+1}) and (u'_{j+1}, s'_{j+1}) are different strings, since $u_{j+1} \neq u'_{j+1}$ and all digests consist of exactly k bits, but they have the same digest $u_j = u'_j$: they form a collision. The case $b_{j+1} = 1$ is identical, with the inputs of Hash swapped.

In both cases, we find a collision of Hash. □

Exercise 28. *We can view* MT.Root *as a hash function that maps lists of* $n = 2^d$ *binary strings to digests of* k *bits. Show that position binding implies that this hash function is collision resistant. More precisely, given two different lists* $x_1, \ldots, x_n$ *and* $x'_1, \ldots, x'_n$ *such that* $\mathsf{MT.Root}(x_1, \ldots, x_n) = \mathsf{MT.Root}(x'_1, \ldots, x'_n)$, *show how to find a position* b, *two different values, and two valid openings of position* b *to these values. Conclude that one can find a collision of* Hash, *and count the evaluations of* Hash *that this requires.*

Why must the depth be public? In the proof of Theorem 18, both openings consist of exactly d digests, because the verifier knows d and rejects openings of any other length. If the verifier did not know d, a cheating prover could use openings that are shorter or longer than they should be. Consider, for instance, any node b of length $j < d$: its digest is the hash of the string obtained by concatenating the digests of its two children

$b0$ and $b1$. The prover could claim that this string of $2k$ bits is an element of the list at position b, as if the tree had depth j, and prove it with an opening consisting of only j digests, which the verifier would accept.[1]

Exercise 29. *Design a variant of Merkle trees in which the verifier does not need to know the depth d, i.e., in which positions can be binary strings of any length and the verifier accepts openings of any length. Show that your variant is secure: given a list $x_1, \ldots, x_n$ with $n = 2^d$, a position b and a valid opening of position b to a value x with respect to the root of the list, one can find a collision of Hash, unless b has length d and $x = x_b$. (Hint: make sure that the digest of a leaf cannot be mistaken for the digest of another node.)*

Merkle trees in blockchains. From now on, we assume that the content x of every block is a list of transactions $x_1, \ldots, x_m$, and that the digest z contained in the header of the block is

$$z := \big(d, \mathsf{MT.Root}(x_1, \ldots, x_m, y)\big),$$

where d is the depth of the Merkle tree of the list $x_1, \ldots, x_m, y$ (padded with dummy elements if needed). In this way, the depth of the tree is public. This replaces $z = \mathsf{Hash}'(x, y)$ in proof of work and $z = \mathsf{Hash}(x, y)$ in proof of stake. Nothing else changes: in particular, the headers can still be checked without knowing x and y (see Section 6.1). To prove that a block contains a transaction tx, a heavy node sends the position b of tx in the block and the opening π of position b. The light node retrieves the depth and the root from the header of the block in its chain, and checks that π is a valid opening of position b to tx (as usual, it also checks that the header is buried deep enough in its main chain). This only requires downloading and hashing the transaction and $O(\log m)$ digests, instead of the whole block.

Exercise 30. *In 2026, a Bitcoin block typically contains around* 4000 *transactions and occupies around* 1.7 *megabytes, while an Ethereum block typically contains around* 300 *transactions and occupies around* 180 *kilobytes. Both blockchains use hash functions with* 32*-byte digests, and a transaction typically occupies a few hundred bytes. Assume, for simplicity, that both blockchains use the Merkle trees of this section (Bitcoin actually uses very similar Merkle trees, while Ethereum uses a variant of the Merkle Patricia tries of Section 6.2.3). For each of the two blockchains, compare the amount of data that a light node needs to download and to hash to check that a block contains a given transaction, with the naive solution and with Merkle trees. By which factor do Merkle trees reduce this amount?*

Exercise 31 (Updatable signatures from Merkle trees)**.** *Let $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ be a digital signature scheme and let $M = 2^d$. In this exercise, we build an updatable signature scheme with M periods (see Definition 35). The idea is to generate M key pairs $(\mathsf{pk}_1, \mathsf{sk}_1), \ldots, (\mathsf{pk}_M, \mathsf{sk}_M)$ of the signature scheme, one for each period, and to use the root of the Merkle tree of the list $\mathsf{pk}_1, \ldots, \mathsf{pk}_M$ as the public key of the updatable signature scheme.*

1. *Describe the four algorithms $(\mathsf{KeyGen}', \mathsf{Update}', \mathsf{Sign}', \mathsf{Verify}')$ of the updatable signature scheme. What should the secret key for period t contain? (Hint: a signature for period t could contain pk_t and its opening.)*

2. *Explain, informally, why the scheme is forward secure (see Definition 36) if the signature scheme is unforgeable and the hash function is collision resistant.*

3. *How large are the public key, the secret keys and the signatures, as functions of M?*

6.2.2 Updating a Merkle Tree

In many applications, the list authenticated by a Merkle tree changes over time, and the root needs to be recomputed after every change. For instance, in an account-based blockchain (see Section 4.2), we may want

[1] This is what happens in Bitcoin, whose headers do not contain the depth of the Merkle tree: a transaction of exactly 64 bytes, i.e., the size of two digests, can be mistaken for the concatenation of the digests of the children of a node, and vice versa. This weakness, discovered in 2017, can be exploited to fool light nodes, although this requires a large amount of computation.

to authenticate the list of the balances of all accounts, by including the root of its Merkle tree in every block. This allows any light node to check the balance of an account (or, in proof of stake, the stake of a validator, see Section 6.1) with a single opening. However, every block only modifies the balances of a few accounts, while the total number of accounts may be huge, so recomputing the whole tree after every block would be wasteful.

Fortunately, this is not necessary. Suppose that we replace the element x_b at position b by a new value x'_b. The only digests that change are those of the nodes along the path from the leaf b to the root, i.e., of the prefixes of b: the digests of all other nodes, including the siblings of the nodes on the path, are computed from the same values as before. Hence, if we keep the digests of all the nodes, we only need to recompute the $d+1$ digests along the path, starting from $\mathsf{Hash}(x'_b)$ and moving up, exactly as in the verification of an opening. For instance, in Fig. 6.2, if x_4 changes, only the purple digests change, and we can recompute them as in the verification, starting from the new value of x_4 and using the blue digests. Notice that this only requires the new value x'_b and the opening of position b, which contains the digests of all the siblings along the path. Hence, anybody who knows the opening of position b can compute the new root, even without knowing the rest of the list. Moreover, the opening of position b remains valid after the update, since the digests of the siblings do not change. We summarise these facts in the following theorem, which we do not prove.

Theorem 19. *Let $n = 2^d$, let $x_1, \dots, x_n$ be a list, and let $x'_1, \dots, x'_n$ be the list obtained by replacing the elements at ℓ positions with new values.*

1. *If only the element at position b changes (i.e., $\ell = 1$), one can compute $\mathsf{MT.Root}(x'_1, \dots, x'_n)$ from the new value x'_b and from the opening of position b in the Merkle tree of $x_1, \dots, x_n$, with $d+1$ evaluations of Hash. Moreover, this opening is a valid opening of position b to x'_b with respect to the new root.*
2. *Given the new values and the digests of all the nodes of the Merkle tree of $x_1, \dots, x_n$, one can compute the digests of all the nodes of the Merkle tree of $x'_1, \dots, x'_n$ with at most $\ell(d+1)$ evaluations of Hash. In particular, updating ℓ elements takes time $O(\ell \log n)$.*

The same technique allows us to remove an element, by replacing it with the dummy element $\perp$, and to add a new element, by replacing a dummy element with it.

Proofs of update. The first part of Theorem 19 is particularly useful when the list is stored by a single party, while many other parties only store its root z. As for openings, we call the former the *prover* and the latter the *verifiers*: for instance, the prover may be a heavy node, and the verifiers may be light nodes. When the prover replaces the element at position b with a new value x'_b, it does not need to send the new list to the verifiers, nor to ask them to trust the new root that it computed. Instead, it can send them a short *proof of update*, consisting of the position b, the old value x_b, the new value x'_b, and the opening π of position b in the Merkle tree of the old list. Every verifier checks that π is a valid opening of position b to x_b with respect to z and, if this is the case, computes the new root z' from x'_b and π, as in the first part of Theorem 19, and replaces z with z'. The proof of update consists of two elements of the list and $d = \log_2 n$ digests, and the verifiers need $2(d+1)$ evaluations of Hash to check it and to compute the new root. This also works for removing and adding elements, which are special cases of replacements.

The verifiers do not need to trust the prover. By position binding (Theorem 18), a valid opening of position b with respect to z can only open it to the old value x_b. Moreover, by a similar argument, from two different valid openings of position b with respect to z, even to the same value, one can find a collision of Hash. Therefore, unless the prover can find a collision of Hash, π is the honest opening of position b, and z' is the root of the new list.

6.2.3 Accumulators

In many applications, we need to hash a *set* S into a short digest, in such a way that we can efficiently prove that an element belongs to S. A scheme that achieves this is called an *accumulator*. Merkle trees give

a trivial accumulator: we write the elements of S as a list $x_1, \ldots, x_n$, in any order, and we use the root of its Merkle tree as the digest of S. To prove that an element x belongs to S, it suffices to open its position in the list. However, it is unclear how to efficiently prove that an element x does *not* belong to S, which is often needed too: for instance, a light node may want to check that a certain transaction output has not been spent yet, i.e., that it does not belong to the set of spent outputs (see Section 4.1). An opening of a position to a value different from x only shows that x does not appear at that position, and x may appear at any of the other positions, so the prover would essentially need to send the whole list.

Ideally, an accumulator should offer the following three properties:

- *efficient proofs of membership*, i.e., proofs that an element belongs to S;
- *efficient proofs of non-membership*, i.e., proofs that an element does not belong to S;
- *efficient updates*: when an element of S is modified, or when an element is added to or removed from S, the digest can be updated efficiently, and the parties who only store it can update their copy using a short proof of update (see Section 6.2.2).

Here, *efficient* means that, as for the openings of Merkle trees, the size of the proofs and the time needed to check them, or to update the digest, grow only logarithmically with the number of elements of S. The trivial accumulator offers efficient proofs of membership and efficient updates (to remove an element, we replace it with the dummy element $\perp$, and to add an element, we replace a dummy element with it), but not efficient proofs of non-membership. We now discuss two accumulators that also offer efficient proofs of non-membership.

Sorted lists. Suppose that the elements of S can be ordered (for instance, they are numbers, or strings in lexicographic order), and that we authenticate the sorted list $x_1 < x_2 < \cdots < x_n$ of its elements with a Merkle tree. To handle elements smaller than x_1 or larger than x_n, we also add two special elements to the list: $-\infty$ at the beginning and $+\infty$ at the end. Here, $-\infty$ and $+\infty$ are not numbers: they are just symbols denoting two special strings that do not belong to the set and that, by convention, are respectively smaller and larger than every element of the set. As in the trivial accumulator, a proof of membership is an opening of the position of the element. To prove that an element x does not belong to the set, it suffices to open the positions of two consecutive elements x_i and x_{i+1} of the list such that $x_i < x < x_{i+1}$: since the list is sorted, x cannot appear anywhere else. The verifier checks the two openings, that the two positions are consecutive, and that $x_i < x < x_{i+1}$. By position binding (Theorem 18), the prover cannot open these positions to any values other than the ones in the list, so the proof is convincing, as long as the list is sorted. The verifier cannot check the latter condition by itself: in a blockchain, the validity rules of the blocks require the list to be sorted, and the honest heavy nodes reject the blocks that violate them. Sorted lists therefore offer efficient proofs of membership and of non-membership, but not efficient updates: inserting or removing an element may shift the positions of all the elements that follow it, so we may need to recompute a large part of the tree.

Merkle Patricia tries. More generally, many applications need to authenticate a list of *key-value pairs* (κ_1, w_1), …, (κ_N, w_N), where the keys $\kappa_1, \ldots, \kappa_N$ are all different, and where we want to prove which value is associated with a given key. For instance, the state of an account-based blockchain associates every account (the key) with its balance and other information (the value). A set S corresponds to the special case in which the keys are the elements of S and all values are equal to a fixed string. We could sort the pairs by key and use a sorted list as above, but, as we have just seen, inserting a new key would be expensive. Ethereum, instead, uses a variant of the *Merkle Patricia tries* described below. For simplicity, we assume that all keys are binary strings of the same length.

The Patricia trie. To hash a list of key-value pairs, we first arrange the pairs in a *Patricia trie*. This is a tree with the following properties:

- Every pair (κ, w) of the list has its own leaf, which we call the leaf of κ, and every leaf belongs to a pair.

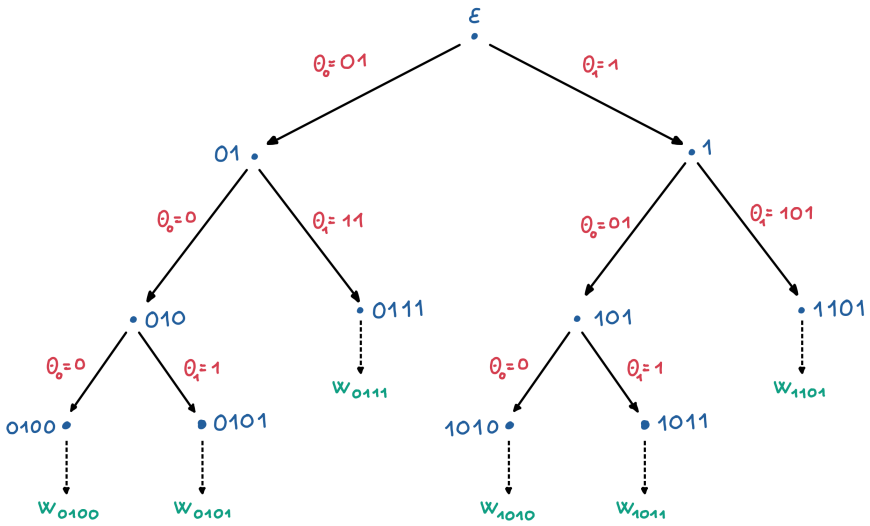

Figure 6.3: The Patricia trie of the six key-value pairs $(0100, w_{0100})$, $(0101, w_{0101})$, $(0111, w_{0111})$, $(1010, w_{1010})$, $(1011, w_{1011})$ and $(1101, w_{1101})$, where w_κ denotes the value associated with the key κ. Every node (in blue) is labelled by the concatenation of the labels of the edges (in red) along the path from the root to it: for instance, the node 101 is reached from the root through the edges labelled 1 and 01, and the leaf of the key 1011 through the edges labelled 1, 01 and 1. Every node other than the leaves has outdegree 2: the label θ_0 of one of its outgoing edges always begins with 0, and the label θ_1 of the other one always begins with 1 (in general, the root may have a single child, if all keys start with the same bit, but this does not happen here). The leaves have no outgoing edges: the dotted arrows only show the values associated with them (in green). The paths of the keys 0100 and 0101 split at the node 010, where the two keys differ for the first time, and the digest of this node is $v_{010} = \mathsf{Hash}(0, v_{0100}, 1, v_{0101})$, where $v_{0100} = \mathsf{Hash}(w_{0100})$ and $v_{0101} = \mathsf{Hash}(w_{0101})$. Similarly, the digest of the node 01 is $v_{01} = \mathsf{Hash}(0, v_{010}, 11, v_{0111})$, where $v_{0111} = \mathsf{Hash}(w_{0111})$: the labels $\theta_0 = 0$ and $\theta_1 = 11$ of the edges leaving 01 are hashed together with the digests of the nodes they lead to.

- Every edge is labelled with a non-empty binary string. For every key κ, concatenating the labels of the edges along the path from the root to the leaf of κ gives exactly κ.
- Every node other than the root and the leaves has exactly two children: the label of the edge leading to one child starts with 0, and the label of the edge leading to the other child starts with 1. The same holds for the root, except that it has a single child if all keys start with the same bit.

Intuitively, the paths of two keys stay together as long as the keys coincide, and they split at an internal node exactly where the keys differ for the first time. One can show that these properties uniquely determine the Patricia trie of every list of key-value pairs. As in a Merkle tree, we identify every node with a binary string, namely, the concatenation of the labels of the edges along the path from the root to the node: the root is the empty string ε, and the leaf of a key κ is κ itself. Figure 6.3 shows an example with six keys of four bits. Notice that a Merkle tree is essentially the special case in which the keys are all the binary strings of length d: in this case, every node other than the leaves has two children, and every label consists of a single bit.

Hashing. As in a Merkle tree, every node p has a digest v_p, and the digests are computed from the

bottom up:

- the digest of the leaf of a key κ is the hash of the value w associated with κ, i.e., $v_\kappa := \mathsf{Hash}(w)$;
- the digest of every other node p is the hash of the labels of the edges leaving p and of the digests of the children of p. More precisely, if the two edges leaving p are labelled θ_0 and θ_1, where θ_0 starts with 0 and θ_1 starts with 1, then the children of p are $p\theta_0$ and $p\theta_1$, and

$$v_p := \mathsf{Hash}(\theta_0, v_{p\theta_0}, \theta_1, v_{p\theta_1}).$$

  If the root has a single child, we replace the missing label and digest with the special symbol $\perp$.

We also assume that the labels are encoded in a way that makes the inputs of Hash unambiguous (for instance, by prefixing every label with its length). The digest v_ε of the root is called the *Merkle Patricia hash* of the list: like the root of a Merkle tree, it authenticates the whole list. Notice that the labels are hashed together with the digests, so the Merkle Patricia hash depends on both the keys and the values.

Merkle Patricia tries offer all three properties of an accumulator: the Merkle Patricia hash of a list allows a prover to convince anybody who knows it that a pair belongs, or does not belong, to the list, using short proofs, and it can be updated efficiently when the list changes. The following exercises guide you through the details.

Exercise 32 (Proofs of membership)**.** *Let z be the Merkle Patricia hash of a list of key-value pairs, and let κ be a binary string of the same length as the keys.*

1. *Following the example of the openings of Merkle trees, design a proof that a pair (κ, w) belongs to the list, and describe how to verify it using z. We call this proof an* opening *of the leaf of κ. (Hint: provide all the material associated with the nodes on the path from the root to the leaf of κ, if this leaf exists.)*
2. *Find an upper bound on the size of the proof and on the time required to verify it.*
3. *Suppose that (κ, w) does not belong to the list. Prove that, given a valid proof that (κ, w) belongs to the list, one can find a collision of* Hash*.*

Exercise 33 (Proofs of non-membership)**.** *Let z be the Merkle Patricia hash of a list of key-value pairs, and let κ be a binary string of the same length as the keys.*

1. *Prove that, if a pair (κ, w) does not belong to the list, then one of the following holds:*
   - *the list contains a pair (κ, w') with $w' \neq w$;*
   - *there exists an internal node p of the trie that is a prefix of κ, such that none of the children of p is a prefix of κ.*
2. *Design a proof that (κ, w) does not belong to the list, and describe how to verify it using z. In the first case, the proof should consist of an opening of the leaf of κ (see Exercise 32). In the second case, it should consist of an opening of the node p, i.e., all the material associated with the nodes on the path from the root to p.*
3. *Find an upper bound on the size of the proof and on the time required to verify it.*
4. *Argue, informally, that if (κ, w) belongs to the list, nobody can produce a valid proof that it does not, unless they find a collision of* Hash*.*

Exercise 34 (Updating a Merkle Patricia hash)**.** *Let z be the Merkle Patricia hash of a list of key-value pairs. In this exercise, we show how to update z efficiently when the list changes, as we did for Merkle trees in Theorem 19.*

1. *Find an efficient way to compute the new Merkle Patricia hash when we change the value associated with an existing key κ.*

2. *Find an efficient way to compute the new Merkle Patricia hash when we add a new pair (κ, w) to the list, where κ is not a key of the list. (Hint: start from a proof that (κ, w) does not belong to the list, see Exercise 33, and derive the new Merkle Patricia hash from it.)*

3. *Find an efficient way to compute the new Merkle Patricia hash when we remove a pair (κ, w) from the list. (Hint: start from an opening of the sibling of the leaf of κ, i.e., of the other child of the parent of this leaf.)*

Compared with the Merkle trees of sorted lists, the main advantage of Merkle Patricia tries is that the path of a key only depends on the key itself. As you will see in Exercise 34, inserting or removing a key only modifies the trie around the path of that key, without moving the other keys. When the keys are spread uniformly, the paths contain about $\log_2 N$ nodes, so updates require $O(\log N)$ evaluations of Hash. On the other hand, the construction is more complex (labels, encoding rules, corner cases), so it is harder to implement and to analyse. Moreover, the shape of the trie depends on the keys: if many keys share long prefixes, some paths become long, which makes proofs and updates more expensive. This is why, in practice, the keys are often hashes, which are spread uniformly.

6.2.4 Other Applications of Merkle Trees

Merkle trees are not only used in blockchains: they are useful whenever a large amount of data needs to be authenticated using a short digest, and parts of the data need to be checked individually. We mention two well-known examples.

Git and GitHub. Git, the version control system on which GitHub is based, identifies every file, every directory and every version (commit) of a project with its digest. The digest of a file is the hash of its content. The digest of a directory is the hash of the list of the names and digests of the files and subdirectories it contains. Finally, the digest of a commit is the hash of the digest of the main directory of the project, of some metadata (such as the author, the date and a description of the changes), and of the digest of the previous commit (or commits, when two versions of the project are merged). In other words, the digest of a commit plays the role of the root of a Merkle tree: it authenticates the whole content of the project at that point in time, as well as its whole history. This has several useful consequences. For instance, two copies of a project can be compared by exchanging only the digests of their latest commits and, if these differ, the differences can be found quickly by exploring only the directories whose digests differ. Moreover, a user who knows the digest of a commit (for instance, because it was published by the developers of the project) can download the project from an untrusted server and check that it has not been tampered with.

BitTorrent. BitTorrent is a peer-to-peer protocol for sharing large files: instead of downloading a file from a single server, users download small blocks of it from many other users (the peers), some of which may be malicious. To start a download, a user needs a small *torrent* file, which describes the file to download. In the second version of the protocol, the torrent file contains the root of the Merkle tree of the blocks of the file. Together with the blocks, the peers send the digests needed to check them against the root, i.e., their openings. Hence, the user can check every block as soon as it arrives, without trusting the peer that sent it, and discard the corrupted ones before passing them on to other users.

6.3 Pools of Validators

A *pool of validators* is a group of validators who collaborate to generate new blocks and share the rewards of these blocks (see Section 4.4) proportionally to the contribution of each of them. Usually, one of them, the *leader* of the pool (or pool operator), coordinates the others, the *members*, and keeps a small part of the rewards as a fee. In proof of work, pools are called *mining pools* (see Section 3.2.2) and, in proof of stake, *stake pools* (see Section 5.8). They are extremely common: for instance, almost all Bitcoin blocks are mined by a handful of pools.

Why do validators join pools? Joining a pool does not increase the average income of a validator (if anything, the fee of the leader decreases it), but it makes this income much more regular. Consider a validator P_j who controls a fraction φ_j of the resources of the system (all the computational power in proof of work, or all the stake in proof of stake), and suppose, for simplicity, that every block is generated by P_j independently with probability φ_j. If R denotes the reward of a block, P_j earns, on average, $\varphi_j R$ per block. However, the number of blocks until P_j generates its first one follows a geometric distribution with mean $1/\varphi_j$ (see Definition 7): a small validator may wait for a very long time before receiving any reward (recall Exercise 19), and it cannot predict how long, while it has to pay for its hardware, its energy or its servers on a regular basis. Most validators prefer a small but reliable income over large rewards that come irregularly, just as most people prefer to pay a small insurance premium rather than risk a large loss. Now, suppose that validators $P_1, \ldots, P_n$, controlling fractions $\varphi_1, \ldots, \varphi_n$ of the resources, form a pool, which therefore controls a fraction $\varphi := \varphi_1 + \cdots + \varphi_n$. Whenever the pool generates a block, P_j receives a fraction φ_j/φ of the reward (ignoring the fee). On average, P_j still earns $\varphi \cdot (\varphi_j/\varphi) \cdot R = \varphi_j R$ per block, but it now receives a small reward every $1/\varphi$ blocks on average, instead of the whole reward every $1/\varphi_j$ blocks. In statistical terms, a pool does not change the average income of its members, but it reduces its variance.

Exercise 35. *The* variance *of a random variable X is $\mathrm{Var}[X] := \mathbb{E}\big[(X - \mathbb{E}[X])^2\big]$: it measures how far X typically is from its average. Let X be the reward that P_j receives for a single block when it works alone (i.e., $X = R$ with probability φ_j, and $X = 0$ otherwise), and let Y be the reward that it receives for a single block when it is a member of a pool that generates every block independently with probability $\varphi \geq \varphi_j$ and gives it a fraction φ_j/φ of each reward. Compute $\mathbb{E}[X]$, $\mathbb{E}[Y]$, $\mathrm{Var}[X]$ and $\mathrm{Var}[Y]$, and show that $\mathrm{Var}[Y] \leq (\varphi_j/\varphi) \cdot \mathrm{Var}[X]$.*

As we will see, pools are much simpler in proof of stake than in proof of work.

6.3.1 Pools in Proof of Work

In a mining pool, the leader acts as a single miner of the protocol of Section 3.2.2, and the members provide the computational power. The leader prepares the content x and the metadata y of a new block, where x assigns the reward of the block to the leader (in Bitcoin, through the coinbase transaction, see Section 4.4). Then, it computes the pointer h to the most recent block of its main chain and the digest z, and sends the puzzle (z, h) to the members, assigning each of them a different set of counters to try (for instance, P_j only tries the counters that start with the binary representation of j). A member who finds a solution, i.e., a counter c such that $\mathsf{Hash}(c, z, h) < W$, sends it to the leader, who completes the block (c, z, x, y, h) and broadcasts it. Notice that the members only need the puzzle: only the leader needs to run a heavy node (see Section 6.1). Moreover, a member cannot steal the reward of a block: modifying x to assign the reward to itself would change z, so c would most likely not solve the new puzzle.

Proving the work: shares. How can the leader measure the contribution of each member? Solutions are so rare that a small member would almost never find one, so counting them does not help. Instead, the leader picks a second threshold W', much larger than W, and asks the members to also send every counter c such that $\mathsf{Hash}(c, z, h) < W'$. These counters, called *shares*, are counters that almost win the election. Since the outputs of Hash look uniformly random (see Section 1.4.1), every counter is a share with probability $W'/2^k$: a member finds, on average, one share every $2^k/W'$ evaluations of Hash, so the number of shares that it sends is, on average, proportional to the computation that it performed for the pool. Moreover, every share is a solution with probability W/W', so all shares contribute equally to the chances of the pool, which finds, on average, one block every W'/W shares. Shares are easy to check (with a single evaluation of Hash) and hard to fake (finding them requires the corresponding computation). They cannot be reused either: they almost solve the puzzle (z, h), and z depends on x, so they only certify work performed on blocks whose reward goes to the leader. The choice of W' is a trade-off: the larger W', the more precisely the leader can measure the contributions of the members, but the more shares it must receive and check.

Sharing the rewards. In the simplest scheme, whenever the pool generates a block that ends up on the main chain, the leader keeps its fee and splits the rest of the reward among the members, proportionally to the number of shares that they sent since the previous block of the pool. In the *pay-per-share* scheme, instead, the leader immediately pays every share with its average reward (a fraction W/W' of the reward of a block), minus the fee: the members then have a completely regular income, and the leader takes all the risk.

The members still need to trust the leader. Shares allow the members to prove their work to the leader, but not the other way around. First, the rewards are paid to the leader, who may pay the members less than it promised (the members cannot even check how many shares the others sent). Second, the leader decides the content of the blocks and the chain they extend, while the members typically only see the puzzle. Hence, the leader controls all the computational power of the pool, which it could use to censor transactions or, if the pool is large enough, to attack the blockchain (recall that security relies on the honest parties controlling the majority of the computational power, see Theorem 8). For instance, in 2014, the Bitcoin pool GHash.io briefly controlled more than half of all the computational power of the system. Conversely, the leader must trust the members to some extent. A member could send all its shares except the solutions: this is called *block withholding*. The member is still paid for its shares, but the pool never wins thanks to its computational power, so the other members earn less. This attack can be used to sabotage a competing pool, and it is hard to detect, since solutions are rare.

6.3.2 Pools in Proof of Stake

In proof of stake, pools are much simpler, at least if the members trust the leader. Consider our construction of Section 5.4. The members $P_1, \ldots, P_n$ give their stakes $V_1, \ldots, V_n$ to the leader (for instance, by transferring the corresponding coins to it), and the leader commits the total stake $V := V_1 + \cdots + V_n$ with its own keys, taking part in the elections as a single validator. In every slot, the leader wins with probability approximately $\frac{V}{V_{\text{tot}}} \cdot \frac{W}{2^k}$ (see Section 5.5), which is the sum of the probabilities with which the members would win on their own: on average, the pool wins as many elections as its members together would, but much more regularly. Whenever the leader wins, it gives each member P_j a fraction V_j/V of the reward, minus its fee, and it returns the stake of the members who leave the pool.

Unlike in proof of work, there is no work to measure: the contribution of each member is its stake, which is recorded on the blockchain, so there is no need for shares. The members do not need to do anything (not even to run a heavy node, see Section 6.1): only the leader evaluates the VRF and signs blocks, which also makes block withholding impossible. Finally, the blocks of the pool, and therefore its rewards, are recorded on the blockchain, so every member can check how much it is owed. Nevertheless, the members still trust the leader to return their stake, to pay them and to follow the protocol: as in proof of work, the leader alone exercises the influence over the protocol granted by all the stake of the pool and, if the protocol punishes misbehaviour by destroying stake (slashing, see Section 5.8), a misbehaving leader also loses part of the stake of the members.

6.3.3 Pools Without a Trusted Leader

Can we remove the need to trust the leader? In proof of stake, this is relatively easy, at least for the coins and the rewards. Some blockchains allow the members to *delegate* their stake: the members keep their coins and only grant the leader the right to use them in the elections, while the protocol itself computes the rewards and pays each member its part. For instance, in Cardano, the delegated coins never leave the wallets of their owners, and the rewards are distributed automatically, so the leader can steal neither the coins nor the rewards. On blockchains without delegation, such as Ethereum, a smart contract (see Section 4.3) can play the same role: it collects the coins of the members, commits them as the stake of validators run by one or more leaders, receives the rewards that the protocol pays to these validators, and distributes them automatically (Lido and Rocket Pool are well-known examples). Since, in Ethereum, the stake of a validator can only be returned to an address fixed when the stake is committed, the leaders never get hold of the

coins. However, the transaction fees of a block are paid to an address chosen by its proposer, so the members must trust the leaders to forward them to the contract. More generally, the leader still holds the secret keys used to sign the blocks: it may go offline (the members then lose rewards) or misbehave (in blockchains with slashing, the members may then lose part of their stake). Two mitigations are common: the leader may have to commit some stake of its own, which is lost first if it is punished (as in Rocket Pool), and the signing key may be shared among several operators using *threshold signatures*, so that blocks can only be signed if enough operators agree (this is called a *distributed validator*).

In proof of work, removing the leader is harder: somebody must check the shares, and the recipients of the reward must be fixed in the block before the search for a solution starts. *P2Pool*, used on Bitcoin for several years, replaces the leader with a separate blockchain of shares, maintained by the members using the threshold W': the content of every share must assign the reward of the block to the members, proportionally to the shares that they found recently on this chain, so a share that is also a solution yields a block that pays all the members directly. *SmartPool* replaces the leader with a smart contract on the blockchain itself (it was deployed on Ethereum when Ethereum used proof of work): the members mine blocks whose reward goes to the contract, and periodically send it the root of a Merkle tree of their shares (see Section 6.2). The contract checks the openings of a few randomly chosen shares and pays the member proportionally to the number of shares in the tree, or nothing at all if it catches the member cheating. These solutions are more complex and more expensive than centralised pools, and they were never widely adopted. For instance, P2Pool is no longer used, partly because, as the pool grew, shares became harder to find, so its small members received irregular rewards again.

6.4 Succinct Arguments

Every validator must check every new block, which may be expensive: for instance, it must evaluate all the smart contracts called by the transactions of the block (see Section 4.3). *Succinct arguments* allow a party to convince anybody that an expensive computation was performed correctly, without asking them to redo it. For instance, in proof of stake, the winner of an election can attach to its block a short proof that the block is valid, and the other validators (and even the light nodes, see Section 6.1) only need to check this proof. In this way, only the winner of the election needs to evaluate the smart contracts, which vastly decreases the computation performed by the validators.

NP relations. Succinct arguments prove statements about problems whose solutions may be hard to find, but are easy to check.

Definition 41 (NP relation)**.** *An* NP relation *is a set R of pairs of binary strings (x, w) for which there exist a polynomial p and a deterministic polynomial-time algorithm M_R such that:*

- *$|w| \leq p(|x|)$ for every $(x, w) \in R$;*
- *for every pair of binary strings (x, w), it holds that $M_R(x, w) = 1$ if $(x, w) \in R$, and $M_R(x, w) = 0$ otherwise.*

We call x a statement *and, if $(x, w) \in R$, we call w a* witness *for x. A statement is* true *if it has at least one witness, and* false *otherwise. We denote the set of true statements by L_R.*

We denote by t_R the time that M_R needs to check a witness. For instance, the pairs (x, w) such that $\mathsf{Hash}(w) = x$ form an NP relation, whose witnesses are preimages. In our example, the statement consists of the transactions of a block and of the roots z and z' of the Merkle trees of the state of the blockchain (e.g., the balances of all accounts, see Section 6.2.2) before and after the block. A witness is a state S with root z such that applying the transactions to S yields a state with root z' (in practice, the witness only contains the entries of S involved in the transactions, with their openings). Here, t_R includes the time needed to evaluate the smart contracts of the block.

Provers and verifiers. In a non-interactive argument, the *prover* knows a statement x and a witness w, and its goal is to convince the *verifier* that x is true by sending a single message π, called a *proof*, which can be published (e.g., in a block) and checked by anybody. The verifier, who only knows x and π, wants to accept every true statement, but never a false one, whatever the prover does. Both use public parameters crs, generated once and for all by an algorithm Setup. Sending $\pi := w$ would trivially achieve these goals, but would not save the verifier any work: we want much shorter proofs, which can be checked much faster.

Definition 42 (Non-interactive argument)**.** *A non-interactive argument for an NP relation R consists of a triple of polynomial-time algorithms* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *with the following syntax:*

Setup**:** *This algorithm is probabilistic and takes no input. It outputs public parameters* crs*, often called the* common reference string*.*

Prove**:** *This algorithm is probabilistic and takes as input the public parameters* crs*, a statement x and a witness w. It outputs a proof π.*

Verify**:** *This algorithm is deterministic and takes as input the public parameters* crs*, a statement x and a proof π. It outputs a bit: 0 stands for "*reject*", 1 stands for "*accept*".*

We additionally require completeness*: for every $(x, w) \in R$,*

$$\Pr\left[\mathsf{Verify}(\mathsf{crs}, x, \pi) = 1 \;\middle|\; \begin{array}{l} \mathsf{crs} \xleftarrow{\$} \mathsf{Setup} \\ \pi \xleftarrow{\$} \mathsf{Prove}(\mathsf{crs}, x, w) \end{array}\right] = 1.$$

Definition 43 (Soundness)**.** *Let $T \in \mathbb{N}$ and $\epsilon \in [0, 1]$. A non-interactive argument* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *for an NP relation R is (T, ϵ)-sound if, for every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

$$\Pr\left[\begin{array}{c} \mathsf{Verify}(\mathsf{crs}, x, \pi) = 1 \\ x \notin L_R \end{array} \;\middle|\; \begin{array}{l} \mathsf{crs} \xleftarrow{\$} \mathsf{Setup} \\ (x, \pi) \xleftarrow{\$} \mathcal{A}(\mathsf{crs}) \end{array}\right] \leq \epsilon.$$

Completeness states that an honest prover who knows a witness always convinces the verifier. Soundness states that a cheating prover running in time at most T convinces the verifier of a false statement (even one chosen after seeing crs) with probability at most ϵ. Typically, ϵ is extremely small (e.g., $\epsilon = 2^{-128}$), even for very large values of T.

Efficiency. A non-interactive argument is a *SNARG* (succinct non-interactive argument) if its proofs are much shorter than the witnesses and much faster to check than by running M_R. Informally, the best known SNARGs achieve the following: the prover runs in time roughly proportional to t_R; apart from reading x, the verifier runs in time that does not depend on t_R, or grows only very slowly with it (like a power of $\log t_R$); and the size of the proofs does not depend on t_R either, or grows only very slowly with it. In our example, computing the proof takes roughly as long as checking the block (although, in practice, the slowdown is still significant), while the other validators check it in a time that is essentially independent of the computation required by the block.

Generating the public parameters. Soundness only holds if crs is generated by Setup. If crs is a uniformly random string, we can in principle use the digest $\mathsf{SHA2}(\texttt{Hello})$ of a fixed message (or $\mathsf{SHA2}(\texttt{Hello}, 1)$, $\mathsf{SHA2}(\texttt{Hello}, 2)$, ..., if we need a longer string): since $\mathsf{SHA2}$ behaves like a random oracle (see Section 1.4.1), this string looks random, and nobody can have tampered with it. If, instead, crs is a more complex object, whoever runs Setup learns secret information that would allow it to prove false statements. We can then rely on a trusted authority, who runs Setup and erases this information (if the authority is malicious, however, the SNARG is not secure), or let a committee of parties generate crs using a *multiparty computation* (MPC) protocol, which guarantees that crs is secure as long as at least one member of the committee is honest (we will discuss MPC in a future lecture). Either way, crs is generated once and for all, so the costs of its secure generation are amortised over time.

SNARKs. Soundness guarantees that the statement is true, i.e., that a witness exists, but not that the prover knows one. A *SNARK* (succinct non-interactive argument of knowledge) is a SNARG that offers a stronger guarantee: informally, whoever produces a valid proof for x must *know* a witness for x (which implies soundness, since nobody can know a witness for a false statement). The difference matters when true statements are easy to find, but their witnesses are not. For instance, since the domain of Hash is much larger than its range, almost every digest has a preimage, so, for the relation of preimages, soundness guarantees almost nothing, while a SNARK guarantees that the prover knows a preimage. Our motivating example is a situation in which we need a SNARK. Since the root z is much shorter than the state, many different states have root z (although nobody can find two of them), so the statement may be true even if z' is wrong: it suffices that some state S', with the same root as the real state S, yields a state with root z'. With a SNARK, a prover who produces a valid proof must know such an S'. If z' is wrong, then $S' \neq S$, so the prover, who also knows S, knows two different lists with the same root, from which it can find a collision of Hash (see Exercise 28). In practice, the succinct arguments used in blockchains are SNARKs.

A note on terminology. In the blockchain world, SNARGs and SNARKs are often called "ZK proofs": crypto enthusiasts (the so-called crypto bros) tend to say "ZK" whenever they mean a SNARG or a SNARK. For instance, the systems that use SNARKs to prove the validity of large batches of Ethereum transactions are called *zk-rollups*, even though many of them do not provide ZK at all. ZK is a different property: we will see in a future lecture what ZK stands for.

6.5 Multisignatures and Aggregatable Signatures

Blockchains are full of digital signatures (see Section 1.5): every transaction carries the signatures of the users who authorise it and, in proof of stake, the validators sign blocks and votes (see Section 5.8). All these signatures must be sent over the network, stored and checked. We now discuss, informally, two extensions of digital signatures that allow us to compress many signatures into a single one.

Multisignatures. Suppose that n signers $P_1, \ldots, P_n$, with public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$, all sign the same message m (for instance, a vote for a block). With an ordinary signature scheme, the verifier would need to receive and check n signatures. A *multisignature* scheme, instead, allows the signers to produce a single signature s on m, as short as an ordinary one, which convinces anybody who knows $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$ that all the signers signed m. Moreover, the public keys can usually be combined into a single *aggregate public key* pk, against which s is verified exactly as an ordinary signature. Computing pk is cheap (and, for a fixed group of signers, it only needs to be done once), so verification costs essentially the same as for a single signature. In some schemes, such as the *BLS signatures* (named after Boneh, Lynn and Shacham), the signers sign independently and anybody can combine their signatures; in others, such as *MuSig2* (based on Schnorr signatures and standardised for Bitcoin), the signers run a short interactive protocol.

Aggregatable signatures. Suppose now that each signer P_i signs its own message m_i (for instance, its own transaction). An *aggregatable signature* scheme (also called aggregate signature scheme) allows anybody to compress the signatures $s_1, \ldots, s_n$ into a single short signature s, which convinces the verifier that, for every i, P_i signed m_i. This is the crucial difference with multisignatures: since the messages are different, the verifier still needs all the pairs (pk_i, m_i), and its running time still grows linearly with n (although it is usually smaller than for n separate signatures). The main gain is therefore in size: n signatures are replaced by a single one. BLS signatures are aggregatable, while the Schnorr and ECDSA signatures used in the transactions of Bitcoin and Ethereum are not.

Why are they relevant for blockchains?

- *Votes in proof of stake.* In Ethereum, a block becomes final once validators holding two thirds of the stake have voted for it (see Section 5.8). Since there are very many validators, their votes are combined

into BLS multisignatures: every block only contains a few multisignatures, each accompanied by a list of bits indicating which validators voted (one bit per validator, instead of one signature). This is what allows Ethereum to support so many validators.

- *Certificates.* More generally, a multisignature and a list of bits suffice to prove that a large group of parties approved a message. For instance, Ethereum's light nodes (see Section 6.1) follow the chain by checking, for every block, a single multisignature produced by a committee of validators. Similarly, multisignatures shorten the messages of protocols like Dolev–Strong (see Theorem 3), which contain many signatures on the same message.

- *Transactions.* Aggregatable signatures can compress the signatures of all the transactions of a block into a single one: for instance, every block of the Chia blockchain contains a single BLS signature.

- *Shared accounts.* With MuSig2, the owners of a Bitcoin account controlled by several parties produce a single signature that looks like an ordinary one, instead of several separate signatures, which saves space and improves privacy. A related primitive, *threshold signatures*, allows any t out of n parties to sign under a single public key (see the distributed validators of Section 6.3.3).

Chapter 7

Game Theory on Blockchains

The security of the protocols of the previous chapters relies on the assumption that most of the resources (computational power or stake) are controlled by honest parties, i.e., by parties that follow the protocol. Validators, however, are not altruistic: they run the protocol because they are paid to do so (see Section 4.4). It is therefore natural to ask whether following the protocol is in their interest. Questions of this kind are studied by *game theory*, the mathematical theory of the interaction between rational parties. In this chapter, we introduce its basic notions, we look at how they are used in blockchains, and we discuss the limits of this approach.

7.1 Basic Notions of Game Theory

Game theory studies situations in which several parties, called *players*, make decisions that affect each other, so that the best decision for a player depends on the decisions of the others. For instance, a miner who finds a new block must decide whether to publish it immediately or to keep it secret for a while, and which choice is more profitable depends on how the other miners behave (we study this example in Section 7.2.2). To describe such a situation, we must specify what each player wants (their objective function), what each player can do (their strategies), and how the choices of all players determine the outcome: together, these ingredients form a game. We can then ask how players who act in their own interest will behave, which leads to the notion of Nash equilibrium.

Objective functions. Each player has an *objective function* (also called *utility* or *payoff*), which assigns a real number to every possible outcome of the interaction: the larger the number, the more the player likes the outcome. We assume that every player is *rational*: they always act so as to maximise their objective function (we discuss whether this assumption is realistic in Section 7.3). In blockchains, the objective function of a validator is typically their profit, i.e., the value of the rewards and fees that they collect, minus their costs (for instance, electricity, hardware and penalties). When the outcome is random, we consider the expected value of the objective function.

Strategies. A *strategy* is a complete description of the behaviour of a player: it specifies which action the player takes in every situation that may arise. The description must be complete because a player does not know in advance how the others will behave, so their strategy must say how they react to every possible behaviour of the others. In the simplest games, each player takes a single decision without seeing the decisions of the others, and a strategy is just a choice of action (the prisoner's dilemma below is a game of this kind). In a protocol, instead, the parties exchange messages over time, and a strategy is simply the algorithm that a party runs, which specifies how the party reacts to every message that it may receive. The *honest strategy* is the algorithm prescribed by the protocol, and every other algorithm is a *deviation*. Deviations play the role of the adversaries of our security definitions, with one difference: an adversary may

behave arbitrarily, for instance, just to cause damage, while a rational player deviates only if this increases their objective function.

Games. The strategies of all players together determine the outcome of the interaction (or its probability distribution), and therefore the value of every objective function. Formally, a *game* with n players $P_1, \ldots, P_n$ consists of

- a set $\mathcal{S}_i$ of strategies for every player P_i;
- an objective function $u_i : \mathcal{S}_1 \times \cdots \times \mathcal{S}_n \to \mathbb{R}$ for every player P_i.

A tuple $s = (s_1, \ldots, s_n) \in \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ is called a *strategy profile*, and $u_i(s)$ is the (expected) value of the objective function of P_i when every player P_j follows the strategy s_j. Notice that u_i depends on the strategies of all players, not only on that of P_i, so the best strategy for P_i may depend on the strategies of the others. Given a strategy profile s, we denote by s_{-i} the strategies of all players except P_i and, for every $s'_i \in \mathcal{S}_i$, we denote by (s'_i, s_{-i}) the strategy profile in which P_i follows s'_i, while every other player P_j still follows s_j.

Nash equilibria. Which strategy profiles do we expect rational players to follow? Suppose that P_i expects the other players to follow the strategies s_{-i}. Then, the best that P_i can do is to follow a strategy s_i that maximises $u_i(s_i, s_{-i})$, which is called a *best response* to s_{-i}. If the strategy of every player is a best response to the strategies of the others, the strategy profile is *stable*: no player has a reason to change their strategy, as long as the others do not change theirs. Stable strategy profiles are called Nash equilibria.

Definition 44 (Nash equilibrium)**.** *A strategy profile $s^* = (s_1^*, \ldots, s_n^*)$ is a* Nash equilibrium *if, for every $i \in [n]$ and every strategy $s_i \in \mathcal{S}_i$,*

$$u_i(s_i, s^*_{-i}) \leq u_i(s^*).$$

In the definition, $u_i(s_i, s^*_{-i})$ is the value of the objective function of P_i when P_i switches to the strategy s_i, while all other players keep following s^*: the inequality says that P_i gains nothing by switching. We can also see a Nash equilibrium as an agreement that enforces itself: if the players agree to follow s^*, nobody needs to force them to keep the agreement, since no player can gain by breaking it alone. For a protocol, we would like the *honest strategy profile*, in which every party follows the honest strategy, to be a Nash equilibrium: in this case, no rational party has an incentive to deviate from the protocol, as long as the others follow it.

Are equilibria good outcomes? A Nash equilibrium is stable, but this does not mean that it is good for the players. Since the players have different objective functions, there is in general no outcome that is the best for all of them. Some comparisons, however, are clear: if $u_i(s') > u_i(s)$ for every $i \in [n]$, then all players prefer the strategy profile s' to the strategy profile s (in the terminology of economics, s is *Pareto-dominated* by s'). It may happen that a Nash equilibrium is Pareto-dominated by another strategy profile which, however, the players cannot reach, because it is not stable: if the players agreed to follow it, some of them would gain by breaking the agreement. The best-known example of this phenomenon is the prisoner's dilemma.

Example: the prisoner's dilemma. Two suspects, P_1 and P_2, are arrested and interrogated separately. Each of them can either stay silent (S) or betray the other by confessing (B), without knowing what the other does. If both stay silent, both are sentenced to one year in prison (for a minor offence). If both betray, both are sentenced to two years. If only one of them betrays, the betrayer is released, and the other is sentenced to three years. The objective function of each prisoner is minus the number of years that they

spend in prison. In the notation above, $\mathcal{S}_1 = \mathcal{S}_2 = \{\mathsf{S}, \mathsf{B}\}$, and the following table shows the pair (u_1, u_2) for every strategy profile.

	P_2 stays silent (S)	P_2 betrays (B)
P_1 stays silent (S)	$(-1, -1)$	$(-3, 0)$
P_1 betrays (B)	$(0, -3)$	$(-2, -2)$

Whatever P_2 does, betraying is the best response of P_1: if P_2 stays silent, betraying gives 0 instead of -1 and, if P_2 betrays, it gives -2 instead of -3. The same holds for P_2. Therefore, (B, B) is a Nash equilibrium. It is also the only one: in every other strategy profile, at least one prisoner stays silent, and they could increase their objective function by betraying. Yet, (B, B) is Pareto-dominated by (S, S), in which each prisoner spends one year in prison instead of two. The prisoners cannot reach (S, S) because it is not stable: even if they had promised each other to stay silent before being arrested, each of them would gain by breaking the promise. We will see a blockchain version of this game, called the miner's dilemma, in Section 7.3.

Collusion. Definition 44 only considers the deviations of a single player. In practice, however, several players may collude: they may coordinate their strategies and share their profits. In blockchains, collusion is easy, since the colluding parties can simply transfer coins to each other, and it is also common: the members of a mining pool or of a stake pool act together and share their rewards (see Section 6.3), and a single person may even run many validators. We therefore assume that a set of players, called a *coalition*, deviates whenever this increases its total objective function: if the total increases, the members can share the gain so that each of them is better off. For every set $C \subseteq [n]$ and every strategy profile s, we denote by s_C the strategies of the players in C and by s_{-C} the strategies of the other players, and we define (s'_C, s_{-C}) as above.

Definition 45 (Resilient equilibrium)**.** *Let $t \in [n]$. A strategy profile s^* is a* t-resilient equilibrium *if, for every coalition $C \subseteq [n]$ with $|C| \leq t$ and every tuple of strategies $s_C \in \prod_{i \in C} \mathcal{S}_i$ for the players in C,*

$$\sum_{i \in C} u_i(s_C, s^*_{-C}) \leq \sum_{i \in C} u_i(s^*).$$

For $t = 1$, this is exactly Definition 44, while larger values of t give stronger guarantees. In blockchains, it is often more natural to bound the resources of the coalition rather than its size: for instance, we may require that no coalition controlling at most a fraction α of the stake can increase its total objective function by deviating. Several variants of this notion appear in the literature: for instance, when the colluding players cannot share their profits, it is natural to only require that no deviation makes every member of the coalition better off.

Let us go back to the prisoner's dilemma. The Nash equilibrium (B, B) is not 2-resilient: the coalition $\{P_1, P_2\}$ can switch to (S, S) and increase its total objective function from -4 to -2. In fact, the game has no 2-resilient equilibrium at all. Such an equilibrium would have to maximise the total objective function of the two players (otherwise, the coalition $\{P_1, P_2\}$ would deviate), so it would have to be (S, S), which is not a Nash equilibrium. To actually switch to (S, S), however, the prisoners must be able to commit in advance to staying silent, for instance, by agreeing that a betrayer must pay a fine to the other prisoner. If the fine is worth more than one year in prison, staying silent becomes the best strategy for both players, whatever the other does, and (S, S) becomes the only Nash equilibrium. The agreement to stay silent then enforces itself, provided that betrayers really pay the fine. Making sure that they do requires a trusted party, such as a court. On a blockchain, smart contracts can play this role, as we see in Section 7.2.1.

7.2 Game Theory in Blockchains

We now look at three uses of game theory in blockchains: a protocol that relies on deposits to make cheating unprofitable, an analysis showing that following the proof-of-work protocol is not always the most profitable strategy, and the consensus protocol of Ethereum, whose security relies heavily on incentives.

7.2.1 Swapping Secrets with Deposits

Suppose that two parties, P_1 and P_2, want to exchange two secrets: P_1 knows w_1, P_2 knows w_2, and each of them wants to learn the secret of the other. For instance, each w_i may be the key needed to decrypt a file that P_i sells to the other party. The secrets were committed in advance: everybody knows the digests $h_1 := \mathsf{Hash}(w_1)$ and $h_2 := \mathsf{Hash}(w_2)$, where Hash is modelled as a random oracle (see Section 1.4.1), and w_1 and w_2 are long random strings, so that the digests reveal nothing about them. Ideally, the exchange should be *fair*: P_1 should learn w_2 only if P_2 learns w_1, and vice versa. The difficulty is that somebody has to reveal their secret first, and the other party could then refuse to reveal theirs. On a blockchain, we can make such a refusal costly: whoever learns the secret of the other party without revealing their own must pay a penalty to the other party. If the penalty is large enough, cheating is never profitable.

The protocol. The parties agree on the digests h_1 and h_2, on an amount d of coins, called the *deposit*, and on two deadlines $\tau_1 < \tau_2$ (for instance, two block heights). Then, they create a smart contract (see Section 4.3) that works as follows.

1. *Deposits.* Each party sends d coins to the contract. As long as the contract has not received both deposits, each party can withdraw their own deposit and abort.
2. *First revelation.* Before the deadline τ_1, P_1 can send w_1 to the contract. The contract checks that $\mathsf{Hash}(w_1) = h_1$ and records w_1. From this moment on, w_1 is public, since it is written on the blockchain.
3. *Second revelation.* If w_1 was recorded, P_2 can send w_2 to the contract before the deadline τ_2. The contract checks that $\mathsf{Hash}(w_2) = h_2$ and records w_2.
4. *Settlement.* If w_1 was not recorded by the deadline τ_1, the contract sends both deposits to P_2. If w_1 was recorded, but w_2 was not recorded by the deadline τ_2, the contract sends both deposits to P_1. If both secrets were recorded, each party gets their deposit back.

The honest strategies are the following: P_1 reveals w_1 and, if w_1 was revealed, P_2 reveals w_2.

Analysis. To analyse the protocol, we need objective functions. For $i \in \{1, 2\}$, let $v_i \geq 0$ be the value that P_i attributes to learning the secret of the other party, and let $k_i \geq 0$ be the value that P_i loses if their own secret becomes public (for instance, the price at which they could have sold it to somebody else). We assume that $v_i \geq k_i$ (otherwise, P_i would not want to take part in the exchange). The objective function of P_i is the sum of these values and of the coins that P_i gains or loses through the contract. For instance, if both parties are honest, then $u_1 = v_1 - k_1$ and $u_2 = v_2 - k_2$, since the deposits are returned. If P_1 reveals w_1, but P_2 does not reveal w_2, then $u_1 = d - k_1$ and $u_2 = v_2 - d$. If P_1 does not reveal w_1, then $u_1 = -d$ and $u_2 = d$.

Proposition 1. *If $d \geq \max\{k_1, k_2\}$, the honest strategy profile is a Nash equilibrium. Moreover, an honest party never ends up with a negative objective function, whatever the other party does.*

Proof. Consider first the deviations of P_2. If P_2 does not deposit, the protocol aborts and $u_2 = 0 \leq v_2 - k_2$. Otherwise, after w_1 is revealed, not revealing w_2 gives $u_2 = v_2 - d \leq v_2 - k_2$, since $d \geq k_2$. Similarly, if P_1 does not deposit, then $u_1 = 0 \leq v_1 - k_1$, and if P_1 deposits but does not reveal w_1, then $u_1 = -d \leq 0 \leq v_1 - k_1$. Therefore, no party can increase their objective function by deviating alone.

For the second claim, notice that, if P_1 is honest, then u_1 is equal to 0, $v_1 - k_1$ or $d - k_1$, depending on whether P_2 does not deposit, reveals w_2, or deposits but does not reveal w_2. All these values are non-negative, since $v_1 \geq k_1$ and $d \geq k_1$. Similarly, if P_2 is honest, then u_2 is equal to 0, d or $v_2 - k_2$, depending on whether P_1 does not deposit, deposits but does not reveal w_1, or reveals w_1. □

Remarks. The exchange is not fair in the strict sense: P_2 may learn w_1 and keep w_2 secret. In this case, however, P_2 pays $d \geq k_1$ coins to P_1, which compensates P_1 for the loss of their secret. The analysis also relies on some assumptions. First, the transactions of the honest parties must be added to the blockchain before the deadlines, so the deadlines must leave enough time (see Theorem 10), and the validators must not censor the transactions. For instance, if P_1 could bribe the validators to delay the transaction of P_2 until after τ_2, then P_1 would learn w_2 (which P_2 broadcast to the network) and would also take the deposit of P_2. Second, the parties must agree on a deposit d larger than k_1 and k_2, which requires estimating how much the secrets are worth.

7.2.2 Selfish Mining

Is the honest strategy profile of the proof-of-work protocol a Nash equilibrium? In Section 3.2.2, we observed that a miner who keeps a new block secret takes a risk: while the block is secret, the other miners may find a different block, and the secret block may then be wasted. For this reason, it was long believed that publishing every block as soon as possible is the most profitable strategy for every miner holding less than half of the computational power. In 2013, Eyal and Sirer showed that this is false: a miner (or a pool) holding a large enough fraction of the computational power can obtain a larger share of the rewards with a strategy called *selfish mining*. The idea is to keep new blocks secret and to publish them only when the honest miners are about to catch up, so that the honest miners waste their computational power on blocks that end up being discarded.

The model. We consider a single selfish miner, which controls a fraction $\alpha < 1/2$ of the computational power. All other miners are honest: they publish their blocks immediately and mine on top of the longest chain of published blocks. As in the security analysis of Section 3.2.5, blocks are found one at a time, and each new block is found by the selfish miner with probability α, independently of the previous blocks. We ignore network delays, with one exception. In the strategy below, the selfish miner sometimes publishes a block as soon as it sees an honest block of the same height. In this case, each honest miner mines on top of the block that reaches them first (as in Bitcoin), and we denote by $\zeta \in [0, 1]$ the fraction of the honest computational power that receives the block of the selfish miner first. If the selfish miner has a privileged position in the network, for instance, because it is directly connected to most honest miners, its block reaches them before the honest one, and ζ tends to 1.

The strategy. The selfish miner keeps the blocks that it finds secret, and mines on top of them. Its *lead* is the number of blocks by which its chain, including the secret blocks, is longer than the longest chain of published blocks. The selfish miner only publishes its blocks when the honest miners are about to catch up. More precisely, when an honest miner finds a block, the selfish miner reacts as follows.

- If the selfish miner has no secret blocks, it starts mining on top of the new block.
- If the new block makes the lead drop from 1 to 0, the selfish miner publishes its secret block, so two chains of the same length compete: the selfish miner and a fraction ζ of the honest computational power mine on top of the block of the selfish miner, while the other honest miners mine on top of the honest block. The next block decides this *race*: if the selfish miner finds it, it publishes it immediately and its chain wins; if an honest miner finds it, the chain that it extends wins.
- If the new block makes the lead drop from 2 to 1, the selfish miner publishes all its secret blocks. Its chain is still longer than the public one, so all honest miners switch to it, and the honest blocks found since the selfish miner started its secret chain are discarded.
- If the new block makes the lead drop to 2 or more, the selfish miner keeps its blocks secret.[1]

[1] In the original strategy of Eyal and Sirer, the selfish miner also publishes its oldest secret block in this case. This does not change which blocks end up in the main chain.

The objective function. We take as objective function of each miner the fraction of the blocks of the main chain that it mines in the long run. The reason is the adjustment of the hardness parameter (see Section 3.2.4), which only counts the blocks of the main chain and keeps its growth rate constant. In the long run, the main chain therefore grows at the same rate whatever the miners do, and the same amount of rewards is distributed per unit of time. Since every block of the main chain gives the same reward to its miner (we ignore transaction fees), each miner receives a share of these rewards equal to the fraction of the blocks of the main chain that it mined. Its costs (for instance, electricity) do not depend on its strategy, so maximising this fraction is the same as maximising the profit. With honest mining, the selfish miner would mine a fraction α of the blocks, so selfish mining pays off if and only if it yields a larger fraction.

Proposition 2. *In the above model, the long-run fraction of the blocks of the main chain that are mined by the selfish miner is*

$$R(\alpha, \zeta) = \frac{\mu_{\mathsf{sel}}}{\mu_{\mathsf{sel}} + \mu_{\mathsf{hon}}},$$

where

$$\mu_{\mathsf{sel}} = \alpha(1-\alpha)\left(2\alpha + \zeta(1-\alpha)\right) + \alpha^2\left(2 + \frac{\alpha}{1-2\alpha}\right) \qquad \textit{and} \qquad \mu_{\mathsf{hon}} = 1 - \alpha + \alpha(1-\alpha)^2(2-\zeta).$$

In particular, $R(\alpha, \zeta) > \alpha$ if and only if $\alpha > (1-\zeta)/(3-2\zeta)$.

Proof. We split the execution into *cycles*: a cycle starts when all miners, including the selfish one, mine on top of the same chain, and it ends the next time this happens. The blocks that are in the main chain at the end of a cycle stay there forever. Every cycle starts from the same situation and evolves independently of the previous ones. By the law of large numbers (the average of many independent samples of a random variable gets closer and closer to its expected value), the long-run fraction of the blocks of the main chain mined by the selfish miner is thus $\mu_{\mathsf{sel}}/(\mu_{\mathsf{sel}}+\mu_{\mathsf{hon}})$, where μ_{sel} and μ_{hon} are the expected numbers of blocks of the selfish miner and of the honest miners, respectively, that a cycle adds to the main chain. We distinguish three cases, depending on the first blocks of the cycle.

- With probability $1-\alpha$, the first block is found by an honest miner. The selfish miner mines on top of it, and the cycle ends with one honest block.
- With probability $\alpha(1-\alpha)$, the first block is found by the selfish miner and the second one by an honest miner, which starts a race. The next block ends the race and the cycle. With probability α, the selfish miner finds it and gets two blocks. With probability $(1-\alpha)\zeta$, an honest miner extends the chain of the selfish miner, and each side gets one block. With probability $(1-\alpha)(1-\zeta)$, an honest miner extends the other chain, and the honest miners get two blocks. In expectation, the selfish miner gets $2\alpha + (1-\alpha)\zeta$ blocks, and the honest miners get $(1-\alpha)\zeta + 2(1-\alpha)(1-\zeta) = (1-\alpha)(2-\zeta)$ blocks.
- With probability α^2, the first two blocks are found by the selfish miner, whose lead becomes 2. From then on, every block of the selfish miner increases the lead by one, and every honest block decreases it by one, until the lead drops from 2 to 1. At that point, the selfish miner publishes its chain, and the cycle ends. All the blocks of the selfish miner end up in the main chain, and all the honest ones are discarded. In expectation, the selfish miner gets $2 + \mathbb{E}[U]$ blocks, where U is the number of blocks that it finds after its lead reaches 2, and the honest miners get none.

Summing up the contributions of the three cases gives the expressions of μ_{sel} and μ_{hon} in the statement, provided that $\mathbb{E}[U] = \alpha/(1-2\alpha)$.

To compute $\mathbb{E}[U]$, let D be the expected number of blocks needed for the lead to drop by one, starting from any value $\ell \geq 2$. This number does not depend on ℓ: until the lead drops to $\ell - 1$, it is at least 2, and every block simply moves it up or down by one. Consider the first block. With probability $1-\alpha$, it is honest, and the lead drops after one block. With probability α, it is found by the selfish miner, and the lead

grows to $\ell+1$. The lead must then drop twice, from $\ell+1$ to ℓ and from ℓ to $\ell-1$, which takes $D+D$ more blocks in expectation. Therefore,

$$D = (1-\alpha)\cdot 1 + \alpha\cdot(1+2D) = 1+2\alpha D,$$

which gives $D = 1/(1-2\alpha)$ (one can show that D is finite when $\alpha < 1/2$). In the third case, the cycle ends when the lead drops from 2 to 1 for the first time, so the number N of blocks found after the lead reaches 2 has expected value D. Since the lead drops by one in total, these blocks include exactly one more honest block than blocks of the selfish miner, so $N = 2U+1$. Therefore, $\mathbb{E}[U] = (D-1)/2 = \alpha/(1-2\alpha)$.

Finally, $R(\alpha,\zeta) > \alpha$ if and only if $(1-\alpha)\mu_{\mathsf{sel}} > \alpha\mu_{\mathsf{hon}}$. Using $\alpha^2\,(2+\alpha/(1-2\alpha)) = \alpha^2(2-3\alpha)/(1-2\alpha)$, a direct computation shows that

$$(1-\alpha)\mu_{\mathsf{sel}} - \alpha\mu_{\mathsf{hon}} = \alpha(1-\alpha)^2\zeta + \frac{\alpha(1-\alpha)^2(3\alpha-1)}{1-2\alpha} = \frac{\alpha(1-\alpha)^2}{1-2\alpha}\cdot(\alpha(3-2\zeta)-(1-\zeta))\,.$$

Since $0<\alpha<1/2$, this is positive if and only if $\alpha(3-2\zeta) > 1-\zeta$, that is, $\alpha > (1-\zeta)/(3-2\zeta)$. □

Consequences. The threshold $(1-\zeta)/(3-2\zeta)$ of Proposition 2 is equal to $1/3$ for $\zeta=0$ and to $1/4$ for $\zeta = 1/2$, and it tends to 0 as ζ tends to 1. Therefore, the honest strategy profile is not a Nash equilibrium if some miner (or pool) controls more than a third of the computational power, and a selfish miner with a privileged position in the network profits even with a small fraction of the computational power. Pools of this size have existed (see Section 6.3.1). Eyal and Sirer proposed a simple mitigation: when two chains of the same length compete, honest miners should choose one of them uniformly at random, instead of the one that they received first. Then, the next honest block extends the chain of the selfish miner with probability $1/2$, whatever its position in the network, which corresponds to $\zeta = 1/2$ and to a threshold of $1/4$. Notice that selfish mining does not contradict the analysis of Section 3.2.5: the chain quality property (Theorem 11) only guarantees that the adversary obtains at most a fraction of about $\alpha/(1-\alpha)$ of the blocks of the main chain, which is larger than α. Variants of the protocol have also been proposed, such as FruitChains, in which honest behaviour is an (approximate) equilibrium even against coalitions controlling less than half of the computational power, in the sense of Definition 45.

Selfish mining pays off later. The reasoning that justifies our objective function only holds after the hardness parameter has been adjusted. Until then, the selfish miner collects at most as many rewards per unit of time as with honest mining: it finds blocks at the same rate, but some of them may be discarded when it loses a race. Above the threshold of Proposition 2, selfish mining thus becomes more profitable than honest mining only after the adjustment. We will meet the same phenomenon in Section 7.3.

7.2.3 The Proof-of-Stake Protocol of Ethereum*

We now describe the consensus protocol of Ethereum, called *Gasper*, which has been running since Ethereum switched to proof of stake in 2022. Gasper combines two mechanisms: a rule that selects the main chain, which plays the role of the longest chain principle, and a *finality layer*, through which blocks become final once validators holding two thirds of the stake have voted for them. Unlike in the protocol of Section 5.4, the proposers are known in advance, and all validators vote on the blocks (see also Section 5.8).

Gasper comes with fewer proven security guarantees than the protocol of Section 5.4, whose security can be proven under the assumption that honest validators hold the majority of the stake (we proved some of its properties in Section 5.5). For Gasper, only the safety of finalized blocks is well understood, and we prove it at the end of this section. The rest of the protocol, and in particular its rule for selecting the main chain, has only been analysed in part: its designers proved some properties of a version of the protocol, attacks against this rule were discovered afterwards, and the deployed protocol, which includes patches against these attacks, has, as far as we know, no complete security proof. Moreover, the security of Gasper relies on incentives much more than that of the protocol of Section 5.4: validators are rewarded for following the protocol and

punished for misbehaving, which only deters validators whose objective is to maximise their rewards (see Section 7.3). For these reasons, the approach of Ethereum should be taken with some criticism: its security rests to a large extent on informal arguments and on experience, and it is important to distinguish what has been proven from what is only believed.

This does not mean that Gasper is insecure, or that its design is worse. Its finality layer provides a guarantee that longest-chain protocols, such as Bitcoin or the protocol of Section 5.4, cannot offer. In these protocols, a block never becomes final: it only becomes less and less likely to be reverted as more blocks are added on top of it, so users must decide how many blocks to wait for. In Gasper, instead, a block is either final or not, and anybody can check which blocks are final from the votes recorded in the blockchain. When the network works normally, blocks become final within minutes, and honest validators never revert them. Most importantly, the safety of finalized blocks does not rely on bounded network delays:[2] two conflicting blocks can only be finalized if validators holding at least a third of the stake sign contradictory votes, and these votes prove their misbehaviour. The culprits can then be punished by destroying their stake, so reverting a finalized block costs at least a third of the total stake (see Theorem 20 and Corollary 5). Ethereum has also been running for years, securing a very large amount of value. In the rest of this section, we describe a simplified version of the protocol, ignoring many details.

Validators. To become a validator, a party deposits at least 32 ETH (the cryptocurrency of Ethereum) in a special smart contract, together with a public key of the BLS signature scheme (see Section 6.5). The deposit is the stake of the validator: it remains locked while the party acts as a validator, and it can only be withdrawn after a delay once the party leaves (see Section 5.7). The weight of every validator in the protocol is (essentially) proportional to their stake. We denote by V_{tot} the total stake of the validators.

Slots, epochs and proposers. Time is divided into slots of 12 seconds, which are grouped into epochs of 32 slots. In every slot, a single validator, called the *proposer*, can propose a new block. Proposers are selected at random, with probability proportional to their stake, using a randomness beacon called *RANDAO*. RANDAO relies on the fact that BLS signatures are *unique*: for every public key and every message, there is exactly one signature that passes verification. A unique signature on a fixed message behaves like the output of a VRF (see Section 5.2): the signer cannot choose among several valid signatures, everybody can check it with the public key, and nobody else can compute it in advance, by unforgeability (see Definition 16). The beacon works as follows. The blockchain stores a value called the *RANDAO mix*. Every block must contain a signature of its proposer on the current epoch number, and the block updates the mix, replacing it with the bitwise XOR of the mix and the hash of this signature. Since the signature is unique, the public key of each proposer fixes their contribution in advance, even though nobody else can compute it before the block is published. The mix at the end of each epoch determines the proposers and the committees (see below) of the epoch after the next one, so the proposers are publicly known at least an epoch in advance. Like the beacon of Section 5.6, RANDAO can be biased: the proposer of the last slot of an epoch can choose between two outputs, by publishing their block or not (at the price of the reward for the block), and an adversary who controls several of the last slots of an epoch has even more choices.

Committees and attestations. At every epoch, the validators are randomly partitioned into 32 *committees*, one for each slot. Every validator votes once per epoch: a third of the way into the slot of their committee, they broadcast an *attestation*, i.e., a signed message containing two votes:

- a *head vote* for the block that the validator considers the most recent block of the main chain, according to the fork-choice rule described below;
- a *checkpoint vote* (s, t), which is used by the finality mechanism described below.

[2] This holds as long as the stake of the validators does not change. Because of the inactivity leak described below, a network partition lasting several weeks can let both sides of the partition finalize conflicting blocks, even if all validators are honest.

Attesting and proposing are separate tasks: an attestation is recorded in the blockchain when the proposer of one of the following slots includes it in their block. Proposers combine the attestations that they receive from each committee into a BLS multisignature to save space (see Section 6.5). Only the attestations included in blocks count for the finality layer and for the rewards described below, which also give the proposers an incentive to include them.

Fork choice. Instead of the longest chain principle, the validators select the main chain using a rule called *LMD-GHOST* (*latest message driven greedy heaviest observed subtree*). For every validator, only their most recent head vote is taken into account. The *weight* of a block is the total stake of the validators whose most recent head vote is for that block or for one of its descendants. Starting from the most recent justified checkpoint (defined below), the rule repeatedly moves to the child with the largest weight, until it reaches a block without children, which is the head of the main chain. Unlike the longest chain principle, LMD-GHOST takes into account the votes of all validators, rather than only the blocks. The deployed protocol also gives some temporary extra weight to the block proposed in the current slot (this is called *proposer boost*), to mitigate some of the attacks that we mention below.

Finality. The finality layer of Gasper is called *Casper FFG* (*friendly finality gadget*). To simplify, we assume that the first slot of every epoch contains a block, which we call the *checkpoint* of the epoch, and we denote by $\mathsf{epoch}(c)$ the epoch of a checkpoint c (when the first slot of an epoch is empty, the deployed protocol uses the most recent block before it). The genesis block is the checkpoint of epoch 0. A checkpoint vote (s, t) consists of two checkpoints, called the *source* and the *target*, such that s is an ancestor of t (and $s \neq t$). An honest validator votes for the checkpoint of the current epoch on their main chain as target, and for the most recent justified checkpoint as source, where justified checkpoints are defined as follows.

- A *supermajority link* $s \to t$ is a checkpoint vote (s, t) cast by validators holding a total stake of at least $2V_{\mathsf{tot}}/3$.
- A checkpoint t is *justified* if it is the genesis block, or if there exists a supermajority link $s \to t$ from a justified checkpoint s.
- A checkpoint s is *finalized* if it is the genesis block, or if it is justified and there exists a supermajority link $s \to t$ such that $\mathsf{epoch}(t) = \mathsf{epoch}(s) + 1$.

Honest validators never revert a finalized checkpoint: their fork-choice rule only considers its descendants. When the protocol is not under attack, every checkpoint is justified at the end of its epoch and finalized at the end of the next one, so blocks become final after about two epochs. As in Bracha's protocol (see Theorem 4), the threshold of two thirds guarantees that any two sets of validators holding two thirds of the stake each share validators holding at least one third of it. The deployed protocol also finalizes checkpoints in a few more situations, which we ignore.

Slashing. A validator is slashed if they sign two conflicting messages, namely, two different blocks for the same slot, or two different attestations whose checkpoint votes (s_1, t_1) and (s_2, t_2) satisfy one of the following conditions:

- *double vote*: $\mathsf{epoch}(t_1) = \mathsf{epoch}(t_2)$;
- *surround vote*: $\mathsf{epoch}(s_1) < \mathsf{epoch}(s_2)$ and $\mathsf{epoch}(t_2) < \mathsf{epoch}(t_1)$, i.e., the first vote surrounds the second.

An honest validator casts a single checkpoint vote per epoch, and the epochs of their sources never decrease, so they never violate these conditions. Since all messages are signed, anybody can include two conflicting messages in a block as evidence. The slashed validator then loses their whole stake and is forced to leave the protocol.

Rewards and penalties. Rewards and penalties are computed by the protocol from the content of the blocks, in the same way as the balances of the accounts: every block updates them. In particular, every chain keeps its own record of rewards and penalties, and only the record of the main chain matters in the end. Validators receive rewards (newly created coins) for proposing blocks and for their attestations. When a block includes an attestation, the protocol checks three of its components, namely the head vote and the source and the target of the checkpoint vote, against the chain to which the block belongs: a component is correct if it coincides with the vote of an honest validator whose main chain is this chain. More precisely, the source must be the most recent justified checkpoint of this chain at the slot of the attestation, the target must be the checkpoint of the epoch of the attestation in this chain, and the head must be the most recent block of this chain at the slot of the attestation. Each correct component earns a reward, provided that the attestation is included quickly enough, while a validator whose attestation is not included in time, or has a wrong source or target, loses a similar amount. The reward of each component is multiplied by the fraction of the total stake whose attestations got that component right, so every validator earns more when everybody participates. When the network works well, all honest validators see the same main chain, and their votes are correct; a validator who voted for a branch that was later abandoned, instead, loses some rewards and may pay a penalty. Finally, the proposer of a block receives a reward for every correct component of the attestations that they include, provided that no earlier block of the chain included the same attestation. This is why proposers include as many attestations of other validators as they can.

Inactivity leak. If no checkpoint is finalized for more than four epochs, the protocol enters an *inactivity leak*: the validators that do not vote for the correct targets lose an amount of stake that grows quadratically with time. Eventually, the active validators hold two thirds of the stake again, and finality resumes. In this way, the chain eventually finalizes new blocks even if a third of the stake goes offline (for instance, because of a network partition), at the expense of the offline validators.

What Can We Prove?*

The finality mechanism satisfies a strong guarantee, called *accountable safety*: if two conflicting checkpoints are finalized, validators holding at least a third of the stake can be identified as culprits. We say that two checkpoints *conflict* if neither of them is an ancestor of the other. To simplify, we assume that the set of validators and their stake do not change.

Theorem 20 (Accountable safety)**.** *If two conflicting checkpoints are finalized, then validators holding a total stake of at least $V_{\mathsf{tot}}/3$ have violated one of the slashing conditions.*

Proof. If two sets of validators hold at least $2V_{\mathsf{tot}}/3$ each, then the validators in their intersection hold at least $2V_{\mathsf{tot}}/3 + 2V_{\mathsf{tot}}/3 - V_{\mathsf{tot}} = V_{\mathsf{tot}}/3$. Therefore, it suffices to find two supermajority links whose votes violate one of the slashing conditions: every validator that cast both votes is then slashable.

Let a and b be two conflicting finalized checkpoints and assume, without loss of generality, that $\mathsf{epoch}(a) \leq \mathsf{epoch}(b)$. Neither of them is the genesis block, which is an ancestor of every other checkpoint, so $\mathsf{epoch}(a) \geq 1$. Since a is finalized, there is a supermajority link $a \to a'$ with $\mathsf{epoch}(a') = \mathsf{epoch}(a) + 1$. Since b is finalized, it is also justified, so there is a sequence of checkpoints $b_0, b_1, \ldots, b_m$, where b_0 is the genesis block and $b_m = b$, such that $b_{i-1} \to b_i$ is a supermajority link for every $i \in [m]$. Since the source of every vote is an ancestor of its target, every b_i is an ancestor of b (or b itself), and $\mathsf{epoch}(b_0) < \mathsf{epoch}(b_1) < \cdots < \mathsf{epoch}(b_m)$. In particular, no b_i is equal to a or a', otherwise a would be an ancestor of b. Let j be the largest index such that $\mathsf{epoch}(b_j) \leq \mathsf{epoch}(a)$ (it exists, since $\mathsf{epoch}(b_0) = 0$). We distinguish two cases.

If $\mathsf{epoch}(b_j) = \mathsf{epoch}(a)$, then $j \geq 1$, since $\mathsf{epoch}(a) \geq 1$. Since a is justified, there is a supermajority link $c \to a$ for some checkpoint c. The votes (b_{j-1}, b_j) and (c, a) are different (since $b_j \neq a$) and their targets belong to the same epoch: this is a double vote.

If $\mathsf{epoch}(b_j) < \mathsf{epoch}(a)$, then $j < m$, since $\mathsf{epoch}(b_m) \geq \mathsf{epoch}(a)$. By the choice of j, we have $\mathsf{epoch}(b_{j+1}) \geq \mathsf{epoch}(a) + 1 = \mathsf{epoch}(a')$. If $\mathsf{epoch}(b_{j+1}) = \mathsf{epoch}(a')$, the votes (b_j, b_{j+1}) and (a, a') are different (since $b_{j+1} \neq a'$) and their targets belong to the same epoch: this is a double vote. Otherwise,

$$\mathsf{epoch}(b_j) < \mathsf{epoch}(a) < \mathsf{epoch}(a') < \mathsf{epoch}(b_{j+1}),$$

so the vote (b_j, b_{j+1}) surrounds the vote (a, a'): this is a surround vote. □

Notice that Theorem 20 holds whatever the network delays and whatever the fraction of corrupted validators: if safety is violated, the culprits are identified. Casper FFG also satisfies a liveness property, called *plausible liveness*: honest validators holding at least two thirds of the stake can always finalize new checkpoints without violating the slashing conditions. Whether they actually do so, however, depends on the fork-choice rule and on the network.

From accountability to incentives. Theorem 20 allows us to give a game-theoretic argument for the safety of finalized blocks. Consider a coalition of validators that deviates from the protocol in order to finalize two conflicting checkpoints, for instance, to spend the same coins on both branches. Let G be the total profit that the coalition obtains in this way. Assume also that the honest validators include the evidence of all the violations in the blockchain before the culprits can withdraw their stake (this is one of the reasons why withdrawals are delayed).

Corollary 5. *Under the above assumptions, the members of the coalition lose a total stake of at least $V_{\mathsf{tot}}/3$. In particular, if $G < V_{\mathsf{tot}}/3$, the deviation decreases the total objective function of the coalition.*

Proof. By Theorem 20, validators holding a total stake of at least $V_{\mathsf{tot}}/3$ violated one of the slashing conditions. They are all members of the coalition, since honest validators never violate the slashing conditions. Since the evidence is included before they can withdraw their stake, all of them are slashed and lose their whole stake. Therefore, the coalition loses a total stake of at least $V_{\mathsf{tot}}/3$. □

In the language of Definition 45, with respect to the deviations that finalize conflicting checkpoints, the honest strategy profile is resilient against coalitions of any size, as long as the profit that they can obtain from such a deviation is smaller than a third of the total stake. This guarantee is often called the *economic security* of finality. Notice that it only covers one kind of deviation, and that it is an economic guarantee: the attack remains possible, but it is expensive.

What we cannot prove. The guarantees above only concern finalized blocks. The safety of the most recent blocks, which are not yet final, and the liveness of the protocol depend on the fork-choice rule, and the attacks mentioned at the beginning of this section target precisely this rule. For instance, in a *balancing attack*, the adversary keeps the honest validators split between two branches, so that no checkpoint gets justified and the chain stops finalizing. Proposer boost was introduced to prevent attacks of this kind. Incentives are not fully understood either. A validator who maximises their protocol rewards has little reason to deviate alone from the honest voting strategy, since missing or wrong votes reduce their rewards, and conflicting votes are slashed. For proposers, however, the honest strategy is not a Nash equilibrium. The fees and the other profits that a proposer can obtain by choosing and ordering the transactions of their block grow during the slot, so a proposer earns more by publishing their block later than prescribed, as long as the block still receives enough votes. Such *timing games* have been observed in practice.

7.3 The Limits of Game Theory

Game theory allows us to analyse protocols in which the parties are neither honest nor malicious, but rational. Its conclusions, however, depend on how we model the players, and real people are hard to model. We now discuss two of the main limits of this approach.

What do people want? The analysis of a game starts from the objective functions of the players and, in blockchains, these are usually assumed to be the profits of the parties. The world, however, is more complicated. Some people maximise their fame: for instance, a hacker may attack a blockchain just to show that it can be done, even if this costs them money. Others act according to their ideological or religious beliefs, and care more about their chances of a good afterlife than about their wealth. Others prefer an

immediate gratification to a larger profit in the future. Even people who only care about money are often risk averse rather than interested in their average profit: this is why pools exist, even though they do not increase the average income of their members (see Section 6.3). Moreover, objectives are not fixed: they change over time, often in unpredictable ways, for instance, with the price of a cryptocurrency or with the personal circumstances of each party. A strategy profile that is an equilibrium for some objective functions may not be an equilibrium for others.

Systems do not work in isolation. A game describes a single system, whose players only care about the outcome of that system. Every blockchain, however, is part of a much larger world, and a party may have an interest in losing money in the system today, because this gives them an opportunity to gain much more tomorrow, possibly in a completely unexpected way. For instance, an attacker may first bet on a fall of the price of a cryptocurrency (for instance, by short selling it, i.e., by selling borrowed coins and buying them back later) and then attack its blockchain. Similarly, a competitor or a government may want to destroy a blockchain at any cost (this is sometimes called a *Goldfinger attack*). None of these gains and losses appears in the objective functions of the game.

Example: infiltrating a pool. Pools of proof-of-work miners illustrate both limits. Consider two pools A and B, which control fractions φ_A and φ_B of the total computational power, and suppose that B shares its rewards proportionally to the shares submitted by its members (see Section 6.3.1). Pool A can use a fraction $x \leq \varphi_A$ of the total computational power to infiltrate B: these miners join B and submit their shares, but they never reveal the solutions they find (this is the block withholding attack of Section 6.3.1).

Since mining is random, we consider expected rewards. Let R_{tot} be the expected total reward per unit of time when all miners publish their solutions (the reward of a block times the expected number of blocks per unit of time). As long as the hardness parameter does not change, a group of miners that controls a fraction φ of the computational power and publishes its solutions earns $\varphi \cdot R_{\mathsf{tot}}$ per unit of time in expectation. The miners who stay in A thus earn $(\varphi_A - x) \cdot R_{\mathsf{tot}}$. The infiltrators earn nothing directly, but B shares with them the rewards $\varphi_B \cdot R_{\mathsf{tot}}$ of its honest members: since B pays its members proportionally to their shares, and thus to their computational power, the infiltrators receive a fraction $x/(\varphi_B + x)$ of these rewards. In total, the expected reward of A per unit of time is

$$r_A(x) = \left(\varphi_A - x + \varphi_B \cdot \frac{x}{\varphi_B + x}\right) \cdot R_{\mathsf{tot}} = \left(\varphi_A - \frac{x^2}{\varphi_B + x}\right) \cdot R_{\mathsf{tot}}.$$

For every $x > 0$, this is less than $\varphi_A \cdot R_{\mathsf{tot}}$, the expected reward of A without the attack. If A maximises its expected profit, its best strategy is thus $x = 0$: the miners of A mine for their own pool.

Suppose, instead, that A maximises its profit relative to everybody else, that is, its fraction of all rewards. Since the infiltrators discard their solutions, the expected total reward per unit of time drops from R_{tot} to $(1 - x) \cdot R_{\mathsf{tot}}$, so the fraction of all rewards that A receives is

$$\frac{r_A(x)}{(1-x) \cdot R_{\mathsf{tot}}} = \frac{1}{1-x}\left(\varphi_A - \frac{x^2}{\varphi_B + x}\right).$$

This is larger than φ_A whenever $0 < x < \varphi_A\varphi_B/(1 - \varphi_A)$.

This example also illustrates the second limit. Recall that proof-of-work blockchains periodically adjust the hardness parameter, so that blocks keep being generated at the same rate (see Section 3.2.4). After the adjustment, the miners who publish their solutions generate, in expectation, as many blocks per unit of time as all miners did before the attack, so every expected reward computed above is multiplied by $1/(1-x)$. The expected total reward per unit of time returns to R_{tot}, and the expected reward of A per unit of time becomes its fraction of all rewards times R_{tot}. As we have just seen, this is larger than $\varphi_A \cdot R_{\mathsf{tot}}$, the expected reward of A without the attack, whenever $0 < x < \varphi_A\varphi_B/(1 - \varphi_A)$. Thus, by accepting a lower expected reward today (until the next adjustment), A obtains a higher expected reward tomorrow: in the long run, even a pool that only maximises its own expected profit has an incentive to infiltrate other pools (as for selfish

mining, see Section 7.2.2). When two pools can infiltrate each other, they face a version of the prisoner's dilemma, called the *miner's dilemma*: not attacking is not an equilibrium and, in the only equilibrium, both pools attack each other, and both earn less in expectation than if neither had attacked.

Chapter 8

Anonymity and Privacy

Blockchains are public: anybody can read the whole history of the transactions (see Sections 4.1 and 4.2). This transparency is what allows everybody to check that the blockchain is valid, but it comes at a price. Imagine a bank that published the statements of all its customers on the Internet: even without names, most customers would object. In this chapter, we study how much the blockchains of the previous chapters reveal about their users, and how they can reveal less. We distinguish two goals. *Anonymity* means hiding who is involved in a transaction, for instance, who pays whom. *Privacy* (or *confidentiality*) means hiding the content of a transaction, for instance, the amounts. As we will see, cryptography allows us to achieve both, while keeping the blockchain publicly verifiable.

8.1 Anonymous Communication

Before looking at what the blockchain reveals, we must look at how transactions reach it. To publish a transaction, a user sends it to some nodes of the network, which forward it to the other nodes. The nodes that receive the transaction directly from the user learn the user's IP address, which internet providers (and, on request, the police) can usually link to a name. An adversary that runs many nodes, or that observes the traffic of many nodes, can often identify the node from which a transaction started spreading: in 2014, researchers showed that an attacker connected to many Bitcoin nodes could link a large fraction of the transactions to the IP addresses of the users who created them. Whatever we do on the blockchain, anonymity is lost if the network reveals the sender. Users who want to be anonymous must therefore connect to the network through an infrastructure that hides their IP address.

The most widely used one is *Tor* (originally short for *The Onion Router*), which most people access through the Tor Browser. Tor is a network of relays run by volunteers. To send a message, the user chooses a short path of relays and encrypts the message in several layers, one for each relay of the path, like the layers of an onion. Each relay removes one layer, which reveals the next relay of the path, and forwards the rest. In this way, the first relay knows who sends the message, but not where it goes, while the last relay knows where the message goes, but not who sent it. Many blockchain nodes can be reached as Tor services, so that the message never leaves the Tor network. Tor is fast enough for browsing the web, but it does not protect against an adversary who observes the traffic entering and leaving the network: by comparing the timing of the messages, such an adversary can often match senders and recipients. *Mix networks* (or *mixnets*) address this problem. Each relay of a mixnet, called a *mix*, collects a batch of messages from many senders, removes a layer of encryption from each of them and forwards them in a random order, often after a random delay, so that even an observer of the whole network cannot match the messages that enter a mix with those that leave it. This stronger protection comes at the price of a higher latency.

An anonymous network hides which of its users sent a message, but not who its users are: the internet provider of a user, or the administrator of their local network, can usually see that the user connects to Tor, although not what they do with it. A user of an anonymous network is therefore only hidden among the

other people who were using it at the same time and, if these are few, the protection is weak. For instance, in 2013, a student at Harvard University sent emails announcing bombs in several buildings of the campus, hoping to avoid a final exam. He sent them through Tor and a disposable email service, so the emails themselves did not reveal where they came from. However, the logs of the university network showed that he had connected to Tor from the campus wireless network in the hours before the emails were sent. Tor had hidden what he did, but not that he was using Tor at that time and, since few people on the campus did so, this was enough to make him a suspect: when the police questioned him, he confessed. We will formalise this idea in Section 8.3, with the notion of anonymity set.

In the rest of this chapter, we assume that users communicate through an anonymous network, and we focus on what the blockchain itself reveals.

8.2 Pseudonymity

Bitcoin is often described as anonymous. It is not: it is *pseudonymous*. In the UTXO system (see Section 4.1), the owner of an output is identified by a public key (the address), not by a name. A public key therefore works as a *pseudonym*: an identifier that is not directly linked to a person, but that is the same in all the transactions that involve it. Everything else is public: the amounts, the outputs spent by each transaction, and the time at which each transaction was added to the blockchain. As soon as a pseudonym is linked to a person, all the transactions that involve it are linked to that person, including those that happened years before. Such links are easy to establish: in most countries, exchanges that convert cryptocurrency into traditional money are required by law to identify their customers, a shop that ships goods learns the postal address of the buyer, and many people publish their public keys, for instance, to receive donations.

One-time addresses. The obvious countermeasure is to never reuse a public key. The recipient of every payment generates a fresh key pair and gives the new public key to the payer, and the change of every transaction (see Section 4.1) is sent to a fresh public key of the payer. The Bitcoin whitepaper already recommended this practice, and today's wallets follow it automatically, deriving all key pairs from a single secret, so that users do not need to back up a new key after every payment. In this way, every public key appears in at most two transactions: the one that creates its output and the one that spends it. If the public key is linked to a person, only these two transactions are revealed.

Why one-time addresses are not enough. One-time addresses hide who owns each output, but not where its coins come from, nor where they go. In the UTXO system, every input of a transaction refers to an output of an earlier transaction (see Section 4.1). Starting from any output, anybody can therefore trace its history backward: they look at the transaction that created the output, at the outputs spent by that transaction, at the transactions that created those outputs, and so on, until they reach the coinbase transactions in which the coins were created (see Section 4.4). In the same way, anybody can follow the coins forward, since every output is spent by at most one later transaction, whose outputs are spent by further transactions, and so on. Together, the transactions form a public graph, in which every transaction is connected to the transactions that created its inputs and to those that spend its outputs. Since the amounts are public too, the graph often shows quite precisely how an amount moved: if a transaction spends a single input, all its outputs come from that input, while if it spends several inputs, the blockchain does not say which input paid for which output, and the history of each output includes all of them.

For example, suppose that Alice withdraws some coins from an exchange, which knows her identity, to a fresh public key of hers. She then uses them to pay Bob, sending the payment to a fresh public key of Bob and the change to another fresh public key of hers. Later, Bob deposits the payment at the same exchange, which knows his identity too. By following the graph from Alice's withdrawal to Bob's deposit, the exchange learns that the coins withdrawn by Alice ended up with Bob, even though neither of them ever reused a public key, and even if the coins passed through other transactions in between. Similarly, Bob can trace his coins backward to Alice's withdrawal, and Alice can follow her payment forward and see how Bob spends it. Combined with a few simple heuristics, the graph also links the different public keys of the same user.

- *Common inputs.* All the inputs of a transaction must be signed by their owners, so a transaction with several inputs is usually created by a single user who owns all of them (for instance, to pay an amount larger than the value of each of their outputs), as the Bitcoin whitepaper itself acknowledged. Applying this rule to all transactions groups the public keys into *clusters*, each of which typically belongs to a single user or organisation.
- *Change.* A payment typically has two outputs: the payment itself and the change. The change output can often be recognised (for instance, the payment is a round amount, while the change is not), which links it, and the transactions that later spend it, to the payer.
- *Identities.* As soon as one public key of a cluster is linked to a person, for instance, through a deposit at an exchange, the whole cluster is.

Companies specialised in this kind of analysis sell their services to exchanges and to law enforcement agencies, who use them to trace stolen funds, ransoms and other criminal proceeds. Account-based systems (see Section 4.2), such as Ethereum, reveal even more: an account is designed to be used many times, and a fresh account needs some cryptocurrency to pay the fee of its first transaction, which links it to the account that sends it.

8.3 Mixing Coins

To break the links of the transaction graph, users can mix their coins with those of other users. The simplest solution is a *mixer*: a service to which many users send their coins, and which sends the same amounts (minus a fee) back to fresh public keys of the users, using coins received from other users. An observer of the blockchain sees coins entering and leaving the mixer, but cannot tell which outgoing coins correspond to which incoming ones. The mixer, however, knows exactly who received what. It may keep this information, reveal it to others, or simply keep the coins.

CoinJoin. Mixing does not actually need a trusted party, since the inputs of a UTXO transaction may belong to different users, as long as each of them signs (see Section 4.1). In a *CoinJoin*, several users create a single transaction together. Each participant contributes one or more inputs and chooses a fresh public key for each of their outputs, and all the main outputs have the same value (the excess of each participant, if any, goes to a change output). Once all inputs and outputs have been collected in a transaction, every participant checks that their outputs are included, and only then signs their inputs. Since the transaction is only valid if all its inputs are signed, nobody can steal the coins of the others: a participant whose outputs are missing simply refuses to sign. An observer sees a transaction with many inputs and many outputs of the same value, and cannot tell which output belongs to which participant. Notice that CoinJoin purposely breaks the common-input heuristic of Section 8.2: the inputs of a CoinJoin belong to different users.

Somebody must still collect the inputs and outputs and assemble the transaction. This *coordinator* cannot steal any coins but, if it learns which participant registered which output, it can undo the mixing. We will see in Section 8.5.2 how blind signatures prevent this.

Anonymity sets. To quantify the anonymity that mixing provides, we look at the problem from the point of view of the adversary. The *anonymity set* of an action (for instance, receiving a given output or, in the example of Section 8.1, sending the emails) is the set of users who, given everything the adversary has observed, could have performed it. If the adversary has no reason to prefer any member of the anonymity set over the others, it identifies the right one with probability one over the size of the set: the larger the anonymity set, the stronger the anonymity. Without mixing, the anonymity set of an output often contains a single user. In a CoinJoin with n participants, the anonymity set of each of the outputs of the same value contains at most all the n participants. Several factors can make it smaller.

- *Participants controlled by the adversary.* If the adversary controls k of the n participants, it knows their outputs, so the anonymity set of the other outputs only contains $n - k$ participants. In particular, an adversary who joins a CoinJoin with $n - 1$ different identities completely deanonymises the remaining participant.
- *Amounts.* Outputs of different values, such as the change outputs, can often be matched with the inputs that funded them.
- *Later transactions.* If a user later spends two of their mixed outputs in the same transaction, or a mixed output together with an unmixed one, the common-input heuristic links them again.

Users can enlarge their anonymity sets by mixing their coins several times, with different groups of participants, at the price of paying more fees.

From identities to amounts. CoinJoin hides who owns which output, but not the amounts: the mixed outputs must all have the same value, precisely because the amounts are public. More generally, in all the systems that we have seen so far, the amount of every transaction is visible to everybody. Next, we introduce two cryptographic tools that allow the validators to check that a transaction is valid, even if its amounts are hidden.

8.4 Commitments and Zero-Knowledge Proofs

Suppose that the amount of every output is hidden. To check that a transaction is valid, the validators need two tools. The first one fixes the hidden amounts, so that the owner of an output cannot later claim that it contains a different amount: this is a *commitment scheme*. The second one allows a user to convince the validators that a statement about hidden values is true, for instance, that the inputs and the outputs of a transaction contain the same total amount, without revealing the values: this is a *zero-knowledge proof*.

8.4.1 Commitments

A commitment is the digital analogue of a safe. The *committer* puts a message x in a safe, locks it and gives the safe to the *receiver*, keeping the key. The receiver cannot see x without the key, and the committer can no longer change x, since the safe is in the hands of the receiver. Later, the committer may *open* the commitment by giving the key to the receiver. In the definition below, the commitment plays the role of the locked safe, and its opening plays the role of the key.

Definition 46 (Commitment scheme)**.** *A commitment scheme with message space $\mathcal{X}$ and randomness space $\mathcal{R}$ is a deterministic polynomial-time algorithm Com, which takes as input a message $x \in \mathcal{X}$ and some randomness $r \in \mathcal{R}$, and outputs a commitment $c = \mathsf{Com}(x, r)$.*

To commit to a message x, the committer samples r uniformly at random from $\mathcal{R}$ and sends $c := \mathsf{Com}(x, r)$ to the receiver. To open the commitment, the committer sends (x, r), and the receiver accepts if $\mathsf{Com}(x, r) = c$.

Definition 47 (Hiding and binding)**.** *Let $T \in \mathbb{N}$ and $\epsilon \in [0, 1]$. A commitment scheme Com is:*

- *(T, ϵ)-*hiding *if, for every pair of messages $x_0, x_1 \in \mathcal{X}$ and every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

$$\left| \Pr\left[\mathcal{A}(c) = 1 \middle| \begin{array}{l} r \xleftarrow{\$} \mathcal{R} \\ c := \mathsf{Com}(x_0, r) \end{array} \right] - \Pr\left[\mathcal{A}(c) = 1 \middle| \begin{array}{l} r \xleftarrow{\$} \mathcal{R} \\ c := \mathsf{Com}(x_1, r) \end{array} \right] \right| \leq \epsilon;$$

- (T, ϵ)-binding *if, for every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

$$\Pr\left[\begin{array}{c} x \neq x' \\ \mathsf{Com}(x, r) = \mathsf{Com}(x', r') \end{array} \middle| (x, r, x', r') \xleftarrow{\$} \mathcal{A}\right] \leq \epsilon.$$

In the hiding property, $\mathcal{A}$ plays the role of a receiver who tries to tell whether c is a commitment to x_0 or to x_1. If it could tell the two cases apart, it would output 1 more often in one case than in the other: hiding requires the two probabilities to differ by at most ϵ, whatever the two messages are. In the binding property, $\mathcal{A}$ plays the role of a committer who tries to find a commitment that it can open to two different messages: as for the collision resistance of hash functions (see Definition 14), it succeeds with probability at most ϵ.

An example. In the random oracle model (see Section 1.4.1), we can commit with a hash function: $\mathsf{Com}(x, r) := \mathsf{Hash}(x, r)$, where r is a long random string. Binding follows from collision resistance, since two different openings $(x, r) \neq (x', r')$ of the same commitment form a collision of Hash. Hiding holds because the digests of a random oracle look random. The randomness r is essential: if we committed to x with $\mathsf{Hash}(x)$, a receiver who suspects that x is either x_0 or x_1 could simply compute $\mathsf{Hash}(x_0)$ and $\mathsf{Hash}(x_1)$ and compare them with the commitment.

8.4.2 Zero-Knowledge Proofs

In Section 6.4, we introduced non-interactive arguments, which allow a prover who knows a witness w for a statement x to convince a verifier that x is true (see Definition 42 for the definition and the completeness property, and Definition 43 for soundness). There, our goal was succinctness: we wanted proofs that are much shorter and much faster to check than the witness. Here, we have a different goal: the proof must not reveal anything about the witness, except that it exists. For instance, the owner of an output with a hidden amount may want to prove that the amount is non-negative, without revealing it.

How can we formalise that a proof reveals nothing? The idea is to compare the real world, in which the verifier receives a real proof, with an ideal world, in which the verifier receives a fake proof, produced by an algorithm called the *simulator*, which knows the statement x, but not the witness w. If the two worlds look the same, the proof reveals nothing about w: whatever the verifier can learn from a real proof, it could also learn from a fake one, which was computed without w. The simulator, however, cannot simply compute valid proofs of x without knowing w: if it could, a cheating prover could do the same for false statements, contradicting soundness. What distinguishes the simulator from a cheating prover is that the simulator is allowed to generate the public parameters itself, together with some secret information td, called the *trapdoor*, which allows it to compute fake proofs. In the real world, the public parameters are generated by Setup, and nobody knows the trapdoor (see the discussion on the generation of the public parameters in Section 6.4). Notice that the simulator is only a mathematical artifact, which we use to prove that a proof system is zero-knowledge: in reality, it is never implemented or run.

Definition 48 (Zero knowledge)**.** *Let $T \in \mathbb{N}$ and $\epsilon \in [0, 1]$. A non-interactive argument* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *for an NP relation R is* (T, ϵ)-zero-knowledge *if there exists a pair of probabilistic polynomial-time algorithms* $(\mathsf{SimSetup}, \mathsf{SimProve})$*, called the* simulator*, with the following property: for every pair $(x, w) \in R$ and every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

$$\left|\Pr\left[\mathcal{A}(\mathsf{crs}, x, \pi) = 1 \middle| \begin{array}{l} \mathsf{crs} \xleftarrow{\$} \mathsf{Setup} \\ \pi \xleftarrow{\$} \mathsf{Prove}(\mathsf{crs}, x, w) \end{array}\right] - \Pr\left[\mathcal{A}(\mathsf{crs}, x, \pi) = 1 \middle| \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \xleftarrow{\$} \mathsf{SimSetup} \\ \pi \xleftarrow{\$} \mathsf{SimProve}(\mathsf{crs}, \mathsf{td}, x) \end{array}\right]\right| \leq \epsilon.$$

Above, $\mathcal{A}$ plays the role of a verifier who tries to tell whether it received a real proof, computed with the witness, or a fake one, computed by the simulator without the witness. As for the hiding property of commitments, the definition requires that the probability that $\mathcal{A}$ outputs 1 is almost the same in the two worlds. A non-interactive argument with this property is commonly called a (non-interactive) *zero-knowledge proof*.

Many proofs. Definition 48 only considers a single proof, for a statement fixed before the public parameters are generated. In a blockchain, however, the public parameters are generated once and for all, and then used by everybody to produce many proofs, for statements that may depend on the public parameters and on the proofs published so far. The following definition considers an adversary that receives the public parameters and then obtains as many proofs as it wants, for statements and witnesses of its choice, chosen one after the other.

Definition 49 (Adaptive multi-theorem zero knowledge)**.** *Let $T \in \mathbb{N}$ and $\epsilon \in [0,1]$. A non-interactive argument* $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *for an NP relation R is* (T,ϵ)-adaptively multi-theorem zero-knowledge *if there exists a pair of probabilistic polynomial-time algorithms* $(\mathsf{SimSetup}, \mathsf{SimProve})$ *with the following property: for every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

$$\left|\Pr\left[\mathcal{A}^{\mathcal{O}_{\mathsf{real}}}(\mathsf{crs}) = 1 \middle| \mathsf{crs} \xleftarrow{\$} \mathsf{Setup}\right] - \Pr\left[\mathcal{A}^{\mathcal{O}_{\mathsf{sim}}}(\mathsf{crs}) = 1 \middle| (\mathsf{crs},\mathsf{td}) \xleftarrow{\$} \mathsf{SimSetup}\right]\right| \leq \epsilon.$$

Above, $\mathcal{A}$ has unrestricted access to an oracle (see Definition 16). On input a pair $(x,w) \in R$, the oracle $\mathcal{O}_{\mathsf{real}}$ responds with a real proof $\pi \xleftarrow{\$} \mathsf{Prove}(\mathsf{crs},x,w)$, while $\mathcal{O}_{\mathsf{sim}}$ responds with a simulated proof $\pi \xleftarrow{\$} \mathsf{SimProve}(\mathsf{crs},\mathsf{td},x)$, computed without w. On inputs that do not belong to R, both oracles refuse to answer.

The word *adaptive* refers to the fact that the adversary chooses each statement after seeing the public parameters and the previous proofs, while *multi-theorem* refers to the fact that it obtains many proofs (in this context, statements are also called theorems), all computed with the same public parameters and, in the ideal world, with the same trapdoor. Definition 49 implies Definition 48, since the adversary may ask for a single proof, of a fixed statement, but the converse does not hold in general. The zero-knowledge proofs used in blockchains are designed to satisfy the stronger definition.

Zero knowledge and succinctness. Zero knowledge and succinctness are independent properties. A *zk-SNARK* is a SNARK (see Section 6.4) that is also zero-knowledge: its proofs are short, fast to check and reveal nothing about the witness. As we observed in Section 6.4, however, the term "zero knowledge" is often used inappropriately in the blockchain world, to refer to any SNARG or SNARK. Many SNARKs used in blockchains are not zero-knowledge, since they only speed up the verification of large computations whose inputs are public anyway: this is the case of many zk-rollups, despite their name. Conversely, many zero-knowledge proofs are not succinct. For instance, verifying the range proofs that we use below, which show that a committed amount lies in a given interval, is not faster than checking the witness itself: their only purpose is to keep the witness secret.

8.4.3 Confidential Transactions in Mimblewimble

We now use commitments and zero-knowledge proofs to hide the amounts in a UTXO system, following the approach of *confidential transactions*, as in Mimblewimble, a protocol used, for instance, by the cryptocurrency Grin and by an optional extension of Litecoin. In a confidential transaction, every output contains a commitment c to its amount v, instead of v itself. The owner of an output knows the opening (v,r) of its commitment. To check that a transaction is valid without learning its amounts, the validators need two guarantees.

Balance: The total amount committed in the inputs must equal the total amount committed in the outputs (plus the transaction fee, which remains public). The commitments used in Mimblewimble have an algebraic structure that allows anybody to check this equality directly from the commitments, without opening them. We do not describe how this works.

No negative amounts: Every output must commit to an amount in a fixed interval of non-negative values. Without this check, balance would not be enough: a user who owns an output with amount v could

create two outputs with amounts $v+a$ and $-a$, obtaining a coins out of nothing. For this reason, every output comes with a *range proof*, a zero-knowledge proof that its commitment opens to a value in the interval: the statement is the commitment c, and the witness is its opening (v, r).

Merging transactions. Mimblewimble has two further features. First, it has no addresses: the payer and the payee build every transaction together, interactively, and an output belongs to whoever knows the opening of its commitment (spending it also requires a signature, which we omit). Second, the transactions of a block are merged into a single large transaction, so that the block does not reveal which inputs and outputs belonged to the same transaction, and the outputs that are created and spent within the same block are removed. Since spent outputs can be removed from the history, the blockchain essentially consists of the unspent outputs and of a small record for every transaction, which keeps it small.

How much anonymity? The amounts are hidden, but the transaction graph is mostly public: every input still points to the output that it spends, and merging only hides the links within a block. Moreover, a new transaction spreads through the network on its own before being merged into a block, so an observer connected to many nodes can see it before it is merged. In 2019, a researcher who connected to a fraction of the nodes of Grin reported that he could link the inputs and outputs of most of its transactions in this way. Mimblewimble therefore offers strong privacy for the amounts, but weak anonymity.

8.4.4 Zerocash

Zerocash, the protocol on which the cryptocurrency Zcash is based, hides not only the amounts, but the whole transaction graph. Its main ingredients are commitments, Merkle trees (see Section 6.2) and zk-SNARKs. We describe a simplified version.

Notes. Each user samples a random secret key sk and publishes the public key $\mathsf{pk} := \mathsf{Hash}(\mathsf{sk})$. Coins are replaced by *notes*: a note is a triple (pk, v, ξ), where pk is the public key of its owner, v is its amount, and ξ is a random string that makes it unique. The blockchain does not contain the notes, but only commitments $c = \mathsf{Com}((\mathsf{pk}, v, \xi), r)$ to them. All the commitments ever created are the leaves of a Merkle tree, whose root z is updated whenever a new commitment is added (see Section 6.2.2). The owner of a note knows its opening $((\mathsf{pk}, v, \xi), r)$, which the payer sends to them, for instance, encrypted so that only the owner can read it (in Zcash, these encrypted openings are published in the blockchain).

Spending a note. To spend a note (pk, v, ξ) with commitment c, its owner chooses the new notes, for instance, a note (pk', v', ξ') for the payee and a note $(\mathsf{pk}, v-v', \xi'')$ for the change, and publishes a transaction containing:

- the commitments c' and c'' to the new notes;
- the *nullifier* $\mathsf{nf} := \mathsf{Hash}(\mathsf{sk}, \xi)$ of the spent note;
- a zk-SNARK proof for the statement $(z, \mathsf{nf}, c', c'')$, where z is the current root of the Merkle tree. The witness consists of the spent note (pk, v, ξ), the randomness r of its commitment c, an opening of c in the Merkle tree, the secret key sk, and the openings of c' and c''. The proof shows that:
  - $c = \mathsf{Com}((\mathsf{pk}, v, \xi), r)$ is a leaf of the Merkle tree with root z, and $\mathsf{pk} = \mathsf{Hash}(\mathsf{sk})$;
  - $\mathsf{nf} = \mathsf{Hash}(\mathsf{sk}, \xi)$;
  - the amounts of the new notes are non-negative and add up to v.

The validators check the proof and check that nf does not appear in any previous transaction. Then, they add c' and c'' to the Merkle tree. Notice that the transaction does not contain c: the validators never learn which note is spent.

Security. Recall that a SNARK guarantees that whoever produces a valid proof knows a witness (see Section 6.4). Since the witness includes a secret key sk such that $\mathsf{Hash}(\mathsf{sk}) = \mathsf{pk}$, only the owner of the note can spend it. Moreover, every note has a single nullifier: the commitment c fixes ξ (by binding), and nobody can find two secret keys with the same public key (they would form a collision of Hash). Spending a note twice would thus publish the same nullifier twice, which the validators reject. Finally, no transaction can create money, since the amounts of the new notes add up to the amount of the spent note.

Anonymity and privacy. The zero-knowledge property guarantees that the proofs reveal nothing about their witnesses (here, we need Definition 49, since all users publish many proofs with the same public parameters): the validators learn that some note in the Merkle tree is being spent, but not which one, nor its amount or its owner. The commitments c' and c'' hide the amounts and the owners of the new notes, and the nullifier, computed with the secret key, looks random and cannot be linked to the commitment of the spent note. The anonymity set of a transaction therefore contains all the notes in the tree, that is, all the notes ever created. This is much stronger than CoinJoin, whose anonymity sets only contain the participants of the same transaction, and than Mimblewimble, which leaves most of the transaction graph visible. Like Mimblewimble, Zerocash hides the amounts. Moreover, the public keys of the payees never appear in the blockchain.

Limitations. This protection comes at a price. Computing a zk-SNARK proof is much more expensive than computing a signature, although it is now fast enough for ordinary computers. The first versions of Zcash relied on public parameters generated by a multiparty computation (see Section 6.4): if all its participants had colluded, they could have created money out of nothing without anybody noticing, since the amounts are hidden. A flaw in the first proof system of Zcash, discovered in 2018 and disclosed in 2019, would have had the same effect: it was fixed, and the developers found no evidence that anybody had exploited it. Since 2022, Zcash also supports a proof system that does not need any trusted generation of public parameters. Finally, Zcash also allows transparent transactions, which work as in Bitcoin, and coins that move between transparent and shielded transactions reveal their amounts. A study of the first years of Zcash found that most transactions were transparent, and that simple heuristics, such as matching the amounts that enter and leave the shielded notes, linked many of the shielded transactions, reducing their anonymity sets.

The same technique is also used on Ethereum. Tornado Cash is a smart contract that works as a mixer for deposits of a fixed amount: a user deposits the amount together with a commitment, and later withdraws it to a fresh account, with a zk-SNARK proof that they know the opening of one of the commitments deposited so far, and with a nullifier that prevents withdrawing twice. The anonymity set of a withdrawal contains all the deposits of the same amount.

What happened to Tornado Cash. The contracts of Tornado Cash are immutable: nobody, not even their developers, can modify or stop them. Tornado Cash was used both by people who simply wanted privacy and by criminals. In August 2022, the US Treasury placed it under sanctions, stating that it had been used to launder billions of dollars, including hundreds of millions stolen by a group of hackers linked to the North Korean government. From then on, US citizens, residents and companies were forbidden from interacting with it: for instance, Circle froze the USDC held by its addresses (see Section 4.5). The contracts, however, kept working. A few days after the sanctions, one of the developers of Tornado Cash was arrested in the Netherlands and, in 2024, a Dutch court convicted him of money laundering and sentenced him to more than five years in prison, a decision that he has appealed. In 2023, two other developers were charged in the United States with conspiring to launder money, to violate sanctions and to operate an unlicensed money transmitting business. In 2025, one of them was tried: the jury convicted him of the last charge, but could not agree on the other two, for which a new trial is scheduled. Meanwhile, some users of Tornado Cash challenged the sanctions in court and, in 2024, a US court of appeals ruled that the Treasury had exceeded its authority, since immutable smart contracts are not property that can be placed under sanctions. The sanctions were lifted in 2025.

8.5 Signatures with Anonymity Properties

Digital signatures (see Section 1.5) identify the signer: a signature that verifies under pk proves that the owner of pk signed. In this section, we discuss two variants of digital signatures that hide part of this information, and their applications.

8.5.1 Ring Signatures and Monero

A *ring signature* allows a signer to sign on behalf of a group of users, called the *ring*, without revealing who signed. A ring signature scheme consists of three polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. Each user generates a key pair $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}$, as for ordinary signatures (see Section 1.5). To sign a message m, the signer chooses a ring $\mathsf{ring} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ of public keys that contains their own public key, say pk_i, and computes a signature $s \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_i, \mathsf{ring}, m)$. The verifier checks the signature by computing $\mathsf{Verify}(\mathsf{ring}, m, s)$, which outputs 1 (accept) or 0 (reject). Correctness requires that Verify accepts every signature that Sign produces with the secret key of a member of the ring. Remarkably, the other members of the ring do not need to cooperate, or even to know that their public keys are used.

Definition 50 (Security of ring signatures)**.** *Let $T \in \mathbb{N}$ and $\epsilon \in [0, 1]$. A ring signature scheme is:*

- (T, ϵ)-unforgeable *if, for every number of users $n \in \mathbb{N}$ and every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

$$\Pr\left[\begin{array}{c|l} \mathsf{ring}^* \subseteq \{\mathsf{pk}_1, \ldots, \mathsf{pk}_n\} & (\mathsf{pk}_j, \mathsf{sk}_j) \xleftarrow{\$} \mathsf{KeyGen}\ \forall j \\ (\mathsf{ring}^*, m^*) \notin \mathcal{Q} & \\ \mathsf{Verify}(\mathsf{ring}^*, m^*, s^*) = 1 & (\mathsf{ring}^*, m^*, s^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{sign}}}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \end{array}\right] \leq \epsilon.$$

  Above, $\mathcal{A}$ has unrestricted access to an oracle $\mathcal{O}_{\mathsf{sign}}$ (see Definition 16) that, on input an index i, a ring ring that contains pk_i and a message m, responds with a signature $s \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_i, \mathsf{ring}, m)$. We denote the set of the pairs (ring, m) queried to the oracle by $\mathcal{Q}$.

- (T, ϵ)-anonymous *if, for every number of users $n \in \mathbb{N}$, every message m, every pair of indices $i_0, i_1 \in \{1, \ldots, n\}$ and every probabilistic algorithm $\mathcal{A}$ that runs in time at most T, it holds that*

$$\left|\Pr\left[\mathcal{A}(\mathsf{ring}, s) = 1 \,\middle|\, \begin{array}{l}(\mathsf{pk}_j, \mathsf{sk}_j) \xleftarrow{\$} \mathsf{KeyGen}\ \forall j \\ \mathsf{ring} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \\ s \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_{i_0}, \mathsf{ring}, m)\end{array}\right] - \Pr\left[\mathcal{A}(\mathsf{ring}, s) = 1 \,\middle|\, \begin{array}{l}(\mathsf{pk}_j, \mathsf{sk}_j) \xleftarrow{\$} \mathsf{KeyGen}\ \forall j \\ \mathsf{ring} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \\ s \xleftarrow{\$} \mathsf{Sign}(\mathsf{sk}_{i_1}, \mathsf{ring}, m)\end{array}\right]\right| \leq \epsilon.$$

In the unforgeability property, the adversary receives the public keys of n honest users and can obtain signatures by any of them, on messages and rings of its choice, which may also contain public keys generated by the adversary. It wins if it produces a valid signature on a message m^* for a ring ring^* formed only by public keys of the honest users, unless the oracle already gave it a signature on m^* for the same ring. Rings that contain a public key of the adversary are excluded, since the adversary can sign for them with its own secret key. The anonymity property is similar to the hiding property of commitments (see Definition 47): the adversary sees a signature produced either by the member i_0 or by the member i_1 of the ring, and must not be able to tell which one. This definition only considers a single signature: stronger variants also give the adversary other signatures, or even the secret keys of the members of the ring.

A classic application of ring signatures is leaking secrets. A member of a government who wants to leak a secret to a journalist can sign it with a ring formed by the public keys of all the members of the government: the journalist is convinced that the leak is authentic, but cannot tell who leaked it.

Monero. The cryptocurrency Monero uses ring signatures to hide which output a transaction spends. To spend an output, its owner chooses several other outputs from the blockchain as *decoys*, and signs the transaction with a ring formed by the public keys of the real output and of the decoys. The validators learn that one of the outputs of the ring is being spent, but not which one: the anonymity set of the input is the ring. Moreover, Monero hides the amounts with confidential transactions, as in Section 8.4.3, and the recipients with stealth addresses.

Stealth addresses. As we saw in Section 8.2, users should receive every payment on a fresh public key. Stealth addresses make this automatic, even for a payee who publishes a single address. They rely on signature schemes whose public keys can be *rerandomised*: from a public key pk and some fresh randomness, anybody can compute a new public key pk' that corresponds to the same secret key sk. Without knowing sk, nobody can link pk' to pk: it looks like a freshly generated public key. To pay a user whose address is pk, the payer rerandomises pk with fresh randomness, and sends the coins to the resulting one-time public key pk'. The payee checks, using sk, which of the new outputs have a public key that corresponds to sk: these outputs are theirs, and they can spend them with sk. In this way, no two outputs share a public key, and outsiders cannot link them to the published address. Monero implements a variant of this idea: in its signature scheme, every secret key has a single public key, so every one-time public key has its own secret key, and the payer attaches to the transaction some information that allows the payee, and only the payee, to recognise the output and to compute its secret key from sk.

Double spending. Hiding the spent output creates a problem: since the validators do not know which output is spent, they cannot mark it as spent, so its owner could spend it again, with a different ring. Monero solves this problem with *linkable* ring signatures. Every signature contains a *key image*, a value that is computed deterministically from the secret key of the signer and from the public key of the spent output, and that does not reveal which member of the ring signed. Two signatures that spend the same output, even with different rings, therefore contain the same key image. The validators store the key images of all past transactions, and reject any transaction that repeats one of them. The key image plays the same role as the nullifier of Zerocash: it identifies the spent output, without revealing which output it is. Notice that linkable ring signatures purposely reveal whether two signatures spend the same output. This does not hurt anonymity, since every output can only be spent once.

Limitations. The anonymity set of a Monero input is small: it only contains the members of its ring, whose number is a small constant fixed by the protocol, while in Zerocash it contains all the notes ever created. Moreover, the decoys must look like real spends: for instance, users tend to spend the outputs they have received recently, so an output much older than the others in its ring is probably a decoy. The early versions of Monero, which also allowed rings without decoys, were particularly vulnerable. An output that appeared alone in a ring was certainly spent there, so it could be excluded from all the other rings in which it appeared as a decoy, which in turn could leave other rings with a single possible output. In 2017 and 2018, researchers showed that this chain reaction, together with the age of the outputs, revealed the real input of a large fraction of the transactions of that time. Monero has since made larger rings mandatory and improved the choice of the decoys and, at the time of writing (2026), it is testing the replacement of ring signatures by zero-knowledge proofs of membership in the set of all outputs, in the style of Zerocash.

8.5.2 Blind Signatures, Electronic Cash and CoinJoin

A *blind signature* allows a user to obtain a signature on a message from a signer, without the signer learning the message. Signing is a protocol between the user, who knows the message m, and the signer, who knows the secret key sk. The user first *blinds* m, for instance, by combining it with some randomness, and sends the blinded message to the signer. The signer signs it, and the user *unblinds* the result, obtaining an ordinary

signature s on m, which anybody can check with $\mathsf{Verify}(\mathsf{pk}, m, s)$.[1] A blind signature scheme must satisfy two properties, which we do not define formally.

Blindness: The signer learns nothing about m and, when it later sees (m, s), it cannot tell in which execution of the protocol it produced s.

One-more unforgeability: After k executions of the protocol, the user cannot produce valid signatures on $k + 1$ different messages.

Electronic cash. The best-known application of blind signatures is Chaum's *electronic cash* (eCash), an electronic currency with the anonymity of cash, which some banks offered in the 1990s. In eCash, a bank issues digital coins of a fixed value, using a dedicated signature key pair.

Withdrawal: A customer identifies themselves to the bank, chooses a long random string, the *serial number* of a new coin, and obtains a blind signature on it from the bank, which debits the customer's account.

Payment: To pay a merchant, the customer gives them the serial number and the signature.

Deposit: The merchant identifies themselves to the bank and sends it the coin. The bank checks the signature and that the serial number has never been deposited before, and credits the merchant's account.

The signature guarantees that the coin was issued by the bank, and the list of deposited serial numbers prevents double spending (the serial number plays the role of the nullifier of Zerocash). Notice that the bank knows the identity of the customer at withdrawal, since it debits their account, and the identity of the merchant at deposit, since it credits theirs. What it cannot do is link the two: thanks to blindness, it cannot tell in which withdrawal a deposited coin was created. The bank therefore learns how much each customer withdraws and how much each merchant receives, but not who paid whom: the anonymity set of a payment contains all the customers who withdrew a coin before it was deposited. Since the bank accepts every serial number only once, every coin can be used for a single payment: to make another anonymous payment, the customer must interact with the bank again, to obtain a blind signature on a new serial number. Unlike a blockchain, eCash relies on a trusted bank, which must be online during payments, to check that the coin has not been deposited before, and which could refuse to serve some customers; in exchange, payments are extremely cheap and fast. There are also variants in which the bank does not need to be online during payments: if a customer spends the same coin twice, their identity is revealed when the two copies are deposited. eCash was not a commercial success, but its approach survives, for instance, in GNU Taler, a payment system in which customers are anonymous, while the income of the merchants is visible, so that it can be taxed.

Blind signatures and CoinJoin. Recall from Section 8.3 that the coordinator of a CoinJoin can undo the mixing if it learns which participant registered which output. Blind signatures prevent this, in a variant called *Chaumian CoinJoin*. All the mixed outputs have the same value, and the coordinator generates a key pair for a blind signature scheme. The protocol has three phases.

Input registration: Each participant connects to the coordinator through an anonymous network, and registers their inputs, proving that they own them. They also choose the fresh public key of their output, and obtain a blind signature on it from the coordinator, which only signs once it has checked that the inputs cover the value of the output.

[1] For readers who know RSA signatures, here is the classic construction. Let (N, e) be the public key and d the secret key of the signer, and let the signature on m be $\mathsf{Hash}(m)^d \bmod N$. The user samples a random integer u coprime with N and sends $m' := \mathsf{Hash}(m) \cdot u^e \bmod N$, which is a uniformly random number coprime with N, independent of m. The signer returns $s' := (m')^d \bmod N = \mathsf{Hash}(m)^d \cdot u \bmod N$, and the user computes $s := s' \cdot u^{-1} \bmod N = \mathsf{Hash}(m)^d \bmod N$.

Output registration: Through a new anonymous connection, which cannot be linked to the previous one, each participant sends the public key of their output together with the signature, which the coordinator checks before adding the output to the transaction.

Signing: As in Section 8.3, every participant checks that their output is included in the transaction, and only then signs their inputs.

The signature convinces the coordinator that the output belongs to one of the participants who registered inputs but, thanks to blindness, not to which one: the coordinator learns the inputs and the outputs, but not the links between them. One-more unforgeability guarantees that no participant can register more outputs than they paid for. Several wallets have implemented CoinJoin in this way, or with refinements that also allow outputs of different values.

8.6 Should Money Be Anonymous?

The techniques of this chapter could, in principle, make all payments as anonymous as cash, or even more. Whether this is desirable is the subject of an intense debate, which involves not only cryptographers, but also governments, regulators and courts. We summarise the main arguments on both sides.

Arguments for anonymity. Payments reveal a lot about a person: their health (payments to doctors and pharmacies), their religion and political views (donations), their relationships and their habits. On a transparent blockchain, this information is available forever to everybody: not only to the authorities, but also to employers, competitors, criminals and foreign governments. Knowing who holds large amounts of cryptocurrency exposes them to theft, and even to physical attacks. Businesses cannot pay their employees and suppliers on a transparent ledger without revealing their salaries and their commercial relations. Transparency also undermines *fungibility*, the property that all units of a currency are interchangeable: if coins carry their history, coins that once belonged to a criminal may be refused or frozen, so not all coins have the same value, and honest users who receive such coins bear the loss. Finally, in authoritarian countries, financial surveillance is a tool of repression, and anonymous payments protect dissidents, journalists and activists. Supporters of anonymity also point out that cash has always allowed anonymous payments.

Arguments against anonymity. Anonymity also protects criminals. Mixers and other privacy-enhancing tools have been used to launder the proceeds of hacks, ransomware, fraud and drug trafficking, and to evade sanctions. For instance, Tornado Cash was used by hackers linked to the North Korean government (see Section 8.4.4) and, according to US prosecutors, more than 237 million dollars of criminal proceeds were laundered through Samourai Wallet, a Bitcoin wallet that offered CoinJoin. Anonymity also makes it harder to collect taxes and to enforce court decisions and sanctions, while the tracing techniques of Section 8.2 are routinely used to recover stolen funds. Total anonymity can also hurt democracy. Money is power, and power is always political: wealth can finance political campaigns, buy newspapers and television channels, and influence legislators. In a democracy, it is fundamental to know who owns a lot of wealth and how they use it, which is why many countries require large political donations to be disclosed, and keep registers of the real owners of companies. In a fully anonymous economy, such rules would be very hard to enforce. Finally, hidden amounts make some bugs impossible to detect: as we have seen for Zcash, a flaw that allows creating money could go unnoticed.

Between the two extremes. Governments have responded in different ways. Exchanges are generally required to identify their customers and, in the European Union, from July 2027, crypto-asset service providers will no longer be allowed to handle privacy-enhancing coins or anonymous accounts. In the United States, besides the case of Tornado Cash (see Section 8.4.4), the founders of Samourai Wallet were sentenced to prison in 2025 for conspiring to operate an unlicensed money transmitting business. Whether writing privacy software should be a crime is itself a contested question. Technical compromises have also been proposed.

Zcash allows its users to share *viewing keys*, which reveal their transactions to an auditor of their choice, such as a tax authority. GNU Taler hides who pays, but not who is paid. Other proposals allow users to prove, in zero knowledge, that their coins do not come from a list of known illicit sources. Finally, the proposals for a digital euro, a digital currency that the European Central Bank may issue, include offline payments with a level of privacy similar to that of cash.