Blockchain Foundations

Draft of June 8, 2023

Dionysis Zindros

Athens and Stanford

June 2021 - January 2023

Contents

Preface

After defending my PhD thesis in 2020, I swore to take a good break from the stress of academia, and spent a gap year in Athens, Greece. This turned out to be the most prolific year of my career. In the absence of external pressure, I could finally get some research done.

It was during that summer of 2021 that I started designing a new course on *Blockchain Foundations*. This 56-hour course aimed to answer three questions:

1. What are blockchains?
2. How do they work?
3. Why are they secure?

I found that there was already a corpus of works describing the details of blockchains. However, that literature was not quite what I was looking for. Some of it was scattered across informal blog posts and YouTube videos that explained high-level ideas tailored towards end users or hobby scientists, and never went down to the mathematical details. Other writings were *very* precise on the engineering level — talking about this or that byte of a packet — but never explained the *why* behind the design decisions, and in particular they were missing security proofs against arbitrary adversaries. Many of the technical specifications of various cryptocurrencies fall into this category. Lastly, a body of works of high quality *does* provide mathematical proofs and justification for the *whys*, but these are in the form of scientific papers that are extremely dense and beyond the reach of even advanced graduate students, let alone undergraduates dipping their feet on blockchain science for the first time. I decided it's about time to write a series of lecture notes on the foundations to accompany the lectures.

As I was designing the course, I trialed it to a group of my computer science colleagues with no prior blockchain experience: Giannis Gkoulioumis, Nikolaos Kamarinakis, Apostolos Tzinas. Over the course of that summer, we spent 52-hours together over notebooks of notes, discussing the proofs of *bitcoin backbone* and the construction of Merkle trees. They also solved a series of programming exercises which I designed to accompany the course and pertained to creating their own blockchain from scratch, each building their own full node. Their feedback gave me the opportunity to refine the course and prepare it for teaching.

I had the first opportunity to teach this course — which I nicknamed *Marabu*[1] after the eponymous poem of Nikos Kavvadias — in an official capacity during the spring quarter of 2022 at Stanford University. It was a graduate-level course named

[1] The misspelling is intentional, as it makes it easier to Google.

EE374 - Blockchain Foundations. While most of the material was based on my past summer, my co-instructor David Tse and I redesigned parts of the course to meet the time constraints of the quarter system, and to include some new material. To our surprise, the course was mostly attended by undergraduates. The course was well received, largerly due to the great work of our first teaching assistants, Srivatsan Sridhar and Kamilla Nazirkhanova.

At the time, the lecture notes were still in my handwritten notebooks. As I taught the lectures on the whiteboard, several students helped out with digitizing the lecture notes to form the first version of this book. These scribes were Kaylee Renae George, Kenan Hasanaliyev, Koren Gilbai, Ben Choi, Stephen Su, Nathaniel Masfen-Yan, Schwinn Saereesitthipitak, Kachachan Chotitamnavee, Alan Zhang, Taher Poonawala, Scott Hickmann, Lyron Co Ting Keh, Sam Liokumovich, Kaili Wang, Edward Vendrow, Cathy Zhou, Yifan Yang, Michael Nath, Coleman Smith, Solal Afota, Suppakit Waiwitlikhit, Lora Xie, Bryan Chiang, Lucas Xia, Albert Pun, Gordon Chi, Jack Liu, Neetish Sharma, Sergio Charles, Priyanka Mathikshara, and John Guibas. Even though these notes were later rewritten many times, I'm deeply grateful to all, because they made the first version happen, and it would not have started without them.

We repeated this course during winter quarter of 2023. There, our teaching assistants were Kenan Hasanaliyev and Scott Hickmann who further helped refine the course material and notes.

This book is the result of assembling those lecture notes into a more organized format. We want the advanced undergraduate or beginning graduate student to be able to comfortably read it. The prerequisites are a basic understanding of probabilities; an expert level of programming knowledge, ideally with some network programming experience; some exposure to computer science through an introductory algorithms or computability course, and familiarity with computational reductions; and, of course, the always elusive *mathematical maturity*.

This book does not talk about Bitcoin or Ethereum. It doesn't speak about the particularities of these implementations, such as how Bitcoin encodes addresses, or how Ethereum's particular programming language works. Instead, we go back to the foundations to understand the basic components that make a blockchain from first principles. The principles that we explore apply to most blockchain systems, and even decentralized ledger technology systems that are not based on a blockchain *per se*. The goal is to learn how to argue about the security of these systems by walking through the components of a simple UTXO proof-of-work blockchain design first. The same design principles apply when designing more complicated systems such as proof-of-stake blockchains. Upon completing this book, the reader will know what blockchains are, how they work, and why they are secure. They will also have developed the tools and background necessary to argue about the security of more complex protocols.

Chapter 1

A Big, Bad World

1.1 The Nature of Money

Before money, there was debt [18]. Money is a yardstick for measuring it. Sometimes it takes the form of a gold coin. Not useful in itself, one accepts it because one assumes other people will. Modern *fiat* money is not backed by gold, but takes the form of pieces of paper bills or, more often, bits in the computer systems of banks. Regardless of their manifestation, all forms of money are debt, which is a social relation [19].

Money has a long history. A tale told about its origins is of a world of *barter* in which people would visit markets to exchange ten chickens for an ox; money, it is said, was invented to ease the burden of figuring out exchange rates. This is a myth. No such barter societies have ever existed prior to the invention of money. Instead, historically, societies used to be gift economies, in which people were mostly self-reliant on their broader families, and they gifted goods to each other regularly within their villages. These relationships are based on *trust*. The reliance on some form of trust on society will be a *motif* which will reemerge as we try to redesign money in the form of a blockchain.

The idea that one can transact with a stranger without trusting her is an idea that came about with the invention of money. Money came about as a means of tracking debt accumulated through violence such as wars and slavery. Historical forms of money had a physical manifestation: sea shells or salt. The word *salary* we use today hints at this history. Gold and silver were later adopted. Modern money, such as USD, used to have *backing* in gold, so that one could exchange their USD money for gold. The gold standard for USD was abolished in 1971. After this, money issued is known as *fiat*, because it is by social agreement that we give it value. Money is a collective delusion: If, one day, we all stopped believing in money, it would instantly be worthless. The same is true for gold, sea shells, and salt. Money is a *social construct.*

Money functions as a *medium of exchange*, as a *common measure of value* or *unit of account*, as a *standard of value* or *standard of deferred payment*, and a *store of value* [20]. These functions of money rely on the relationship of the individual with the economic community that accepts money. Each monetary transaction between two parties is never a "private matter" between them, because it translates to a claim upon society [31].

This gives rise to the need of *consensus*. The economic community must be able to ascertain, in principle, whether a monetary transaction is *valid* according to its rules. In a good monetary system, parties of the economic community must globally agree on the conclusions of such deductions. In simple words, when someone pays me, I must know that they have sufficient money to do so, and that this money given to me will be accepted by the economic community when I later decide to spend it. This judgement of validity consists of two parts: First, that the money in use has been *minted* legitimately in the first place. Secondly, that this money rightfully belongs to the party who is about to spend it, and has not been spent before, to protect against *double spending*, or ownership tracking. Consensus pertains to ensuring both correct minting and correct transfers. For money to have value, it must be *scarce*. Scarcity must be ensured both during minting and during transfers. Scarcity is a necessary, but not sufficient, property of money.

The problem of consensus is solved differently in different monetary systems. Gold coins had stamps whose veracity could be checked, while paper bills have watermarking features making them difficult to duplicate. Such physical features ensure the legitimacy of minting. The problem of double spending is trivial when it comes to physical matter: If I give a gold coin to someone, I no longer hold that gold coin and cannot also give it to someone else. When coins are digitized, the problem of *who owns what* is solved by the private bank and payment processors. A private bank centrally maintains the balance of a bank account to ensure a corresponding debit card cannot spend more money than it has. In this case, a vendor's terminal connects to the bank's servers to check the validity of the payment (and security can only be ensured while the terminal is online). These cases involve a *trusted third party*, the bank or the payment processor, to maintain a balance and make a judgement on whether a transaction is valid. The central bank is relied upon for the legitimacy of minting. Payment processors and banks who maintain account balances and make a judgement on whether a transaction is valid are relied upon to prevent double spending. The economic community depends on these third parties and trusts them for availability and truthfulness.

The cypherpunk political movement and the wave of cryptographers working on *protocols* in general have an inherent hatred for trusted third parties. For the former, they amount to centralization of political power which they wish to see eliminated. For the latter, it constitutes a technical and intellectual challenge – if the role of the trusted third party is fully algorithmizable, why not replace the party by a protocol ran by the economic community themselves? It is somewhere in the intersection of the two that *blockchain* protocols appeared.

Trusted third parties are undesirable for four reasons. First, the authority may fail, not because of nefarious purposes, but because of a mistake. There might be a power loss and its servers may be shut down, causing availability issues. Secondly, a trusted third party may become corrupted in the future, creating the possibility of abuse. While the authority is trustworthy today, who knows about tomorrow? Third, the authority may be honest in and of itself, but an external adversary may breach into its systems, especially if it is digital. Fourth, different parties may not have a mutual authority that they both trust. For example, the US government and the Chinese government may not both want to rely on either of the US federal reserve, or the Chinese central bank. Trusted parties are liabilities for the people using them, but also liabilities for themselves. If a bank falls victim to a digital breach, it may be held responsible for losses incurred. It is thus often in the best interest of both the community and the central authority itself to remove trust to the central authority.

How can a trusted central party misbehave? A private bank can conjure up more money in someone else's account, or remove money from yours, and one's only recourse against such actions, which can be damaging to the economy, is legal. We rely on the functions of government, a trusted third party, to prevent such actions. If a bank illegally takes away money from one's bank account, they can sue the bank. This is a *treatment* of adversarial behavior. In the systems we will design, our goal will be to create systems that *prevent* adversarial behavior by making it impossible in the first place; not by detecting it and *treating* it when it emerges. These systems will be *self-enforcable*. Prevention is preferable to treatment.

The question we try to answer is whether we can *decentralize* money by removing some of these institutions of trust. We can remove private banks and people can *be their own bank*; and we can remove the central bank, and money issuance can be in the hands of the people. However, some trust in society will necessarily remain, as money is a social construct. Governments are elected by the people and, in that sense, express the will of the people. It is a political question whether we *want* to remove central parties from the picture. Removing the central bank removes an important macroeconomic tool from the hands of government, which may have long lasting and disastrous recession effects. Removing the private bank from the picture makes each and everyone responsible for their own money: In case your house in which your computer is stored burns down, you lose your money, contrary to the case of a bank, where all your documents can be recovered through some form of legal process. We will provide the *means* to eliminate centralization, but it is not always clear that we *should*. Once we describe the system to do so, we can choose *which* centralization parts we want to eliminate. For example, we can create a system where private banks are unnecessary, but money issuance is still centralized.

Because money is a social construct and it is conjured by social delusion, it does not need legal backing to have value. We can rebuild money in the form of code, as long as we can recreate scarcity, minting and ownership tracking, and we convince society to adopt it as currency. This is what gives rise to *cryptocurrencies*. Similarly, because private contracts between individuals are also a social construct, we can also recreate these in the form of code. This is what gives rise to *smart contracts*.

Money and its functions, as well as contracts and their function, have been traditionally codified in the form of law. These laws have been created through centuries of experience and contain a lot of wisdom. As computer scientists, our role when implementing cryptocurrencies and smart contracts will be to identify the *computational* properties of money and contracts. What *computational* role does each of the virtues of money play in ensuring its correctness and security? Which of these can be modified? What are the computational aspects of rules, regulations, and processes? When money and contracts are implemented in code, and analyzed in the theoretical framework of computer science, these will become precise and explicit rather than implicit.

1.2 The Adversary

Our systems will be designed in the presence of an *adversary*. This adversary will have various nefarious goals and may try to act against the rest of the parties. We will highlight the parties whose interests we want to defend and designate them as *the honest parties*. The honest parties follow the protocol as described by us, the protocol designers. A party beyond this group of designated honest parties is considered *adversarial*. The adversary can deviate from the honest protocol and behave differently from what we designed. We will only provide assurances to the honest parties when we embark on our security proofs. This follows the path of cryptography: If you want security assurances, you must play honestly.

We will assume our adversary has access to our source code and we will not keep this secret. This is a general principle of cryptography known as Kerckhoff's Principle. This makes the adversary more powerful. Hence, if we can prove security against this adversary, we have a stronger protocol. Of course we will need to keep *some* secrets from this adversary. These will be things like passwords and secret keys, which we will be exploring soon.

We consider only *one* adversary, not multiple. That single adversary can *spawn* nodes that are acting on her[1] behalf. The treatment in which the adversary is considered to be a single party with an overarching goal in mind gives the adversary more power. She is a more powerful adversary than an adversary who is fighting against another. We will design our protocols to be secure against this single, overarching adversary.

We will design our systems to be resilient against very powerful adversaries, such as state actors. Our adversary can really be truly malicious. She can break laws. She might be *irrational* and decide to lose money, just so that we suffer, even if there is no monetary gain for her. She can control corporations. She can control governments, including the legislative, executive, and judicial branches of the government. This means she can change the laws and outlaw our protocol. She can take over a country's or multiple countries' courts, issue subpoenas, or kill people to achieve her goals, and do this all in secret. We will not rely on these centralized institutions for our security, but will try to design protocols that are resilient in these settings. In light of this model, it becomes clear that there is very little we can rely on. For example, we cannot rely on someone proving their identity by presenting their government-issued passport, as an adversary controlling the government can issue an arbitrary number of fake passports.

Ideally, we want our protocols to survive and remain operational as long as a country's Internet infrastructure is operational, and people are allowed just a modicum of private life. Compare this to centralized services, such as Google's search, or Amazon's market. These services really cannot hope to survive an adversarial government. A subpoena issued by a court can order them to shut down, and they must comply. On the contrary, our decentralized protocols will not be subject to court decisions. In that sense, our protocols are *sovereign* — they enjoy the same level of independence as a stand-alone country. For a court to shut down a de-

[1] As a convention, we will use the female pronouns for the adversary, the male pronouns for the honest parties, and the neutral pronoun for the challenger. This helps write succinct and easy to read sentences in which the "he" and "she" pronouns are used with clarity. As blockchain designers in which adversarial thinking is a central tenet, we will take both roles of the honest party and the adversary and argue from both sides when designing a protocol and reason about its security.

centralized protocol, it cannot order its servers to shut down, because there are no servers. Instead, it must target each of its participants, a much more difficult task.

The Cryptographic Model

Following the cryptographic tradition, and highlighting our computer science methodology, our protocols are structured upon three pillars [24]:

1. **Formal definitions** play a central role. They specify the desirable properties of our protocols. As we will see, these can often be quite tricky to develop. One such example is what it means for a ledger to have *safety*, a topic we will return to when we speak about ledgers.

2. **Clearly articulated assumptions** allow us to understand the limitations of our protocols. Our protocols never work unconditionally, and we must restrict our model to obtain security. One such example is the *honest majority assumption*, a topic we will return to when we speak about proof-of-work.

3. **Rigorous proofs of security** give us the *guarantee* that our protocols are secure, as long as our assumptions hold. Instead of employing handwavy arguments, the proofs are mathematical theorems employing computational reductions ane exact probability calculations. They assert that the protocols are secure *for all* adversaries.

We will model the adversary as a Turing Machine interacting with the honest parties, each of which will also be modelled as a Turing Machine. For the time being, the Turing Machine formalism is unimportant: Intuitively, we will simply imagine our adversary as a computer running an adversarial computer program which we will denote $\mathcal{A}$. Similarly, we will imagine the honest parties as separate computers all running the same program, the honest program, which we will sometimes denote Π. The adversary and the honest parties are all directly or indirectly connected to each other in a common communication network. We will return to the formal model of computation and the network at a later time to make our arguments rigorous.

The critical part that will allow us to prove our security through computational arguments is that we will limit the power of the adversary: We will require that the adversary runs in *polynomial time* with respect to its input size. We will also allow the adversary access to randomness. The same constraints are applied to the honest parties. We will denote such parties PPT, probabilistic polynomial-time, parties. Formally speaking, these are modelled as Turing Machines [34] with additional access to a random tape. In practice, when thinking about these machines, we simply think of them as regular computer programs (in, say, Python, C++ or TypeScript) in which we have access to a random number generator, which we assume produces fresh, uniform and completely fair randomness every time it is called.

1.3 Game-Based Security

We will soon give detailed and rigorous definitions of what security properties we want our protocols to achieve. These will be our design goals. Once we have clearly articulated these goals, we will attempt to formally prove that our protocols attain the desired properties.

We want to rigorously define what security means. A first attempt, trying to write out a definition in English, looks like this:

> A protocol Π is secure if it is impossible for an adversary $\mathcal{A}$ to break it.

But what does it mean for the adversary $\mathcal{A}$ to *break* our protocol exactly? This will depend on the exact protocol. When the time comes to talk about hashes, signatures, or blockchains, we will define these rigorously. These security goals will be written out in the form of a *cryptographic game*. These games are algorithms that, given a particular PPT adversary, attest to whether the adversary has been successful in breaking the protocol.

You can imagine the game as a piece of code that evaluates the success of an adversary. The game will be specific to the protocol and property we wish to describe, and be given a relevant name. The game is also known as the *challenger* or *experiment*. To use one of the games, first, we decide which adversary we want to evaluate, and fix the source code that defines this adversary. We call this adversary $\mathcal{A}$, denoting a particular computer program. We are only interested in evaluating the performance of PPT adversaries against the game. We then run the game, which is a different computer program, and give it the source code of the adversary as a parameter. We also give the game access to run the honest protocol Π. The game will simulate some interaction between the adversary and the honest parties. The game executes the adversary and the honest parties and facilitates some data exchange between them. It then takes the output of the adversary and decides whether the adversary has been successful in her endeavour to break the protocol. The game outputs a boolean output: *true* if the adversary was successful in breaking the protocol, and *false* if the adversary was unsuccessful. It is bad for us, the designers of the protocol, if there exists some adversary such that the game outputs *true*. That's why we call this a *bad event* The execution of the game never occurs in real implementations of the protocol. It is simply a tool we use at the mathematical level to argue about the security of our design.

In its foundations, our security is analyzed with a *security parameter*: The parameter κ. This parameter denotes what probability of failure we are willing to accept in our protocols: We can tolerate probabilities that are roughly $2^{-\kappa}$. For $\kappa = 256$, this probability is extremely small: It is extremely more probable that a global earth catastrophy is caused by an asteroid hitting it *during the second you read this particular sentence* than a probability 2^{-256} occurring. Simply put, these events never occur.

When calling the adversary and allowing her to perform an attack, we will hand her some information, including the particular value κ that we are interested in. The adversarial source code must be the same for all values of κ (we say that we are interested in *uniform* adversaries).

The adversary and honest party run in polynomial time in the security parameter κ. Similar to the analysis of algorithms in all of computer science, when we say that the adversary $\mathcal{A}$ is *polynomial*, we mean that the adversary runs in polynomial time with respect to its input length. More specifically, there exists a polynomial $p(\kappa)$ such that, given any input of size κ, the adversary runs for at most $p(\kappa)$ steps. If we want to give the adversary the option to run in polynomial time with respect to the parameter κ, we must issue the call to the adversary with an input of size κ.

We denote this by writing $\mathcal{A}(1^\kappa)$, meaning that we call the adversary with an input of a string consisting of just the character 1 repeated κ times. Because $|1^\kappa| = \kappa$, the length of the input is κ, and the adversary can run in $p(\kappa)$ time. Note that it would be inappropriate to call the adversary without arguments as $\mathcal{A}()$, as in this case the adversary has no time to perform the attack. It would also be inappropriate to call the adversary using $\mathcal{A}(\kappa)$, as in this case the adversary would only have $\log \kappa$ time available (since $|\kappa| = \log \kappa$). We may have more information to pass to the adversary as input, such as a public key. If that information already has length κ, this is sufficient for our purposes, and we do not need to pass the adversary the extra argument 1^κ. You can think of the argument 1^κ as *giving the adversary enough time to operate.*

The format of a generic game is illustrated in Algorithm 1. The challenger is parameterized by the code of the honest party Π and the code of the adversary $\mathcal{A}$. It is invoked with the security parameter κ (note that there is no restriction that the challenger runs in polynomial time, as it is merely a mathematical tool). It invokes the honest party, giving him polynomial time in κ to run, as well as some additional arguments that will be defined by the particular protocol. It then runs the adversary, giving her polymomial time in κ to run, as well as some additional arguments which may depend on the honest party's behavior. Depending on the game, the challenger may invoke the honest party and adversary multiple times, creating some interaction between them. Lastly, the challenger evaluates the output of the adversary to ascertain whether she has been successful in breaking the protocol, and outputs a boolean value: 0 indicating that the protocol remained unbroken, or 1 indicating that the adversary broke the protocol. The challenger is illustrated diagramatically in Figure 1.1.

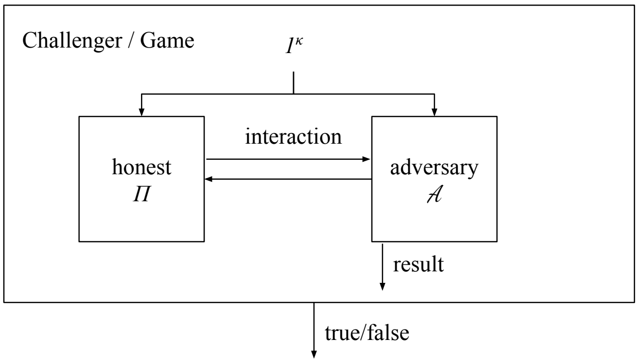


Figure 1.1: A game-based security definition shown diagramatically. The challenger invokes both the honest party and the adversary before deciding whether the adversary was successful.

Algorithm 1 The form of a challenger for a game-based security definition.

```
function MY-GAME$_{\Pi,\mathcal{A}}(\kappa)$
    ▷ Invoke the honest party with poly $\kappa$ time and more arguments
    $\cdots \Pi(1^\kappa \cdots)$
    ▷ Invoke the adversary with poly $\kappa$ time and more arguments
    result $\leftarrow \mathcal{A}(1^\kappa \cdots)$
    ▷ Evaluate whether the adversary has been successful
    if result indicates adversarial success then
        return true
    else
        return false
    end if
end function
```

Negligibility

In an ideal world, we would like to state that our protocols are unbreakable:

$$\mathsf{my\text{-}game}_{\Pi,\mathcal{A}}(\kappa) = \mathsf{false}$$

However, this goal is sadly unattainable. When the time comes to generate a secret key or a password, we will make these have bit length κ. Unfortunately, the adversary can *guess* such secrets by taking a random guess. If our secret is sampled from the set $\{0,1\}^\kappa$, the set of κ-bit strings, the probability of the adversary guessing correctly will be $2^{-\kappa}$. This is a probability of failure that we are willing to accept, as it is unavoidable.

What happens if an adversary attempts to perform multiple guesses for the secret key? Each of these guesses has a probability of success amounting to $\frac{1}{2^\kappa}$. Since the adversary has polynomial time $p(\kappa)$, the number of guesses she can perform must be polynomial, too. What is the probability that *at least one of these guesses* is correct? We can apply a *union bound* [30] to find this.

Theorem 1 (Union Bound). *Consider n events $X_1, X_2, \cdots, X_n$. Then the probability that* any one of them *occurs is given by their* union bound*:*

$$\Pr[X_1 \vee X_2 \vee \cdots \vee X_n] \leq \Pr[X_1] + \Pr[X_2] + \cdots + \Pr[X_n]$$

Note that this is just an upper bound and these probabilities may not be exactly equal. To see why, consider the simple example of rolling a die 6 times, hoping to get a 6. For any one roll, the probability of getting a 6 is $\frac{1}{6}$, and the union bound tells us that the probability of getting a 6 in *any roll* across our whole game of 6 rolls is at most 1. However, the actual probability is in fact a little less: $1-(1-\frac{1}{6})^6 = 0.665$. Here, we calculated the probability of *not* winning in a single roll, which is $1-\frac{1}{6}$. We then calculated the probability of failing to win in any single roll, which is $(1-\frac{1}{6})^6$. Lastly, we took the complement of this probability, interpreting this to mean that we won in at least a single roll, obtaining $1-(1-\frac{1}{6})^6$. We will use this style of arguments a lot when counting probabilities about blocks and chains.

Returning to our polynomial adversary, and applying a union bound, we see that this adversary can succeed with probability bounded by $\frac{p(\kappa)}{2^\kappa}$.

If we have one adversary who can succeed with some probability $Pr_\mathcal{A}$, then a different adversary $\mathcal{A}'$ can succeed with probability bounded by $p(\kappa)Pr_\mathcal{A}$ for any polynomial p. This is known as *amplification.* We wish to define a class of probability functions that we deem *acceptable* probabilities of failure. Clearly, if an adversary has a *constant* probability of success (such as 0.5) that does not depend on the security parameter κ, this is *not* acceptable, as we want κ to be our *tuning knob* of how secure our protocol will be. We will deem *acceptable* the class of functions denoting a probability which is not amplifiable to a constant by this manner.

Any inverse polynomial probability such as $\frac{1}{\kappa^3+\kappa+9}$ can be amplified by a polynomial adversary to be close to the union bound by repeating the experiment a polynomial number of times. Therefore, we must ask that our probability is *smaller than any inverse polynomial.* Such functions are called *negligible.*

Definition 1 (Negligible function)**.** *A function $f(\kappa)$ is* negligible *if for any polynomial degree $m \in \mathbb{N}$, there exists a κ_0 such that for all $\kappa > \kappa_0$:*

$$f(\kappa) < \frac{1}{\kappa^m}$$

We choose to accept negligible functions exactly because the probability of failure cannot be amplified in this manner. If an adverary $\mathcal{A}$ succeeds with negligible probability, an adversary $\mathcal{A}'$ that simulates $\mathcal{A}$ must run the simulation an *exponential* number of times in order to achieve anything beyond a negligible probability. Given that we have constrained our adversaries to be polynomial-time, this is impossible.

The negligible probability of failure is the standard treatment in modern cryptography [24]. Beyond the above argument pertaining to the polynomiality of adversaries, negligible functions are easy to work with because they observe certain *closure* properties. In particular, multiplying a negligible function with a polynomial yields a negligible function. As constants are polynomials, scaling a negligible function by a constant yields a negligible function too. Of course, multiplying something negligible with something negligible keeps it negligible, and taking any constant power of a negligible function keeps it negligible.

$$\begin{aligned}
\mathsf{negl} \cdot \mathsf{negl} &= \mathsf{negl} \\
\mathsf{const} \cdot \mathsf{negl} &= \mathsf{negl} \\
\mathsf{poly} \cdot \mathsf{negl} &= \mathsf{negl} \\
\forall k \in \mathbb{N} : \mathsf{negl}^k &= \mathsf{negl}
\end{aligned}$$

Definitions of Security

As designers, the ideal goal for us would be to design a protocol for which *no adversary* succeeds in breaking the game, no matter what code she is running. If we can achieve this, it will be a truly magnificent achievement. Observe what we are trying to say here: The protocol works *no matter what the adversary decides to do*, as long as our assumptions are respected (such as the polynomiality bounds on the adversary). We are not merely enumerating a bunch of attacks that we considered ourselves and arguing that our protocol is secure against *these*! Instead, we are arguing against *all adversaries*, even adversaries that we do not know about and have not imagined. The ability to argue against *all* adversaries is the epitome of modern cryptography, a feat only possible through the formalism and models of computer science. This proof style is recent and has only appeared within the last 50 years.

The ideal protocol Π satisfies security against all adversaries:

$$\forall PPT \mathcal{A} : \mathsf{Game}_{\Pi,\mathcal{A}}(\kappa) = 0$$

However, this is an unattainable goal. To see this, consider the case where the honest party generates a secret of length κ such as a private key or password. In that case the adversary can simply attempt to *guess* this private key at random. This will be possible with probability $\frac{1}{2^\kappa}$. As such, the above goal of requiring that the game *always* outputs 0 for all adversaries cannot be attained. Instead, we will require that any adversary only has negligible probability of succeeding in breaking these games. Remember that the probability of randomly finding the secret key is $\frac{1}{2^\kappa}$ and this is a negligible value in κ.

A security definition will look like this:

Definition 2 (Security)**.** *A protocol* Π *is secure with respect to game* Game *if there exists a negligible function* negl(κ) *such that*

$$\forall PPT \mathcal{A} : \Pr[\mathit{Game}_{\Pi,\mathcal{A}}(\kappa) = 1] \leq \mathrm{negl}(\kappa)$$

We are using probability notation here because the execution of the challenger *with the same honest protocol* Π *and against the same adversary* $\mathcal{A}$ *and using a fixed security parameter* κ will not always yield the same result! Since both the honest party and the adversary have access to generate randomness, the challenger will sometimes report 0, and other times report 1. For a fixed Π and $\mathcal{A}$ and κ, there is a certain probability that the challenger will report 0, and a certain probability that the challenger will report 1.

Fixing Π and $\mathcal{A}$, but leaving κ to take any value, we obtain different probabilities for each value of κ. Therefore, the value denoted by $\Pr[\mathsf{Game}_{\Pi,\mathcal{A}}(\kappa) = 0]$ for a fixed Π and $\mathcal{A}$ is a function of κ. What we are saying here is that this function that counts the probability must be below some negligible function. Said differently, that probability must *eventually* (for sufficiently large κ) become smaller than all inverse polynomials.

The Honest/Adversarial Gap

Note here how great the requirements of security that cryptography mandates are: In a *secure* protocol, the honest party can act within polynomial time, but an adversary needs superpolynomial time to break it. The successful honest party is efficient and lives within the complexity class P, but the successful adversary is inefficient, and lives within the complexity class NP but not in P. This is illustrated in Figure 1.2. This is a much bolder claim that the security of traditional money! In a traditional banknote monetary system, the honest party (such as the government) has many more resources than the adversary (say, a forgery criminal). If the adversary acquires resources equivalent to the honest party (for example access to the same banknote-printing machines), the system's security will be compromised. Here, we are achieving something significantly stronger: An honest party needs only polynomial time to successfully participate in the protocol, but a successful adversary will require superpolynomial time to successfully break it — a huge discrepancy.

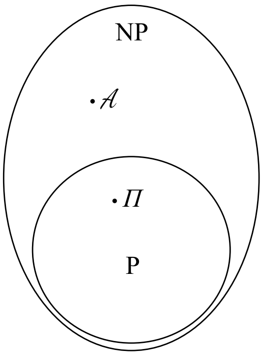


Figure 1.2: In a secure protocol, a successful honest party needs polynomial time, while a successful adversary needs superpolynomial time.

Proofs of Security

When the time comes to prove a protocol secure, we will sometimes make an assumption that an existing, underlying protocol is secure. Our new protocol will be built *on top of* the existing protocol. In the blockchain world, we will take many underlying primitives for granted: We will make use of *hash functions* and *signatures* assuming they are secure, and leave their design to the cryptographers. Our theorems will state that *if* the underlying protocol is secure, *then* the protocol we are building on top of the existing primitive is also secure. Said differently, if no PPT adversary wins in the underlying protocol except with negligible probability, then also no PPT adversary can win in our new protocol, except with negligible probability.

The proofs of these theorems will take the form of a *computational reduction*, and they will look roughly as follows, when we are designing a new protocol Π^*:
Claim. If protocol Π^* is secure, then protocol Π, built on top of Π^*, is also secure.
Proof. Suppose, towards a contradiction, that protocol Π is *insecure*. Then, by the game-based security definition, there must exist a PPT adversary $\mathcal{A}$ that breaks Π with non-negligible probability (but we don't know the exact inner workings of this adversary, because she is arbitrary). We design a PPT adversary $\mathcal{A}^*$, for which we write the code and know her inner workings *exactly*. Somewhere in the code of $\mathcal{A}^*$ we make use of the code of $\mathcal{A}$ as a black box. The adversary $\mathcal{A}^*$ attempts to break the protocol Π^* within the confines of the challenger for the protocol Π^* (a particular game). The adversary $\mathcal{A}$ attempts to break the protocol Π within the confines of the challenger for the protocol Π (a different game). When $\mathcal{A}^*$ runs, she *simulates* the execution of $\mathcal{A}$ by invoking her code, as illustrated in Figure 1.3. When $\mathcal{A}^*$ invokes $\mathcal{A}$, she must do so behaving *as if she were* the challenger for protocol Π. The adversary $\mathcal{A}^*$ can invoke $\mathcal{A}$ multiple times with different inputs and collect her outputs before producing an output of her own. Because $\mathcal{A}$ runs in polynomial time, and because $\mathcal{A}^*$ only performs a polynomial number of operations beyond invoking $\mathcal{A}$ a polynomial number of times, therefore $\mathcal{A}^*$ is also a PPT. We can now evaluate the probability of success of $\mathcal{A}^*$ and relate it to the probability of success of $\mathcal{A}$, arguing that *if* the probability of success of $\mathcal{A}$ is non-negligible, then so is the probability of success of $\mathcal{A}^*$. However, this contradicts the assumption that Π^* was secure, completing the proof. $\Diamond$

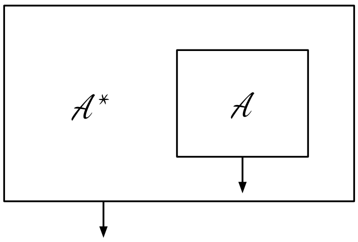


Figure 1.3: A computational reduction between two adversaries. Given an adversary $\mathcal{A}$ against protocol Π, we construct an adversary $\mathcal{A}^*$ against a protocol Π^*.

This proof style is by contradiction. We can write the same proof in a *forward direction* without resorting to a contradiction. This gives a shorter proof, and we will prefer this style in our writing, following the example of Katz and Lindell [24]. These proofs look like this:

Proof. Consider an arbitrary PPT adversary $\mathcal{A}$ attempting to break the protocol Π. We construct an adversary $\mathcal{A}^*$ against the protocol Π^* by making use of $\mathcal{A}$ as before. For the same reasons as before, $\mathcal{A}^*$ is also PPT, and their probabilities of success are related. By the security assumption on Π^*, we know that the probability of success of $\mathcal{A}^*$ against its challenger is negligible. From the relationship between the probabilities of success of $\mathcal{A}$ and $\mathcal{A}^*$, we also deduce that the probability of success of $\mathcal{A}$ is negligible, completing the proof. $\Diamond$

The two proofs are identical, with the exception that the second one is a little more straightforward. Of course, these are rough proof outlines provided to give a sketch of what to expect next, but are still quite abstract. You will become acquainted with the particular workings of this style of proof as we work through particular theorems, particular protocols, and particular games in the next chapters.

1.4 The Network

In our quest to decentralize money, our participants will be nodes on a computer network. These nodes will each run their software and communicate with one another. Each of them is connected to some of their *peers* as illustrated in Figure 1.4. Contrary to more traditional Internet systems where there is a designated role of a *client* and a *server*, here all peers play the same role: They function both as clients and as servers of requests.

The Non-Eclipsing Assumption

In this network, not everyone is connected to everyone else, but messages can reach from one side of the network to the other by travelling through intermediaries. This is achieved through the *gossiping* protocol: When a node receives a message it hasn't seen before, it forwards it to its peers. That way, everyone eventually learns about the message. In order to avoid denial-of-service attacks, messages may be validated in a basic manner before they are gossiped. For example, syntactically invalid messages will not be gossiped.

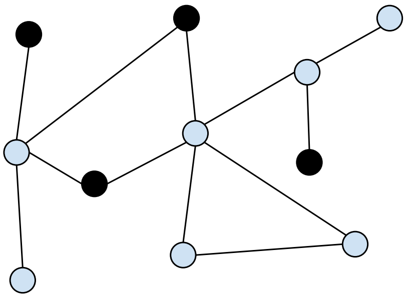


Figure 1.4: The peer-to-peer network. Nodes are shown as circles and connections as lines. The honest nodes are shown in blue, while the adversarial nodes are shown in black.

We will make a central assumption about the network: That there exists a path between any two honest parties on the network, which consists of only honest nodes. Said differently, the network is not split into components whose connection is controlled by the adversary.

Definition 3 (Non-eclipsing)**.** *The* non-eclipsing assumption *states that, between every two honest parties on the network, there exists a path consisting only of honest nodes.*

Note that, for the non-eclipsing assumption to hold, it is *not* sufficient that every honest party has a connection to an honest party. There might be components of honest parties that remain isolated from the rest of the network, as illustrated in Figure 1.5.

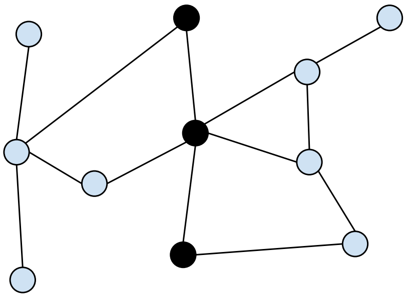


Figure 1.5: An eclipsed peer-to-peer network. Even though every honest party has an honest connection, the network is partitioned into two disconnected components by the adversary.

We are introducing this assumption out of necessity. We cannot hope to build any currency in an eclipsed world. To see why, imagine two completely isolated civilizations, both maintaining their own separate currency. These civilizations, given a lack of communication between them, cannot hope to be able to deduce who owns how much money in their respective counterpart world.

The Sybil Attack

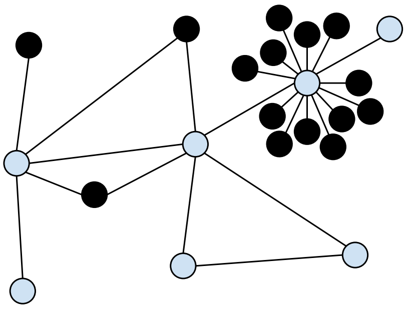


Figure 1.6: A Sybil attacked peer-to-peer network. The non-eclipsing assumption is not violated.

Following our pattern of a powerful adversary, we give the adversary the ability to create as many identities on the network as she desires. This is termed a *Sybil attack* [12]. The adversary may overwhelm an honest party with adversarial connections as illustrated in Figure 1.6.

Definition 4 (Sybil Attack)**.** *In a* Sybil attackable *network model, the adversary may create as many identities (nodes) as she desires. The honest parties cannot distinguish which identities have been created by the adversary in this manner.*

It is possible that the adversary controls all the connections of an honest party, except for one connection to an honest party, which is necessary to maintain the non-eclipsing assumption. *Every honest party will certainly be connected to at least one other honest party.*

Peer Discovery

Ensuring that the non-eclipsing assumption is maintained is a practical engineering problem and there are many heuristics employed in achieving this. The process of connecting to other nodes, attempting to ensure at least one honest connection, is termed *peer discovery*.

Let us briefly discuss how peer discovery is performed in practical peer-to-peer networks. When a peer-to-peer node is first booted, it must connect to some of its peers for the first time. This is the *network bootstrapping* phase. At this phase, the node typically will attempt to connect to a list of hard-coded peers whose IP addresses appear in the implementation source code. Some of these connections may fail, but if one of them succeeds and connects to an honest party, the newly booted node can begin to operate. After bootstrapping, whenever the newly booted node connects to a peer, it asks the connected peer to tell it about the addresses of *its own peers*. These peers are then recorded and can be used for further connections.

They can also be reported to other peers asking for peer discovery. The policy for reporting discovered peers may vary. For example, some nodes may not share all of their known peers. In case the bootstrapping phase fails, the user is given the option to manually connect to a peer by entering its address. This allows the software to survive cases of censorship, or of broad compromise of all the hard-coded peer addresses.

Problems

1.1 Which of the following functions are negligible in κ?

a. $f(\kappa) = 0$

b. $f(\kappa) = 1$

c. $f(\kappa) = 2^{-128}$

d. $f(\kappa) = \frac{2^\kappa}{128}$

e. $f(\kappa) = \frac{128}{2^\kappa}$

f. $f(\kappa) = \frac{1}{3\kappa^3+7\kappa^2+12}$

g. $f(\kappa) = \frac{\kappa^7}{7^\kappa}$

h. $f(\kappa) = \frac{1}{\log \kappa}$

i. $f(\kappa) = \frac{1}{\kappa!}$

1.2 Use induction to prove the *Union Bound* theorem.

1.3 Let f and g be negligible functions. Show that $h(\kappa) = \max\{f(\kappa), g(\kappa)\}$ is negligible.

1.4 Prove that

a. $\mathsf{negl} \cdot \mathsf{negl} = \mathsf{negl}$

b. $\mathsf{const} \cdot \mathsf{negl} = \mathsf{negl}$

c. $\mathsf{poly} \cdot \mathsf{negl} = \mathsf{negl}$

d. $\forall m \in \mathbb{N} : \mathsf{negl}^m = \mathsf{negl}$

Further Reading

Blockchain science is founded on cryptography. For a great introduction to modern cryptography, consult *Introduction to Modern Cryptography* by Katz and Lindell [24]. It is a beautifully written book. It explores how to build many of the primitives we will make use throughout this book, including hash functions and signature schemes. More importantly, it is a good way to learn about the adversarial mindset and to look into complexity reduction-based security proofs. The book is filled with theorems and proofs that show that, for all PPT adversaries, the protocol is secure, except with negligible probability. In *Further Reading* paragraphs at the end of the next chapters, you will find some references in chapters of *Modern Cryptography* (2nd edition). Another good book on cryptography is *Foundations of Cryptography* [16,17]. An easier and pleasant to read textbook is Smart's *Cryptography Made Simple* [33].

For a more in-depth treatment of Turing Machines and our computational model, consult Sipser's *Introduction to the Theory of Computation* [32]. It is a very well written book, with great examples and proofs that are written to be educational. It's an easier book than *Modern Cryptography*, and a good way to learn computational reductions.

Throughout this book, we use many elements of discrete mathematics and probability theory. For discrete mathematics, you can use Liu's *Elements of Discrete Mathematics* [25]. For probability theory, you can use Ross's *A First Course on Probability Theory* [30]. You can read both cover-to-cover, but they also function well as a reference in case you want to look something up.

Chapter 2

Cryptographic Primitives

2.1 Hash Functions

We already discussed the *gossip protocol* that allows peer-to-peer nodes to exchange objects on the network. Before exchanging an object, it is useful that the nodes can talk *about* these objects and ask each other whether they have a particular object. To do this, it will be useful to give each object a unique identifier. We cannot use increasing integers as identifiers, as these objects may be created in different parts of the network and there is no global shared counter. We also cannot use a simple random number as the identifier, as we want the identifier to be unfakeable: Given an identifier we want to be able to check that the object really does correspond to the identifier.

For this, we will use *cryptographically secure hash functions.* The hash function is a function $H : \{0,1\}^* \longrightarrow \{0,1\}^\kappa$, where κ is the security parameter. As you can see, the hash function accepts *any* string as input, but *always* returns a κ bits long string. This makes it useful as a *compression* mechanism, as these identifiers are short (κ will be 256 bits in practice) and can be exchanged on the network prior to the actual objects. In order for H to be practically useful, we require that it is polynomially computable.

Collision Resistance

Ideally, we would like each different object to correspond to exactly one hash output:

$$\forall x_1, x_2 : x_1 \neq x_2 \Rightarrow H(x_1) \neq H(x_1)$$

However, this ideal goal is unattainable. Because the hash function has *unlimited* inputs and *limited* outputs, there will necessarily exist some *collision* in which multiple inputs correspond to the same output. To find a collision, we can start enumerating all the possible inputs to the hash function starting at 0 and going up to 2^κ. If we have not found a collision when we reach $2^\kappa - 1$, then this means that we have taken up all of the possible 2^κ outputs. When we then evaluate the hash function on the input 2^κ, we will certainly find a collision. This process is shown in Algorithm 2.

Algorithm 2 An exponential search for a collision in a hash function that *certainly* finds a collision.

```
function COLLISION-SEARCH$_H(\kappa)$
    for $i \leftarrow 0$ to $2^\kappa$ do
        for $j \leftarrow i + 1$ to $2^\kappa$ do
            if $H(i) = H(j)$ then
                return (i, j)
            end if
        end for
    end for
end function
```

Of course, as this function has to run through $2^{2\kappa}$ combinations, its running time is exponential. The result that hash functions must *necessarily* have collisions stems from the Pigeonhole Principle [25]:

Theorem 2 (Pigeonhole)**.** *Consider a function $f : A \longrightarrow B$. If $|A| > |B|$, then there must exist x_1 and x_2 such that $f(x_1) = f(x_2)$.*

Instead, we will require that *finding* such collisions is *computationally* difficult. We can define this in the form of the collision finding cryptographic game, illustrated in Algorithm 3.

Algorithm 3 The collision-finding game for a hash function H.

```
function collision-game$_{H,\mathcal{A}}(\kappa)$
    $x_1, x_2 \leftarrow \mathcal{A}(1^\kappa)$
    return $H_\kappa(x_1) = H_\kappa(x_2) \land x_1 \neq x_2$
end function
```

In this game, we ask the adversary to produce two different inputs x_1 and x_2 that have the same hash. Note that the hash function H is different for every value of κ (while the code that produces the hash output for every κ is the same, it must take κ into account when running), so we denote it H_κ. It gives κ bits of output. The adversary can have some hard-coded collisions in her source code, but, if our hash function is secure, these won't work[1] for sufficiently large values of κ. The security definition follows from this game:

Definition 5 (Collision Resistance)**.** *A hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^\kappa$ is* collision resistant *if for all PPT adversaries $\mathcal{A}$*

$$\Pr[\textit{collision-game}_{H,\mathcal{A}}(\kappa) = 1] \leq \text{negl}(\kappa)$$

Gossiping with Hashes

Collision resistance ensures that, if we are given a hash of something, we cannot later be given something else that hashes to the same value. Each object's hash is uniquely identified by its content. We say that objects are *content addressable*

[1] If you come from the field of cryptography, note that here we're bypassing the gory details of *keyed* hash functions by requiring the adversary be uniform.

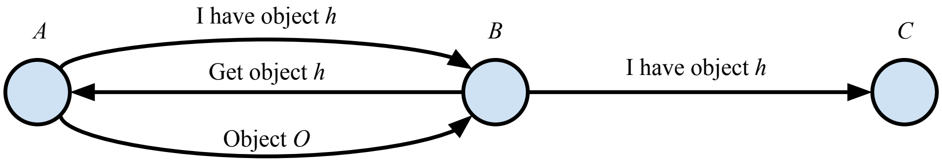


Figure 2.1: Gossiping via hashing. The hash function H is used to give content-addressible hash h to object O.

by their hashes. This makes hashes suitable for use as identifiers of objects as we exchange them on the network. When gossiping about objects, instead of sending a whole object to each of our peers, we can optimize the process by advertising the ownership of an object through its hash. The hash of an object used in this manner is called the *objectid*. If the peer already knows about this object, they can ignore our advertisement. If the peer has not seen the object before, they can request the object through its objectid. Only at this point, we send the full object to the peer. Upon receiving the object, the peer can verify that it is indeed the requested object by hashing it and comparing it to the stored objectid. Towards this purpose, each node must maintain a set of known objectids for quick lookup.

We now have a more complete understanding of how the gossiping protocol works, illustrated in Figure 2.1:

1. Node A first becomes aware of a new object O, either by receiving it from a peer, or by generating it locally. Object O has objectid $h = H(O)$, where the input to the hash function is a string-encoded version of the object.

2. A advertises its knowledge of O by sending a message indicating *I have an object with objectid* h to its peer B.

3. B receives the objectid h and checks against its database whether it has already seen this object. Suppose that it has not. At this point, B sends to A a message requesting the contents of the object with objectid h.

4. A sends to B the object O. Upon receiving O, the node B can verify that $h = H(O)$, where h is the requested objectid. This ensures that A sent the correct object to B.

5. In turn, B advertises to *its* peers that it now knows of an object with objectid h. It sends node C a message indicating this.

6. At this point, if C has already received O from A, it will not request the object from B. This is how the propagation in the gossiping algorithm stops.

The node B does not know whether h was newly generated by A, or if A is simply relaying. This gives a modicum of anonymity: When a new object is first sent to us from an IP address, we cannot deduce that the message is actually originating from that IP address.

Algorithm 5 The preimage-finding game for a hash function H.

```
function preimage-game_{H,A}(κ)
    $x \xleftarrow{\$} \{0,1\}^\kappa$
    $y \leftarrow H_\kappa(x)$
    $x' \leftarrow \mathcal{A}(y)$
    return $H_\kappa(x') = y$
end function
```

Preimage resistance

Hashes are also useful for allowing a party to *commit* to a value. The party reveals that hash, but not the object itself. Anyone who has the hash can verify that the object is correct once the full object has been received, but it is useful that this is not possible when seeing only the hash: It should be difficult to find the *preimage* of a hash given its image. Of course, the preimage of a hash can be found by performing an exhaustive search as illustrated in Algorithm 4, but this will take exponential time.

Algorithm 4 An exponential search for a preimage in a hash function that *certainly* finds the preimage.

```
function PREIMAGE-SEARCH_H(h)
    ctr ← 0
    while true do
        if $H(\text{ctr}) = h$ then
            return ctr
        end if
        ctr ← ctr + 1
    end while
end function
```

Naturally, we can define the property of *preimage resistance* using a cryptographic game.

In this game, the challenger chooses a random κ-bit value as the input. This is denoted by the $x \xleftarrow{\$} S$ symbol that indicates that an element x is chosen uniformly at random from the set S. Note here that to do this, the challenger, too, has access to randomness, and so any probabilities are also taken with respect to this randomness. The adversary is given $H(x)$ and would like to find x. As there are other inputs that produce the same $H(x)$, we ask her to produce some x' (equal or different from x) that has the same hash value as x. Preimage resistance is then defined as follows:

Definition 6 (Preimage Resistance). *A hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^\kappa$ is* preimage resistant *if for all PPT adversaries $\mathcal{A}$*

$$\Pr[\textit{preimage-game}_{H,\mathcal{A}}(\kappa) = 1] \leq \text{negl}(\kappa)$$

Algorithm 6 The second-preimage-finding game for a hash function H.

```
function 2nd − preimage − game$_{\mathcal{A},H}(\kappa)$:
    $x_1 \overset{\$}{\leftarrow} \{0,1\}^{2\kappa+1}$
    $x_2 \leftarrow \mathcal{A}(x_1)$
    return $x_1 \neq x_2 \wedge H_\kappa(x_1) = H_\kappa(x_2)$
end function
```

But, perhaps, we are simply asking too much of this adversary. It would already be a big problem for us if the adversary, given some x_1, can find a *different* x_2 that hashes to the same value. We call this property *second primage resistance*, and it is defined through the following game.

In this game, the adversary is given a randomly sampled $(2\kappa + 1)$-bit string input x_1 by the challenger. The adversary is successful if she can come up with an $x_2 \neq x_1$ that hashes to the same value as x_1, as illustrated in Algorithm 6. Second preimage resistance is then defined as follows:

Definition 7 (Second Preimage Resistance)**.** *A hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^\kappa$ is* second preimage resistant *if for all PPT adversaries $\mathcal{A}$*

$$\Pr[\textit{2nd-preimage-game}_{H,\mathcal{A}}(\kappa) = 1] \leq \mathrm{negl}(\kappa)$$

It seems that all three properties, collision resistance, preimage resistance, and second preimage resistance, are desirable. Let us examine whether some of these properties are stronger than the other. Finding a collision seems to be the easiest: The adversary is asked to come up with her own x_1, x_2, without any input from the challenger. The preimage adversaries seem stronger: If we have an adversary who can produce a preimage, we can use her to create a collision. We now show that collision resistance implies 2nd preimage resistance.

Theorem 3 (Collision Resistance $\implies$ 2nd Preimage Resistance)**.** *If a hash function H is collision resistant, then it is 2nd preimage resistant.*

Proof. Suppose, towards a contradiction, that H is not 2nd preimage resistant. Then, there exists an adversary $\mathcal{A}$ that can win the 2nd preimage game with non-negligible probability. We will construct an adversary $\mathcal{A}'$ against the collision challenger which uses $\mathcal{A}$ as a black box.

The adversary $\mathcal{A}'$ works as illustrated in Algorithm 7 and Figure 2.2. She first chooses an x_1 uniformly at random from the message space. She then hands this x_1 to $\mathcal{A}$, who, hopefully, produces an x_2 such that $x_1 \neq x_2$ and $H(x_1) = H(x_2)$. The adversary $\mathcal{A}'$ outputs the pair (x_1, x_2).

Algorithm 7 The adversary $\mathcal{A}'$ in the proof of Theorem 3.

```
function $\mathcal{A}'(1^\kappa)$:
    $x_1 \xleftarrow{\$} \{0,1\}^{2\kappa+1}$
    $x_2 \leftarrow \mathcal{A}(x_1)$
    return $(x_1, x_2)$
end function
```

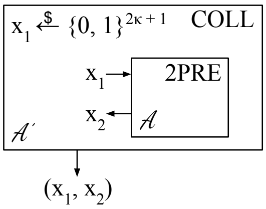


Figure 2.2: A visualization of Theorem 3.

If the adversary $\mathcal{A}$ is successful in breaking the 2nd preimage game, then $\mathcal{A}'$ will be successful in breaking the collision game:

$$\Pr[\mathsf{2nd\text{-}preimage\text{-}game}_{H,\mathcal{A}}(\kappa)] = \Pr[\mathsf{collision\text{-}game}_{H,\mathcal{A}'}(\kappa)]$$

Since $\Pr[\mathsf{2nd\text{-}preimage\text{-}game}_{\mathcal{A}}(\kappa)]$ is non-negligible, then so is $\Pr[\mathsf{collision\text{-}game}_{\mathcal{A}'}(\kappa)]$.

Furthermore, the adversary $\mathcal{A}'$ works in polynomial time, and so is also PPT. This contradicts the assumption that H is collision resistant. □

In addition, an adversary who breaks preimage resistant is stronger than an adversary who breaks 2nd preimage resistant. This is not surprising. An adversary who works towards finding a second preimage already has a first preimage, whereas an adversary who attempts to break preimage resistance only has an image.

This is captured in the following theorem:

Theorem 4 (2nd Preimage Resistance $\implies$ Preimage Resistance)**.** *If a hash function H is 2nd preimage resistant, then it is preimage resistant.*

Proof. Consider a PPT adversary $\mathcal{A}$ against the preimage game. We will construct an adversary $\mathcal{A}'$ against the 2nd preimage game. The adversary $\mathcal{A}'$ is depicted in Algorithm 8 and Figure 2.4. It is clear that $\mathcal{A}'$ is PPT because $\mathcal{A}$ is PPT.

Algorithm 8 The 2nd preimage adversary $\mathcal{A}'$ of Theorem 4.

```
function $\mathcal{A}'(x_1)$:
    $y \leftarrow H(x_1)$
    $x_2 \leftarrow \mathcal{A}(y)$
    return $x_2$
end function
```

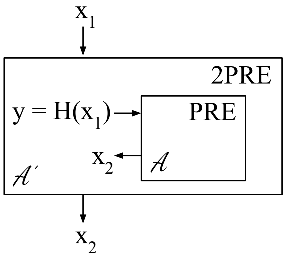


Figure 2.3: The reduction in the proof of Theorem 4.

Even though $\mathcal{A}$ could win the preimage game, this only guarantees that $H(x_1) = H(x_2)$. We need one additional property for $\mathcal{A}'$ to win the 2nd preimage game: It must hold that $x_1 \neq x_2$. We will now argue that this is often the case.

To do this, we draw the input space as a big box with each input illustrated as a pink dot, as shown in Figure 2.4. There are $2^{2\kappa+1}$ dots in the big box. We now partition this big box into smaller boxes, grouping each input with the other inputs that have the same image. We move the larger boxes (containing more dots) to the left of the picture and the smaller boxes (containing fewer dots) to the right of the picture.

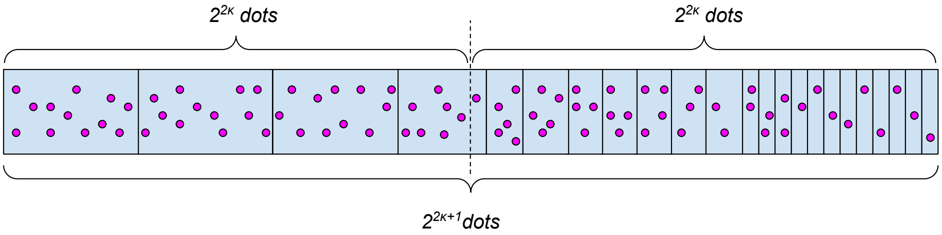


Figure 2.4: A visualization of the counting argument in Theorem 4.

We now split the big box into two big partitions exactly in the middle, as illustrated by the vertical dashed line, with $2^{2\kappa}$ dots on the left, and $2^{2\kappa}$ dots on the right (the dashed line may cut one of the smaller boxes in the middle, but this is fine).

We now argue that the boxes on the left of the dashed line are large.

Claim: *Each of the boxes to the left of, or on, the dashed line contains at least* 2^κ *dots.* To see this, we need a counting argument. Towards a contradiction, suppose that one of the boxes to the left of, or on, the dashed line contains fewer than 2^κ dots. Then, since the boxes are ordered, each box on the right of the dashed line contains fewer than 2^κ dots. However, because the output space is only 2^κ, this means that there are at most 2^κ boxes to the right of the dashed line. This means that, on the right of the dashed line, there can only be fewer than 2^κ dots $\times 2^\kappa$ boxes $= 2^{2\kappa}$ dots. But there are exactly $2^{2\kappa}$ dots to the right of the dashed line by construction. Therefore the claim is true.

Now that we have proven this claim, we can compare the probabilities of success of $\mathcal{A}$ and $\mathcal{A}'$. We will only consider the case where the uniform sampling of x_1 by the 2nd preimage challenger falls to the left of the dashed line. This happens with probability $\frac{1}{2}$. If this is the case, then there are at least 2^κ dots in the box of $H(x_1)$.

Therefore, *conditioned* on the fact that $\mathcal{A}$ is successful and that we have landed to the left of the dashed line, the probability of success of $\mathcal{A}'$ is $1 - 2^{-\kappa}$. The overall probability of success of $\mathcal{A}'$ is

$$\Pr[\textsf{2nd-preimage-game}_{\mathcal{A}'}(\kappa) \geq \frac{1}{2}(1 - 2^{-\kappa}) \Pr[\textsf{preimage-game}_{\mathcal{A}}(\kappa)] \, .$$

Since H is 2nd preimage resistant, the probability $\Pr[\textsf{2nd-preimage-game}_{\mathcal{A}'}(\kappa)]$ is negligible. Then so is $\Pr[\textsf{preimage-game}_{\mathcal{A}}(\kappa)]$, because $\frac{1}{2}$ is a constant and $(1-2^{-\kappa})$ is a value larger than $\frac{1}{2}$. Therefore, H is also preimage resistant. □

Note that in the above proof, the probabilities of success of $\mathcal{A}'$ and $\mathcal{A}$ are related by an inequality. This is because $\mathcal{A}'$ may also succeed in case we land to the right of the dashed line, but we are not accounting for this probability in our counting.

Additionally, observe how we went in the *forward direction* in this proof: Contrary to previous proofs, we did not assume that $\mathcal{A}$ succeeds with non-negligible probability at any point in the proof. As we have discussed in the previous chapter, this is a cleaner way of writing security proofs, although it takes some getting used to.

Hash Security

We can now define what it means for a hash function to be *cryptographically secure* or simply *secure*. Since collision adversaries are the weakest adversaries, we will simply require that our hash functions are collision resistant.

Definition 8 (Secure Hash Function)**.** *A hash function* $H : \{0,1\}^* \longrightarrow \{0,1\}^\kappa$ *is* secure *if there is a negligible function* negl *such that*

$$\forall PPT \mathcal{A} : \Pr[\textsf{\textit{collision-game}}_{H,\mathcal{A}}(\kappa)] = 1] \leq \textsf{\textit{negl}}$$

Applied Hashes

In practice, the hash functions most commonly used to build blockchains are `SHA256` (used by Bitcoin), `SHA3` or `keccak` (used by Ethereum), `blake2`, or Poseidon. As an example, `SHA256` is a hash function that takes any input and outputs $\kappa = 256$ bits (or 32 bytes). Here is the `SHA256` hash of the word "hello", displayed in hexadecimal format:

```
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

2.2 Signatures

When Alice sends money to Bob, she needs to *authorize* this payment. This means that the rest of the network needs Bob to *prove* that Alice really gave him her money instead of taking his word for it.

We will use a cryptographic *signature scheme* to do that. A cryptographic signature scheme is a means for a party to say "this message was really written by me" and it is not a forgery.

Public Key Cryptography

Before we discuss signature schemes, we must discuss the notion of identity in the cryptographic setting. In a traditional legal system, an identity is tied to a person's physical body and authorized by a government using papers such as a passport. In our case, we do not want to rely on physical bodies or centralized governments for proving identity. Anyone should be able to create a new identity *pseudonymously*, without necessarily associating it with their real person. To do this, a participant uses his computer to create a *key pair*. The key pair consists of two keys: A *public key* and a *secret key* (or *private key*). The public key portion of the key pair can be shared freely and even be made public. For example, it can be published on the owner's website, social media, or on a newspaper, without any security problems. On the contrary, a private key must be kept secret. We use the public key to specify *the identity about which we are speaking*. So, instead of saying "Alice" with such and such legal name, we refer to her by her public key. The private key can be used by Alice herself to prove her identity. That's why the private key must remain secret: If it falls into the hand of someone else, this someone else *really is* Alice. Of course, any physical person can create multiple different key pairs and maintain multiple identities that are not necessarily associated with one another. This idea will play an important role in achieving a basic level of pseudonymity.

The public key and private key are created together as a pair, because they are associated with one another mathematically in a unique manner. For every private key, there is a unique associated public key. For every public key, there is a unique associated private key. Given a private key, it is *easy* to get the respective public key. Given a public key, it is *hard* to get the respective private key, even though there is a unique such key. This is essential. If it were easy to get the private key from a public key, anyone who knew your public key could impersonate you. We will denote the public key pk and the secret (or private) key sk.

Unforgeability

A cryptographic signature is created using a particular private key sk and is denoted σ. It is associated with a particular message m. This means that the person who holds the private key has authorized this message m. The message will say something like: "I, Alice, gave 5 monetary units to Bob." Of course, the messages will be in a computer-readable format. We will make these messages more precise very soon.

A signature is associated with a particular message. If the user wants to sign a different message, a different signature must be created. If σ is the signature pertaining to the message m, then a signature σ' must be created for a different message $m' \neq m$. The signature σ will be invalid for message m'. This shows that cryptographic signatures are very different from hand-written signatures. Hand-written signatures are useless from a security point of view, as they can be copied and pasted around and their veracity cannot be checked. Cryptographic signatures cannot be copied underneath unauthorized messages. This would constitute a *forgery*, and we will make a precise computer science claim about how likely such forgeries are using a cryptographic game. We emphasize that we use the word *signature* because there is some analogy in physical signatures, but what we are achieving here is something truly different and much more powerful than pen-and-paper signatures. There is no reliance on courts of law and pseudoscientific "graphologists" to tell whether a signature is genuine. Instead, the reliance is on hard computational problems and formal cryptographic claims.

Before we define our security game, let us precisely state how a signature protocol works. Initially, Alice generates her key pair (pk, sk) by invoking a special algorithm $\mathsf{Gen}(1^\kappa)$. The public key and the secret key are both simple strings, κ bits long (in practice typically 256 bits each). She keeps sk secret and publishes pk by sending it to her friends. When the time comes for Alice to write a message m that she wishes to sign, she uses her private key sk to invoke the function $\mathsf{Sig}(sk, m)$ to obtain a signature σ. She sends the message m together with the signature σ to Bob. Bob already holds the public key pk of Alice. He uses the public key to invoke the function $\mathsf{Ver}(pk, m, \sigma)$, which returns true if the signature was genuinely created by Alice, or false otherwise. If the adversary sends a different message $m' \neq m$ together with this σ to Bob, the *Ver* function will return false.

If both the sender and the verifier are honest, the signature scheme should always work. This is what constitutes a *correct* signature scheme.

Definition 9 (Signature Correctness)**.** *Consider a signature scheme* $(\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$*. The scheme is* correct *if for any key pair* (pk, sk) *generated by invoking Gen, and for all messages* m*, it holds that* $\mathsf{Ver}(pk, m, Sig(sk, m)) = \mathsf{true}$*.*

For the security definition, we want the adversary to not be able to produce messages that were not authorized by their rightful owner. Since we are protecting an honest verifier who holds a correctly generated public key, our challenger will invoke *Gen* to obtain the key pair (pk, sk). Additionally, the adversary will be given access to pk, since this is public, but not access to sk (if the adversary has access to sk, we can have no hope). The adversary will then attempt to generate a signature σ that verifies for a message m using the public key pk. The adversary does not have to use the *Sig* algorithm, but can use any method she likes, as long as the Ver algorithm returns true. The game will output true if the Ver algorithm outputs true.

But note that this approach misses something: The adversary is trying to generate signatures *in the blind*, but in the real world, the adversary may see some authorized signatures that the honest signer really *did* make. The adversary can then make use of these signatures as she sees fit. For example, she might try to copy/paste a signature on a different message, or alter an existing signature on one message to create a signature on a different message. We would like our game to capture the fact that the adversary has this kind of access. To make her even more powerful, in our game we allow the adversary to ask the signer to sign *any message of her choice*. As long as the adversary can produce a signature for *any message she did not ask a signature for*, we consider it a successful forgery. This is a very powerful notion. The adversary has a lot of power, so if we can create a signature scheme that is resilient to such adversaries, we will have a lot of confidence in our protocol.

Algorithm 9 The existential forgery game for a signature scheme (Gen, Sig, Ver).

```
function existential-forgery-game$_{Gen,Sig,Ver,\mathcal{A}}(\kappa)$
    $(pk, sk) \leftarrow \mathsf{Gen}(1^\kappa)$
    $M \leftarrow \emptyset$
    function $\mathcal{O}$(m)
        $M \leftarrow M \cup \{m\}$
        return $\mathsf{Sig}(sk, m)$
    end function
    $m, \sigma \leftarrow \mathcal{A}^{\mathcal{O}}(pk)$
    return $\mathsf{Ver}(pk, \sigma, m) \wedge m \notin M$
end function
```

The *existential forgery game* is depicted in Algorithm 9. Initially, the challenger generates a keypair (pk, sk) using the honest key generation algorithm *Gen*. He then invokes the adversary, giving her access to *pk*. Since *pk* is κ bits long, we do not need to pass 1^κ to this adversary. A closure function $\mathcal{O}$ is defined within the challenger. When invoked with a message m, this function gives out a signature σ to the message m using the secret key *sk*, but without revealing the secret key. The closure also records the requested message in the set M. When the adversary is invoked, she is given *oracle access* to call the function $\mathcal{O}$. This is like a callback, and is denoted using the exponent notation. It means that $\mathcal{A}$ can call $\mathcal{O}$, but cannot look at its code. Critically, $\mathcal{A}$ cannot see the value *sk*. The adversary can make multiple *queries* to the oracle to obtain many signatures. Based on the signatures she sees, she can make yet further queries in an adaptive manner. When she is finally ready, the adversary is expected to produce a signature σ on a message m that was not queried to the oracle $\mathcal{O}$ (the adversary can trivially succeed in providing a signature for messages queried to the oracle). If the message and signature provided by the adversary pass the *Ver* check using the public key *pk*, the adversary is deemed successful.

The security definition is straightforward. Since we have already seen a few identical security definitions, try writing out the definition before looking at it.

Definition 10 (Secure Signature Schemes)**.** *A signature scheme* (Gen, Sig, Ver) *is called* secure *if there exists a negligible function* negl *such that*

$$\forall PPT \mathcal{A} : \Pr[\textit{existential-forgery}_{\mathsf{Gen},\mathsf{Sig},\mathsf{Ver},\mathcal{A}}(\kappa) = 1] < \mathrm{negl}(\kappa)$$

Secure signature schemes are sometimes called *existentially unforgeable signature schemes.*

Applied Signatures

Since the hash of a message is a unique identifier for it, it is sufficient that the hash of a message is signed instead of the message itself. This is often done in practice since it simplifies the implementation of signature schemes (libraries that implement signatures will do this for you). One class of secure signature schemes is called `ECDSA` and is based on the mathematical structure of *elliptic curves.* These curves define how public keys are structured, and they involve some algebra which makes it hard for private keys to be calculated based on the knowledge of just the public key. The computational problem on which hardness is based is called the *discrete logarithm problem*. There are different curves with different names, and each of them defines a different format for key pairs. A popular curve in cryptocurrencies is `secp256k1`. Another is `ed25519`.

Here is a public key of the `ed25519` signature scheme:

```
10b4b0f158afb93e3fd6111b564ad4c4054ae9a142362d8d9e05a9f2d6444530
```

Here is the respective private key:

```
7aa064fb575c861d5af00febf08c1c31620d5a70094c4bcb11cb2720630ee98a
```

Here is a signature generated with the above private key:

```
c538752e628c9ca43b3328f68afc76af40cf68732db00a8c9a885a6d41045b49
5ef44fb625a6742895d6819a63c254e352537998961a6802687140115811a409
```

As you can see, all of these look pretty much like random bytes. As blockchain protocol designers, the details of these curves and the exact underlying meaning of private keys and public keys do not matter to us, as long as the resulting signature scheme is secure. When implementing a cryptocurrency, it is best to use a library to do the signing and verification for us instead of implementing the signature scheme ourselves. Like many cryptographic primitives, it is extremely difficult to write a good, safe implementation for signature schemes. There are many pitfalls such as bad randomness and timing attacks. *Do not roll your own crypto.*

We will use signature schemes for basic money transfer. When Alice wishes to participate in the cryptocurrency, she will initially create an identity (pk, sk) by invoking *Gen*. When she is ready to get paid, she will hand out *pk* to the person wishing to pay her. Later, when the time comes for her to spend her money, she will authorize a payment by invoking the function *Sig* using her private key *sk*. The message describing the payment must contain both the amount that she is spending as well as the public key pk' of the receiver. Both of these must be included in m so that nobody can forge the amount that Alice paid or the identity of the receiver and swap it out for something else. Lastly, Alice's payment can be verified by invoking *Ver* using *pk* on m and the signature.

Problems

2.1 What is an example of a hash function that is not collision resistant, nor preimage resistant, nor 2nd preimage resistant?

2.2 The proof of Theorem 4 is a proof with reference to the illustration. Make it rigorous so that it doesn't speak of images, boxes, and dashed lines. Instead, use exact counting formulas and define appropriate notation to represent the boxes as equivalence classes.

2.3 In the proof of Theorem 4, the probabilities

$$\Pr[\textsf{2nd-preimage-game}_{\mathcal{A}'}(\kappa)] \geq \frac{1}{2}(1 - 2^{-\kappa}) \Pr[\textsf{preimage-game}_{\mathcal{A}}(\kappa)]$$

are compared by $\leq$. Why are they not exactly equal?

2.4 Give a simpler proof of Theorem 4.

2.5 Combine Theorems 3 and 4 to directly show that collision resistance implies preimage resistance.

2.6 Given a collision resistant hash function H, construct a preimage and 2nd preimage resistant hash function G which is not collision resistant. Prove these properties of G.

2.7 Given a collision resistant hash function H, make a collision resistant hash function G whose first bit is reliably predictable and prove that it is collision resistant. Prove these properties of G.

2.8 Construct a correct but insecure signature scheme.

2.9 Let $H : \{0,1\}^* \to \{0,1\}^\kappa$ be a collision-resistant hash function and define $G(x) = H(H(x))$. Show that G is a collision-resistant hash function.

Further Reading

Any cryptography book contains more information about hash functions and signature schemes. Our treatment here was superficial (as we did not, for example, treat *keyed* hash functions). Read the security definitions in *Modern Cryptography* [24] pertaining to *collision resistance*, *pre-image resistance* and *second pre-image resistance.* Read the security definitions for *existential unforgeability.* In the same book, you can find constructions for signature schemes using various methods, including some details on elliptic curves. Other good books that review these topics and talk about *how* to build a hash function or a signature scheme are *Introduction to Modern Cryptography* [24], *A Graduate Course in Applied Cryptography* [7], or *Foundations of Cryptography* [16, 17].

In our examples, we considered a $(2\kappa + 1)$-bit message space as an illustrative example. For a complete and nuanced cryptographic treatment of arbitrary-sized message spaces, which is beyond the scope of this book, refer to the seminal paper by Rogaway and Shrimpton [29] that formalized and proved these notions.

Chapter 3

The Transaction

3.1 Coins

We are now ready to start creating money. Given our insight that money comes to be through mutual social agreement—a social construct—we can create money simply by conjuring it through software. As long as it is difficult to forge and everyone agrees *who has what*, it will become something that can take on value through social agreement.

To solve the problem of knowing *who has what*, we will employ an unusual strategy: We will require that *every node on the network knows who owns what.* There are privacy and efficiency issues with this, and we will resolve both later.

Let us imagine how we can model the transfer of money between two parties. We need to represent that Alice made a payment to Bob of some particular amount. We will represent this through a *transaction.* We will draw a transaction as a node (a circle) with an *incoming edge* and an *outgoing edge.* The incoming edge is called the *input* and illustrates *who is paying.* The outgoing edge is called the *output* and illustrates *who is getting paid.* We will draw the *amount being transacted* above the edge and *the owner* below the respective edge. A transaction of 1 unit between Alice and Bob is illustrated in Figure 3.1. This may seem like an unusual way to illustrate things, but it will soon become clear why we are adopting it.

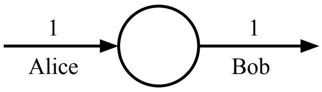


Figure 3.1: A transaction paying 1 unit of money from *Alice* to *Bob.*

For Alice to spend this money and give it to Bob, she must have been given this money previously. We will illustrate this by the output of one transaction connecting to the input of another, as illustrated in Figure 3.2.

Money changing hands in this manner is referred to as a *coin.* A coin has a current owner, denoted in the final outgoing output edge which is not connected to another transaction as input. It has a history of previous owners. The outgoing output edge of a transaction that is not connected to another transaction is an

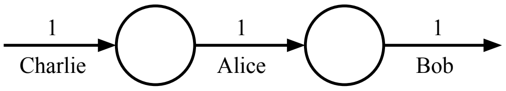


Figure 3.2: *Alice* pays 1 unit of money to *Bob*. She received this money from *Charlie*.

output *available for spending*. It is a *dangling output* and it is known as an *Unspent Transaction Output* (UTXO).

As we discussed in the signatures section, we will use public keys for identities. Our transactions will not contain a payment from “Alice” to “Bob”, but from some public key (whose respective secret key is held by Alice) to some other public key (whose respective secret key is held by Bob). This is illustrated in Figure 3.3. However, for convenience, we will write out *Alice* and *Bob* in place of their public keys, understanding that the payments are made to public keys and not legal identities. An appropriately encoded public key to which a payment can be made is also known as an *address*. An address can be exchanged between counterparties even before any transaction takes place.

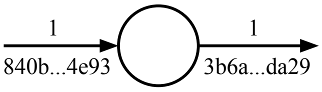


Figure 3.3: A transaction pays from one public key to another. It does not contain real names or other identifying information.

3.2 Multiple Outputs

It may be useful to pay for multiple things with a single transaction. If Alice receives her salary through one transaction, she may want to spend it on both her rent as well as on a book. A transaction can have multiple outputs. Each of the outputs may have a, potentially different, recipient public key and a, potentially different, amount. An example is illustrated in Figure 3.4. Each of the outputs can be spent independently. For example, Alice’s landlord can spend his money while the bookstore doesn’t. This transaction consumes one input and produces two outputs.

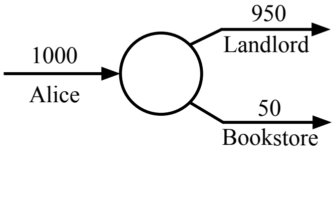


Figure 3.4: Alice uses her salary, the incoming edge, in a single transaction to pay for both her rent and a book in two different outgoing edges.

An outgoing edge can either be spent (if it is connected to another transaction) or unspent (if it is not connected to another transaction). It cannot be partially spent. A UTXO can only be spent in its *entirety* by being connected as input to a new transaction. If Alice wants to use *part* of her salary to buy a book, and keep the rest of her salary for later spending, she must still spend her salary output in its entirety and use it as a transaction input. She creates *two* outputs in this transaction: One paying the bookstore, and the other paying back to herself. The second output is the new UTXO that she can use to spend her remaining salary at a later time. This is known as a *change output* and is illustrated in Figure 3.5.

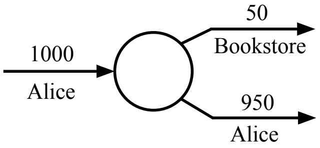


Figure 3.5: Alice uses her salary, the incoming edge, in a single transaction to pay for a book (top output edge). She uses the rest of her money to pay the *change* back to herself (bottom output edge).

Coins are often spent in a series of transactions like that. Alice uses her salary to pay for a series of things. She first pays for a book, then gives herself the change of that transaction. In a next transaction, she pays for an apple, and then gives herself the change of that. She then pays for a coffee, and gives herself the change for that. This process is illustrated in Figure 3.6. This graph has four UTXOs: Alice's remaining salary of 944 units, the payment for the book store of 50 units, the payment to the fruit market for 1 unit, and the payment to the coffee shop for 5 units. The left-most edge, Alice's original salary of 1000 is not a UTXO, since it is spent. Even though there are four UTXOs, only three transactions are depicted in this graph.

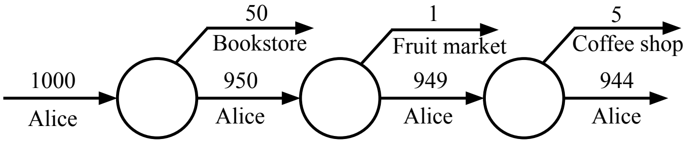


Figure 3.6: Alice uses her salary, the left-most edge, in a series of transactions, always giving change back to herself. The bottom-right edge is her remaining salary.

3.3 Multiple Inputs

It is also possible to *combine* multiple inputs into a single transaction to make a larger payment. For example, Alice can use two of her salaries, each of which resides in a different transaction output, to make a down payment for the house she is buying. This is illustrated in Figure 3.7. This transaction consumes two outputs and produces one new output.

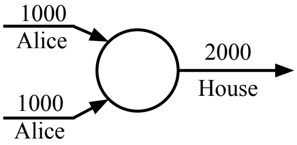


Figure 3.7: Alice combines two of her salaries (left, incoming edges) to pay for her house (right, outgoing edge).

Typically, a transaction will have one or more inputs and exactly two outputs. Alice will use one or more payments she has received (including previous change) to pay for something she is purchasing, and give herself the change remaining from the purchase.

3.4 The Conservation Law

Money needs to be scarce. When money is transacted, new money must not be created out of nothing. The input amounts to a transaction must match the output amounts of the same transaction. This is known as the *conservation law.* Let us denote by tx a transaction, by tx.ins its array of inputs, and by tx.outs its array of outputs. For each input *in* in tx.ins, let us denote by in.v the amount in the particular input, and similarly for *out.* We can write the conservation law in an equation.

Definition 11 (Conservation Law)**.** *Given a transaction* tx*, we say that it obeys the* Conservation Law *if*

$$\sum_{in \in \mathsf{tx}.ins} in.v = \sum_{out \in \mathsf{tx}.outs} out.v$$

Most transactions will obey this law. However, money must come from *somewhere*, and so there must be some initial transactions that do not obey this law. These are known as *coinbase* transactions. Even though they have valued outputs, they have no inputs. They are the only ones that do not respect the Conservation Law. Coinbase transactions follow very particular rules and they must be designated and limited, in order to have scarcity. We will explore the exact rules in more detail when we speak about macroeconomics in Chapter 5. Even though there can be transactions with no inputs (the coinbase transactions), every transaction must have at least one output.

3.5 Outpoints

Each transaction is given an *identifier* known as the txid . This is obtained by hashing the transaction data (including all of its inputs and outputs).

Since an input of a transaction is always spending a previous output, the input can just be a reference to a previous output. To reference an output, we need to specify the transaction it belongs to, using its txid, as well as the *index* of the output (whether it is the first output of the transaction, or the second output of the transaction, and so on). The pair (txid, idx) is used in place of an input and is sufficient to uniquely specify a previous output. The value idx is simply a number $0, 1, 2, \ldots$. This pair is known as an *outpoint*. An outpoint example is illustrated in Figure 3.8. We will illustrate the outpoint pair on top of an incoming edge to a transaction, although this will typically be implicit.

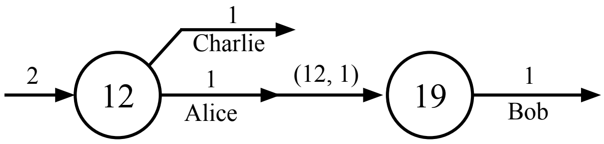


Figure 3.8: The transaction number 19 has a single input that spends the output number 1 (to Alice) of transaction number 12. The output number 0 of transaction number 12 (to Charlie) is still unspent. Here, we have highlighted the outpoint pair (12, 1) that connects the input of transaction 19 with the specific output of transaction 12.

3.6 The UTXO Set

The whole history of payments in our system forms a *transaction graph*. This is a Directed Acyclic Graph (DAG). It cannot contain cycles because transactions must be strictly orderable in the way that they spend: the input of a next transaction refers to outputs of previous transactions through outpoints that contain their hashes in the form of txids. An example transaction graph is illustrated in Figure 3.9. In this diagram, we are not showing the edge owners or amounts for conciseness. As new payments are made in the system, new transactions are added to the graph, but existing transactions are not modified, and previously added transactions are not removed. This is an append-only graph.

Some transactions in the graph have outputs that have all been spent, and we will never need to care about them again. Some transactions have dangling outputs, and so their outputs are available for spending. The money that is available for spending in the system is in the UTXOs. The set of all UTXOs forms the *UTXO set*.

Transactions in the transaction graph can be ordered in a sequence of transactions. We can do this by ordering the graph in topological order. We start with an empty sequence of transactions and we place the transactions from the graph into the sequence one by one, ensuring each transaction appears only once. The strategy we use to place transactions in the sequence is that we always choose a transaction whose inputs point to transactions that have all already been placed in the sequence. Since coinbase transactions have no inputs, they can always be placed in the sequence. We continue in this manner until all transactions have been placed into our sequence. There may be multiple ways to order transactions in this manner, but there will always be one way to do it. All of the ways are *consistent*: each transaction that spends from another transaction is placed in the sequence *after* the transaction that it spends from. This sequence of transactions, ordered in this consistent manner, is known as a *transaction ledger*. In Figure 3.9, transactions are labelled in one possible consistent order. As new transactions are added to our graph, they can also be appended to the transaction ledger while maintaining consistency.

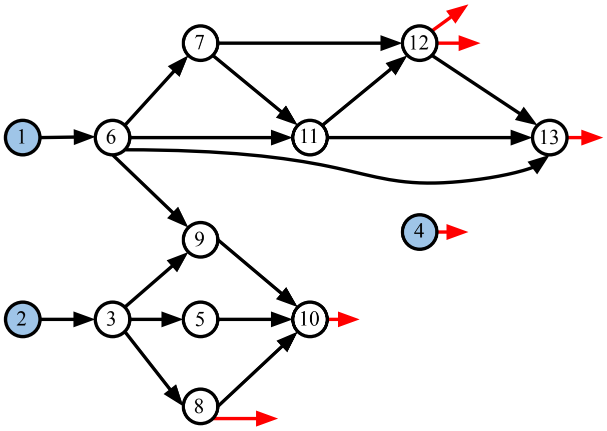


Figure 3.9: A transaction graph with 3 coinbase and 10 non-coinbase transactions. Coinbase transactions are shaded blue. There are 23 outputs, of which 6 belong to the UTXO set. The UTXO set is shaded red. Transaction number 8 contains both a spent and an unspent transaction output.

Each node in our network stores the *whole* transaction graph. When a party wishes to make a payment, they create a new tranasction and broadcast it to the network. This transaction is received by the other peers, who add it to their local transaction graph. Now everyone knows *who owns what* by looking at their local UTXO set. The sum of all the values in the UTXO set is equal to the total amount of money in the system.

In particular, if I, as an honest node, want to know *how much money I have*, I look at my local UTXO set and collect all those UTXOs who are marked with a public key whose respective private key I am in possession of. Summing all of their values gives me my current holdings.

3.7 Transaction Signatures

For a transaction to be valid, its inputs must point to outputs whose spending has been authorized by their rightful owner. This can be done by *signing* the new transaction data using the secret key that corresponds to the public key annotated on the previous output being spent. Let us look at the transaction that Alice creates in Figure 3.10. This new transaction, transaction 19, is spending from an output that belongs to Alice. The output being spent is the output with index 0 of transaction 7. The new transaction is paying Bob 1 unit and Charlie 2 units, for a total of 3 units.

Alice must authorize this spending by signing using her secret key. The data that she signs are the contents of the new transaction: The owners and amounts in the

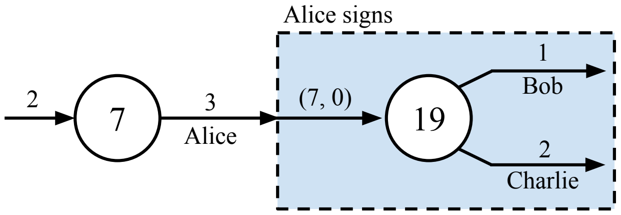


Figure 3.10: Alice creates a new transaction, transaction 19, by which she pays Bob 1 unit and Charlie 2 units. The transaction data include Bob's public key, Charlie's public key, the outgoing amounts 1 and 2, and the outpoint $(7, 0)$. These must all be signed by Alice's secret key.

outputs, and the outpoint of the input. It is not necessary to include the value of the input here, as the outpoint uniquely identifies it. It is imperative that Alice includes the public key of Bob in her signature when she creates this transaction. Otherwise, a malicious party, Eve, on the network could swap out Bob's public key with her own. If a secure signature scheme is used, any such forgery will be impossible due to existential unforgeability. The same applies in case Bob attempts to alter the amounts allocated to him and Charlie: Alice's signature will be invalidated, and the transaction will no longer look valid to any observers. Alice's signature on the transaction is packed together with the transaction and accompanies it whenever it is broadcast on the network.

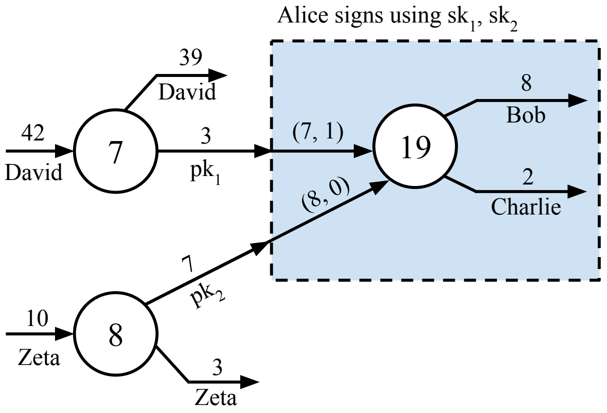


Figure 3.11: Alice creates a new transaction, transaction 19, by which she pays Bob 8 units and Charlie 2 units. The transaction data include Bob's public key, Charlie's public key, the outgoing amounts 1 and 2, and two outpoints, $(7, 1)$ and $(8, 0)$. All of these data must be signed twice: Once using Alice's sk_1 secret key, and once using Alice's sk_2 secret key, giving two different signatures σ_1 and σ_2 on the same data.

If Alice wishes to create a transaction with multiple inputs she controls, she must provide a signature corresponding to each of the inputs. Consider the example illustrated in Figure 3.11. Here, Alice wishes to spend two outputs that she owns. The first output has been paid to Alice's public key pk_1. The second output has been paid to Alice's public key pk_2. Alice controls the respective secret keys sk_1 and sk_2. Alice creates a new transaction, transaction 19, containing the desired inputs and outputs. In the inputs, she places two outpoints pointing to the two outputs she wishes to spend. The new transaction data, including the inputs and outputs, must all be signed twice: First, using Alice's sk_1 (and verifiable using pk_1), and secondly using Alice's sk_2 (and verifiable using pk_2). This will yield two different signatures σ_1 (created using sk_1) and σ_2 (created using sk_2). Both of these signatures verify on the *same* data, but using different public keys. A signature for the outpoint connected to each of the transaction's inputs must accompany the transaction whenever it is broadcast. A transaction is accompanied by as many signatures as it has inputs.

3.8 Transaction Creation

When the honest node Alice wishes to create a new transaction, she performs the following steps:

1. She requests the public key of the recipient through some off-chain means (e.g., via e-mail or a QR code).
2. She picks the UTXO outputs she wishes to spend from.
3. She creates a new transaction with the output *public keys* and *amounts* she wishes to pay to.
4. She creates transaction inputs where she places outpoints pointing to the UTXOs she wishes to spend from.
5. She collects all the above transaction data into a message.
6. For each of the outpoints, she uses her respective private key to sign the message.
7. She broadcasts the transaction and its signatures to the network.

3.9 Transaction Format

So far, we have treated a transaction as an abstract object. It is time to make this concrete. A transaction consists of its *inputs* and *outputs*:

1. Its list of *inputs*. Each element in this list is an outpoint, a pair in the form $(\mathsf{txid}, \mathsf{idx})$.
2. Its list of *outputs*. Each element in this list is a pair containing a public key (the owner of the output) and an integer amount.

An example transaction object looks like this:

```
{
  inputs: [
    {
      outpoint: (
        "cc6a88afaca94fec238258e3665d64cd
         de592e3ea13f151eca37d5e6589cd169",
        0
      )
    },
    {
      outpoint: (
        "3a648c42b90af46b9bba7ae723451002
         aa53baba187020051e0c32112bf458a0",
        3
      )
    }
  ],
  outputs: [
    {
      pk: "36dec8c46741efdab93a77f8fc75acec
           6e290b1ce0f280e04753db5c864a6469",
      amount: 5012900000000
    },
    {
      pk: "f71c1e7478459d90764d59dd9cb0ea9f
           62406d2c1412dea7c6de0c0355183066",
      amount: 3170000000000
    },
    {
      pk: "d9169b9601f3121f316f44e4d1bd0ecc
           d6d07df4d152fdcbf59a210e7c77e467"
      amount: 135000000000
    }
  ]
}
```

The amounts are given as integers. These are given in the smallest denomination possible that we want to allow our cryptocurrency to take. For example, Bitcoin uses the *satoshi* unit, equal to 10^{-8} bitcoin, whereas Ethereum uses the *wei*, equal to 10^{-18} ether. The coins cannot be further divided beyond integer amounts. This avoids floating point errors.

The transaction example above is an *unsigned* transaction, and cannot be broadcast to the network as-is. Signatures must be included with each outpoint to authenticate the respective input. A *signed* transaction looks something like this:

```
{
  inputs: [
    {
      outpoint: (
        "cc6a88afaca94fec238258e3665d64cd
         de592e3ea13f151eca37d5e6589cd169",
        0
      ),
      sig:
        "680b733e57690024d18025603f6df238
         b6f181c2de76b96961a90c1f0fc0d1e2
         42795ddd48f06e6ac2760579459b6b98
         ef77d91c4278dd0a9feaac91a57eae08"
    },
    {
      outpoint: (
        "3a648c42b90af46b9bba7ae723451002
         aa53baba187020051e0c32112bf458a0",
        3
      ),
      sig:
        "13f1eb894921c7535ac51aac874187ac
         993b0dcf7b3b439b15cf408dbf66db2b
         7739605ef9ab0bc79be7957a9e15ef0b
         e0dd92524c8881cc3fd3742c621e8c0b"
    }
  ],
  outputs: [
    {
      pk: "36dec8c46741efdab93a77f8fc75acec
           6e290b1ce0f280e04753db5c864a6469",
      amount: 5012900000000
    },
    {
      pk: "f71c1e7478459d90764d59dd9cb0ea9f
           62406d2c1412dea7c6de0c0355183066",
      amount: 3170000000000
    },
    {
      pk: "d9169b9601f3121f316f44e4d1bd0ecc
           d6d07df4d152fdcbf59a210e7c77e467",
      amount: 135000000000
    }
  ]
}
```

Naturally, the particularities of how transactions are encoded differ from blockchain to blockchain in practice. For example, Bitcoin uses the secp256k1 signature scheme to produce the public keys to receive payments and sha256 as the hash by which txid is calculated. In contrast, Ethereum uses the keccak hash to calculate txid, and Cosmos Hub uses the ed25519 signature scheme to generate public keys. The encoding of addresses also differs from cryptocurrency to cryptocurrency. For example, Bitcoin encodes addresses by taking the public key, hashing it with two different hashes (ripemd and sha256), and then applying the base58 encoding function which adds some checksums to avoid mistypes. These details are implementation details and are immaterial to the foundational functionality of the system.

3.10 Transaction Validation

In order to verify an incoming transaction from the network, each node must maintain the current transaction graph and in particular the *current UTXO set*. When a node sees a new transaction arriving from the network, it checks the new transaction's inputs to see if they belong to its current UTXO set. If the transaction is valid, the node adds the new transaction to its transaction graph. It also removes the new transaction's inputs from its *current UTXO set*, and adds the new transaction's outputs to its *current UTXO set*. This is how the transaction graph and the current UTXO set of each node evolve.

When a new transaction arrives at the door of a receiver for the first time, he must check that it is a valid transaction. This process is called *transaction verification*. It involves checking that this transaction is rightfully spending the money that it is claiming. If a transaction is deemed *valid*, then it is gossiped to the rest of the network. If a transaction is deemed *invalid*, then it is rejected, and it is not gossiped to the network. This protects from spammy transactions occupying the network. The checks performed when verifying a transaction include checking the Conservation Law and checking the signatures on the new transaction.

To perform these checks, he must follow the outpoints to find out the corresponding public keys and amounts. When the honest party Bob wishes to verify a transaction tx received from the network, he performs the following steps:

1. For each transaction input, he resolves the respective outpoint.
   (a) He checks that this outpoint is in his current UTXO set.
   (b) He retrieves the public key and amount of this outpoint.
   (c) He checks that a signature on the new transaction data verifies using the public key of the outpoint.
2. He checks that the Conservation Law holds (or that this is a valid coinbase transaction).
3. He removes the outpoints from his current UTXO set.
4. He adds the new outputs to his current UTXO set.

Let us discuss the step 1a above. This is a necessary condition to ensure that the money really *does* belong to its rightful owner and has not been previously spent. Consider what it would mean if this step failed. The verifier here is seeing a *new* transaction, a transaction he has never seen before. Yet, this transaction is spending from an output that is *not* in his UTXO set.

It's possible that *this outpoint was never added to the UTXO set in the first place*. This can occur for two different reasons. The first reason is malicious. The adversary is creating a transaction spending from a non-existing outpoint. This transaction must be rejected. The second reason is benign, and it is a *race condition*. If Alice pays Charlie in one transaction tx_1 and then Charlie pays David in another transaction tx_2, which spends from tx_1, then tx_1 and tx_2 will be broadcast in this order. However, the verifier may receive them on the network in a different order than they were sent. He can see tx_2 first, and tx_1 only later. If a verifier sees tx_2 first, then he cannot verify this transaction before he has seen tx_1. After all, he doesn't have the necessary public key to verify the respective signature, and he doesn't have the necessary amounts to verify the Conservation Law. He must necessarily *reject* tx_2. It is not the responsibility of the verifier to hold onto tx_2 until tx_1 is received, because he cannot know if such a tx_1 exists in the first place. For all it knows, an adversary could be attempting to spend a non-existent outpoint.

Alternatively, it's also possible that *this outpoint was added to the UTXO set, but was later removed from the UTXO set.* This means that there exists a different transaction which spends from the *same* output. This is called a *double spend*, and it must be rejected. Double spends are necessarily adversarially created. Honest parties do not produce double spending transactions.

Definition 12 (Double Spend)**.** *In the UTXO model, a transaction* tx *is a* double spend *or a* conflicting transaction *with another transaction* $\mathsf{tx}' \neq \mathsf{tx}$*, if* tx *and* tx' *both have some outpoint in their inputs that is the same, i.e.,*

$$\exists i, j \in \mathbb{N} : \mathsf{tx}.\mathsf{ins}[i].\mathsf{outpoint} = \mathsf{tx}'.\mathsf{ins}[j].\mathsf{outpoint}$$

Transactions broadcast from different parts of the network may arrive in a different order in other parts of the network. As different nodes on the network see different transactions at different times, each node may have a different opinion on what their current UTXO set is. This can lead to race conditions, which we tackle in the next chapter.

Problems

TBD

Further Reading

The UTXO model was first put forth in the context of Bitcoin by Satoshi Nakamoto. Satoshi introduced blockchains, and his paper, *Bitcoin: A Peer-to-Peer Electronic Cash System* [26] is the seminal paper that spoke about them for the first time. Consider it mandatory reading. It is an easy and short paper that includes details about the UTXO model that we explored in this chapter, but also blocks, chains, and SPV proofs that we will explore in the next chapters. The denomination *satoshi* in Bitcoin is named after Satoshi Nakamoto. Satoshi was a pseudonym of an alleged Japanese man who created both the Bitcoin paper and the first Bitcoin implementation in C++ during the years 2008-2009. After two short years of participation in the community, he disappeared mysteriously, never to be heard from again. His identity remains a mystery.

For many more details on transaction format particularities in the specific implementation of the UTXO model in Bitcoin, refer to the Bitcoin Developer Guide [1]. The books Mastering Bitcoin [2] and Mastering Ethereum [3] go into a lot of detail about the particularities of transaction, key, and address formats for Bitcoin and Ethereum respectively.

Chapter 4

Blocks

4.1 The Network Delay

In the last chapter, we created a monetary system in which participants can issue transactions and transfer money between one another while maintaining scarcity. We ensured participants can only spend their own money by using an unforgeable signature scheme to authenticate transactions that everyone verified. By gossiping transactions on the network, every participant assembled them, upon verification, into a transaction graph, and reading the UTXO set of that transaction graph enabled participants to determine *who owns what.* Furthermore, we made that transaction graph append-only. Our intuition is that, since every transaction is gossiped, everyone will eventually arrive at the same transaction graph, and the population will reach consensus on the UTXO set, even if some transactions take a moment to arrive to distant parts of the network. Note that, it doesn't matter to us if different honest parties observe transactions arriving on the network in different order, as long as all the parties compute the same UTXO set. This is the only thing that's important to determine *who owns what.* If two honest nodes accept the same set of transactions, even if they have processed them in different order, they will arrive at the same transaction graph and the UTXO set computed by them will be the same.

Of course, if a transaction is delayed while in transit on the network *for ever*, some nodes will not receive it and they will not be in consensus with the rest of the network, but this contradicts our non-eclipsing assumption that we introduced in Chapter 1. To make our intuition more precise, let us quantify how long it takes for a message to reach the whole network when it is broadcast by any party. We call this the *network delay parameter.*

Definition 13 (Network Delay)**.** *The* network delay parameter Δ *measures the maximum time it takes for a message to travel from one honest party to every other honest party on the network.*

Because honest parties gossip adversarial messages, this network delay ensures that even adversarial messages make it across the network within Δ time. That is, if an honest party receives an adversarial message at some point in time, then every honest party will see the same adversarial message within time Δ.

Now we can express our intuition that nodes reach consensus more precisely: While some transactions may be delayed up to Δ time, if no transactions are broadcast for a time of Δ, everyone's transaction graph will converge to be the same, and the UTXO set will be shared among along all honest parties. Unfortunately, this intuition is misguided, and things are not that simple. Things break down when double spend transactions are introduced by the adversary.

4.2 The Double Spend

Let's try to understand the double spending problem a little more carefully. Eve receives 1 unit of money from Alice through a transaction tx_1 as illustrated in Figure 4.1. The transaction tx_1 was created honestly by Alice and has one input of 1 unit coming from Alice, and one output of 1 unit paying Eve's public key. Eve now creates two transactions: The first transaction, tx_2 consumes the single output of tx_1 and pays 1 unit back to Alice. The second transaction, tx'_2 also consumes the single output of tx_1 and pays 1 unit, this time to Eve.

Suppose that two other parties, Charlie and Dave, have already seen tx_1 on the network, but have not yet received either of tx_2 or tx'_2. If Charlie receives tx_2, he will accept this transaction as valid. The transaction's input contains an outpoint that points to an element of the UTXO set in his view, since the output of tx_1 has not been previously spent. Furthermore, the transaction contains a valid signature by Eve created with the correct secret key, and it satisfies the Conservation Law. Upon accepting tx_2, Charlie updates his UTXO set, removing the output of tx_1 and adding the output of tx_2 to it. If Charlie now receives tx'_2, he will reject this transaction, as it is spending from an output that is not in the UTXO set in his view.

On the contrary, Dave receives tx'_2 first, and tx_2 after. When Dave receives tx'_2, he considers this a valid transaction, because it is spending from the UTXO set in his view. Dave, contrary to Charlie, believes that the output of tx_1 is still in the UTXO set. Dave then updates his UTXO set, removing the output of tx_1 and adding the output of tx'_2 to it.

At this point Charlie's and Dave's view are in disagreement. This is a problem. If Alice has also received tx_2 prior to tx'_2, she will justifiably believe that Eve paid her. While Charlie will accept Alice's money, because it is in his UTXO set, Dave will not accept Alice's money. We have arrived at a situation where Alice's money is not acceptable to everyone. We have lost consensus on *who owns what*.

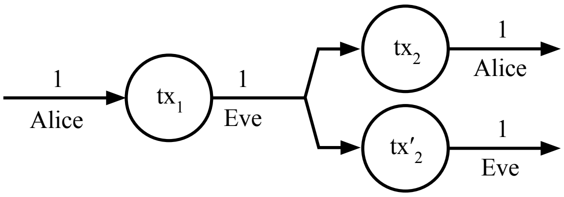


Figure 4.1: A double spend transaction. Alice paid Eve 1 unit through tx_1, but Eve spent it in both tx_2 and in tx'_2, which have different recipients.

4.3 Simple Ideas Don't Work

Let us consider three simple ideas to resolve the double spending problem that first come to mind. Sadly, these ideas won't work.

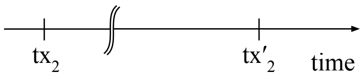


Figure 4.2: The first idea is unviable because it allows the adversary to retroactively invalidate an earlier transaction much later. Here, tx'_2 is initially withheld, but broadcast much later, causing an invalidation to the earlier tx_2 transaction.

Idea 1: Reject double spends altogether. Honest parties never double spend. Since the adversary is the only one creating double spends, why do we need to provide any assurances? We can opt to simply invalidate that money. We add the following rule to our protocol:

> If you see a transaction that is a double-spend, then consider *all* of the transaction outputs that pertain to the double spend transactions invalid.

This approach is problematic. The reason is that the adversary can retroactively take back a payment: She initially broadcasts tx_2 to the network paying Alice, but keeps tx'_2 withheld, as illustrated in the timeline of Figure 4.2. Alice, like everyone else, observes tx_2 on the network, but not tx'_2. She thinks this is a normal transaction and accepts the payment. In exchange for this payment, Alice provides a service to Eve: she serves her coffee. At a later time, Eve has enjoyed the coffee and has departed from Alice's establishment. At this point, Eve broadcasts tx'_2, a double spend of tx_2. Suddenly, everyone on the network considers both tx_2 and tx'_2 invalid. Alice's money is gone. Therefore, we cannot adopt this construction.

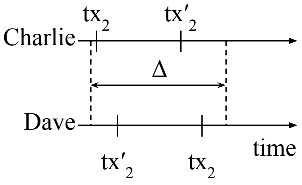


Figure 4.3: The second idea is unviable because parties have views in disagreement about transaction order. Here, Charlie believes tx_2 precedes tx'_2, whereas Dave believes tx'_2 precedes tx_2.

Idea 2: Accept the first transaction seen. As we saw, two different honest parties can disagree on the order in which two transactions arrived on the network. The situation is illustrated in Figure 4.3. Therefore, the following simple construction does not work:

> Among double spending transactions, accept the first, and reject every subsequent transaction.

However, we now note that these two transactions must be broadcast to honest parties in close succession, and in particular within time Δ. If the adversary were to reveal tx_2 to Charlie first, but then wait for more than Δ time until she revealed tx'_2 to Dave, then Charlie would have gossiped tx_2 and Dave would have receive it within Δ and prior to seeing tx'_2. In that case, Charlie and Dave would be in agreement. In order for the adversary to cause disagreement, she must broadcast the two double spending transactions to two different honest parties within time Δ of each other. Yet, this is simple for an adversary to do, so we cannot adopt this construction either.

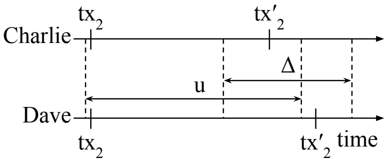


Figure 4.4: The third idea is unviable because parties disagree about whether tx_2 and tx'_2 have arrived within time u, and follow different policies. Here, Charlie rejects both transactions, whereas Dave accepts tx_2 and rejects tx'_2.

Idea 3: Reject double spends within u. Why don't we combine ideas 1 and 2? We saw that Idea 1 is problematic because it allows the adversary to retroactively take back a transaction after a long time into the future. We also saw that Idea 2 is problematic because it allows an adversary to cause disagreement when two transactions are broadcast in close succession. This creates a natural new construction idea:

> Upon seeing a transaction, wait for some time $u \geq \Delta$. If a double spend appears within the window u, reject all double spending transactions. However, if u has passed and we have not seen any double spends, accept the single transaction that we have seen. If a double spend appears in the future, just reject that one.

At first sight, this third idea seems to work: After time u has passed, either the money is accepted or not. The adversary cannot walk away as she did against Idea 1. If the adversary broadcasts two conflicting transactions within Δ time, as she did against Idea 2, they will both be rejected. It won't matter that different parties saw them in different order.

Sadly, upon closer inspection, this idea does not work, either. The strategy of the adversary is now to cause disagreement between Charlie and Dave with regards to *whether or not* the two transactions appeared within time u. The problematic situation is illustrated in Figure 4.4. The adversary initially broadcasts tx_2 to both Charlie and Dave. Both Charlie's and Dave's clocks start ticking to measure the time u. Right before time u hits, the adversary broadcasts tx'_2 to Charlie. Now, Charlie has seen a double spend within time u, and so he rejects both tx_2 and tx'_2. He also rebroadcasts tx'_2 to Dave, but this message will require time Δ to reach Dave. In the meantime, time u has passed, and Dave accepts tx_2. When tx'_2 arrives on Dave's end, Dave has already accepted tx_2 and now rejects tx'_2. Now Charlie and Dave are in disagreement: Dave thinks tx_2 is valid, whereas Charlie thinks tx_2 is invalid.

Try to think of more simple ideas to resolve this ordering issue. You'll see that none of them will work. For example, placing a timestamp within a transaction to ensure that every honest party simultaneously applies a transaction doesn't work, either (why?). Neither does the policy of accepting the transaction with the smallest `txid` among double spends (why?). We'll need to invent heavier artillery to attack this problem. Over the next few sections, we'll gradually derive the concepts of a *block* and a *chain*. While we do this, we will find issues with our scheme and keep augmenting it until we arrive at a fully working safe and live protocol.

4.4 Ledgers

Even though honest parties receive transactions in a different order on the network, we would like to have them coordinate with one another to report them in the same order. Each honest party will report a *ledger*, an ordered sequence of transactions. This ledger will not necessarily contain the transactions in the order they were received from the network. It may also not immediately report some transaction as soon as it is received from the network, but, akin to Idea 3 of Section 4.3, it may need to delay reporting it on its ledger for a bit. By reading that ledger reported by an honest party one transaction at a time from left to right, we can reconstruct the transaction graph and arrive at the UTXO set. If the honest parties agree on their reported ledgers, they will agree on the UTXO set. We'll therefore concern ourselves with the question of whether we can achieve consensus among the ledgers reported by honest parties.

We will soon figure out what each honest party should do internally in order to produce a ledger that is consistent with every other honest party, but before we get to *how* to do that, let us first more clearly articulate what exactly it is that we want to achieve.

We wish to build an honest party construction Π which we call the *full node*. This will be a piece of code which will be identically executed by all honest parties. It will implement peer discovery, the gossiping network communication to exchange messages on the network, and so on. In addition, we'll make Π expose two functionalities: A *write* functionality, and a *read* functionality. The *write* functionality accepts a brand new transaction. This transaction is broadcast and gossiped to every other party on the network by the full node. The *read* functionality returns a ledger of transactions. These functionalities are used by the *wallet*. Together, the full node and the wallet constitute the software that is running on the human user's machine. The human user only interacts with the wallet, while the full node sits on the backend.

The wallet is a piece of software that is capable of creating and signing new transactions to make payments as instructed by the user. The wallet also shows whether a payment was received, and displays the balance of the user. These functionalities are made possible by having access to the *write* and *read* functionalities exposed by the full node. The *write* functionality is what the wallet uses to broadcast a new transaction into the network, whereas the *read* functionality allows the wallet to obtain a ledger that it can then use to obtain the UTXO set in order to display whether a payment has been completed and to calculate the user balances. This architecture is illustrated in Figure 4.5.

We'll have more to say about the wallet portion in the next chapters. For now, let us focus on how to build the full node. Our goal will be to have all the honest

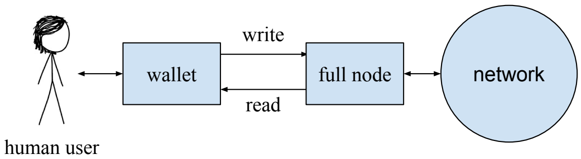


Figure 4.5: The human user interacts with the wallet. The wallet interacts with the full node by invoking its *write* and *read* functionalities. The full node interacts with the network.

full nodes agree on the ledgers they are reporting when their *read* functionality is invoked. That way, everyone will arrive at the same UTXO set and agree *who owns what.*

Definition 14 (Ledger)**.** *A* ledger *of an honest party P reported at time r, denoted L_r^P is a finite* sequence of transactions *returned when the honest party P invokes the* read *functionality of its honest protocol* Π.

We take note here that ledgers are dependent on both P and r. It is nonsensical to speak about "the ledger" without specifying *who* and *when.* While these parameters may sometimes be implicit, it is imperative that we understand what we are talking about. Ignoring the *who* and the *when* of the ledger, and speaking about "the ledger" as if it is a global view, is a common cause for confusion and misconception. As we saw in the previous section, the ledgers of different honest parties may be in disagreement, and there might not even be a well-defined global ledger.

We want the ledgers reported by honest parties to have two *virtues.* We give an intuitive definition of these virtues now, but we will return to define them more formally in Chapter 12.

Definition 15 (Safety (informal))**.** *For any two honest parties, their reported ledgers at any point in time are be equal.*

Definition 16 (Liveness (informal))**.** *If an honest party* writes *a transaction into its ledger, then this transaction appears in the ledgers of all honest parties "soon".*

Let us think what these virtues are saying. If a ledger has *safety*, then *bad things don't happen.* We do not run into double-spend situations, or into disagreements about *who owns what.* If a ledger has *liveness*, then *good things happen.* When an honest party attempts to issue a transaction, this transaction actually does take place.

It is easy to build a protocol that has *safety* or *liveness*, but not both together.

Safe but not live. A safe but not live protocol acts as follows. Whenever the *read* functionality is invoked, it returns the empty sequence of transactions as the ledger. Whenever the *write* functionality is invoked, it ignores the transaction being written. It never reads or writes anything from the network. Because the *read* functionality always returns the same thing, safety is trivially satisfied. Liveness, however, is not satisfied. When an honest party attempts to make a transaction, it is never reported in the ledgers of other honest parties.

Live but not safe. A live but not safe protocol acts as follows. Each honest party begins with an empty initial ledger. Whenever the *write* functionality is invoked with a transaction, this transaction is appended to the local ledger and broadcast to the network. When a new transaction is received from the network, it is gossiped to the rest of the network and appended to the local ledger. The *read* functionality returns the local ledger. This protocol is live because every honest transaction makes it to the ledger of every honest party. However, it is not safe, because transactions are reported on the ledgers in the order they are received from the network, and this order may be different for different parties.

Our design goal for the rest of this book will be to build protocols that are *both* safe and live simultaneously. Those protocols are called *secure*.

Definition 17 (Security). *A protocol is called* secure *if it produces ledgers that are both* safe *and* live*.*

4.5 Rare Events

We previously determined that the root cause of double spending transactions being problematic is because the adversary can issue them in short succession, and in particular within time Δ. It would be useful to enforce that the adversary issues transactions at a slower rate. We'd like to limit the rate at which the adversary can issue transactions at once every Δ, with *periods of silence* in between. If transactions are spaced apart by Δ time, then no harm can come from double spends. We can simply adopt the strategy of accepting the first among multiple double spending transactions, and ignoring subsequent double spends.

To force the adversary to issue transactions Δ apart, we'd like to require her to obtain a *ticket* before she can issue a transaction, and each of these tickets should be obtained every Δ time. Then, when she issues a transaction, the adversary will be required to associate the transaction with the ticket, and that will cause the ticket to be expended. The same ticket cannot be used for multiple transactions. Of course, our protocol does not know who is adversarial and who is honest and must treat everyone equally, so the ticket system applies to honest and adversarial parties alike.

If our tickets are issued more often than Δ apart, this will not be a good protocol, as double spends will still be possible, harming safety. On the contrary, if our tickets are issued much longer than Δ apart, the honest parties will take a long time to issue transactions, and liveness will deteriorate. The choice of how often to allow tickets to be issued highlights a theme that we will keep returning to throughout this book: A trade-off between safety and liveness. This trade-off is illustrated in Figure 4.6.

But how can we create such a ticketing system? How can we get them to behave this way in a permissionless world, where there is no authority to issue these tickets and ensure they are spread Δ apart?

4.6 Proof-of-Work

There is a natural candidate for creating rare events like the tickets we need in a permissionless world. When we introduced hashes in Chapter 2, we gave a brute-force algorithm (Algorithm 4) that breaks the preimage property of a hash, but takes exponential time. However, we can tweak the problem of finding a preimage so that it is not an *exact* preimage, but a "close enough" solution. This will allow

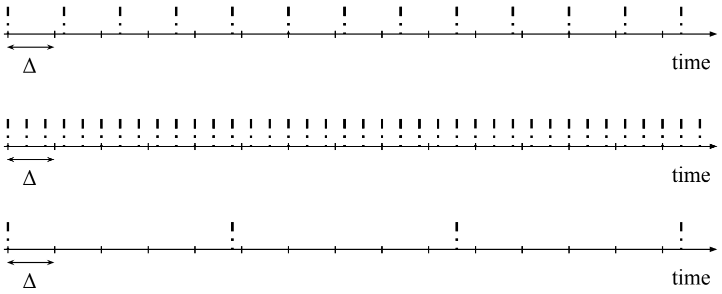


Figure 4.6: Positioning tickets in time. Tickets spread apart by slightly more than Δ (top) achieve good safety and liveness. Tickets spread apart less than Δ (middle) can cause safety violations. Tickets spread apart much more than Δ (bottom) cause liveness to deteriorate.

us to create a problem that is *moderately hard*; it is not *hard* in the computational sense, but we can tweak how long it takes to solve it. We are interested in "close enough" hash preimages of the image $h = 0$; that is, we want an input B to the hash function H, so that $H(B)$ is a small enough number. *How close* is defined by the value T, the *target*. We write this requirement in the form of an inequality:

$$H(B) \leq T$$

This inequality is known as the *proof-of-work inequality*. Finding a B that satisfies this equality is known as *performing work*. Remember that the output of the hash function $H(B)$ is a κ-bit binary number. We are comparing this number against the value T, another number. The inequality is satisfied if $H(B)$, treated numerically, is not larger than the number T.

The challenge is to find the unknown value B so that the inequality is satisfied. Of course, there will be multiple B values that satisfy this inequality. Upon solving this inequality, one obtains a ticket to issue a transaction. While solving the inequality is moderately hard, and requires many trial-and-error invocations of the hash function, verifying that the particular B satisfies the equation is very easy: It requires just one hash invocation.

Algorithm 10 The proof-of-work algorithm.

```
function $\textsc{PoW}_{H,T}$
    $\mathsf{ctr} \overset{\$}{\leftarrow} \{0,1\}^\kappa$
    while $\mathsf{true}$ do
        $B \leftarrow \mathsf{ctr}$
        if $H(B) \leq T$ then
            return B
        end if
        $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
    end while
end function
```

To find a ticket, one can execute a strategy similar to the exhaustive search for finding a preimage. We start at a random $B = \mathsf{ctr}$ and keep looking for a ctr that satisfies the proof-of-work inequality by incrementing ctr and checking whether the equation is satisfied. This is illustrated in Algorithm 10 and is known as *mining*. The value ctr is known as the *nonce*, and has no significance beyond ensuring that the proof-of-work inequality is satisfied. Once a party finds a ticket B, this ticket can be gossiped to the network and relayed by others. Any party can check that the ticket has a valid proof-of-work, without performing the exhaustive search again. Intermediary nodes on the gossip network cannot modify the contents of a ticket in transit, as this will invalidate the proof-of-work, and they would have to mine a new ticket.

The larger we make the target T, the easier it is to solve the inequality. The smaller we make the target T, the more difficult it becomes. That's why we call the value $\frac{1}{T}$ the *difficulty*: The larger the difficulty, the more difficult it is to find a solution to the inequality. By increasing T, we space tickets closer together. By decreasing T, we space tickets further apart from each other. At the extremes, setting $T = 0$ makes the proof-of-work problem equivalent to finding the hash preimage of 0, which needs exponential time. On the other end of the spectrum, setting $T = 2^\kappa$ makes the proof-of-work problem trivial, because every hash satisfies the inequality $H(B) \leq 2^\kappa$.

4.7 The Block

We have introduced a moderately hard problem that forces the adversary to spread out her tickets in time by more than Δ. But how do we associate tickets with transactions? We cannot just require the adversary to send a ticket together with a transaction. The adversary may reuse a ticket multiple times, or may use a ticket together with one transaction when communicating with one party and the same ticket with another transaction when communicating with another party. We need to somehow tie the ticket together with a transaction so that a ticket is only usable for a particular transaction. To do this, we set $B = \mathsf{tx} \parallel \mathsf{ctr}$ and require the ticket to satisfy $H(B) \leq T$, as before, where tx is the transaction we want the ticket to be tied to. The notation $\parallel$ denotes the concatenation of two strings (separating them appropriately so that they don't accidentally mix with each other by using an appropriate encoding such as JSON). Now, upon receiving a transaction/ticket pair, we can check that the ticket corresponds to the particular transaction. Furthermore, if the adversary attempts to replace the transaction within B with a different one, then this will cause $H(B)$ to change. The value $H(B)$ is *committing* to a particular transaction; changing the transaction changes the value $H(B)$ in a way that cannot be predicted. If she changes the transaction, this will invalidate the proof-of-work and the adversary will have to perform yet another exhaustive search to find a new ticket.

We've solved one problem, but have introduced another: If we require one ticket per transaction for the adversary, then we must do the same for the honest parties. This means that at most one transaction can be executed every Δ. If the honest party wants to issue multiple transactions, this will take a long time, and liveness will deteriorate. In order to solve this, we will *bundle* transactions together into a sequence $\overline{x} = (\mathsf{tx}_1, \mathsf{tx}_2, \ldots, \mathsf{tx}_n)$, and place this within the ticket, setting $B = \overline{x} \parallel \mathsf{ctr}$. Instead of each ticket being associated with *one* transaction, each ticket is associated with a sequence of transactions $\overline{x}$. Such a ticket is known as a *block*. The sequence of transactions $\overline{x}$ is also known as the *block payload*. Now each honest party can issue as many transactions as he wants in one go, as long as he is able to mine one block. In practice, to save on communication, $\overline{x}$ may be a list of transaction ids instead of the literal transactions themselves. We'll see more ways to optimize communication when we talk about Light Clients in Chapter 10. From now on, we will no longer speak of tickets, but will speak of blocks, even if a block contains just one transaction.

A natural question now arises: If we've bundled multiple transations together into one block, hasn't the double spending problem resurfaced? The answer is *no*, because, in order to verify a block, all the transactions must be sent together in the bundle. As part of our block validation rule, we will require that there are no conflicting transactions within the payload $\overline{x}$ of a block. If $\overline{x}$ contains two mutually double spending transactions, the whole block will be rejected. Blocks are either accepted or rejected as a whole. We will not partially accept transactions within a block.

Double spending transactions can still appear in different blocks, but the moderate difficulty of the proof-of-work puzzle ensures that blocks are produced sufficiently far apart. Like transactions, blocks are gossiped over the network. If two blocks are transmitted at least Δ time apart, then the first will arrive at the doorstep of every honest party before the second. The second block, containing a transaction that double spends a transaction in the first block, can then be rejected by all honest parties. Again, here, the whole block will be rejected, not just the particular pathological transaction.

4.8 The Mempool

Even though we allow each mining party to include many transactions into their own blocks, honest parties are still encumbered with the responsibility to mine a block before they can get their transactions accepted by others. This wait time may still be long and this scheme is not very good for liveness. Worse yet, some honest parties may have large computational power, while other honest parties may have smaller computational power, and so the block generation time for each honest party might be vastly different. It would be nice if we didn't tie the liveness of each honest party to that particular party's computational power, but instead allowed the honest parties to work together to confirm each others' transactions.

Towards that purpose, we design the system as follows: A transaction can still be issued independently of a block and broadcast to the network. The transaction is gossiped until it reaches everyone on the network as usual. However, the transaction is placed into a temporary waiting area called the *mempool* , ensuring double spends are resolved one way or another (any of the simple ideas we discussed previously can be used as strategies to resolve double spends). Each honest party maintains their own mempool $\overline{x}$ of transactions that are still in limbo. Because the double spending problem has not yet been solved by finding a block, different honest parties may disagree about what their mempool looks like. However, each mempool is independently locally consistent.

Definition 18 (Mempool)**.** *The* mempool $\overline{x}$ *of an honest party* P *at time* r *is the sequence of transactions that have been received and validated, but have not yet been* *included in a block.*

Upon receiving a transaction from the network, an honest party performs the usual transaction validation checks on it before adding the transaction to its own mempool, so the mempool does not contain double spends. Of course, the mempool of one honest party may contain a transaction which conflicts another transaction in the mempool of another honest party. As honestly generated transactions can always be appended to the transaction graph, the mempool of every honest party will contain every honestly generated transaction that has not yet made it into a block, as long as that transaction was broadcasted at least Δ ago. When an honest party tries to mine a block, he includes all the transactions in his mempool into this block. This has the benefit that, whenever *any* honest party finds a block, *all* pending honest transactions are included in that block, as long as they have been issued more than Δ time ago. The transactions in the mempool are ordered, and so are transactions within a block.

We illustrate a block in Figure 4.7. We'll follow the convention of drawing a block as a rectangle and a transaction as a circle.

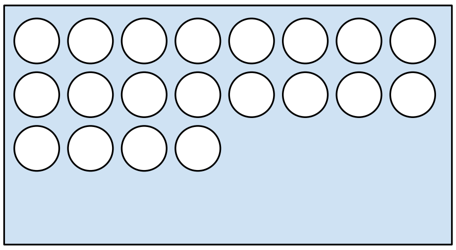


Figure 4.7: A block containing ordered transactions. We will draw a block using a rectangle and a transaction using a circle.

When a transaction makes it into a block, we term the transaction *confirmed* . An honest party invokes the *read* functionality of its ledger, only confirmed transactions are reported. This ensures that ledgers are consistent, salvaging safety. Liveness is guaranteed from the fact that an honest block will eventually appear, and this will cause all the transactions in the mempool to become confirmed.

4.9 Chain of Blocks

Similar to transactions, we give blocks identifiers, which are the hashes of their contents. A blockid of block B is the value $H(B)$. Contrary to transactions, these blockids will generally be a small value, because they all satisfy the proof-of-work equation. Therefore, these hashes will begin with a run of 0s.

At this point, we have introduced blocks that function as tickets to allow for the spaced-out broadcasting of transactions. However, our system is not yet safe. There's a problem: While the proof-of-work process ensures that the adversary gets a block in regular spaced out intervals, the adversary is not guaranteed to use these block at the time of issuance. Instead, the adversary could *withhold* a couple of blocks and broadcast them all later in close succession, and in particular closer than Δ apart in time. This defeats the purpose we set to achieve. The attack stems from the ability of the adversary to use blocks that are stale and have been set aside for long. We can resolve the issue by requiring each block to be *fresh.* We can do this by having the block include, in addition to its transactions, a pointer to a previous recent block. This pointer, denoted by s, is the blockid of the previous block known to the miner, and is known as the *previd.* Our modified block format is now $B = s \,\|\, \overline{x} \,\|\, \mathsf{ctr}$. Note that, similarly to before, the adversary cannot retroactively change s after mining a block, because this will invalidate the proof-of-work. This helps with freshness: If a block B is old and includes a pointer s to an even older block, this s cannot be retroactively chnaged to point to a newer block to make B appear fresh. The final proof-of-work algorithm illustrated in Algorithm 11. accepts both s and $\overline{x}$ as parameters and tries to find a ctr that satisfies the proof-of-work equation.

Algorithm 11 The mining algorithm for a block associated with multiple transactions $\overline{x}$ and a previous blockid s.

```
function $\textsc{PoW}_{H,T}(s, \overline{x})$
    $\mathsf{ctr} \xleftarrow{\$} \{0,1\}^\kappa$
    while $\mathsf{true}$ do
        $B \leftarrow s \,\|\, \overline{x} \,\|\, \mathsf{ctr}$
        if $H(B) \leq T$ then
            return B
        end if
        $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
    end while
end function
```

Because each block points to a previous block, the blocks form a chain. This is known as the *blockchain* and is illustrated in Figure 4.8. The arrows follow the direction of the pointers, pointing from one block to the block it refers to. While the pointers have a right-to-left direction, the blockchain was produced from left-to-right. From now on, we will draw blocks as simple squares, omitting the respective transactions inside for brevity.

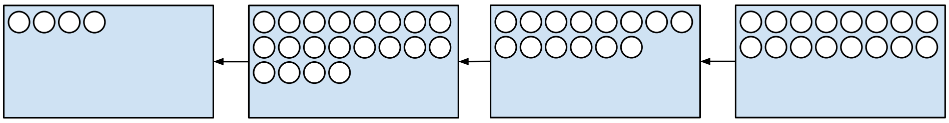


Figure 4.8: A chain of blocks. The blocks were mined from left to right.

4.10 Genesis

We've added pointers between blocks to ensure freshness, and each block attests about its freshness by a pointer to a recent block. But what about the freshness of the first block on the chain? That first block cannot have a pointer to a parent block. How do we ensure its freshness? In particular, how do we prevent an adversary from having produced many blocks in secret prior to the blockchain protocol even being announced, anticipating its announcement, perhaps in collusion with the creator of the protocol? To prove the freshness of the first block, the first block contains a reference which ties the theoretical and mathematical world of the blockchain with real world events. The first block, known as the *genesis block*, or simply *genesis*, contains in its metadata the headline of a recent newspaper describing impactful and unpredictable world events. Because these data are committed into the proof-of-work of the genesis block, similarly to the values s and ctr, they cannot be retroactively changed without invalidating the proof-of-work. This is the way of the protocol creator indicating there's *nothing up his sleeve*, and that the genesis block was produced *after* a certain point in time. This is an *anchor in time* and guards against the fear of *premining attacks*, situations in which the protocol inventor is adversarial and has mined blocks prior to making the protocol public. The genesis block is denoted $\mathcal{G}$. A blockchain beginning with the genesis block is illustrated in Figure 4.9.

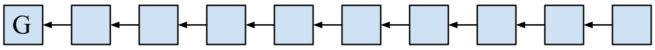


Figure 4.9: A blockchain beginning from genesis.

The Bitcoin genesis block was mined on January 3rd, 2009 and contains the following quote:

> The Times 03/Jan/2009 Chancellor on brink of second bailout for banks

The quote is both an anchor in time and a political message by Satoshi Nakamoto. The frontpage of The Times newspaper of that day appears in Figure 4.10.

We will make all honest parties check that all blockchains begin with $\mathcal{G}$ to avoid premining. The genesis block is hard-coded into the source code of each honest party. By convention, we consider $\mathcal{G}$ to be an *honestly generated* block.

4.11 Mining

Honest parties *constantly* attempt to mine blocks by following the mining algorithm. The steps that an honest miner follows are as follows:

1. Maintain a consistent local mempool $\overline{x}$ of transactions.
2. Attempt to mine a block $B = s \| \overline{x} \| \mathsf{ctr}$ by finding a ctr that satisfies $H(B) \leq T$.
3. If mining is successful, broadcast the newly created block B.
4. Otherwise, keep mining.

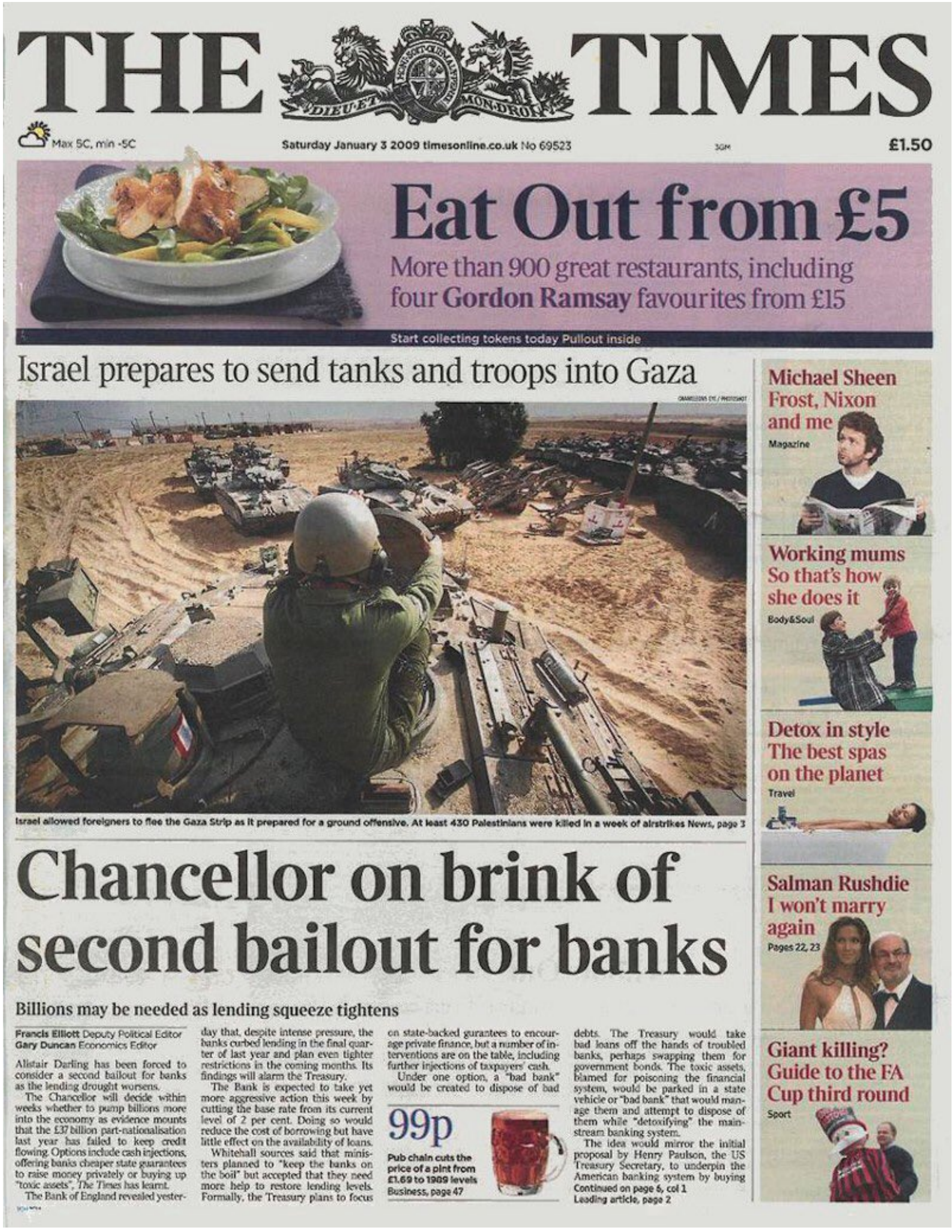
The honest parties keep mining even if their mempool is empty. This means that the blockchain might consist even of empty blocks.

Problems

4.1 When mining in Algorithm 11, an honest party begins their search at a value of ctr which is initialized at a uniformly randomly chosen sample from $\{0,1\}^\kappa$. After this, the honest party searches sequentially by incrementing ctr. Consider the case when there is a constant number of parties, each of which is running for a polynomial amount of time, and all of them are trying to mine with the same s and $\overline{x}$. Use a union bound to estimate whether the probability that any two honest parties will use the same ctr when attempting to find proof-of-work is negligible or not.

Further Reading

Proof-of-Work was invented long before blockchains, in 1992, by Cynthia Dwork and Moni Naor in their seminal paper *Pricing via Processing or Combatting Junk Mail* [13] published in CRYPTO '92. Satoshi Nakamoto invented the blockchain [26] by using proof-of-work as a building block.



THE TIMES

Max 5C, min -5C

Saturday January 3 2009 timesonline.co.uk No 69523

£1.50

Eat Out from £5

More than 900 great restaurants, including four **Gordon Ramsay** favourites from £15

Start collecting tokens today Pullout inside

Israel prepares to send tanks and troops into Gaza

Israel allowed foreigners to flee the Gaza Strip as it prepared for a ground offensive. At least 430 Palestinians were killed in a week of airstrikes News, page 3

Michael Sheen Frost, Nixon and me

Magazine

Working mums So that's how she does it

Body&Soul

Detox in style The best spas on the planet

Travel

Salman Rushdie I won't marry again

Pages 22, 23

Giant killing? Guide to the FA Cup third round

Sport

Chancellor on brink of second bailout for banks

Billions may be needed as lending squeeze tightens

Francis Elliott Deputy Political Editor
Gary Duncan Economics Editor

Alistair Darling has been forced to consider a second bailout for banks as the lending drought worsens.

The Chancellor will decide within weeks whether to pump billions more into the economy as evidence mounts that the £37 billion part-nationalisation last year has failed to keep credit flowing. Options include cash injections, offering banks cheaper state guarantees to raise money privately or buying up "toxic assets", *The Times* has learnt.

The Bank of England revealed yesterday that, despite intense pressure, the banks curbed lending in the final quarter of last year and plan even tighter restrictions in the coming months. Its findings will alarm the Treasury.

The Bank is expected to take yet more aggressive action this week by cutting the base rate from its current level of 2 per cent. Doing so would reduce the cost of borrowing but have little effect on the availability of loans.

Whitehall sources said that ministers planned to "keep the banks on the boil" but accepted that they need more help to restore lending levels. Formally, the Treasury plans to focus on state-backed gurantees to encourage private finance, but a number of interventions are on the table, including further injections of taxpayers' cash.

Under one option, a "bad bank" would be created to dispose of bad debts. The Treasury would take bad loans off the hands of troubled banks, perhaps swapping them for government bonds. The toxic assets, blamed for poisoning the financial system, would be parked in a state vehicle or "bad bank" that would manage them and attempt to dispose of them while "detoxifying" the mainstream banking system.

The idea would mirror the initial proposal by Henry Paulson, the US Treasury Secretary, to underpin the American banking system by buying

Continued on page 6, col 1
Leading article, page 2

99p

Pub chain cuts the price of a pint from £1.69 to 1989 levels
Business, page 47

Figure 4.10: The genesis block contains metadata with a time anchor to real-world events.

Chapter 5

The Chain

In the last chapter, we discussed the necessity for blocks as a mechanism to enforce *periods of silence* so that double spends can be adequately separated in time. We then linked blocks together into chains in order to ensure freshness so that an adversary cannot retroactively bring blocks mined and withheld in the old past. However, we left the notion of "freshness" and block validation undefined. In this chapter, we will fill in all the missing details of how blocks and chains are verified. By the end of this chapter, we will have a rudimentary, yet complete and secure, blockchain protocol.

5.1 The Target

Previously, we designed the proof-of-work inequality $H(B) \leq T$ in order to create *rare events* and *periods of silence* as a moderately hard version of the exponentially hard hash preimage problem. In order to prevent double spends, we needed to ensure that blocks are spaced Δ apart. Let us now calculate the correct value of T so that block production is spaced more than Δ time apart. To do this, we need to first find out when the proof-of-work inequality holds. If we start with a freshly generated ctr and place it into $B = s \parallel \overline{x} \parallel \mathsf{ctr}$, what is the probability that $H(B) \leq T$? This question is not straightforward to answer, because the output of a hash function may not be uniformly distributed (see also Problem 2.7).

In fact, not all collision resistant hash functions are suitable for proof-of-work. We want to demand that, whenever the hash is queried with a fresh input, the output is uniformly distributed in $\{0, 1\}^\kappa$. This extra assumption is known as the *random oracle model* and we will return to it in Chapter 11. Using the random oracle model, we can calculate the probability that a particular nonce trial is successful:

$$p = \Pr[H(B) \leq T] = \frac{T}{2^\kappa}$$

During the exhaustive search of proof-of-work, if a particular hash trial $H(B)$ is found to be $H(B) \leq T$, we call that hash function query a *successful query*. Otherwise, if $H(B) > T$, we call the query *unsuccessful*. The expected number of queries needed until a successful query is found is $\frac{1}{p} = \frac{2^\kappa}{T}$.

As protocol designers, we need to set T correctly so that blocks are produced at a rate smaller than once per Δ. For this, we need to have an estimate of how fast the participating nodes on the network can compute hash queries. Let's start with some definitions. Suppose that a typical honest computer can process $q \in \mathbb{N}$ computations of H during every unit of time. Let us also suppose that the total number of computers mining on the network is $n \in \mathbb{N}$ and the adversary controls $t \in \mathbb{N}$ of them. Here, we are assuming that all computers have equal processing power, but if there is some computer on the network that has more processing power, we can just think about that powerful computer as a collection of smaller, less powerful computers, each of which has q computational power. The model still works out. We will call each of these computers a *party*. Each mining party will roughly correspond to a node on the network, but in practice these are not necessarily the same: A single network node can correspond to multiple parties if it is a powerful computer. On the other hand, because our system is permissionless, a party may spawn multiple nodes that all share the same computational power. Additionally, even though we are saying that the adversary is in control of t adversarial parties, we are still considering the situation where the adversary is a "puppet master" who controls every adversarial party of the network by a central "master plan" $\mathcal{A}$. We say that such a party controlled by the adversary is *adversarial*, *corrupt*, or *malicious*, and we will use these terms interchangably.

The total number of hash queries that can be executed in the unit of time is nq. Since all parties are simultaneously trying to get a block all the time, the expected block generation time is $\frac{1}{pnq} = \frac{2^\kappa}{Tnq}$. If we want this value to be above Δ we must demand

$$\frac{2^\kappa}{Tnq} > \Delta \Rightarrow T < \frac{2^\kappa}{nq\Delta}\,.$$

In a nutshell, the more computational power there is on the network, the larger we must make the difficulty. Also, the larger the network delay, the larger we must make the difficulty. We will use f to denote the honest block production rate (expected blocks per unit of time), and $\eta = \frac{1}{f}$ the expected time between two blocks being generated. For now, we'll estimate that $f \simeq (n-t)qp$, but we will give a more formal definition for f in Chapter 11.

As the designers of the protocol, we will make some reasonable estimations for Δ, n, t and q. Based on these, we will calculate the value T. For the time being, we will fix T to be a constant that never changes. This model is called the *static difficulty model*. We'll revisit this assumption in Chapter 8.

5.2 The Arrow of Time

In the previous chapter, we decided to link blocks together into a chain in order to ensure each block has *freshness* and cannot be brought back from the past. It turns out that this natural and intuitive chain structure actually has many more nice properties than just ensuring freshness, which we will explore over the next few chapters.

In order to be able to speak about chains more precisely, let us introduce a bit of notation. A chain $\mathcal{C}$ is a finite sequence of blocks $\mathcal{C} = (B_0, B_1, B_2, \ldots, B_{n-1})$ ordered chronologically. The chain *length* $|\mathcal{C}|$ is the number of blocks in the chain. We address these blocks by their zero-based index, so $\mathcal{C}[0]$ is the first (and oldest) block on the chain, $\mathcal{C}[1]$ is the second, and so on. The first block among a complete chain is the genesis block, so $\mathcal{C}[0] = \mathcal{G}$. We use negative indexing to address blocks from the end of the chain, so $\mathcal{C}[-1]$ is the last (and most recent) block, $\mathcal{C}[-2]$ is the penultimate block, and so on. The block $\mathcal{C}[-1]$ is called the *tip*. The index of a block within its chain is called the block's *height*, so genesis has height 0 and the tip has height $|\mathcal{C}| - 1$. The *depth* of a block within its chain is the distance of a block from the tip. For example, the tip has depth 1, the block preceding the tip has depth 2 and so on. It will also be useful to speak about *chunks* of a chain, continuous portions of the chain. We'll denote by $\mathcal{C}[i{:}j]$ the portion of the chain starting at block with index i (inclusive) and ending at block with index j (exclusive). For example, $(B_0, B_1, B_2)[0:2] = (B_0, B_1)$. These i and j can also be negative, again meaning indexing from the end. Omitting i means starting from the beginning, while omitting j means going until the end of the chain. For example $(B_0, B_1, B_2, B_3, B_4, B_5)[-2:] = (B_4, B_5)$.

We have not yet specified how to actually verify a block, and what are the conditions for accepting a block. How exactly do we determine if a block is *fresh* or *unfresh*? Let us try the following strategy to see where it leads.

> A fresh block must extend the most recent block we've seen. If we receive a fresh block, we accept it. Otherwise we reject it as unfresh.

Having considered the simple ideas that don't work for transaction ordering, this strategy seems eerily fragile. Let us go over an example where it breaks. Consider the successful queries illustrated in the timeline of Figure 5.1. In this timeline, the successful queries are spaced apart more than Δ by an appropriate choice of the T parameter. Some of the queries happen to be honest (black solid diamonds), whereas other queries happen to be adversarial (red hollow circles). The honest party who performed the successful query 1 will mine a block on top of $\mathcal{G}$, and broadcast it to everyone. Let's call this honest party *party* 1 and the block just generated *block* 1. Since block 1 will reach the whole network by time Δ, everyone will have adopted it prior to the next successful query. Honest party 2 will then mine a block on top of 1 and broadcast it to everyone, and so on, until the adversary gets the successful query 5. At this point, the adversary mines block 5 on top of 4, but can choose to selectively disclose it. The adversary does *not* disclose block 5 to party 6, but discloses it to party 7 at some time t_A right before query 6 occurs. Since t_A and query 6 are less than Δ apart, party 6 will not receive block 5 before mining block 6, and so block 6 will extend block 4. At a later time t_B, occurring after query 6 but before query 7, party 6 receives block 5, but it is too late: He has already mined block 6 and cannot go back and change it. On the contrary, party 7 has seen block 5 prior to block 6 and is made to falsely believe that party 6 wrongly chose not to extend block 5. Party 7 rejects block 6 as *unfresh* and mines block 7 on top of block 5.

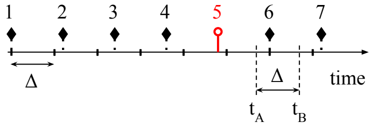


Figure 5.1: A series of successful honest (black solid diamonds) and adversarial (red hollow circles) queries regularly spaced apart more than δ.

This scenario leads to the *blocktree* illustrated in Figure 5.2. Worse yet, now some parties who detected that 5 was broadcast late will accept 6 as valid, whereas the parties who received 5 early will accept 7 as valid. While we were hoping that the proof-of-work construction would give us a single chain as a global source of truth, we have failed, and, again, we have honest views in disagreement.

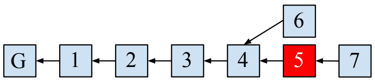


Figure 5.2: A block*tree* instead of a block*chain* arising out of the timeline of figure 5.1.

There is a disagreement about what is fresh and what is unfresh. We need to adopt a more robust measure of time. Towards this, let us reinspect the chain structure we discovered in the previous chapter.

Because the hash of a block cannot be predicted prior to the proof-of-work actually taking place (otherwise proof-of-work would be solvable by a faster algorithm than exhaustive search), the value s of the previd must be computed prior to the value $H(s \,\|\, \overline{x} \,\|\, \mathsf{ctr})$. In other words, the previd s must already be known when mining for a new block that includes s as its previd. Blocks are mined in the order they appear. The pointers between blocks in the chain are *causality links*. They point backwards in time. The blockchain grows forward in time and defines an *arrow of time*. When we observe a blockchain, we can deduce that all of its blocks were mined one after the other in the order that we observe them. In fact, if we have a blockchain consisting of a sequence of blocks, if we change the payload $\overline{x}$ in an earlier block, this will cause the proof-of-work in that same block and all subsequent blocks to become invalid: Changing the $\overline{x}$ of the block will cause its own blockid to change and its proof-of-work to be invalidated. This, in turn, requires changing the previd s of the next block to maintain the chain structure, causing *its* proof-of-work to be invalidated, and so on and so forth for every subsequent block on the chain, as illustrated in Figure 5.3. To change the contents of one block of the chain, the adversary would have to go back and redo all the proof of work. Hence, the adversary cannot detach a block from its parent and append it to a different parent to make it appear newer than it is.

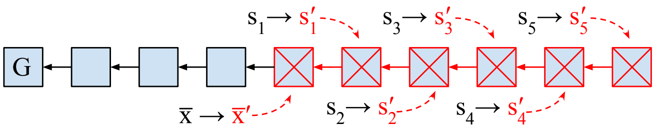


Figure 5.3: Changing just one small part of a block's content causes a cascading effect invalidating the proof-of-work of the whole chain inheriting from that block.

A longer blockchain takes more time to generate, whereas a shorter blockchain takes less time to generate (we will prove this relationship between blockchains and time precisely in Chapter 13). Therefore, we can use the *length* of a blockchain to estimate how much time it took to generate, and how fresh its tip is.

This allows a more objective comparison about which chain to adopt. Since a longer chain takes more time to produce than a shorter chain, its tip is roughly the “freshest”. We now adopt the following rule for which chain to choose among multiple candidate chains:

> **Longest Chain Rule.** Among all chains on the network, adopt the *longest* chain (the one with the most blocks) as your *canonical chain.* If there are multiple competing chains of the same length, choose any chain arbitrarily.

Using this rule, a node doesn’t have to stay online on the network all the time to observe which block is fresh and which one isn’t. Additionally, two honest nodes that have seen the same network messages, regardless of the order in which they received them, agree on their verdict of which chain is best.

We can now summarize the honest miner’s rules, which every honest miner is constantly running:

1. Maintain a local canonical chain $\mathcal{C}$.
2. Keep mining on the canonical chain.
3. If mining is successful, broadcast the new block to the network, and update the local canonical chain.
4. If not, keep mining.
5. In the meantime, when a new valid chain arrives on the network, check its length, and update the local canonical chain if the newly arriving chain is longer than the existing canonical chain.

The most common situation is for the canonical chain to be replaced by a new chain that extends the previous chain by one more block. However, sometimes a chain appears that is longer but does not extend the previous chain. In this case, we say that the chain *reorgs* (reorganizes) and certain blocks are *rolled back*.

Once a party has adopted a canonical chain, reporting a ledger of transactions to implement the *read* operation of a full node is straightforward and is depicted in Algorithm 12. Given a chain $\mathcal{C} = (B_0, B_1, \ldots, B_{n-1})$, the ledger reported by the honest party when the *read* functionality is invoked is the sequence of transactions in the same order as they appear on the chain. Here, we use the notation $B.\overline{x}$ to denote the $\overline{x}$ component of block $B = s \parallel \overline{x} \parallel \mathsf{ctr}$. First, the ledger is initialized to be the empty sequence $[\,]$. The algorithm then traverses each block B of chain $\mathcal{C}$ in chronological order (while we use the notation $B \in \mathcal{C}$, remember that $\mathcal{C}$ is an ordered sequence and not a set). The transactions of the genesis block B_0 are appended to the ledger, in the same order as they appear within the genesis block’s transaction sequence $B_0.\overline{x}$ (which, again, is a sequence and not a set of transactions); next, the transactions of the block following genesis are appended; and so on, until the transactions of the tip are appended.

Algorithm 12 The naïve *read* rule.

```
function READ($\mathcal{C}$)
    $L \leftarrow [\,]$
    for $B \in \mathcal{C}$ do
        for $\mathsf{tx} \in B.\overline{x}$ do
            $L \leftarrow L \,\|\, \mathsf{tx}$
        end for
    end for
    return $L$
end function
```

As you can judge by the name of this rule, this is a rule which we plan to soon revise (in Chapter 6).

5.3 The Stochastic Nature of Work

Despite adopting the Longest Chain Rule, the situation with successful queries is far worse than we anticipated. So far, we have presented successful queries as being spaced apart *exactly* $\frac{1}{pnq}$, but mining is a stochastic process which will have irregularities. The value $\frac{1}{pnq}$ is just the *expected* block generation time, but sometimes successful queries will be spaced apart more closely and sometimes sparsely, as illustrated in Figure 5.4. While we attempted to space queries reasonably apart, it can just so happen that certain queries are successful in close proximity (less than Δ apart). This means that, even without the presence of an adversary, the honest parties can happen to mine blocktrees and be in disagreement.

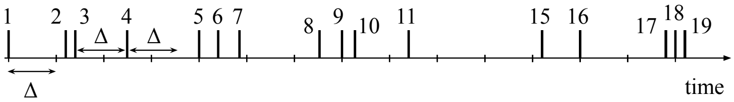


Figure 5.4: The stochastic nature of work makes honest successful queries irregular.

In Figure 5.5, we illustrate a blocktree that can arise out of the timeline of queries in Figure 5.4. Even though *all* successful queries were honest, the honest parties who used the longest chain rule built a blocktree and not a single blockchain. At the end of the execution, the honest parties are split between the canonical chains they have adopted: Some parties have adopted the chain ending on block 17, some have adopted the chain ending on block 18, and some have adopted the chain on block 19, as their canonical chain. If these three blocks contain double spends, the different honest parties could be reporting different ledgers. This happened even though the adversary was not mining any blocks. This is a loss of safety.

Are we back to square one? Not quite. We'll soon see that, with a few last modifications to our protocol, we can soon achieve security. We'll finish the protocol design in the next chapter, and prove its security in Chapters 11-13. For the moment, let us try to understand how the longest chain protocol works a little more precisely.

Let's go back to Figure 5.5. In this picture, we notice that there really is only one longest chain in the blocktree, except with some extra blocks here and there. This

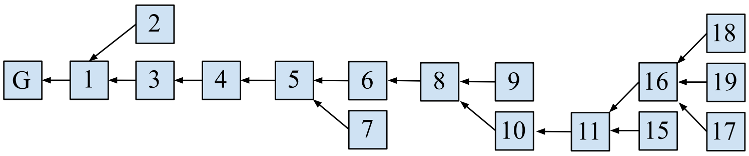


Figure 5.5: A blocktree arising out of the timeline of honest-only queries of Figure 5.4.

longest chain begins from genesis and goes on until block 16. Here and there on the sidelines of the longest chain, we see a few blocks popping up such as block 2 and block 7. These smaller branches alongside the longest chain are called *temporary forks* and can be one or more blocks long, but typically these temporary forks will be quite short. We'll explore the reason for this in the next chapter.

Our chain's lifecycle begins at genesis, with every honest party mining on top of it. When there is a single block mined by just one honest party, and that block makes it to the doorstep of every other honest party before they've had the opportunity to mine their own block, the chain grows for everyone. This is the usual case. In rare occassions, two different honest parties will mine two different blocks at approximately the same time. Because the one honest party has not received the block of the other honest party while he is mining his own, both of these blocks will extend the same parent, so we will have a temporary fork. We say that the chains of honest parties have diverged. These two forks will now have the same height, and so the honest population may be split among which one to adopt. If it so happens that the honest parties again mine two different blocks at about the same time across the different forks, these temporary forks will grow. However, as soon as there is a block mined on any of the two temporary forks, and no other block is mined in close temporal proximity to it, there will be one definitive longest chain, and every honest party will adopt it. We say that the honest chains have converged again.

We notice that, in the optimistic setting where no adversary is mining, the honest parties *converge* on one chain whenever there is an honest successful query separated Δ from all other honest successful queries (it is also possible that honest parties converge in other moments if they get lucky). Therefore, we call these moments in time *convergence opportunities*:

Definition 19 (Convergence Opportunity (informal))**.** *An honest successful query is called a* convergence opportunity *if it is Δ separated in time from all other honest successful queries.*

Notice that the definition of a convergence opportunity only requires that an honest successful query is separated from other *honest* successful queries. It does not concern itself with adversarial queries. It is possible that an adversary will cause a divergence even during a convergence opportunity.

Convergence opportunities are events that are visible in *God's eye* who has a full and current global view of each and every party. The individual parties may not be able to tell whether a successful query is also a convergence opportunity. The hope is that, even though honest parties cannot detect convergence opportunities, these will still happen behind the scenes and will lead to convergence. While this convergence will take place often, the honest parties will not be aware of whether convergence has occurred, but this will not cause any issue.

5.4 The Honest Majority Assumption

Even though we used the longest chain rule to ensure that blocks from the long past are not retroactively revived, an adversary with large mining power compared to the honest parties can still mine in secret. Look at Figure 5.6. Here, the honest parties mine sequentially on top of block 1 and produce block 2 and its descendants. The adversary independently mines on top of block 1 and produces $2'$ and its descendants, which the adversary does not broadcast to the network yet. When the adversary has accumulated a bunch of more blocks than the honest party, he broadcasts the whole chain to the network. The honest parties adopt the newly broadcasted adversarial chain, abandoning their own.

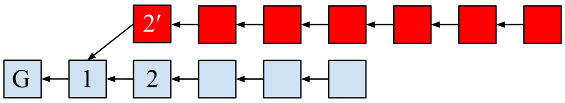


Figure 5.6: An adversary with dishonest majority (top red chain) can outpace the honest parties (bottom blue chain).

This situation is pretty bad. It causes a *loss of safety*, as some of the honest parties have switched from their ledgers to the adversary's ledger, causing disagreement. It also causes a *loss of liveness*, because the adversary may choose not to include any honest transactions. However, if the adversary only has a minority of mining power, even though these attacks can start small, they cannot grow for much longer, as the adversary will be left behind in the dust by the honest parties who will be mining more quickly. We'll explore these attacks much more deeply in Chapter 7, and we'll formalize them in Chapter 11. However, it should already be clear that we must require that the adversary controls only a minority of the compute of the network. We will call this the *honest majority assumption*.

Definition 20 (Honest Majority Assumption)**.** *The* honest majority assumption *mandates that the adversary controls less compute than the honest parties:*

$$t < n - t$$

Using the honest majority assumption, and with a small modification to the way we read ledgers, in the next chapter we will intuitively argue that our protocol is now secure.

5.5 Coinbase Transactions

Now that we have completed the design of our longest chain protocol, we have finally solved the problem of how money is *transferred* in a publicly verifiable and decentralized manner. However, we never tackled the problem of how money is *created* in a decentralized manner. If there is no central authority to issue money, how can we create new money, and who receives any new money that is created? With the invention of blocks, there is a natural choice for this. Since blocks are generated in regular intervals, we can use the block to inject the money supply with new money in a manner that maintains controlled scarcity. This is where the *coinbase* transaction comes into play. Recall that a coinbase transaction has no inputs and a single output with a certain value coming out of it, as illustrated in Figure 5.7. It is the only type of transaction that does not respect the Conservation Law. Our rule says that every block must have *at most one* coinbase transaction, and that transaction must appear *first* in the list of transactions of the block, if at all (different blockchain systems may have slightly different details on what rules they enforce on the syntax of coinbase transactions). The public key in the output of the coinbase transaction is freely chosen by the miner. Naturally, because this is free money, the miner will typically make that public key be their own, so that they can reap the benefits of mining.

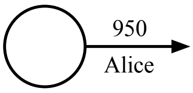


Figure 5.7: A coinbase transaction paying Alice the amount of 950 units. The coinbase has no inputs and a single output.

We now need to make a decision about how much money can appear in the output of the coinbase transaction of each block. The value on the coinbase transaction has two parts: A block *reward* and the *transaction fees*.

The *block reward* f_r corresponds to money that is newly created. This money comes out of thin air, and is injected into the system without coming from anywhere. It is how new money is created. The block reward must follow an algorithmic, prespecified rule that is hard-coded into the system. One simple way is to set a fixed amount to be the limit of the block reward, and ensure every full node has this amount hard-coded into its source code. For example, we can set the block reward to simply be the constant $f_r = 50$ units. During the validation of a coinbase transaction, each client checks that this amount is respected. In case the coinbase transaction deviates from this rule, the coinbase transaction is considered invalid, and so is the block it is contained in. The decision on how the block reward is algorithmically determined is called the *macroeconomic policy* of the chain. We'll have more to say about this in Chapter 8. Let's observe what exactly is happening with this construction. Instead of having an authority decide how much new money is created, this decision is fixed upfront and known to everyone beforehand. And instead of having an authority receive the newly created money and decide what to do with it, we *randomly* allocate this new money to the lucky node who happened to be the successful miner of the block.

The second part of the value of the coinbase transaction output consists of the *transaction fees*. Revisiting the Weak Conservation Law, remember that it's possible that some transactions can have more input value than output value, because we only demanded that

$$\sum_{i \in \mathsf{tx.ins}} i.v \geq \sum_{o \in \mathsf{tx.outs}} o.v\,.$$

If this inequality is strict, then there is more input value than output value, as illustrated in Figure 5.9. That difference in value is not lost, but the miner is allowed to reclaim it as part of their coinbase value.

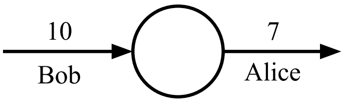


Figure 5.8: A transaction with an input of 10 and an output of 7 pays a fee of 3 units that go to the coinbase output.

Overall, the coinbase transaction is allowed to include up to a value which is the sum of the block reward plus the transaction fees of all the transactions. The total amount of fees incurred in a block B will be

$$f_f(B) = \sum_{\substack{\mathsf{tx} \in B.\overline{x} \\ \text{tx non-coinbase}}} \sum_{i \in \mathsf{tx.ins}} i.v - \sum_{o \in \mathsf{tx.outs}} o.v\,.$$

The coinbase output value is allowed to be up to the total $f_t = f_r + f_f(B)$ which includes both the reward and the fees. Typically, the miner will claim the maximum allowed value in the coinbase. In case the miner chooses not to claim the full value, that money is *burned* and does not belong to anyone.

To verify a coinbase transaction, we follow these steps:

1. Check that is is the only one in that block.

2. Check that it is the first transaction in the block.

3. Check that it has no input and exactly one output.

4. Check that its output has a value which is less than or equal to the sum of the block reward and the difference between the input and output amounts of all *other* transactions in the block.

5. Check that the coinbase is not spent in the same block.

The last condition is known as the coinbase *maturation* condition, and sometimes a longer waiting time than one block is required.

One more technical detail is required in the data that is included in the coinbase transaction. Because the miner in two different blocks can be the same, and the value f_t in two different blocks may also be the same, all the data of two different coinbase transactions can be identical. If we didn't take any action to prevent this, their txid would be identical as well. However, we would still like to distinguish the two different coinbase transactions of different blocks, so that they can be spent independently. It is therefore important to include some extra metadata to distinguish one coinbase transaction from another. One such piece of metadata that can be included is the block height of the block the coinbase transaction is included in.

With coinbase and regular transactions, we've concluded both the publicly verifiable *creation* and *transfer* of money in a way that respects scarcity, completing our monetary design.

5.6 Block Validation

So far, we described how blocks are mined, but we have not given the block validation algorithm. This algorithm is straightforward and consists of the following steps. Whenever an incoming block $B = s \parallel \overline{x} \parallel \mathsf{ctr}$ arrives:

1. Validate the proof-of-work $H(B) \leq T$.
2. Use s to locate its parent block and ensure it is valid recursively.
3. Validate the transactions $\overline{x}$.

To avoid spam, only valid blocks are gossiped by honest nodes. As it is moderately difficult to create proof-of-work, this is validated first to avoid spam attacks. In case s points to a block that is not known to the node, this is downloaded and validated recursively. Similarly if $\overline{x}$ refers to any transactions that are unknown, these are downloaded from the network to validate the block.

The validation of transactions $\overline{x}$ in a block works as follows. For each block B, we remember a UTXO set which we associate with the state $\mathsf{st}(B)$ of the world *after* the particular block has been adopted. We also define a *genesis state* st_0, the state of the world *before* any transaction ever took place. The genesis state, which is where the UTXO set begins its lifetime, is defined to be the empty set.

To validate the transactions in a block $B = s \parallel \overline{x} \parallel \mathsf{ctr}$, we take the state $\mathsf{st}_{B'}$ associated with its parent block B' whose blockid is $s = H(B')$, or st_0 if $B = \mathcal{G}$. We then attempt to apply each of the transactions in the block, in the order they are defined in $\overline{x}$, updating the UTXO set as we go along by consuming and producing elements in the set, and validating every transaction along the way, in the manner we already discussed in Chapter 3. If at any point the transaction validation fails, the whole block is rejected, even if just one transaction was invalid. If all transaction validations succeeded, we assign the state st_B after our block to be the UTXO set we arrived at after this process is completed. This is illustrated in Figure **??**. In this example, $B.\overline{x} = (\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_4)$.

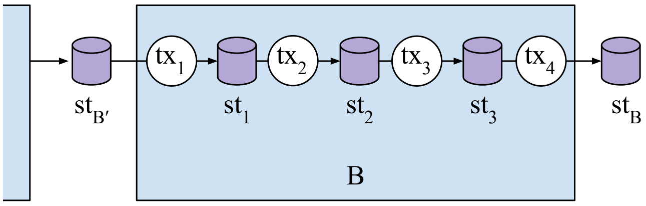


Figure 5.9: Validating the $\overline{x}$ part of block B. The intermediate states are shown as purple cylinders. The state $\mathsf{st}_{B'}$ comes from a block which has already been validated (shown partially on the left).

While we will sometimes say that *chains* are exchanged on the network, it is really *blocks* that are sent over the network. Due to the fact that each block contains a reference to its parent, a block uniquely identifies the chain of which it is a tip. When a party wishes to send a chain to another party, it is sufficient that the tip is sent over. The other party can then request the ancestor blocks if he doesn't already have them. In practical protocols, sometimes these requests for objects are bundled together in batches to improve communication complexity. For example, all the transactions of one block can be downloaded in one go instead of requesting each of them independently. Validating a chain equates to validating its tip, as this requires validating the parent block recursively. The genesis block is hard-coded into every full node and is considered valid by definition. This is where the recursion stops.

5.7 Maintaining the Mempool

As we have already discussed in the previous chapter, each node maintains a *mempool* with transactions in limbo, waiting to be confirmed into a block. A *mempool state* is maintained pertaining to the current mempool. This state is where newly arriving unconfirmed transactions are checked against for validity. When a new transaction arrives from the network, after it is validated, this mempool state is updated to reflect the consumption and creation of elements in the UTXO set. If the miner is able to get a block, the mempool transactions make it into the block and the mempool is emptied. At this point, the mempool state becomes equal to the state of the newly mined block. However, if a *different* party successfully mines a block, we have to be a little more careful about updating our mempool and the mempool state, as our mempool may not exactly match the transactions that were received in the newly arriving block.

Consider the case where a party P_1 has some chain $\mathcal{C}$ and party P_2 mines a new block B on top of $\mathcal{C}$. As usual, the block B is validated with respect to the state $\mathsf{st}(\mathcal{C})$ of $\mathcal{C}[-1]$. After validation, we have calculated the state $\mathsf{st}(B)$ of B. Now, P_1 calculates the *new* mempool $\overline{x}'$ as follows. He first sets the mempool state to be equal to the state $\mathsf{st}(B)$ of the latest block. He looks at the transactions $\overline{x}$ in his *old* mempool and processes them one by one. He attempts to apply each of these transactions $\mathsf{tx} \in \overline{x}$ in the order that they appeared in his old mempool on top of $\mathsf{st}(B)$, updating the mempool state every time a transaction is successfully applied. If a transaction from the old mempool cannot be applied, that transaction is thrown away and the mempool state remains unaffected. The transactions that are applied successfully make it to the new mempool. This process is like pretending that each of the old mempool transactions appeared on the network in order after block B was processed. If block B contains a transaction that was in the old mempool (which will typically be the case), then this transaction has already been applied in $\mathsf{st}(B)$, and so will not make it into the new mempool, because it is spending already spent outputs. If block B contains a transaction that is conflicting with a transaction in the old mempool, then the transaction in the old mempool will also be thrown out, because it is conflicting with the transaction in B that has already been applied to $\mathsf{st}(B)$. As such, if two mutually double spending transactions appear in a block and in the mempool, the transaction in the block takes precedence.

The situation becomes slightly more complicated when there is a reorg. In that case, the state is rolled back to after the latest common ancestor between the old canonical chain and the new reorged chain, and state transitions are applied from that point onwards. As for the new mempool, it is reconstructed by attempting to apply first all the transactions in the abandoned fork, and then the transactions in the old mempool.

This algorithm will be made clearer with an example. Consider the situation illustrated in Figure 5.10, and suppose our party has adopted the chain whose tip is B'_2 (bottom) and has a mempool of $\overline{x}'$. Suddenly, block B_3 arrives. The party downloads the chunk (B_1, B_2, B_3) and validates it. As before, this validation begins by looking at the state of the latest common ancestor B. This has the effect that all the transactions in B'_1 and B'_2 are undone prior to applying the transactions in B_1. First, the transactions in B_1 are applied and $\mathsf{st}(B_1)$ is calculated. Then B_2 is applied and $\mathsf{st}(B_2)$ is calculated. Lastly, B_3 is applied and $\mathsf{st}(B_3)$ is calculated. If at any point a transaction cannot be applied, the whole block containing it is rejected.

Now suppose that the newly arriving chain was valid. Since B_3 has a higher height than B'_2, the chain ending in B_3 is adopted by the longest chain rule. At this time, the party wants to compute his new mempool $\overline{x}$. This is done as follows. The party begins by setting the mempool state to $\mathsf{st}(B_3)$. First, all the transactions $B'_1.\overline{x}$ are trialed for placement into the new mempool in the same order they appear within $B'_1.\overline{x}$. Similarly, the transactions in $B'_2.\overline{x}$ are trialed for placement into the new mempool. Lastly, the transactions in the old mempool $\overline{x}'$ are trialed for placement into $\overline{x}$. Every time a transaction can be applied successfully, it is added to the new mempool, and the mempool state is updated. In case a transaction is unapplicable, the transaction is thrown away and the mempool state remains unaffected.

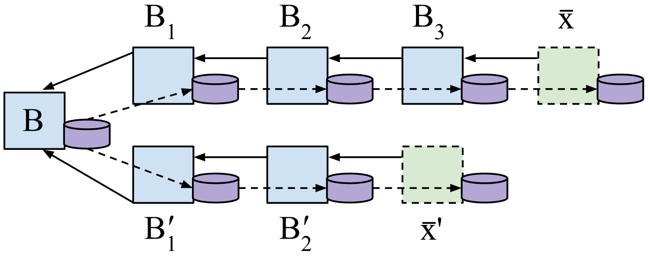


Figure 5.10: The state and mempool calculation in the case of a reorg. Ancestry relations are shown in solid arrows, whereas state updates are shown in dashed arrows. Blocks are depicted as blue solid bloxes; states are depicted as purple cylinders; mempools are depicted as dashed green boxes.

This has the effect that, after a reorg, between any transactions that have a double spend in both the abandoned fork and the newly adopted chain, the transaction in the new chain takes precedence, whereas the double spend within the abandoned fork is also abandoned (and is not even placed in the new mempool). This illustrates why reorgs can be dangerous: Money that is already considered *confirmed* can be rolled back, and a double spend can later become confirmed in its stead.

5.8 Problems

TBD

5.9 Further Reading

TBD

Chapter 6

Chain Virtues

Now that we have developed the longest chain protocol and understand what constitutes a chain, let's familiarize ourselves with executions of this protocol, in honest and adversarial settings, by looking at the combinatorial properties of chains.

6.1 Transactions – Blocks

Before we get into the combinatorial properties of chains, let us spend a moment comparing *transactions* (Chapter 3) and *blocks* (Chapters 4 and 5), to ensure that we don't confuse the two and to built a high-level mental model of the previous couple of chapters.

Recall from Chapter 3 that chain of transactions is what constitutes a *coin* (although this is not a formal notion, since coin provenance may not be uniquely determinable because coins can be joined and split through transactions). Anyone can create a transaction spending their own money instantaneously, because they can easily create a signature. Long chains of such transactions can be created rapidly from a low-end device, such as a mobile phone, with no significant mining power required. On the other hand, no one can create a transaction that spends someone else's money, because they don't hold the respective private key, no matter how much computational power they have (within the bounds of our polynomial world). A chain of transaction conveys no information about time taken.

As for blocks, creating them does not require producing any signatures, but does require having a lot of computational power, and cannot be performed on a laptop or mobile phone.

A summary comparing blocks and transactions is shown in Table . *Time to create honest* describes the time needed to create a transaction spending my own money (including double spending it) and the time needed for an honest party to mine an honest block. *Time to create adversarial* describes the time needed to create a transaction spending someone else's money and the time needed for an adversarial party to mine an adversarial block (for example containing a double spend). The last few rows summarize the inductive process of validating a transaction and validating a block. For transaction validation, the inductive base is the coinbase transaction, where money is generated and the Conservation Law does not need to be verified, and neither do they need signatures. The inductive hypothesis there is that we have a valid UTXO set, and the inductive step involves updating

	Transaction	**Block**
Time to create honest	Instantaneous	Moderately hard
Time to create adversarial	Exponential	Moderately hard
Inductive base	Coinbase	$\mathcal{G}$
Inductive hypothesis	Outpoint UTXO	Previous blockid (s)
Inductive step	Update UTXO set Check σ Conservation	$H(B) \leq T$ Validate $\overline{x}$

Table 6.1: A high-level comparison of the nature of transactions and blocks.

the UTXO set by consuming and producing elements, checking transaction signatures, and verifying the law of conservation. For block validation, the inductive base is the genesis block, whose previd does not need to be checked. The inductive hypothesis is that the previous chain up to s has been validated, and the inductive step involves checking the proof-of-work equation and validating the transactions within.

Given this table, we can observe what protects a transaction in transit from being altered. Imagine the adversary sitting in the middle of a gossip path on the network. This adversary relays transactions and can modify them as they travel from one end of the network to the other. If the adversary attempts to change a transaction's data (such as the recipient), she will fail. The reason of failure is different for a normal transaction and a coinbase transaction. When it comes to a normal transaction, the polynomial adversary who attempts to change the recipient will fail because of the existential unforgeability of the signature scheme. When it comes to a coinbase transaction, these transactions are not signed, as they do not have any inputs. However, because the coinbase transaction is always associated with a block, changing the coinbase transaction invalidates the proof-of-work of the block the coinbase transaction is included in. A coinbase transaction is never valid unless it accompanies a block. The adversary would therefore need to solve the proof-of-work puzzle from scratch in order to usurp this payment. While this is not impossible, it is moderately difficult.

6.2 Safety, Revisited

When we gave the first intuitive definition of ledger safety in Definition 15, we mandated that all honest parties agree *exactly* on their ledgers. Now that we've talked about network delays and arranged transactions into blocks, we see that this will be impossible to achieve. In the best case scenario, we have a chain that is growing without any forks. Still, in this case, if two honest parties P_1, P_2 both have a chain $\mathcal{C}$, whenever P_1 mines a block B on top of $\mathcal{C}$, it will take Δ time until P_2 receives this block and updates his ledger. Remember that the reported ledger of a party only includes the transactions that have been confirmed into blocks.

We need to revise our safety definition to account for the fact that there can be discrepancies. We will accept that ledgers of honest parties may disagree. However, we want the ledgers to be *consistent* with one another: If one honest party has reported a transaction at position i in his ledger, then every other honest party must either have that same transaction at position i, or their ledger must be shorter than i, indicating that the party has not yet decided which transaction to place at that location. However, it is imperative that two different honest parties do not have different transactions at the same location. In different words, we want the ledger of honest party P_1 and honest party P_2 to be *prefixes* of one another, even if they are observed at different points in time. Let's revise our *ledger safety* virtue to state it formally.

Definition 21 (Safety)**.** *A protocol is* safe *if for any two honest parties P_1, P_2 and any two times r_1, r_2, it holds that either $L^{P_1}_{r_1} \preccurlyeq L^{P_2}_{r_2}$ or $L^{P_2}_{r_2} \preccurlyeq L^{P_1}_{r_1}$.*

The notation $A \preccurlyeq B$ between two finite sequences A and B means that $B[:|A|] = A$. An example of safe ledgers is illustrated in Figure 6.1. The image shows the ledgers reported by all honest parties at potentially different points in time. The same transaction is illustrated as a circle of the same color on different ledgers. Not all parties have seen all transactions yet: Party P_2 has seen more transactions than any other party, while P_5 has seen the fewest. At position 8, only P_2 has seen the white transaction, and every other party has not. However, if a transaction ever appears in position 8 of any other honest party (illustrated as a dashed-line ghost transaction), it will be the same exact transaction that party P_2 has seen at that position.

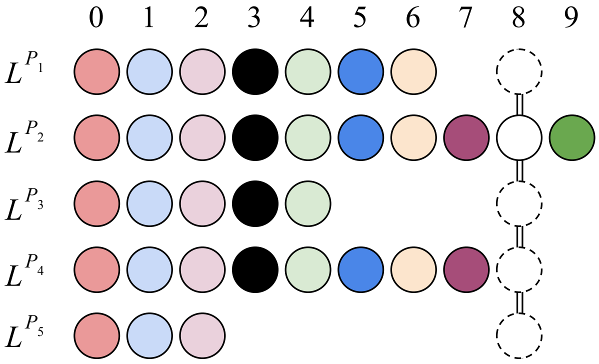


Figure 6.1: A god's eye view of the *safe* ledgers of various honest parties at various moments in time.

Thinking back to the UTXO model, safety means that the transaction graph of one party is outdated, but not inconsistent, as compared to the other party. In particular, if one party has confirmed a transaction tx, then the other party cannot have confirmed, and can never confirm, a transaction tx' which is a double spend of tx. Let's try to understand why.

Lemma 5 (Safe ⇒ No double spend (informal))**.** *In a safe protocol that ensures transaction validity locally, two honest parties can never confirm two conflicting transactions.*

Proof. Suppose, towards a contradiction, that party P_1 at time r_1 has confirmed tx. This means that tx appears in P_1's ledger $L^{P_1}_{r_1}$ at time r_1, and let's call i the position of the transaction in the ledger: $\mathsf{tx} = L^{P_1}_{r_1}[i]$. Additionally, party P_2 at time r_2 has confirmed tx', a conflicting transaction to tx. Similarly, let j be the index: $\mathsf{tx}' = L^{P_2}_{r_2}[j]$. Now, if $i = j$, then this means that $\mathsf{tx} = \mathsf{tx}'$, which is a contradiction since these are conflicting. If $i < j$, then by safety this means that $L^{P_2}_{r_2}[i] = L^{P_1}_{r_1}[i] = \mathsf{tx}$. But then, party P_2 has included in his ledger *both* tx at position i *and* tx' at position j. This is a contradiction, because the honest party validates tx' before accepting it. Therefore, the two parties cannot have accepted conflicting transactions. □

This is a good sanity check that our definition of safety makes sense. Is this revised and precise, yet weaker, notion of ledger safety achieved by our *longest chain* construction? Not quite. The problem is that, as we saw in the previous chapter, the chain may have *temporary forks* even when no adversary is mining. Whenever the chain *reorgs*, there is a potential for safety loss. Consider the case illustrated in Figure 6.2. Imagine that honest party P_1 had initially adopted the chain whose tip is block 6 and contains the top (white) transaction tx. In the meantime party P_2, who has not seen block 6 yet, is still mining on top of block 4. If party P_2 successfully mines block 5, he can include a double spending transaction in it, the bottom (black) transaction tx', conflicting with tx. Now the two chains at blocks 5 and 6 are at a tie. The tie will be broken when the next proof-of-work is found (block 7) and one branch becomes longer; the nodes that were working on the other (block 6) branch will then switch to the longer one. If we now compare the ledger reported by party P_1 when he had adopted block 6 and the ledger reported by party P_2 when he had block 7, we see that the ledgers obtained by *reading* these two chains are mutually unsafe.

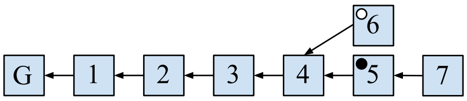


Figure 6.2: A chain reorg during a temporary fork causes a safety loss.

We will have to revise our *reading rule* to account for these temporary forks.

6.3 Honest Convergence

We now argue that, in a correctly parametrized system, temporary forks will generally be short. Let us initially analyze this claim in the setting where every miner is honest.

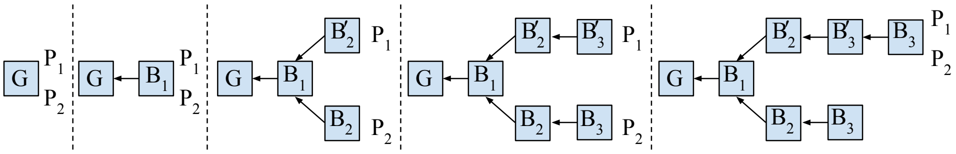


Figure 6.3: The timeline of two honest parties P_1 and P_2 mining in a way that causes the honest population to temporarily split between two factions.

Before we give a proof of this, let us think through an example of chain divergence and convergence. It is illustrated in Figure 6.3. Initially, all honest parties are mining on top of genesis $\mathcal{G}$. If it happens that an honest party gets a block B_1, and broadcasts it to the rest of the network, this block will arrive at the doorstep of every other honest party within time Δ. If no other block was mined during that time, now all honest parties will switch to B_1. Convergence is maintained. This can continue happening, and the chain will keep growing, with all parties agreeing on one tip (with potentially a small delay). Now, the divergence case can happen when two honest parties mine a block almost simultaneously: First, one honest party mines a block B_2 on top of block B_1 and broadcasts it to the network. But before Δ time has elapsed, a second honest party mines block B'_2. The block B'_2 will also be on top of block B_1, as the second party has not received B_2 yet. At this point, two different chains, with tips B_2 and B'_2 respectively, exist and have the same length. Now the honest mining power is split into two factions: Some honest nodes have seen B_2 first, whereas other honest nodes have seen B'_2 first, so some honest nodes are trying to extend B_2 and some others are trying to extend B'_2. These factions may hold different, or similar, portions of compute. Now it's possible that an honest party mines another block B_3 extending B_2. The party broadcasts it to the network, but, before Δ time has elapsed, another party, who was mining on top of B'_2, also mines block B'_3 which extends B'_2. Again the honest parties are split between two different chains of the same length. This situation can continue. At some point, an honest party will mine a block B and broadcast it to the network, but no other block will be mined within Δ time. At this point, every honest party will switch to block B. This is the convergence opportunity.

The idea is that, if we choose T appropriately, then convergence opportunities will happen regularly. Once a convergence opportunity happens, if there is no adversarial miner, the honest parties will converge into one chain.

Lemma 6 (Honest Convergence)**.** *In a setting where only honest parties are mining, exactly Δ time after every convergence opportunity, all honest parties will hold the same chain.*

Proof. Consider an arbitrary convergence opportunity producing block B occurring at time r. Since the convergence opportunity is Δ separated from every *preceding* successful query, and all successful queries are honest and broadcast to the network, all honest parties will have seen all the blocks produced so far (because they all happen prior to $r-\Delta$ and it takes at most Δ time to receive the message containing those blocks). Therefore, the honest party who mined B will be mining on top of the longest chain among all. This means that, at time r, the block B has a height larger than any other block. Because the miner is honest, he broadcasts B, and it is received by all honest parties within Δ time. Since the convergence opportunity is Δ separated from every *succeeding* successful query, no other blocks are mined or during the time interval $r \dots r+\Delta$. At time $r+\Delta$, every honest party has received B and adopted it since it is the longest chain. At this point, all the honest parties agree on their chains. □

In other words, convergence opportunities really do lead to convergence in honest executions. But will convergence opportunities happen often, as we are hoping? This depends on the choice of parameter T, which affects $p = \frac{T}{2^\kappa}$, and it should be tuned based on our best guess for n, q, Δ we have for the network. Let us visualize the effect that the expected block production rate nqp (if all n parties are actively

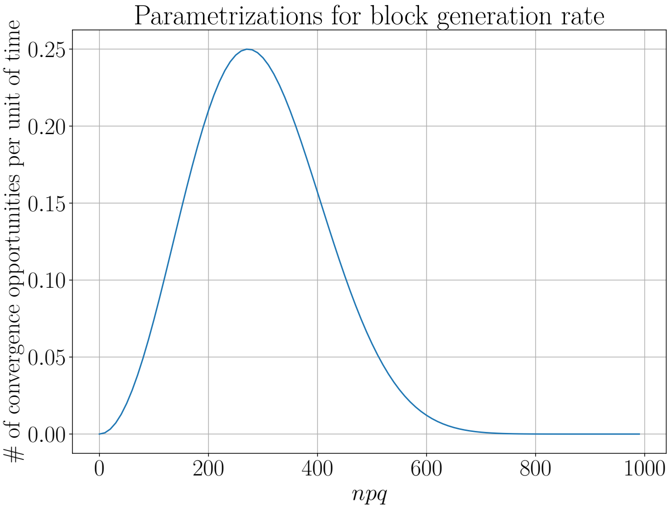


Figure 6.4: The relationship between the expected block production rate nqp and the density of convergence opportunities.

mining), of which we can tweak p, has on the density of convergence opportunities in the unit of time. Their relationship is visualized in Figure 6.4. As you can see in this graph, there's a sweet spot for the expected block production rate that maximizes convergence opportunities. If we make block production rate too slow (left side of the plot), convergence opportunity density in the unit of time drops. The reason is that, while almost all successful queries are convergence opportunities, they are very rare in time. On the other hand, if we make block production rate too quick (right side of the plot), while successful queries are dense in time, convergence opportunities are rare, because successful queries clash with one another and there is no sufficient Δ separation between them. It is our task to design a system that *always* remains within the realm of good density of convergence opportunities. A system that produces blocks very fast may appear to have a high throughput of “transactions per second”, but this misconfiguration can be leading to insecurity, because throughput has been increased to the detriment of convergence opportunity density. Any claims of blockchain performance must always be accompanied by how the parametrization that achieves that performance affects the security of the network.

6.4 Common Prefix

Under a correct parametrization of the system, we have now developed some intuition that temporary forks will be short, at least as long as everyone is honest. These forks may develop some length, but soon enough, a convergence opportunity will arise. At that point, all forks apart from one will be abandoned, and the honest parties will convergence on one chain. It takes time to mine a temporary fork of longer length. Because convergence opportunities will occur often (we will calculate precisely how often in Chapter 12), there is not enough time to build long temporary forks. Our blocktrees will look roughly like Figure 6.5.

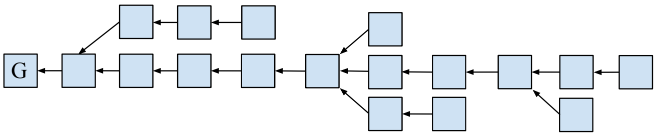


Figure 6.5: An honest execution with some short temporary forks.

We will now define a parameter $k \in \mathbb{N}$ that specifies our belief of how long these temporary forks can get. Our hope is that there will be no temporary fork longer than k. If this is respected, we will be able to ensure safety. We call this virtue of chains the *Common Prefix* virtue.

Definition 22 (Common Prefix)**.** *A system is said to satisfy* Common Prefix *with parameter $k \in \mathbb{N}$ if for all honest parties P_1, P_2 and for all times $r_1 \leq r_2$, the chains adopted by the honest parties satisfy the property that*

$$\mathcal{C}_{r_1}^{P_1}[: -k] \preceq \mathcal{C}_{r_2}^{P_2} .$$

This property is telling us that temporary forks do not extend beyond k blocks long. For example, the Common Prefix virtue holds in Figure 6.5 with parameter $k = 3$, because temporary forks are at most 3 blocks long. The precise statement of the virtue states that, if we take the chains adopted by two honest parties at the same point in time, and we *chop off* k blocks from the end of the chain of the chain of one party, we get a *prefix* of the chain of the other party. In other words, the two honest parties' chains share a long common prefix, and their different suffixes can only be up to k blocks long. This further holds if we take chains of different parties at different points in time. In that case, we have to make sure we chop off blocks from the *old* chain to find that it is a prefix of the *newer* chain. The property also applies when $P_1 = P_2$, in which case it is telling us that a particular honest party cannot perform chain reorgs more than k blocks deep.

If this virtue holds, and we can know a k parameter that bounds all of our temporary forks, we can modify the *read* functionality to achieve safety. Instead of reporting as *confirmed* the transactions that are in our adopted chain $\mathcal{C}$, we report as *confirmed* the transactions that are within $\mathcal{C}[: -k]$, our chain with the last k blocks chopped off. If the Common Prefix virtue holds, we know that the blocks in $\mathcal{C}[: -k]$ will never be abandoned in the future.

Our new, and final, confirmation rule, is illustrated in Algorithm 13. Compare this rule to Algorithm 12 of the previous chapter. The only difference is that we have replaced $\mathcal{C}$ with $\mathcal{C}[: -k]$. The portion $\mathcal{C}[: -k]$ is called the *stable* part of the chain, because we are assured it will not change.

Algorithm 13 The stable *read* rule.

```
function READ_k($\mathcal{C}$)
    $L \leftarrow [\,]$
    for $B \in \mathcal{C}[:-k]$ do
        for tx $\in B.\overline{x}$ do
            $L \leftarrow L \,\|\, \mathsf{tx}$
        end for
    end for
    return $L$
end function
```

We have finally achieved safety. This is captured in the following theorem, whose proof we will complete in Chapter 12.

Theorem 7 (Common Prefix ⇒ Safety (informal))**.** *If the blocktree satisfies Common Prefix with parameter k, then the longest chain protocol with the k-confirmation rule is safe.*

Note the effect of k on the protocol. If we make k too small, then we risk Common Prefix being violated, and safety becoming jeopardized. If we make k too large, then liveness will deteriorate. Similarly to how we must tune the T parameter appropriately to balance safety and liveness, so we must also tune the Common Prefix parameter k.

With this small change to the implementation of *read*, our protocol is finally complete. We have created a secure cryptocurrency. The rest of this book will be devoted to studying the various properties of our construction, augmenting it so that it remains secure in different models, and optimizing it. However, the central construction is done.

6.5 The Nakamoto Race

We now bring Eve, our double spending adversary, back into the picture. Eve has picked up her game. Instead of stealing a cup of coffee, she's now attempting to steal a brand new car! However, she's now up against our k-confirmation protocol and must work a lot harder than before. The attack is pictured in Figure 6.6. Initially, all parties have converged on some block B. Eve first creates two conflicting transactions tx (the white transaction on the bottom chain) and tx′ (the black transaction on the top chain). In this case tx is paying for the brand new car, whereas tx′ is spending the same output and handing the money back to herself. She broadcasts tx to the network, hoping that it will become confirmed by the honest parties and observed by the car dealer so that she can be handed the car. The honest parties receive tx from the network and faithfully include it into their mempool. At this point, Eve starts *secretly* mining on her own on the top (red) fork, while the honest parties are mining on the bottom blue fork. The honest parties do not mine on top of adversarial blocks, because Eve does not broadcast these blocks to the network. Eventually, the honest parties mine a block containing tx (the block on the lower fork containing the white transaction). But the car dealer will not see tx marked as *confirmed* in his wallet before it is buried under k blocks, so Eve must wait. Eve allows the honest parties to keep mining on the bottom chain until k blocks have been mined on top of the transaction and the transaction appears as confirmed. That's when the car dealer hands over the car to Eve. She takes the car and drives away. In the meantime, Eve has been mining on the top fork during for all this time. If at this time she has managed to create a longer chain than what the honest parties have already adopted, she reveal this chain to the network by broadcasting it. Then, every honest party will reorg their chains and switch to Eve's chain. She will take her money back, and she will have gotten away with the car.

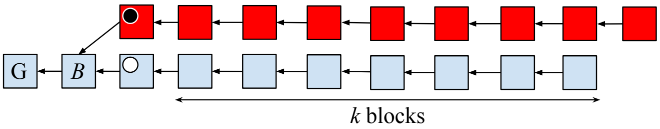


Figure 6.6: The Nakamoto Race.

To see when this attack is successful, we argue as follows. As soon as block B was found and broadcast to the network at some time r, a *race* began between the honest parties and the adversary: the so-called *Nakamoto Race*. This race concludes as soon as the honest parties have mined $k + 1$ blocks on the bottom fork. It is pointless for Eve to broadcast her top fork if the bottom fork does not have $k+1$ blocks in it, because this will simply leave her car-purchasing transaction unconfirmed and Eve carless. Because blockchains take time to produce, it will take time to mine these k blocks. In order for Eve to win she must have mined more blocks than the honest parties within that time window. Whether she is successful in doing this depends on whether she has more mining power than the honest parties or not. If Eve has less mining power than the honest parties ($t < n - t$), then the picture looks somewhat like Figure 6.7. Eve has some successful queries (labelled Z at the bottom), but the density of these successful queries in time is smaller than the density of the honest parties' successful queries (labelled X at the top). While Eve can get lucky for a couple of blocks (in the picture, she gets two successful queries before the honest parties have mined anything at all), given enough time, the honest parties will get more blocks than Eve. If we choose k to be large enough, we can make this time window long enough so that the honest parties will have more successful queries than Eve, and Eve cannot win the race.

If Eve is left behind the honest parties after $k + 1$ honest blocks have been mined, it does not make much sense for her to try and catch up, because she's in an *even worse* situation than she was when she started. Not only must she mine more blocks than the honest parties from now on while they are still mining, but she must also mine some extra blocks to make up lost ground. This gives an intuitive argument of why a minority adversary cannot break k Common Prefix.

On the other hand, if Eve has the upper hand ($t > n - t$), then the picture looks quite differently. The longer we wait, the longer the adversarial fork will become, and it will certainly overtake the honest fork. There is no hope if the adversary has more mining power than the honest parties. This reaffirms our requirement for the honest majority assumption.

However, this picture is slightly incomplete. While we have argued that Eve will mine fewer blocks than the honest parties within the window of time in question, any honestly successful queries that are not convergence opportunities may be wasted

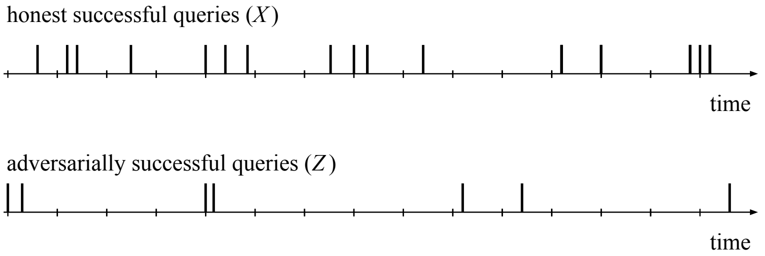

honest successful queries (X)
time
adversarially successful queries (Z)
time

Figure 6.7: Honest *vs* adversarially successful queries with a minority adversary.

into temporary forks. On the contrary, because Eve is a puppetmaster adversary, if she plays her hand right, her queries are coordinated and always lead to one consistent chain without forks. Therefore, it is not the honest successful queries we should be comparing against the adversarially successful queries. Instead, we should be comparing the *convergence opportunities* against the adversarially successful queries. The full picture is illustrated in Figure 6.8. As you can see, even though the honestly successful queries (labelled X at the top) might be more dense than the adversarially successful queries (labelled Z at the bottom) due to honest majority ($t < n - t$), it is still possible that the convergence opportunities (labelled Y in the middle) are fewer than the adversarially successful queries. In order for the honest parties to win the Nakamoto Race, honest majority is not enough! In addition to honest majority, the blockchain must be correctly configured (with an appropriately chosen T) so that the convergence opportunities are denser than the adversarial successes. Of course, because convergence opportunities are always sparser than successful queries (every convergence opportunity is a successful query, but not vice versa), if honest majority does not hold, then there is no hope that convergence opportunities are denser than adversarial successes.

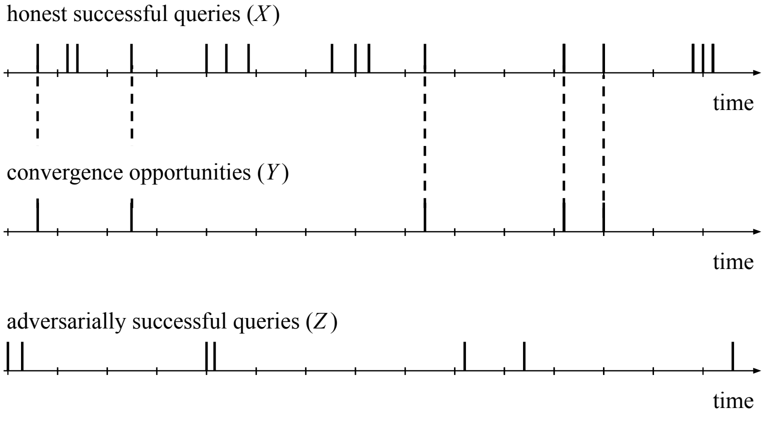

honest successful queries (X)
time
convergence opportunities (Y)
time
adversarially successful queries (Z)
time

Figure 6.8: Convergence opportunities *vs* adversarial queries with honest majority in a misconfigured blockchain.

6.6 The Fan-Out

We just observed that honest majority is not enough for our system to be functional. It is not the honestly successful queries that we must be comparing against adversarially successful queries; rather, it is the convergence opportunities that we must be comparing against adversarially successful queries. A misconfigured system, in which the honest parties have more successful queries than the adversary, but the adversary has more successful queries than the honest parties have convergence opportunities, can lead to a situation of a *fan-out*.

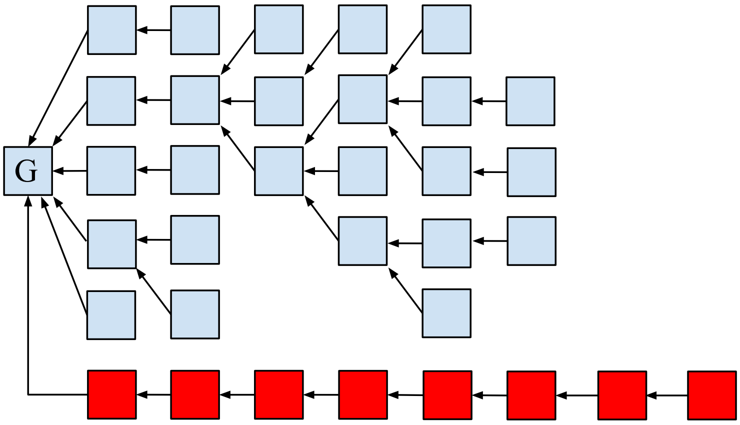


Figure 6.9: A misconfigured blockchain can be subject to fan-outs, even under honest majority.

One such situation is illustrated in Figure 6.9. This system has a target T that is too easy, and so blocks are produced with high probability p, more quickly than we want. As such, the blocktree looks nothing like a blockchain. The honest parties mine the various blue forks on top. The adversary mines the red fork at the bottom, in secret. Even though the adversary has only expended 8 successful queries to build her chain, the honest parties have expended 25 successful queries into building their fan-out structure, whose longest chain is shorter than the adversarially mined chain. The reason why this situation has occurred is because none of the honest successful queries were convergence opportunities. Because the minority adversary can concentrate her mining power into building *one* chain and, contrary to the honest parties, is not subject to the delay Δ (remember that we assume she controls all of her minions from one central puppetmaster), she can win the race against the honest parties even though they control the majority of compute. Because all but one of the temporary forks among the honestly mined ones will eventually be thrown out when a convergence opportunity occurs, all this honest mining power is going to waste.

6.7 Chain Quality and Chain Growth

We spoke about how Common Prefix gives rise to *safety*. Just having our chains have common prefix does not give *liveness*, however. It is possible that the chains have simply stopped growing, and no new transactions are ever making it in. In order for liveness to hold, we need honest transactions to make it into the stable chains of honest parties. Towards this, we will hope our chains will have two more virtues.

On the one hand, we want honest chains to grow. If chains do not grow at all, then there is no hope of new transactions making it in. As long as our chain grows at a positive rate, however, we will have some liveness. The rate at which the chain of an honest party grows over a period of time if the chain velocity.

Definition 23 (Chain Velocity)**.** *Consider an honest party and his chain C_1, C_2 at times $r_1 < r_2$ respectively. We say the chain has grown with a* velocity $\tau = \frac{|C_2|-|C_1|}{|\mathcal{C}_{r_2}|-|\mathcal{C}_{r_1}|}$ *between those times.*

In order to achieve liveness, we will hope that our chain has *chain growth*: Given enough time, the chain grows with a minimum velocity τ.

On the other hand, just because the chain has grown does not mean that any honest transactions are getting confirmed: If the chain has only grown by adversarially mined blocks, it is possible that the adversary has simply been ignoring the honestly generated transactions. In order to get ledger liveness, we hope to occassionally see an honestly mined block in every sufficiently long chain. We call this *chain quality.*

Definition 24 (Chain Quality)**.** *Consider the chunk $\mathcal{C}[i{:}j]$ of an honest party's adopted chain $\mathcal{C}$. We say that the chunk has chain quality $\mu = \frac{z}{|\mathcal{C}[i:j]|}$, where z is the number of honest blocks in $\mathcal{C}[i{:}j]$.*

In other words, the chain quality metric defines the *proportion* of honest blocks within an honestly adopted chain. The *chain quality* virtue demands that sufficient long chain chunks of honest parties have a minimum quality μ. An example is illustrated in Figure 6.10. This is an honestly adopted chain $\mathcal{C}$ which contains both honestly and adversarially generated blocks. Blocks $1, 2, 5, 7, 8$ are adversarially generated, and blocks $\mathcal{G}, 3, 4, 6$ are honestly generated. This chain has quality $\mu = \frac{4}{9}$. If we just focus on the chunk $\mathcal{C}[3{:}6]$, the quality of this chunk is $\mu = \frac{2}{3}$.

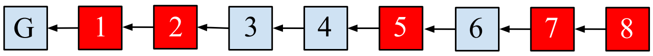


Figure 6.10: An honesty adopted chain with both honestly and adversarially generated blocks. Adversarial blocks are numbered $1, 2, 5, 7, 8$ and shown in red.

Liveness follows from chain growth and chain quality. Liveness mandates that an honestly generated transaction is confirmed "soon". When the honest transaction is broadcast into the network, it makes it into every honest party's mempool (whereas the adversary can ignore it). As soon as any honest block is mined, the transaction will make it into the block if it has not been included in a previous block. Due to chain growth, the chain will grow at a minimum rate of $\tau > 0$. When the chain grows enough, the fresh chain chunk will have a minimum quality $\mu > 0$, and so will contain at least one honest block. That block will contain the transaction in question. This is captured in the following theorem:

Theorem 8 (Chain Growth and Chain Quality ⇒ Liveness (informal))**.** *If the blocktree satisfies Chain Growth and Chain Quality with positive parameters τ and μ respectively, then the longest chain protocol is live.*

Notice here the Common Prefix parameter k is a part of our *construction* and is a value hard-coded into our implementation of the *read* functionality. If we choose this value wisely, we can prove that the Common Prefix virtue with parameter k is respected. The τ and μ parameters are not part of our construction. We will simply calculate these values when we prove that our chains attain all three virtues.

In summary, chain virtues give rise to ledger virtues. Safety follows from Common Prefix and liveness follows from Chain Growth and Chain Quality together.

6.8 Further Reading

The two chain virtues of Common Prefix and Chain Quality were first defined in the Bitcoin Backbone paper [14] by Juan Garay, Aggelos Kiayias and Nikos Leonardos. The Chain Growth virtue was identified in a later paper [22] by Aggelos Kiayias and Giorgos Panagiotakos. The ledger virtues of safety and liveness have been adapted, with appropriate modification, from older literature from the 1980s and 1990s in the context of two problems known as the Byzantine Generals Problem and the Byzantine Agreement Problem. These two problems are very related to the type of problem that blockchains are trying to solve. The Byzantine Generals Problem was originally studied in a seminal 1982 paper [23] by Leslie Lamport, Robert Shostak, and Marshall Pease.

6.9 Problems

TBD

Chapter 7

Attacks

In the previous chapter, we saw two basic attacks: the Fan-Out and the Nakamoto Race. The Fan-Out is more of a misconfigured network situation, because no mining adversary is required to mount it. The Nakamoto Race is a foundational attack. It will function as a prototype for other attacks which we will explore in this chapter.

In this chapter, we will talk about how the various ledger virtues we defined in the last chapter can be broken by adversaries of varying mining power. Because ledger virtues follow from chain virtues, the adversary will have to break the chain virtues first. So we will look at *examples* of attacks that break the chain virtues. Remember, if we describe an attack that works, this is sufficient to illustrate that the system is broken. However, if an attack we think about happens *not* to work, this is not sufficient to argue that the system is secure. We may have just happened to think of the wrong attack.

7.1 Attacks of a Minor Adversary

A minor adversary is an adversary for which $t < n - t$. This is also known as a $< 50\%$ attack. What virtues can such an adversary break? Let us go over all the virtues in question and explore how much an adversary can break them.

Common Prefix

As we saw in the last chapter, it is possible to have *good* or *bad* configurations of the system if the parameters T and k are set incorrectly when the Δ, t, n, q that exist on the network are taken into account. For example, consider the extreme value $T = 2^\kappa$. Then every query is a successful query, and every successful query failed to be a convergence opportunity. Then, if there are multiple honest parties $n - t > 1$, it doesn't make a difference how much we wait for k; we will never have convergence. As such, Common Prefix is not achieved, and therefore safety is violated. In these very misconfigured networks, the adversary can cause a Common Prefix violation even with no mining power, just by allowing the honest parties to diverge on their own.

However, in proper blockchain protocol configurations, the density of convergence opportunities will be more than the density of adversarially successful queries. In these cases, *the best attack that a minor adversary can do is to perform the*

Nakamoto Attack [11]. As we've seen, using this attack she cannot break Common Prefix, except with negligible probability in k. The more we allow k to grow, the more the actual successful queries of the adversary and the actual convergence opportunities will tend to approximate their expectations. We will make this argument precise when we prove the Common Prefix against *all* adversaries in Chapter 13.

Chain Growth

Even though a minor adversary can affect the chain velocity τ of honest parties by stopping her mining, she cannot prevent the rest of the honest network from mining. If she is a minority adversary, this can mean she can drop the expected chain growth rate to at worse $\frac{1}{2}$ of what it would be were she to mine honestly. *The best strategy for harming chain growth is simply not to mine.* Even if the adversary mines withheld chains in the fashion that the Nakamoto Race attack does, these chains will not be adopted, unless they are longer than what the honest parties have adopted. In these cases, even though the honest parties are switching from one chain to another, the chain they are switching to is still longer than what they have, and so Chain Growth is happening.

Chain Quality

For a minor adversary to harm Chain Quality, she would have to try and supress as many honestly generated blocks as possible from becoming adopted by honest parties. Ideally, she wants honest parties to adopt chains that only consist of adversarially generated blocks.

In case the adversary plays honestly, and no forks occur on the chain, then the number of blocks she gets is proportional to her mining power. The honest adversary[1] therefore achieves quality $\mu = \frac{n-t}{n}$. This matches our expectation that the proportion of blocks on a chain generated by a particular party match his mining power.

7.2 Selfish Mining

We might try to think about different strategies in which the adversary may try to harm quality. There is an adversarial strategy known as *Selfish Mining* which is a modified version of the Nakamoto Race, and specifically targets the minimization of the quality of the honestly adopted chain.

The adversary works as follows. Initially, all honest parties start at some block B. Similar to the Nakamoto Race, the honest parties and the adversary mine on separate forks, both extending B. If at any point of time the adversary finds a block on her fork, she keeps it secret, similarly to the Nakamoto Race. She can keep growing her secret fork more and more. If at any point in time the honest parties find a block, then the adversary wishes to suppress it from the honestly adopted chain. There are two cases: Either she has at least one withheld block up her sleeve on her secret fork, or she does not. If, when the honest parties mine a block, the adversary does not have a withheld block up her sleeve, then there

[1] The *honest adversary* is not a misnomer. Remember that the adversary controls any party which we did not designate as honest. She can do whatever she wishes with that party – including playing honestly.

isn't much she can do. Since she doesn't have any secret blocks, she accepts that the honestly mined block that was just broadcast will be accepted as part of the honestly adopted chain, and restarts mining her secret fork on top of the honestly generated block.

On the contrary, if she has one or more withheld blocks up her sleeve, she releases exactly *one* of her withheld blocks in an attempt to kick out the block produced by the honest party from the honestly adopted chain. If the adversary's block makes it to the doorstep of other honest parties before the honestly mined block arrives, the adversarially generated block will be adopted, and the honestly generated block will be rejected. The adversary keeps the rest of her withheld blocks secret so that they can be used in the future.

Whether the adversarial block can make it to the doorstep of an honest party before the honest block makes it there is an issue of how well positioned the adversary is on the network. Remembering our threat model, we want to make the adversary as powerful as possible, so that, when the time comes to prove the security of our protocol, the proof of security will be the strongest. As such, we will assume that the adversary has significantly better network presence than the honest parties. We model this as follows. Whenever any honest party broadcasts a message to the network, the adversary gets a chance to see it *first*, prior to any other honest party seeing it. The adversary then has the choice to broadcast her own message *prior* to the honest message making its way. In this way, the adversary can *rush ahead* in her message delivery as compared to the honest message delivery. She can *frontrun* honest messages after seeing them so that adversarial messages get delivered prior to the honest message. We call such an adversary a *Rushing Adversary.* This may seem like giving too much power to the adversary, but is not an unreasonable assumption in practice. When we deploy our real systems, we want to give assurances to honest parties even if they have very basic network connectivity, for example through a slow Internet connection in a remote area. On the contrary, it is rather inexpensive for the adversary to buy good network presence by deploying some servers on central Internet backbone locations. This would allow the adversary to actually mount rushing attacks against parties located on the Internet's last mile quite successfully.

The basic idea of the selfish ming attack is that, whenever the adversary has a headstart of secretly kept blocks, any honest mining power goes to waste, because any blocks produced by the honest parties during those moments will be kicked out by the competing blocks broadcast by the rushing adversary.

This is a more complicated attack than what we've seen before, so let's look at an example execution of the attack. The example is illustrated in Figure 7.1. The honestly generated blocks are colored blue, while the adversarially generated blocks are colored red. The final honestly adopted chain is displayed at the bottom and consists of a mixture of honestly generated and adversarially generated blocks, whereas abandoned honest blocks are seen at the top as temporary forks. The blocks are numbered in the order they were mined. As you can see, adversarially mined blocks are never abandoned. The only blocks ever abandoned are honestly mined. Additionally, all temporary forks have a length of exactly 1. The example begins with everyone having genesis. Initially, both the adversary and the honest parties attempt to mine a block on top of genesis. An honest party manages to get a block first, and this is block 1. As the adversary does not hold any withheld blocks, she adopts block 1, and so does every other honest party. Now the honest parties and the adversary all mine on top of block 1. At this point, the adversary manages to get block 2, but keeps it secret. Now the adversary is mining on top of block 2, but the honest parties are mining on top of block 1. At this point, the adversary gets lucky and manages to mine another block 3 on top of block 2. She keeps both block 2 and block 3 secret. Now the adversary is mining on top of block 3, but the honest parties are still mining on top of block 1. An honest party manages to get block 4 and broadcasts it to the network. The adversary sees this block being broadcast on the network and rushes to send her own block 2 in order to suppress honest block 4. Since the adversary is rushing, she manages to get block 2 on the doorstep of every honest party prior to them seeng block 4. Therefore every honest party (beyond the one who mined block 4) now adopts block 2 and keeps mining on top of it. Note how the honest mining power that went into mining block 4 went completely to waste here, because the adversary knew ahead of time that she would be able to replace this block. Now every honest party is mining on top of block 2, but the adversary is still ahead and mining on top of block 3. The honest parties generate block 5, but the adversary counters by releasing block 3. The adversary maintains her lead by finding block 6, and broadcasts it upon seeing block 7. At this point, both the honest parties and the adversary have adopted and are mining on top of block 6, and the adversary has no headstart any more.

The honest parties manage to get block 8, which the adversary is forced to accept, since she has no withheld blocks at this time. Next, she gets 9 and keeps it secret. The honest parties get 10, but the adversary uses 9 to suppress it. At this point, both the honest parties and the adversary are mining on 9. The adversary gets 11. The honest parties get 12 and the adversary suppresses it with 11. The honest parties get 13, which the adversary accepts. Next, the adversary gets 14. The honest parties get 15, and the adversary suppresses it with 14. Lastly, the honest parties get 16 and everybody adopts it.

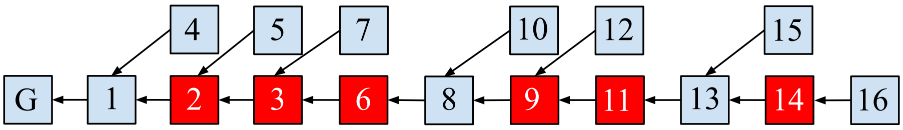


Figure 7.1: The selfish mining attack in action. Red blocks (2, 3, 6, 9, 12, 14) are adversarially generated, whereas blue blocks are honestly generated.

We wish to discover how low the chain quality μ gets for various values of $\frac{t}{n}$. This attack seems harder to analyze analytically, so we will use a simulation method to analyze its effectiveness. In the simulation, we will code a simplified model of the attack in which the honest parties only get convergence opportunities, and there are no successful queries that are not convergence opportunities. This gives an advantage to the honest parties, so, if our adversary is successful against these honest parties, she is definitely going to also be successful against more modest honest parties who also have to deal with the lack of convergence opportunities. We use this simplification because we consider temporary forks due to network desynchronization immaterial to the attack in question, and we want to get an approximate feeling for how the chain quality behaves.

The pseudocode for the simulation is illustrated in Algorithm 14. This simulation uses the Monte Carlo method: An experiment is repeated many times (Line 4), and a quantity of interest (here, the chain quality μ) is measured during each experiment (Line 21). An average is then taken across all experiment executions (Line 24). The hope is that, even though each experiment may yield an unusual value, the average value will be indicative of the attack's nature.

Each experiment proceeds as follows. Initially, all honest parties have only the genesis block, so all honestly adopted chains have just one honest block (Line 6). Each experiment allows the chain adopted by the honest parties to grow until it reaches a predetermined length c (Line 8). While blocks are produced in the experiment, the simulation keeps track of how many blocks adopted by the honest parties have been honestly generated (honest), as well as how many blocks adopted by the honest parties have been adversarially generated ($\mathsf{adversary}$). The simulation also keeps track of the number of blocks the adversary has generated but is still keeping secret ($\mathsf{advantage}$). At every iteration, we allow one block to be mined, and we give the block to the adversary with probability $\frac{t}{n}$, or to the honest parties with probability $\frac{n-t}{n}$. This is done by selecting a uniformly random number in the interval $[0, 1)$ and then comparing it with $\frac{t}{n}$ (Line 9). For simplicity, as discussed before, we assume all honestly generated blocks are instantly communicated to the rest of the honest parties, so no divergence occurs between honest parties. Nevertheless, the rushing adversary has the option to rush whenever an honest block has been produced, and the adversary has an advantage.

If the honest parties get a block, then the adversarial advantage is checked. If the adversary has no advantage, this means that the number of honestly generated blocks in the honestly generated chain increases by one (Line 17), and the adversary starts mining on top of the new honestly generated block. The adversarial advantage remains 0. If the adversary has a positive advantage, then the adversary broadcasts one of the previously secret blocks, and so displaces the honestly generated block, which is discarded by all honest parties. In this simple simulation, we ignore the fact that the honest party who mined this block will keep mining on it. This is an accurate simplification if the mining parties are many, with a small amount of mining power each. This causes the adversarially generated block to become honestly adopted, and the adversarial advantage to decrease by one (Line 14). Lastly, if the adversary happens to get a new block, then this block is kept secret for the moment, and the adversarial advantage increases by one (Line 11).

The Monte Carlo method is a very useful method to get an initial feel of whether an attack can be successful and to analyze probabilities and other values of interest. While it is not an analytical method, it can often function as a first indicator of whether an attack makes sense, and whether further theoretical analysis is warranted. It can also be used as a mechanism to double-check our theoretical results.

Algorithm 14 A Monte Carlo simulation to calculate the chain quality achieved by a minor selfish miner.

```
function SIMULATE-SELFISH($n, t, c, m$)
    $\mathsf{pr} \leftarrow \frac{t}{n}$    ▷ Probability of the next successful query being adversarial
    $\sigma \leftarrow 0$
    for $i = 1$ to $m$ do    ▷ Monte Carlo iteration
        $\mathsf{advantage} \leftarrow 0$    ▷ Length of current secret adversarial fork
        $\mathsf{honest} \leftarrow 1$    ▷ Honestly generated honestly adopted block count
        $\mathsf{adversary} \leftarrow 0$    ▷ Adversarially generated honestly adopted block count
        while $\mathsf{honest} + \mathsf{adversary} < c$ do    ▷ Make a $c$-long chain
            $r \xleftarrow{\$} [0, 1)$    ▷ Roughly simulate mining stochastic process
            if $r < \mathsf{pr}$ then    ▷ Adversary gets block
                $\mathsf{advantage} \leftarrow \mathsf{advantage} + 1$
            else    ▷ Honest parties get block
                if $\mathsf{advantage} > 0$ then
                    $\mathsf{advantage} \leftarrow \mathsf{advantage} - 1$    ▷ Reveal previously secret block
                    $\mathsf{adversary} \leftarrow \mathsf{adversary} + 1$    ▷ Rushing adversary succeeds
                else    ▷ Advantage remains 0
                    $\mathsf{honest} \leftarrow \mathsf{honest} + 1$    ▷ Unsuppressible block
                end if
            end if
        end while
        $\mu \leftarrow \frac{\mathsf{honest}}{\mathsf{honest}+\mathsf{adversarial}}$    ▷ Quality of the particular honestly adopted chain
        $\sigma \leftarrow \sigma + \mu$
    end for
    return $\frac{\sigma}{m}$    ▷ Average quality
end function
```

Plugging some numbers into this simulation, we observe that, for large values of c and m, the average chain quality converges to $\frac{1}{2}$ for $\frac{t}{n} = \frac{1}{3}$, and to 0 as $\frac{t}{n} \to \frac{1}{2}$. You are asked to verify these results in Problem 7.1.

We used a simulation to calculate these numbers to get a feel for them, but we can also support these numbers theoretically and develop a closed-form formula for the achieved chain quality. To see how this attack performs, note that *every* adversarially mined block is accepted in the honestly adopted chain. For every adversarially generated block that is accepted on the honest chain, there is one honestly generated block that was displaced from the honestly adopted chain. Those are exactly the honestly generated blocks that are wasted. The rest of the honestly generated blocks make it to the honestly adopted chain.

Let L be the total number of blocks produced, of which $L_\mathcal{A}$ are adversarial and L_h are honest. The ratio $\frac{L_\mathcal{A}}{L}$ will approach $\frac{t}{n}$, the ratio $\frac{L_h}{L}$ will approach $\frac{n-t}{n}$, and the ratio $\frac{L_\mathcal{A}}{L_h}$ will approach $\frac{t}{n-t}$ after long executions. Let $|\mathcal{C}|$ denote the length of the ultimate honestly adopted chain, $|\mathcal{C}_\mathcal{A}|$ be the number of adversarial blocks in it, and $|\mathcal{C}_h|$ be the number of honest blocks in it. We have that $|\mathcal{C}_\mathcal{A}| = L_\mathcal{A}$ since all adversarially generated blocks are eventually adopted, but $|\mathcal{C}_h| = L_h - L_\mathcal{A}$, since every adversarially generated block displaces an honestly generated block. Lastly, $|\mathcal{C}| = |\mathcal{C}_\mathcal{A}| + |\mathcal{C}_h| = L_h$. The overall chain quality will be $\mu = \frac{|\mathcal{C}_h|}{|\mathcal{C}|} = \frac{L_h - L_\mathcal{A}}{L_h} = 1 - \frac{t}{n-t} = \frac{n-2t}{n-t}$. This verifies our simulation results.

The conclusion is that a minor adversary with close to 50% mining power can bring chain quality very close to 0. Regardless, if the adversary has mining power strictly less than 50%, the chain quality will be non-zero. Incidentally, the fact that in the above calculations $|\mathcal{C}| = L_h$ also shows that the selfish mining attack is minimizing *both* chain quality *and* chain growth at the same time, since the adversarially mined blocks do not contribute to the honestly adopted chain growing more than it would grow even if no adversarial blocks were to be mined.

A minor adversary cannot break Common Prefix for large k, can reduce but not eliminate the chain velocity τ, and can reduce Chain Quality μ close to, but not equal to 0, the minor adversary cannot break the ledger virtues. Safety survives because of Common Prefix. Liveness survives because the chain grows and the chain quality is non-zero. Remember that, to ensure liveness, it is required that just a single honest block survives once in a while. While the time it takes for transactions to confirm will be harmed, all honest transactions will eventually make it to the ledger of all honest parties, and these ledgers will remain consistent.

7.3 Attacks of a Major Adversary

We have explored the attacks of a minor adversary, and showed that ledger virtues survive them in a well-configured blockchain network. Let us now turn our attention to a *major adversary*, an adversary with $t > n - t$. Such attacks are also known as 51% attacks (or > 50% attacks).

Common Prefix

A major adversary can easily break Common Prefix by performing the Nakamoto Race, as discussed in the previous chapter. In fact, the larger the duration of the race, the more blocks the adversary gets. Increasing the Common Prefix k parameter in our confirmation rule does not help! Concretely, if the honest parties have converged to a tip B, the adversary can start mining on top of $B[{:}{-1}]$. Once she has mined $k+2$ blocks and holds a chain $\mathcal{C}_\mathcal{A}$, she waits for the honest parties to also mine their own k blocks on top of B forming the chain $\mathcal{C}$. She then releases her chain and, since it is longer, the honest parties switch to it. The Common Prefix property is violated because, prior to the switch, the honest parties were holding a chain whose stable chunk $\mathcal{C}[{:}{-k}]$ was not a prefix of the adversarially produced unstable chain $\mathcal{C}_\mathcal{A}$. In particular, $\mathcal{C}[{:}{-k}][-1]$ does not appear in $\mathcal{C}_\mathcal{A}$. Refer back to Figure 6.6 for a visualization.

Chain Quality

A major adversary can easily break Chain Quality and achieve exactly $\mu = 0$ by, again, performing the Nakamoto Race. In fact, the adversary can simply just ignore all honest blocks, and mine on her own from the genesis block. No matter what the honest parties do, she will eventually surpass them and build a longer chain. The honest parties will adopt the adversarially generated chain. Since the adversarially generated chain only consists of adversarially generated blocks (with the exception of the genesis block), the chain quality of the chain chunk $\mathcal{C}[1{:}]$ will be 0.

Chain Growth

For the same reasons that a minor adversary fails to completely eliminate Chain Growth, the major adversary also fails to eliminate Chain Growth. The best attack the adversary can perform is to remain idle. Then velocity τ then drops to roughly a proportion $\frac{n-t}{n}$ of the rate it would have if the adversary were honest. Nevertheless, if $n - t > 0$, the chain continues to grow. The fact that the adversary cannot significanty harm Chain Growth is a consequence of the longest chain rule of our protocol.

Safety

Since the major adversary can break Common Prefix, she can also break ledger safety. The double spending attack using the Nakamoto Race is one example. A transaction that is rolled back from an honest party's ledger and then replaced by a different transaction in the same position is a safety violation.

As we discussed when we first introduced the double spending attack in Chapter 3, the double spending is problematic because it has real-world ramifications. When the adversary double spends, she receives some real-world goods, such as a car. When a conflicting transaction appears on the ledger of the seller and the original transaction is reverted, the adversary has long since departed with her brand new car. However, note that, while the adversary can break safety, the ledgers adopted by the honest parties are always *locally consistent.* There is never the case of an honest party adopting a ledger that contains two conflicting transactions. The only thing that happens is that an honest party switches from one, locally consistent, ledger to another, also locally consistent, ledger.

It is worth noting that a major adversary cannot steal an honest party's money from the system. Spending an honest party's money would require creating a signature that validates with the honest party's public key. That would require stealing the honest party's secret key, or breaking the existential unforgeability property of signature schemes. Therefore, even a major adversary cannot take our money.

A major adversary also cannot create more money than what is allowed by the macroeconomic policy, because blocks that violate the macroeconomic policy are not accepted by the honest nodes. Recall that when a block is validated, the coinbase transaction is checked, and part of that check includes validating the macroeconomic policy. Even though the honest nodes are a minority, they do not accept an invalid chain which violates these basic rules.

Liveness

While the major adversary cannot break Chain Growth, she can break Chain Quality and therefore she can cause a liveness violation. This can be used to mount censorship attacks. As soon as the censoring adversary sees an honest transaction she dislikes, she starts mining on top of the currently adopted block, excluding the transaction in question from her block. She keeps mining on top of that block, eventually creating a longer chain than the honest parties because she holds the majority of compute. She must continue to mine, because she cannot allow any honest block to ever make it into the chain, as honest parties are still including the transaction in question in their mempool. Contrary to a Nakamoto Race, she does not need to keep her fork secret, and she can allow it to be extended by the honest parties at any time.

Such a situation is depicted in Figure 7.2. The white circle represents a transation that the adversary wishes to censor. Whereas the honest parties are continuously attempting to include it into their blocks, the adversary keeps displacing their blocks by mining on top of the parent block. The honest blocks with the transaction in question never makes it into the ultimate honestly adopted longest chain.

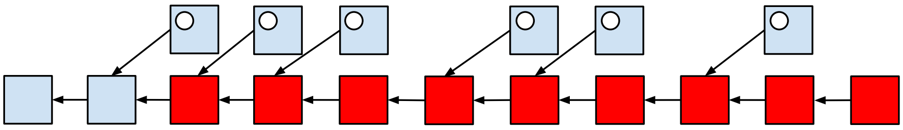


Figure 7.2: A major adversary mounts a censorship attack.

7.4 Healing

Even though a major adversary can break Common Prefix, Chain Quality, Safety, and Liveness, it is worth considering whether these properties *heal* when the adversarial power goes back to a low value. The idea is that we may have a *temporary dishonest majority*, which overtakes the network for a short duration of time, but the adversary soon goes back to controlling the minority of compute. While we will not prove the exact formulae which determine the healing parameters of the chain in this book, we want to give intuitive arguments about why a proof-of-work chain heals very naturally.

The temporary dishonest majority situation is depicted in Figure 7.3. The system begins with a minor adversary whose compute increases until it exceeds 50%. The dishonest majority compute remains for a period of time, until the adversary's compute drops back to below 50%.

Common Prefix

We already know that the major adversary can break Common Prefix, no matter what parameter k we choose. However, if she only has *temporary* dishonest majority, she cannot revive very old temporary forks, because it would take a long time to have them catch up with the current times, so she must mine on a recent temporary fork. She will likely be able to create a temporary fork longer than k blocks long, but not *arbitrarily* longer, because she has limited time. The length of temporary forks will still be bounded. When the adversary's mining power drops below the majority, she has limited time until she can reveal any secret temporary forks she has mined, because the honest parties will soon create a longer chain and the, now minor, adversary will not be able to catch up. Hence, such withheld temporary forks will soon be useless. As such, Common Prefix can only be broken during the time of dishonest majority, and perhaps a little earlier and a little later. However,

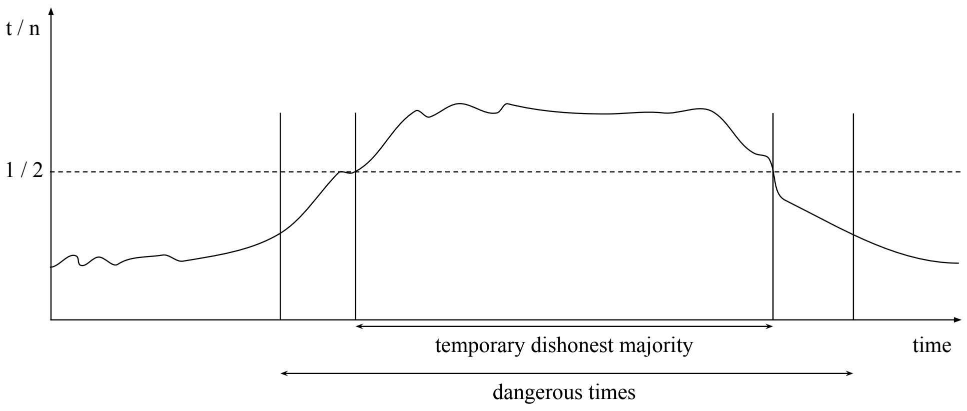


Figure 7.3: The temporary dishonest majority.

if we wait enough time after the adversary has gone back to being minor, we will still have some Common Prefix assurance.

More concretely, there will be certain *dangerous times* which extend from the moment the temporary dishonest majority takes over and slightly after the dishonest majority period, during which we cannot give Common Prefix assurances. Nevertheless, the chains of honest parties adopted *before* these dangerous times, and the chains of honest parties adopted *after* these dangerous times will be consistent. In fact, chains adopted *before* the dangerous times will also be consistent with chains adopted *after* the dangerous times. Common Prefix heals.

Chain Quality

We can apply a similar argument to show that Chain Quality heals. The adversary cannot go back and rewrite very old chunks of the honestly adopted chain. Additionally, soon after the adversary goes back to being minor, honest blocks will keep making it into the honestly adopted chain. The problematic chunks concern only those produced during the dangerous times, which extend from a duration before to a duration after the dishonest majority interval. Chain Quality heals.

Safety

It is not safe to use the system when an adversary has the upper hand. Additionally, it is hard to tell *when* an adversary has the upper hand and when one should not use the system. However, if a particular user happens not to use the system during and around times of dishonest majority, the user will be safe. In particular, one is not susceptible to transaction rollbacks or double spending if one waits sufficiently long after the system has recovered. Because the adversary can roll back chains that have appeared as confirmed retroactively after the dishonest majority period commences, the dangerous times extend for a period *prior* to the dishonest majority period. Ledgers reported by honest parties at times *before* the dangerous times and *after* the dangerous times will be consistent with one another. Safety heals, because Common Prefix heals.

Liveness

The adversary can censor transactions for the duration of time she controls dishonest majority. However, soon after, Chain Quality recovers, and honest blocks make it into the chain. This means that censorship attacks will eventually be recovered from, soon after the adversary loses the upper hand. At that point, any transactions that were previously censored will be included in the honestly generated blocks. Additionally, any newly produced transactions will make it into the new honestly generated blocks as well. Liveness heals, because Chain Quality heals.

Problems

7.1 Implement Algorithm 14 in your favourite programming language, and run it for values t, n, c, m of your choice. Can you reproduce the results described in the text?

7.2 Use the Monte Carlo method to implement a simulation to calculate the density of convergence opportunities in time for various values of n, t, T, Δ. Plot your result. Does your graph resemble Figure 6.4?

Further Reading

The Nakamoto Race was explored in Satoshi's Bitcoin paper. There, he analyzed the probability of the success of a minority adversary mounting the Nakamoto Race attack against a majority of honest parties. This is the only attack that Satoshi analyzed. He did not prove that his protocol is secure against all adversaries. The longest chain protocol was analyzed and proven secure against *all* adversaries in the later work of the Bitcoin Backbone, which we explore in the later chapters of this book.

In an even later work from 2020, *Everything is a Race and Nakamoto Always Wins [11]*, the authors discuss hod Satoshi has been right all along, but we did not know it: In fact, *any* mining attack an adversary can possibly perform is roughly equivalent to a Nakamoto Race.

While the Fan-Out was known to Nakamoto, but was made precise in the Bitcoin Backbone line of work, where they argue that

> f cannot be too large, because [this would mean] that uniquely successful rounds will not produce sufficiently many PoW's to overcome the PoW's produced by the adversary.

Chapter 8

Economics

So far, we have analyzed our protocols in the *honest majority* setting, assuming that most parties will follow our protocol. We provided assurances such as safety and liveness to honest parties. But it's not clear why, even though these assurances are provided, anyone should follow our honest protocol, especially if they can achieve better benefits by deviating from it. If there is money to be made by violating the protocol, it is likely that a majority of participants will do so. Instead of assuming *honesty*, it would make more sense to assume that participants are *rational.* A rational party does not necessarily follow the prescribed honest protocol. Instead, he may deviate from the rules in order to make money. The ideal scenario would be if the honest protocol we designed is also *incentive-compatible*: that there is no better strategy that a party can follow which deviates from the honest strategy and makes them more money (a Nash equilibrium).

8.1 Rationality Models

There are a few different models in which we can analyze our protocols. In the *cryptographic* model, the majority of parties are assumed to be honest (and PPT), whereas the rest are allowed to be arbitrarily adversarial (they can be any PPT algorithm). In the *game theoretic* model, *all* parties are typically assumed to be rational, and no computational restrictions are imposed upon them. However, the game theoretic model does not typically account for adversaries who are willing to deviate from the protocol even if they are about to lose money. Such parties can be realistic in cases where, for example, a nation-state wishes to invest resources simply to shut down a decentralized protocol. But the cryptographic model *does* capture such adversaries, as long as most other participants are honest.

It would be nice to capture the best of both worlds. The most powerful results would be achieved in a model where a *majority* of parties are assumed to be PPT rational, whereas a minority are assumed to be *arbitrarily PPT adversarial.* This is holy grail model and its comparison to the other models is depicted in Figure 8.1. Unfortunately, analyzing protocols in the rational majority setting is very difficult with the current mathematical tooling available to us. Analyzing the incentives of blockchain protocols is only possible for very simple protocols, or for very simple models, and having an arbitrary PPT adversary in the picture makes things even harder. While we would like to have analyses in this model, it is rarely the case that we can. There are many reasons for this, among others the complexity of the protocol, the large number of parties, the broad range of available strategies, the fact that the protocol is iterated over large periods of time, and the fact that the practical instances of these protocols have externalities such as payments that can be received outside of the protocol.

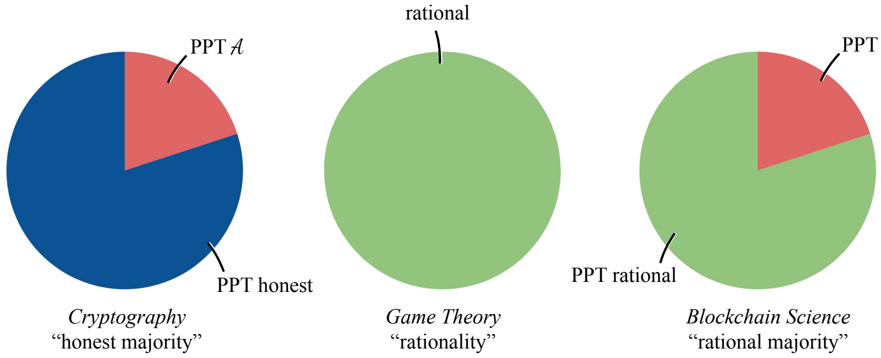


Figure 8.1: The cryptographic, game theoretic, and blockchain science model.

Regardless of the limitations of our understanding and ability to prove things precisely in the rational majority setting, it is still worthy to think about the rationality of our design. While we might not be able to prove that our honest protocol is incentive-compatible and survives against arbitrary adversaries, we also want to make sure the honest protocol is not costly to operate without providing appropriate incentives to honest participants.

In the particular case of the proof-of-work protocol we have studied, running the honest mining protocol is actually extremely costly. At the time of writing, Bitcoin consumes more electricity than the whole country of Argentina, which is an expensive endeavour. Therefore, miners need to be incentived to secure the network, for example by having their electricity paid for.

We have already examined the mechanisms by which miners are paid when they successfully mine a block in Chapter 5, where we mandated that a block's coinbase transaction pays out a total amount f_t which consists of the *block reward* f_r and the *block fees* f_f:

$$f_t = f_r + f_f$$

Both of these terms are denominated in the system's native currency. We will now study these two terms in more depth.

8.2 The Block Reward

The block reward part constitutes the *macroeconomic policy* of the system. This block reward plays a dual role for the system: Firstly, it incentivizes miners to participate in the honest protocol. Secondly, it is a natural mechanism for the decentralized creation of money. Whereas the first purpose is necessary to make the protocol incentive-compatible, the second part is a policy decision of the protocol. For example, it is equally possible to have protocols in which new money creation is left up to a central party, which can be the party to whom the newly issued money is paid out to, and only a small proportion of the proceeds to go to the miner. This would centralize money issuance, but still enable decentralized public verifiability. Such decisions are a matter of policy.

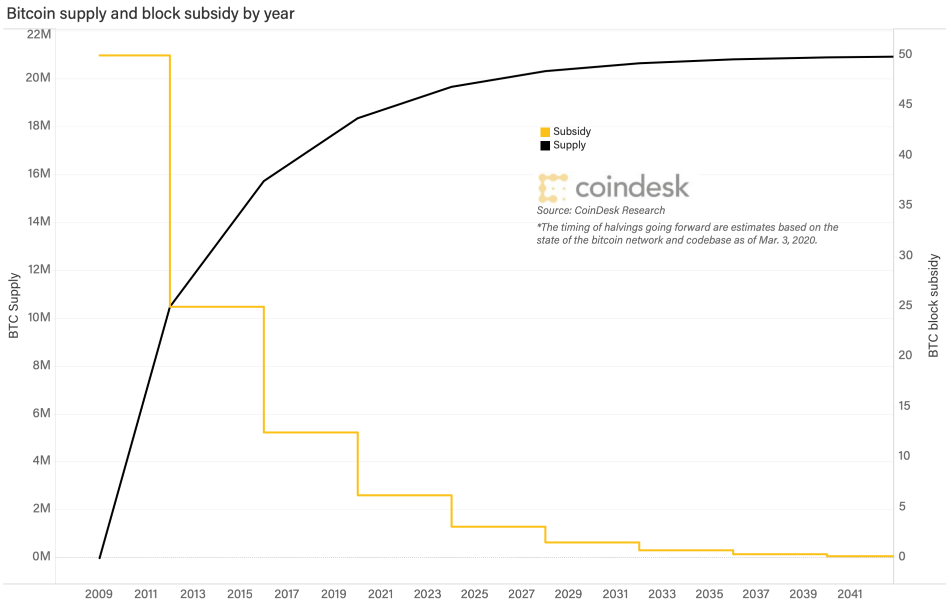
The exact algorithm determining the value of f_r is *hard-coded* into each full node and is part of the block validation algorithm. When a block is validated, its coinbase is also validated, and this is where the macroeconomic policy is checked for validity. It is typically not possible to change these rules of the cryptocurrency, unless the community of the currency all decide to move to a new policy and upgrade their software together. Such decisions can be difficult and contentious.

There are a few alternatives that can be followed in the macroeconomic policy of a cryptocurrency. The policy is defined as a function of the block $f_r(B)$. One option is to make the reward *constant*, which means that the total supply will continue increasing at a constant rate. Another option is to make the reward *decreasing* over time until it becomes 0. In such a system, the function f_r depends on the block height. With such a policy, the total supply is increasing over time, but converging to a particular value.

This is the policy that Bitcoin follows. Bitcoin's macroeconomic policy is an interesting concrete example, because many other cryptocurrencies follow a similar policy. The function f_r for Bitcoin follows a staircase pattern, with long periods of constant supply per block. The supply began at a rate of 50 bitcoin per block at genesis. Every four years in block time (which corresponds to 210,000 blocks because $\eta = 10$ minutes in Bitcoin), the reward per block drops to a half (a process known as *reward halving*). The reward schedule for Bitcoin therefore began at 50 bitcoins per block during the years 2008-2012, dropped to 25 bitcoins per block during the next four years, and so on. At some point in the future the block reward in Bitcoin will drop below one Satoshi, at which point it will become 0 and no new coins will enter the supply. In about a century, Bitcoin will reach a total supply of 21 million bitcoin, and the reward will drop to 0. This macroeconomic policy is depicted in Figure 8.2 (in this graph by CoinDesk, *subsidy* refers to the block reward).

One of the problems with such a policy is that the reward function $f_r(h)$ is discontinuous with block height, and the finite difference of the total supply function $\sum f_r$ changes abruptly at the points of reward halving. This behavior can cause market shocks because the incentives to the miners shift abruptly. Consider a miner who has invested in a large business to mine blocks. The cost of electricity to mine is a certain amount. At the moment of reward halving, the miner is still spending the same amount per month in electricity, but only getting half of the nominal rewards. Electricity is typically paid out in fiat (i.e., non-crypto) currency such as USD. If the price of Bitcoin in USD remains the same, then the miner will be forced to stop half of their operation overnight. Alternatively, if the miner is to continue operating at the same rate, the price of Bitcoin in USD needs to double overnight. Both of these are undesirable situations.

To avoid such market shocks, other cryptocurrencies have adopted a more nuanced macroeconomic policy. One such example is Monero which pioneered the concept of *smooth emission*. In such a system, the rewards are reduced slightly in every block, following a pattern similar to Bitcoin's macroeconomic policy, but



Bitcoin supply and block subsidy by year

Source: CoinDesk Research

*The timing of halvings going forward are estimates based on the state of the bitcoin network and codebase as of Mar. 3, 2020.

Figure 8.2: Reward Halving for Bitcoin Supply (source: coindesk)

adjusted to work in a more continuous manner.

The reward and total supply functions for Bitcoin and Monero are illustrated in Figure 8.3. The actual reward and supply over real (and not block) time can fluctuate less smoothly than anticipated because of the stochastic nature of proof-of-work. Note how Bitcoin's block reward forms a staircase, while Monero's is a smooth curve. Due to a decision taken by the community, Monero's rewards flatten out at the point indicated by the vertical red line.

Because the total supply stops growing at some point in time, Bitcoin and Monero are *deflationary cryptocurrencies*. Many orthodox economists believe that deflationary cryptocurrencies are inappropriate as currencies, because they encourage hoarding and not spending when the emission period is over. This is one reason why other cryptocurrencies have no limit on their total supply. Instead, they continue to inflate the currency and are therefore known as *inflationary cryptocurrencies*. While it is important that the total supply of a cryptocurrency is limited at every point in time to ensure scarcity, it is not necessary that there is a global total bound across all of time. One such example cryptocurrency is Dogecoin.

Overall, the decision of the macroeconomic policy of a system is up to the economists and the community, and there is no exact science that can define it in an objectively acceptable manner. Any emission algorithm which is socially acceptable by the economic participants can be used.

The block reward is the main way miners are incentivized to mine.

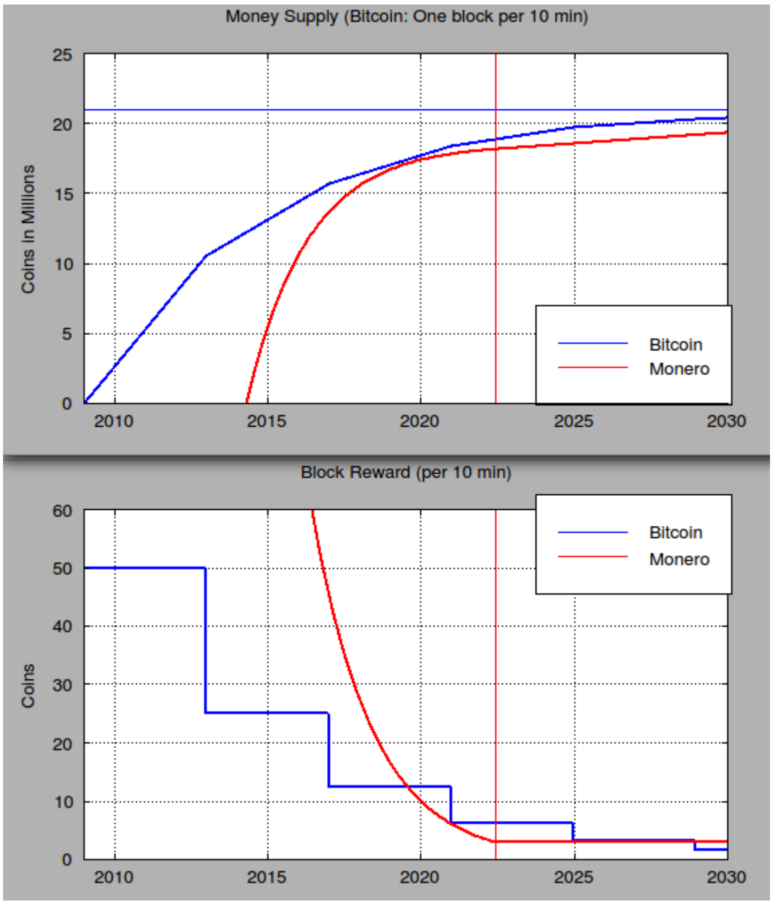


Figure 8.3: The total supply (top) and block reward (bottom) for Bitcoin and Monero.

8.3 Miner Fee Optimizations

While miners are incentivized to mine blocks by giving them a block reward, they need to also be incentivized to include transactions in their blocks. Otherwise, they will simply create *empty blocks*, blocks that contain only a coinbase transaction and don't confirm any part of the mempool. This is solved by introducing transaction fees.

The other term of the total payout is the fees $f_f(B)$ of a block B. Recall that the fees are calculated as the excess money that goes into a transaction but does not go out. Given a transaction tx in the UTXO model, the fees that this transaction is paying will be

$$\sum_{i \in \mathsf{tx.ins}} i.v - \sum_{o \in \mathsf{tx.outs}} o.v\,.$$

The creator of a transaction can choose how many fees he wants to pay, and these can be 0 or larger.

Each block contains many transactions, each of which may pay out fees. All these transaction fees make up the *fee* part $f_f(B)$ of the block payout:

$$f_f(B) = \sum_{\substack{\mathsf{tx} \in B.\overline{x} \\ \mathsf{tx\ non\text{-}coinbase}}} \left(\sum_{i \in \mathsf{tx.ins}} i.v - \sum_{o \in \mathsf{tx.outs}} o.v \right) .$$

Including an extra transaction has minimal cost to the miner and does not affect the mining rate. If there was no limit in how big blocks can be, a rational miner would include all transactions paying even minimal fees. But contrary to what we have described so far, we will need to introduce a block size limit so that our assumptions can work out.

The Block Size Limit

So far, we have not imposed any limits on how big blocks can be. We have also assumed that blocks can traverse the network within Δ time, just like any other network message. This should make you a bit suspicious by now. The larger a block is, the more time it takes to send over the network. If we don't impose a bound on how large a block can get, then Δ cannot be a fixed constant. There are some simple ideas to make blocks of bounded size that don't work. One such example is to include just the transaction ids within the block, and then allow the client to request each transaction as needed at a later time. Another example is to just include the hash of the transaction sequence $\overline{x}$, which already commits to the transactions, and not the transactions themselves, again allowing the interested client to download the transactions in question. We'll explore these strategies in more detail in Chapter 10 to optimize our protocol, but for now let's observe why these don't fix the Δ problem: The time Δ really captures the time that is needed for *all* the data to transmit over the network so that a block can be properly validated. If we separate out transaction bodies from blocks, then a miner who observes a new block on the network has to download all transactions in order to ensure block validity. Otherwise, the miner cannot know if the block is valid or not and where to mine.

To ensure our assumption of a Δ delay holds, we will limit the size of a block to $B_{\max}$:

$$|B| \leq B_{\max}$$

This size limit must be denominated in actual *bytes* (or bits) transmitted over the network, not in the number of transactions per block. The reason is that the delay Δ depends on the literal size of the object transmitted.

Miner Strategies

Generally speaking, which transactions to include in a block is completely up to the miner, as long as the block ends up being valid. The miner can choose to leave the block empty, fill it to the brim, include transactions that are paying small fees, or include transactions paying large fees. The miner can also include their own transactions at any position they want within the block, and reorder the rest of the transactions in the block in any way they like. Despite all of this freedom, some transaction inclusion *strategies* are better than others.

A rational miner is looking to optimize the fees that they can gain by including the highest paying transactions that they can. When the miner is mining, they create a template block in which the included transactions are $\overline{x}^*$. This $\overline{x}^*$ is computed as follows. If the transactions in their mempool $\overline{x}$, plus the new coinbase transaction, are enough to fit within B_{max}, then the miner has no reason to exclude any transactions. He can simply include everything to maximize his profits.

On the other hand, if the mempool $\overline{x}$, plus the new coinbase transaction, exceed the block size limit B_{max} in size, then the miner must make a choice on which transactions to include. He tries to maximize the profits by including the highest paying transactions. In order to do that, each transaction is associated with a *score*. The score of a transaction is defined as the fees it is paying divided by the bytes it takes up:

$$\text{score}(\mathsf{tx}) = \frac{|\mathsf{tx}|}{\sum_{i \in \mathsf{tx.ins}} i.v - \sum_{o \in \mathsf{tx.outs}} o.v}$$

The easiest way is then to order transactions by score, in descending order, and take transactions until no more fit and B_{max} would be exceeded. This strategy is also known as the *greedy strategy*. The reason why the ratio fee/byte is used instead of just the fee is that, naturally, a bigger transaction takes up more space in the block, and must therefore pay accordingly. This concept that block space costs a certain amount per byte is also known as *block space rent*.

Unfortunately for the miner, things are not so simple. The first issue that arises is that transactions can either be included *as a whole* or *not at all*. It is not possible to cut a transaction in half and just include a portion of it into the block. This means that the strategy to just include the top-score transactions will not perform optimally. One such example is illustrated in Figure 8.4. Here, the block is illustrated as the outer blue box, with its size indicating the block size limit B_{max}. Transactions in the mempool (breaking away from our usual circle notation) are displayed as smaller boxes containing numbers. The numbers within the transactions indicate their score, and the size of a box indicates the transaction's size in bytes. The transactions have been ordered by the greedy strategy from left to right in descending order according to their scores. The green transactions are included in the block, as they are the transactions with the top scores. The red transactions are not included in the block, as they have lower scores. The transaction with a score of 4 barely doesn't fit in the block. However, the greedy strategy misses the fact that the two smaller transactions with a score of 3 can both still fit in the block. A more clever solution would include these also. If it were possible to cut the transaction with the score of 4 and partially include it, then this solution would have been optimal, but we are forced to exclude it altogether, so the transactions with a score of 3 are our next best option.

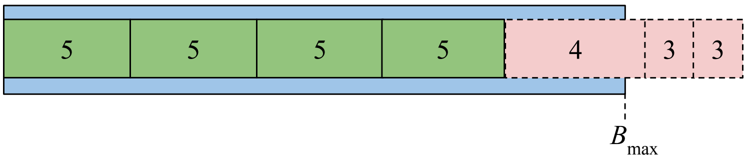


Figure 8.4: The greedy strategy is not always optimal.

This problem is known as the 0/1 Knapsack problem and is known to be an NP-hard problem. Its best known solution is only pseudopolynomial (which means it takes exponential time to execute). Therefore, the problem of optimizing which transactions to include is, generally, difficult, and only heuristics can be employed to solve it. These heuristics sometimes necessarily yield suboptimal solutions.

Even this simple rendition of the problem is NP-hard, but in fact the problem is harder still. The transactions that can be included must also respect the topological order induced by the transaction graph. We cannot include a transaction tx_1 before the transaction tx_2 it is spending from. Therefore, the problem is similar to a 0/1 Knapsack, except that topological relations must also be preserved.

Lastly, there is one more complication. The simplistic honest miner maintains a mempool that outright rejects double spends. This is fine for a miner who is not interested in maximizing his profits. However, a rational miner wants to keep track of double spends. If two conflicting transactions tx and tx' are available, the miner wishes to choose the one with the highest paying fee. Even if a double spending transaction does not pay large fees, it may still worth be caching for later inspection, because it may be the input to a next transaction that *does* pay large fees. Therefore, a rational miner will also cache transactions outside of his mempool and hold onto them in case they are later helpful in optimizing his profits.

In complex blockchains, such as Ethereum, the order of transactions in a block may give rise to more nuanced profits than just those obtained from fees, and so even more advanced techniques are needed. The strategies employed by modern rational miners to optimize profits are extremely complicated and have given rise to a whole industry on their own. In the modern world, *computing* the near-optimal $\overline{x}^*$ and actually *mining* a block are two separate tasks. The computation of $\overline{x}^*$ is a task performed by *searchers* and *builders*, whereas the mining is performed by *miners*. These are different, somewhat mutually trusted, parties running different software and communicating with one another to optimize rational block production and share these profits, which can range to tens of millions of dollars per month.

8.4 Wallet Fee Optimization

While the miner is trying to *maximize* their fees, the *wallet* of the user who is spending money is trying to *minimize* their fees. What is observed on-chain is the economic equilibrium between the two.

Refer back to Figure 4.5 to recall that the wallet is the piece of software that sits between the human user and the full node. The job of the wallet is to take requests from the human user to make payments, and to issue the relevant transactions to make these payments. The human user places such a request by choosing a particular recipient and value to transfer. It is up to the wallet to decide how to build the transaction.

The timeline is roughly as follows. First, the human user places a request to the wallet to make a payment of amount v to some recipient pk. The wallet receives the human request. The wallet then decides an amount of fees per byte (the score) that it wishes to spend on the transaction to be created. Based on this score and an estimation of the size of the transaction which will be created, it estimates the total fee ϕ that the transaction is about to pay out. It then builds a new transaction tx, with an output of value v and recipient pk. It then asks the full node for the current UTXO set. Among the UTXO set, the wallet (which maintains a set of private keys) can distinguish those outputs that belong to the user, since they correspond to addresses encoding public keys that it holds the respective private key to. It then can make a choice of outputs. This choice must make up a value of at least $v + \phi$, but can also exceed $v + \phi$. The wallet adds those outpoints as inputs to the transaction. If the total value in the input exceeds $v + \phi$, then it also adds a change output whose value is the total value of the inputs reduced by $v + \phi$. The address in the change output is the encoding of a public key of a new private key it generates and stores internally for future spending. After the choice of inputs has been completed, the wallet inspects the new transaction for its size and may adjust the ϕ amount accordingly. Overall, this is a heuristic process.

Some aspects of the above procedure are worth explaining in more detail. The step of choosing the inputs of the transaction must be done so as to minimize the total transaction size. The size of each outpoint is roughly the same, so what needs to be minimized is the number of inputs. Additionally, if the change output can be avoided, this is also desirable to reduce the size of the transaction. So, one strategy is to simply consume all the largest outputs that the user owns until the value $v + \phi$ is reached. However, this is not necessarily the best strategy, as this may lead to future transactions that are more expensive. Generally, there is no exact answer to this problem, because the wallet cannot know what future values the user will want to spend. An ideal wallet would minimize the user's fees across all of his spending habits. Different wallets employ different heuristics to reduce the cost.

One more detail of the above procedure is to determine the score that the wallet wishes to use for the upcoming transaction. To estimate the score, the wallet asks the human user roughly how soon he wants the transaction to become confirmed by being included in a block. This choice is a tradeoff between confirmation latency and cost in fees. If the user wishes for fast confirmation, then the fees must be high. If the user is willing to wait for some time, then the fees can be low. If the fee is too low, the transaction may take a long time to be included, or may even never be included. The user sets a desired confirmation time u (in, say, minutes). The wallet knows that blocks are produced in expectation every $\eta = \frac{1}{f}$. Once the transaction makes it to a block, it will take another k blocks for it to become confirmed (recall that k is the confirmation parameter). As such, the wallet attempts to adjust the score so that the transaction makes it within the next $\frac{u}{\eta} - k = uf - k$ blocks. The wallet then inspects the most recent $(uf - k)$ chunk of the currently adopted blockchain by the full node $\mathcal{C}[-(uf - k){:}]$, sorts transactions by score, and takes the minimum score. If the next $(uf - k)$-long chain chunk about to be mined happens to behave similarly to the last $(uf - k)$-long chunk, then using this score will ensure that the transaction will (marginally) make it within the next $uf - k$ blocks, and so will become confirmed within u time.

Some improvements in this heuristic can be made by looking into older chunks and taking an average to ensure that recent blocks are not outliers. Also, instead of simply taking the minimum score, the wallet can consider several among the minimum-scoring transactions, considering that the miners of recent blocks may have injected some transactions of their own with a very low score. Of course, the mining process is stochastic, and so blocks may not appear exactly η apart. Additionally, the network may become more congested, and people will tend to increase their fees in that case. Fee estimators are inherently inaccurate. They can only function heuristically and hope for the best.

If the fee of a transaction is miscalculated and it takes too long to make it into a block, there is a strategy to rectify this. The user can issue a new double spending transaction that spends from all the exact same outputs, but pays a higher fee by including a smaller amount in its *change* output. This new transaction can then be broadcast to the network. Because miners are rational, they will choose to adopt this transaction instead of the lower-fee-paying alternative which already exists in their mempool. Such transactions are often issued by wallets in practice and are known as *replace by fee* transactions, and do not constitute attacks. They are double spends, but are not inherently malicious. While we previously said that only the adversary issues double spend, now that fees have been introduced, it also makes sense for honest wallets to do so on occasion. An example *replace by fee* transaction is illustrated in Figure 8.5.

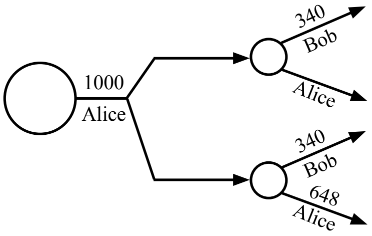


Figure 8.5: A replace-by-fee transaction. The same transaction is recreated, but with a lower value on the *change* address.

8.5 Types of Wallets

The most risky part of a wallet concerns the responsibility of maintaining the secret key private. A *software wallet* that runs on a mobile phone or a laptop computer stores the secret keys encrypted with a user-provided password at rest. These are safe for storing small amounts of money.

For larger amounts, a dedicated device known as a *hardware wallet* is used. These wallets are USB devices that can be connected to a computer using a USB cable. They are responsible for generating a (sk, pk) key pair by invoking the signature scheme's Gen function. The secret key sk never leaves the hardware wallet, and this is enforced at the hardware level. The wallet sends the pk to the computer so that the computer can generate the address to receive money. The computer also has the responsibility of building transactions based on human user requests. These unsigned transactions are then sent to the hardware wallet to obtain a signature. Before signing the transaction, the hardware wallet will prompt the human user for confirmation on the hardware device, displaying the total amount to be paid and the recipient address. When the transaction is signed by the hardware wallet and sent to the computer, the computer can then broadcast this transaction to the network.

The hardware wallet ensures that, in case the computer becomes compromised, the adversary cannot exfiltrate the secret key. Even if the adversary places a request for a signature to the hardware wallet, the human user will see the request and reject it. The hardware wallet is also protected by a PIN that must be entered by the user before any transactions can be signed. Rate limiting mechanisms for repeated PIN trials are enforced at the hardware level. The hardware is built in such a way that, even when physical access to the wallet is gained by the adversary, it is somewhat difficult to extract the private key or generate a signature, although the protection mechanisms at the hardware level have historically been subject to various attacks, which were later patched.

Two popular hardware wallets in the current market are Ledger and Trezor. Their photographs appear in Figures 8.7 and 8.6.

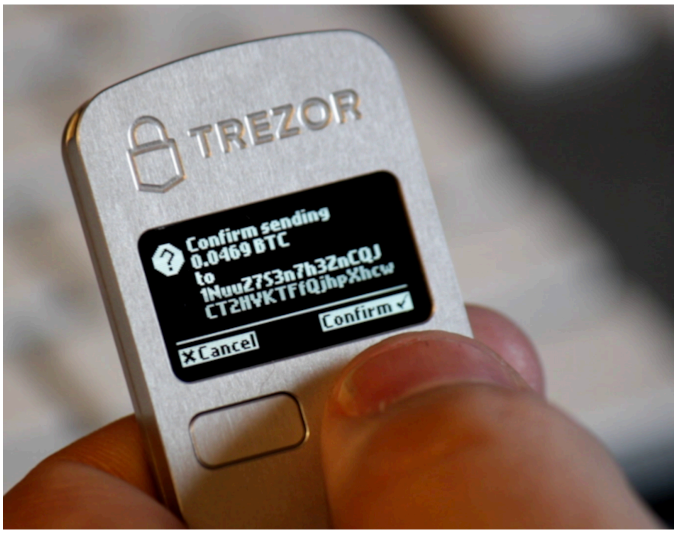


Figure 8.6: A Trezor hardware wallet requesting human confirmation for a transfer.

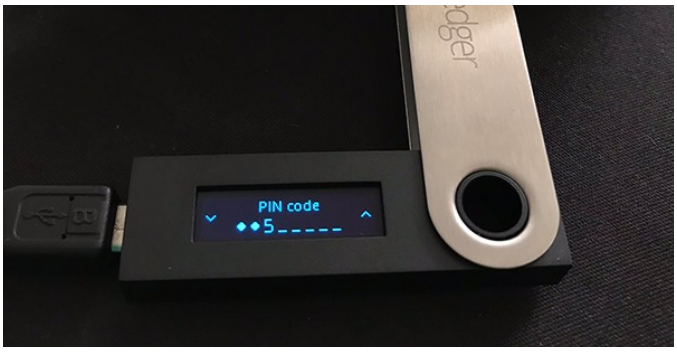


Figure 8.7: A Ledger hardware wallet connected to a computer using a USB cable and requesting human input of the PIN.

We have already discussed that multiple secret keys can be generated so that some basic level of pseudonymity is achieved by the wallet. Many wallets in the UTXO model generate a new public/secret key pair every time they receive a new payment. In contrast, in the accounts model, an account (which corresponds to a public/secret key pair) is typically generated only when the user requests so. Regardless of the model, wallets can be asked to generate and store a series of secret keys. If this generation happens anew every time a key is requested, this can be problematic, because the human user may have kept a backup of the keys at an earlier time, and the backup will have to be taken again. Instead of generating a brand new key every time it is needed, wallets often make use of a *seed*. The idea of a seed is that it is a master secret key on which all other secret keys are based.

The seed is generated by sampling a κ-bit randomness uniformly at random. This randomness is then encoded into a list of human-readable dictionary words by splitting up the seed into chunks and using the chunk as an index within a predetermined word list. Through this mechanism, the human-readable version of the seed can be encoded to something that looks as follows:

```
online response lounge afraid slow renew bright ritual boring
cram taxi page
```

Different cryptocurrencies use a different number of words and different word lists, so these seeds may look different. Some cryptocurrencies include checksums in their encodings so that some errors of transcription can be detected.

The secret keys can then be generated from the initial seed. One simple way to do this is to take the seed and hash it together with a counter $i = 0, 1, 2, \cdots$ to obtain the respective secret key. The first secret key is $H(\text{seed} \,\|\, 0)$, the second secret key is $H(\text{seed} \,\|\, 1)$ and so on. This method requires that the signature scheme has a mechanism of deriving the public key from the secret key, and that its Gen function returns a secret key uniformly at random from $\{0, 1\}^\kappa$, except with negligible probability. Some signature schemes that are used in cryptocurrencies (for example many elliptic curves) follow this form more or less, with some care required to translate the hash output into the appropriate format for the private key.

In practice, a slightly more complex mechanism is used to derive the secret keys from the seed, which allows generating keys for multiple different cryptocurrencies from one shared seed. Such constructions are known as *hierarchical deterministic wallets* (HD-wallets). The details of such constructions pertain to the engineering of the various cryptocurrencies.

8.6 Variable Difficulty

As we briefly discussed in the healing section of Chapter 7, it is possible for the adversarial power t to fluctuate, and for the system to alternate between periods of honest majority and dishonest majority. Honest mining power $n - t$ can also fluctuate. In our calculations for the target T, we have assumed that all the system parameters, including n and t remain constant. In reality, however, mining power wildly fluctuates. As a blockchain system becomes more popular and the price of its token increases, more rational mines join the system hoping to make profit from it. This functions as a reinforcement mechanism for its security: The more value a system stores, the more miners tend to want to mine for it because it is more profitable, and the more expensive it becomes to attack, because the adversary needs to expend a lot of money to reach a high t value. However, it is also possible at times for the system to be undersubscribed by miners. At the same time, as technology improves, the q value also tends to increase with time. It is not rare for popular systems to see a tremendous increase of mining power over time. As an example, Bitcoin's hash rate in the unit of time (nq in our terms, assuming the adversary is also mining in her full capacity) is illustrated in Figure 8.8. This hash rate is not directly observable, but can be deduced by the block production rate and the proof-of-work target T.

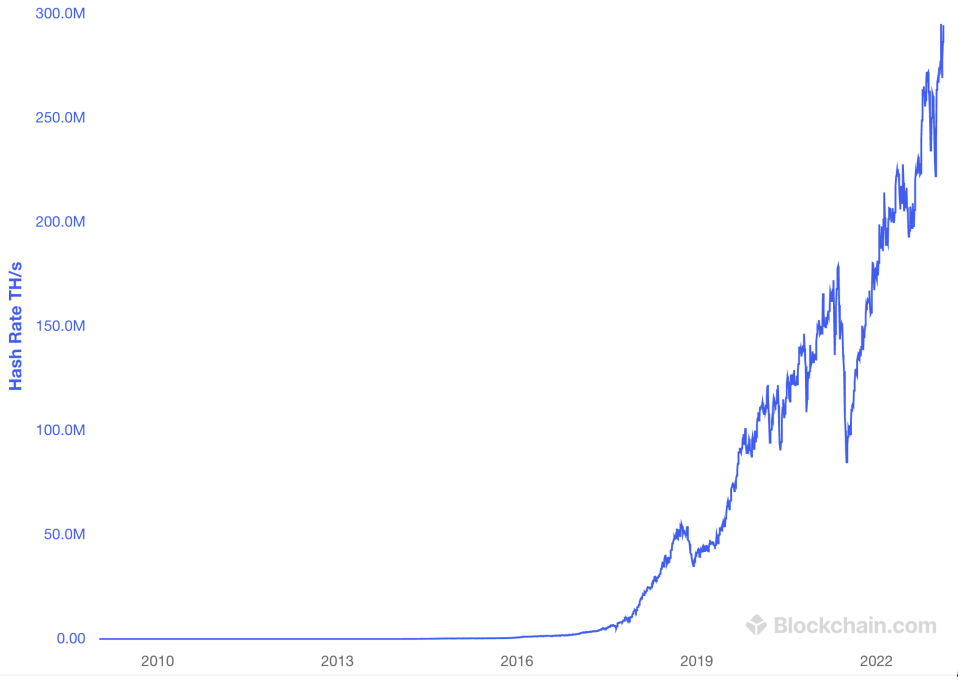


Figure 8.8: Bitcoin's hash rate from its inception to 2023.

When we calculated the T parameter, we based our calculations on n and q. As these values change over time, the parameter T will need to be adjusted. Otherwise, successful queries may become too rare or too dense, leading to issues of liveness or safety respectively.

When the system was first launched, the T parameter was calibrated according to the network parameter Δ, which we assume does not change much over time (a change in Δ implies technical innovations on the physical network level, which we do not treat in this book), and the mining power nq which existed at the first stages of the system's lifetime. Based on the estimated Δ, the system was calibrated in order to achieve a particular expected block generation rate η. For example, in Bitcoin, the system was calibrated to achieve an η of 10 minutes per block. We wish this η to remain constant throughout the system's execution, even though the mining rate may fluctuate wildly. We cannot directly measure the values n, t, and q, but we can sample η by taking observations now and then, and making adjustments to T accordingly. If η is too large, this means that blocks are arriving farther apart than we like, so we must increase T to make mining easier. If η is too small, this means that blocks are arriving close together, so we must decrease T to make mining harder.

As mining is a stochastic process, it can happen that a block here and there may arrive quickly or slowly simply by chance, not having to do with the actual underlying mining rate. Therefore, we will not recalibrate our system every time a block arrives. Instead, we'll choose a somewhat large number of blocks and recalibrate the system every this many blocks. We call this number of blocks m, the *epoch duration*. Every m blocks, the target T is recalculated based on how quickly these m blocks arrived. The target T remains the same for the next m blocks. These m-long chunks of blocks in which the target remains the same are known as *epochs*. These epochs are illustrated in Figure 8.9.

Definition 25 (Epoch)**.** *Given an* epoch duration $m \in \mathbb{N}$*, an* epoch *of a chain is any chunk* $\mathcal{C}[im{:}(i+1)m]$ *for any* $i \in \mathbb{N}$.

It is desired that each epoch takes ηm time to complete. By measuring how much time it *actually* took to complete, we can proportionally adjust the target T. However, we cannot do this by simply measuring *when* we locally received a block, because different nodes may disagree about the time at which they received the block. Instead, we use the chain itself to reach consensus about when a block was produced. We augment our block structure by adding one more field to it: The time r at which it was generated. Our block format now looks like this: $B = s \,\|\, \overline{x} \,\|\, \mathsf{ctr} \,\|\, r$.

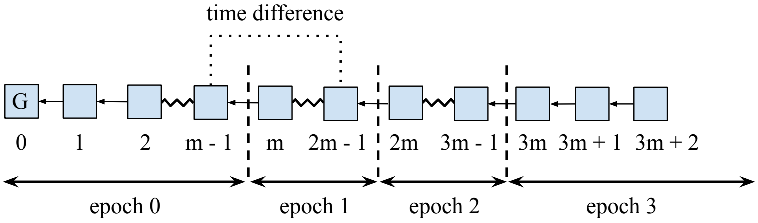


Figure 8.9: Epochs of m blocks in which difficulty remains fixed. Squiggly lines illustrate that some blocks have been omitted from the figure.

Given this new block format, we can now adjust our difficulty at the first block of every epoch, namely when a block's height is divisible by m. For each i with $i \equiv 0 (mod m)$ marking the beginning of the epoch, we look at $\mathcal{C}[i].r$, the timestamp reported at the beginning of the next epoch, and compare it against the timestamp $\mathcal{C}[i-m].r$, the timestamp at the beginning of the previous epoch. The difference $\mathcal{C}[i].r - \mathcal{C}[i-m].r$ marks the *actual* time the epoch reportedly took. This must be compared to the value ηm, the *expected* time the epoch *should* have taken. Calculating the ratio between the two gives us the target for the j^{th} epoch:

$$T_j = T_{j-1} \frac{\mathcal{C}[jm].r - \mathcal{C}[(j-1)m].r}{\eta m}$$

This process is known as *difficulty adjustment* or *target recalculation*. If the actual time is larger than the expected time, the difficulty is decreased. If the actual time is smaller than the expected time, the difficulty is increased.

Lastly, we need to ensure that the r written on a block is somewhat accurate. We do this by ensuring that the chain is *temporally consistent*, which involves performing the following checks. Given a chain $\mathcal{C}$:

1. For all i, check that $\mathcal{C}[i].r \leq \mathcal{C}[i+1].r$, i.e. timestamps are non-decreasing.
2. Check that $\mathcal{C}[-1].r$ is not in the future by comparing it to the local clock.

The last check may cause a block to be rejected if the receiving party has a clock that is slightly off. This is fine, because the party will receive the block again later from others as it is rebroadcast through the network. These two checks are added to the chain validation procedure.

These two simply checks make the timestamps reported on the chain quite accurate. A minor adversary cannot lie about timestamps significantly. Due to the unfavorable conditions of the Nakamoto race against the minor adversary, the adversary cannot mine on top of a very old block. The adversary can only mine on top of a recent block. Specifically, it only makes sense for the adversary to mine on top of a block in $\mathcal{C}[-k:]$, because any attempt to mine on top of older blocks will be in vain if k Common Prefix holds. This means that the adversary has a very small wiggle room about fraudulently reporting timestamps: She can only place her timestamp between $\mathcal{C}[-k].r$ and the current timestamp. This means that the adversary can have a small adverse effect on the difficulty recalculation, but this is not particularly significant. These small timing variations are not very important, because such errors can already happen in the system as it is. One reason for this is the stochastic nature of proof-of-work. The second reason is that the minor adversary can already cause the mining rate to fluctuate by close to a factor of 2 by simply turning her mining power on and off. Therefore, small timing variations in difficulty recalculation are not very important. It's the big picture that counts. This means that the value η should be calculated to be large enough so that, even under these small fluctuations (which can range to more than a factor of 2) should not affect safety.

At this point, it should be clear that η must be chosen to be significantly larger than Δ to achieve a good density of convergence opportunities. We will calculate good values for η (or, equivalently, its inverse $f = \frac{1}{\eta}$) precisely in the next few chapters.

Problems

TBD

Further Reading

While it is folklore belief that a cryptocurrency can remain incentive-compatible when payouts consists only of block rewards and no fees, this belief is actually false. This was studied in the paper *On the Instability of Bitcoin Without the Block Reward* [8].

The field of optimizing miner strategies was pioneered by two works, *Clockwork Finance* [5] and *FlashBoys 2.0* [10]. These gave rise to the concept of *MEV*, or Miner Extractable Value, and gave rise to a whole industry of *searchers*, *builders*, and *proposers*, allocating different roles to the different portions pertaining to this optimization problem. The most popular platform in the field currently concerning itself with such optimizations is *FlashBots*.

The taxonomy of various wallets was studied in *A Taxonomy of Cryptocurrency Wallets* [21], where various types of wallets are described. The security of hardware wallets is analyzed in *A Formal Treatment of Hardware Wallets* [4].

The way keys are generated from seeds depends on the particular blockchain. The first blockchain to use seeds was Bitcoin. The structure that Bitcoin uses is complex and is described in technical detail in the standards BIP32 [35], BIP39 [28], and BIP44 [27].

The target recalculation formula given in this chapter is a simplified version of the real formula. The real formula contains a *clamping parameter*. Constructing a variable difficulty protocol is not a simple matter. There are insidious attacks that can appear. One prominent such attack is due to Bahack [6]. For the detailed analysis of the security of the variable difficulty protocol, consult the seminal work *The Bitcoin Backbone Protocol with Chains of Variable Difficulty* [15].

Chapter 9

Accounts DRAFT

In our treatment so far, we have dealt with the UTXO model. There, the state that we maintained was the UTXO set, and transactions expressed a *transition* from one UTXO set to the next by updating it appropriately (adding and removing elements from it). By ordering the transactions into a ledger using a blockchain, we achieved consensus on a mutually shared state between mistrustful participants, among which an honest majority assumption holds. The machinery we have created solves the *State Machine Replication* problem.

However, the machinery that we used to achieve consensus, namely the block and the chain, was quite separate from the semantic interpretation of the transactions that are confirmed in the produced ledgers. During the block validation process, we asked the full nodes to validate transactions by ensuring that the UTXO set can appropriately transition from one state to the next.

Certain aspects of the validation concern the *consensus layer*, which pertains to checking the consensus contents of blocks: The ancestry relation, the genesis block, the proof-of-work, the timestamps, the difficulty adjustment mechanism. These checks pertain to the *structural validation* of blocks.

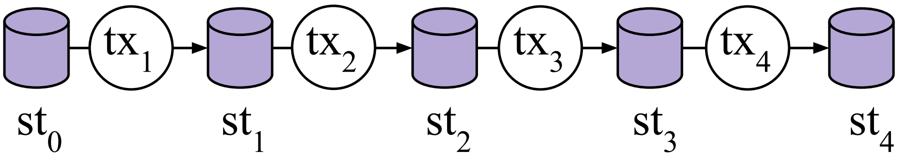


Figure 9.1: Transitioning state by applying transactions.

9.1 Accounts Model

9.1.1 Accounts Model Compared with UTXO Model

Recall the previous UTXO model: we store a set of unspent transaction outputs (UTXOs). When a transaction occurs, UTXOs corresponding to the transaction's inputs are removed and UTXOs corresponding to the transaction's outputs are added into the UTXO set to produce a new UTXO set, as shown in Figure 9.2.

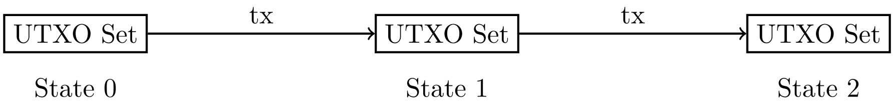


Figure 9.2: State transitions in the UTXO model

The accounts model is another model of transactions. In the accounts model, transactions contain 1) the account that sends balance (from), 2) the account that receives balance (to), 3) the value of the transaction (val), 4) the transaction fee (fee), and 5) the signature on the transaction (σ).

From	To	Val	Fee	σ

Figure 9.3: Structure of a transaction in the accounts model

For the accounts model, the state is maintained by accounts (public keys) and balances, as shown in Figure 9.4 and Figure 9.5.

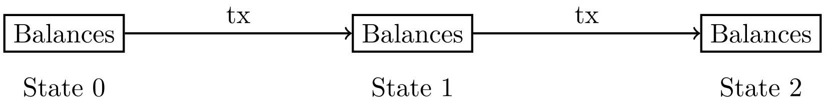


Figure 9.4: State transitions in the accounts model

The function that takes in a state and a transaction and returns a new state is called a **transition function**. It has a general form of:

$$\delta(st, tx) = \begin{cases} st' & \text{if } tx \text{ valid w.r.t. } st \\ \perp & \text{otherwise} \end{cases} \tag{9.1}$$

Specifically, the transition function of UTXO model is given by:

$$\delta_{UTXO}(st, tx) = \begin{cases} st \setminus tx_{in} \cup tx_{out} & \text{if } tx \text{ valid w.r.t. } st \\ \perp & \text{otherwise} \end{cases} \tag{9.2}$$

Where tx_{in} is the set of unspent outputs in the "inputs" field of tx, and tx_{out} is the set of newly generated outputs in the "outputs" field of tx.

The transaction validation process of UTXO model is

- Check σ
- Check conservation

Balances State

Alice	5 bu
Bob	100 bu
Dionysis	1 bu

Figure 9.5: Balance State

- Check inputs are in *st*

Similarly, the transition function of the accounts model could be written as:

$$\delta_{acc}(st, tx) = \begin{cases} st' & \text{where } st'[tx.from] = st[tx.from] - tx.value, \\ & st'[tx.to] = st[tx.to] + tx.value, \text{ if } tx \text{ valid w.r.t.} \\ & st \\ \perp & \text{otherwise} \end{cases} \tag{9.3}$$

The transaction validation process of the accounts model is

- Check σ

- Check $st[tx.from] \geq tx.value$

9.1.2 Accounts Model Replay Attack

Here comes a problem. In the accounts model, if the same transaction is sent to the network twice, should the second transaction be included or not? For example, one morning, Bob bought a cup of coffee from Starbucks. The next morning, he bought a cup of coffee again. These two transactions have the same fields, even the signature.

If the network decides to accept transactions that are the same, the following replay attack could happen: an adversarial coffee shop could replay the transaction even if Bob didn't buy a coffee. However, if the network decided not to include transactions that are same, then Bob could only buy a coffee once.

The solution is to add a nonce field to transactions. The nonce is an 256-bit integer per source account which is incremented every new transaction. The transaction structure now looks like Figure 9.6.

From	To	Val	Fee	**nonce**	σ

Figure 9.6: Structure of tx in accounts model

And therefore, while validating transactions, an additional step of validating the nonce should be included. Transactions in which the nonce has already been used is rejected. This means that the state contains the current nonce for each account, in addition to the balance. The state transition function must also update the nonce for the "from" account of the transaction.

A side by side comparison between the two models of transactions is shown in Figure 9.7.

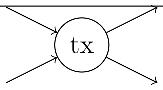
	UTXO	Accounts
Real System	Bitcoin	Ethereum
Transaction tx		From | To | Val | Fee | **nonce** | σ
Transistion δ	Remove consumed outputs and add produced outputs	Update balances $st'[from] := st[from] - value$ $st'[to] := st[to] + value$
Validation	Signature, Law of Conservation, Inputs exist in st.	Signature, Sufficient balance, Nonce unique.
Genesis State	$\emptyset$	$\{\}$

Figure 9.7: Side by Side Comparison of Two Models

9.2 State Machine Replication

We talk briefly about State Machine Replication (SMR). A state machine consists of a state, inputs and a transition function. The machine has an initial state. Based on its inputs and the state transition function, the machine updates its state. In SMR, multiple nodes in the network run a state machine in a distributed manner. The term "replication" signifies that each node in the network maintains the state of the machine and runs its transition functions locally. The goal of SMR is that each node runs the same set of state transitions and in the same order so that there is agreement or consensus on the state of the machine.

A blockchain can be considered as a distributed replicated database. A blockchain can help us run SMR. We have seen two examples of state machines that the blockchain can run — the accounts model and the UTXO model. In both cases, there is a state st, state transition functions δ, and inputs (which are transactions in this case). The initial state is specified by the genesis state.

9.3 Light Clients

How to run a blockchain node efficiently? Efficiency has multiple dimensions: storage, communication, and computation. For most application scenarios, the blockchain node has limited resources. For example, if we store all the data of the chain, it would take gigabytes of storage. Validating every transaction in the network would be very heavy work for a phone. Therefore, a light client is needed for these resource-limited nodes.

9.3.1 Storage Efficiency: Merkle Trees

For a light client, it is better to save the data at a server and retrieve data at usage. However, we need to prove the integrity of the retrieved data. Hash functions are useful in this case. Suppose that we wanted to store a file on a server and verify that we receive the correct file from the server. We could hash the file and store the hash (checksum) locally. When we request files from the data server, we validate the checksum of the retrieved file to verify that it is the exact file we saved on the server. However, this requires clients to retrieve the entire file to validate its integrity even if only a 1 kilobyte chunk is needed.

We can also split the file into chunks and hash each chunk. This reduces the communication complexity: clients only need the chunk to be transferred. However, this requires more hash key storage for the client. The client needs to store one hash per chunk, making the storage complexity linear in the size of the file. There is a trade off between communication complexity and storage complexity: with large chunks, comes high communication complexity and with small chunks, comes high storage complexity.

Our goal is to achieve low storage and low communication. Specifically, storage with $O(1)$ complexity and communication with $O(\log n)$ complexity where n is the number of chunks of the file. And this is done with a data structure called Merkle tree.

9.3.2 Data Structure: Merkle Tree

Files are split into n data chunks.

$$D : D[0], D[1], ..., D[n-1]$$

A binary tree of depth μ is created, where there are $2^\mu = n$ leaves (for simplicity, assume that n is a power of 2). Each node in the binary tree stores a hash h which is the hash of its children concatenated.

$$h \coloneqq H(h[\text{left}] \parallel h[\text{right}])$$

Nodes on the leaves store the hash of the corresponding data chunk. The client stores the Merkle tree root (MTR) h_ϵ. When a data chunk is requested, the server sends the data chunk, along with every sibling hash value to the clients as shown in Figure 9.8. For example, when data chunk at index j is requested, the server sends $D[j]$, π_0, π_1, π_2, and π_3 to the client. The client walks from the received data chunk all the way up to the root to check if the hash values are intact. From calculating e_0 by hashing the data chunk, to the top level $e_{\mu+1}$, the client calculates $e_k = H(e_{k-1} \parallel \pi_{k-1})$ or $e_k = H(\pi_{k-1} \parallel e_{k-1})$(left child first). In this example, the client computes the values $e_0 = H(D[j])$, $e_1 = H(e_0 \parallel \pi_0)$, $e_2 = H(\pi_1 \parallel e_1)$, $e_3 = H(e_2 \parallel \pi_2)$, and $e_4 = H(\pi_3 \parallel e_3)$ and then compares e_4 with h_ϵ.

With this data structure, the data transferred is a list of π values and the data chunk of fixed size, which gives $|\pi| = O(\log n)$ succinct communication and $O(1)$ constant storage.

The Merkle tree structure is described by the functions

$$\text{compress}(D) \to h_\epsilon, \tag{9.4}$$

$$\text{prove}(D, j) \to \pi, \text{ and} \tag{9.5}$$

$$\text{verify}(h_\epsilon, d, j, \pi) \to \begin{cases} \text{true} & \text{if valid} \\ \text{false} & \text{otherwise} \end{cases}. \tag{9.6}$$

The correctness of the Merkle tree is specified as:

$$\forall D, \forall j, \text{verify}(\text{compress}(D), D[j], j, \text{prove}(D, j)) = \text{true} \tag{9.7}$$

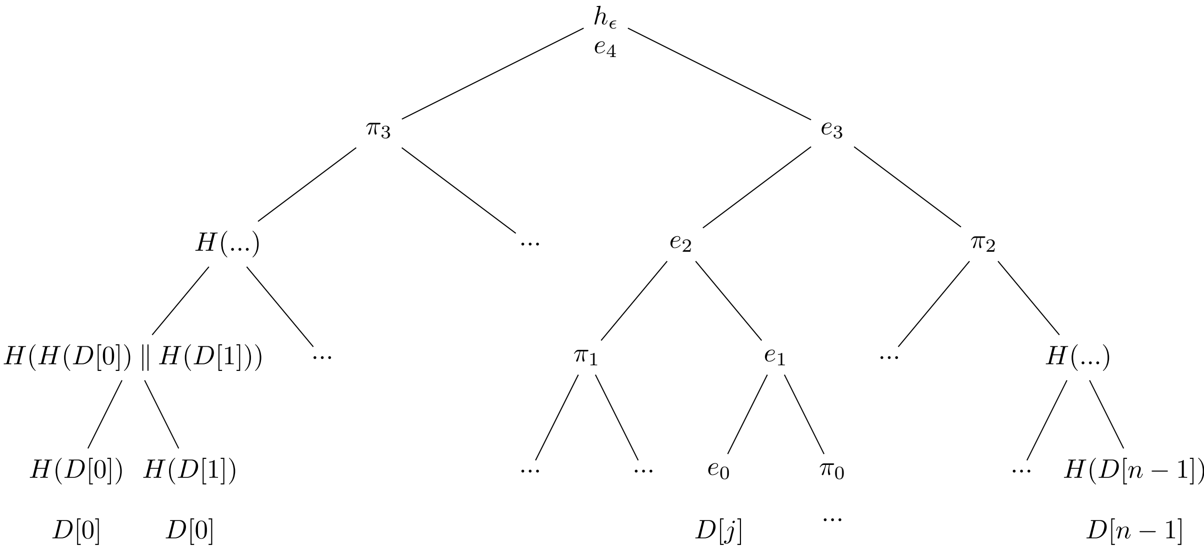


Figure 9.8: Merkle Tree

9.3.3 Security of Merkle Trees

MT-security means that if the client outputs true after verifying the received data chunk and proof, then the received data must be the same data that was originally stored. To define security of Merkle trees formally, we create the following game that lets an adversary try to break the protocol.

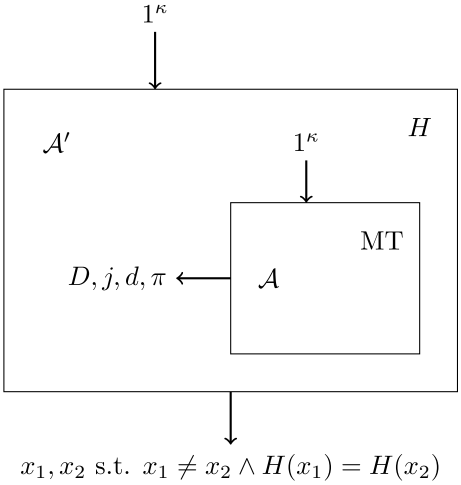
```
$\text{MERKLE}_{\mathcal{A}}(\kappa):$
    $D, \pi, j, d \leftarrow \mathcal{A}(1^\kappa)$
    return  $\text{verify}(\text{compress}(D), d, j, \pi) \land d \neq D[j]$
```

Our goal is to prove that

$$\forall \text{ PPT } \mathcal{A} : Pr[\text{MERKLE}_{\mathcal{A}}(\kappa) = 1] \leq negl(\kappa)$$

Theorem 9. *Let H be a collision-resistant hash function. Then Merkle trees constructed with H are MT-secure.*

Proof. Suppose for contradiction, $\mathcal{A}$ breaks MT-security. We will construct an adversary $\mathcal{A}'$ that breaks collision-resistance of H.



We use e for the hash value calculated by the client, h for the expected hash value in the correct Merkle tree, and π for the hash values returned by the server.

Consider the event that $\mathcal{A}$ succeeds, i.e. $\text{verify}(\text{compress}(D), d, j, \pi) = 1 \wedge d \neq D[j]$.

$\mathcal{A}'$ works as follows:

Given that $\mathcal{A}$ succeeds, the returned list of π is used to calculate hashes of the nodes to verify the returned data chunk, which involves calculating e values by concatenating the children of the nodes. The hash of the root is the same ($h_\epsilon = e_{top}$) and the hash of the data chunk is different ($e_0 \neq h_0$). (If not, $\mathcal{A}'$ has already found a collision because different data chinks have the same hash.) Therefore, there must exists a node, some level k in the tree, such that $e_k = h_k$ but its children e_{k-1} or π_{k-1} not equal to the expected h values. (These must exist because roots are the same, but leaves are different). In this case, we have two different inputs that hash to the same value.

Then, adversary $\mathcal{A}'$ returns children of e_k from the verifier tree at level k as x_1 and children of h_k from real tree at the corresponding position as x_2. Then, x_1 and x_2 satisfy $x_1 \neq x_2 \wedge H(x_1) = H(x_2)$.

Therefore, the probability of breaking the Merkle tree protocol is the same as the probability of breaking the collision-resistant hash function H.

$$Pr[\text{MERKLE}_{\mathcal{A}}(\kappa) = 1] = Pr[\text{Collision}_{\mathcal{A}'}(\kappa) = 1]$$

But $Pr[\text{Collision}_{\mathcal{A}'}(\kappa) = 1]$ is negligible by assumption, which means the probability of breaking Merkle tree protocol is also negligible.

□

Chapter 10

Light Clients DRAFT

10.1 Motivation

Our current model has three main scalability limitations: storage requirements (full nodes store the entire chain, $\sim$ 1TB in Bitcoin), communication requirements (full nodes broadcast, request and download every transaction and block submitted to the network), and computation requirements (full nodes validate all incoming blocks and transactions, compute the UTXO set after each block, etc.).

We seek to design a light client (also referred to as a light node) such that it needs less storage, communicates less with the network, and requires less computation power. Our design will need only $\sim$ 100MB of storage, will only download transactions pertinent to our own address, and will not validate all transactions in the transaction graph.

Before designing such a node, we also wish to distinguish between full nodes and miners: full nodes are peers running our protocol (validating blocks and transactions, gossiping new objects, adopting longest chain, etc.), where miners perform all these tasks in addition to querying for new blocks.

10.2 Definition

To guide our discourse, we will define light clients as nodes on the blockchain network which are able to verify payments to and create transactions from a specified address, but without downloading the entire blockchain (specifically the transaction graph), and without validating said transaction graph in its entirety. Further, we will assume that light clients connect only with full node peers, rather than with other light clients, to ensure the availability of information which is available to any other full node in the network.

10.3 Header Chains

To solve the storage problem, we introduce block headers. We want our light clients to be able to interact with the network without downloading the entire blockchain, and block headers provide a way of maintaining chain virtues, while only needing to download a subset of the transaction graph.

Instead of a chain of blocks, each containing a vector of transactions, header chains are chains of block headers. Block headers are of the format: $s||x||\text{ctr}$, where s refers to the hash of the previous block header, x refers to the root of the merkle tree of all transactions in $\bar{x}$ (the transactions contained in the block), and ctr is the nonce discovered to satisfy the PoW equation.

While this solves our storage problem, it is not immediately clear how a header chain can be used to verify transactions (as the header contains no actual transaction hashes). However, as the block header contains x, light nodes may request merkle proofs π from their full node peers for the inclusion of pertinent transactions in the chain, which no PPT adversary can create for a transaction which was not included. Therefore, header chains provide the same guarantees with regards verification of transaction inclusion.

10.3.1 Block Validation

In order to verify a block in a block header chain, we modify the process slightly such that a full node must now

1. download the new block header
2. validate the header PoW and ancestry
3. download the block body
4. check the block body validity—transaction set validity and merkle tree validity.

10.3.2 Benefits

Header chains provide mitigate all three of our limitations listed above.

Unlike blocks, whose size depends on the number of transactions it contains, block headers have an lower-bounded size of 3κ, each κ coming from each hash being concatenated to make the block header. This enables the block header to be significantly more compact than a block, and solves the problem of storage for light clients which don't wish to download the entire transaction history. We say that the size is lower bounded because implementations may choose to include additional metadata in the block header. Thus, the minimum storage requirements for a light client is reduced from $O(h|\bar{x}|)$ to $O(h\kappa)$, where h is the height of the longest chain.

Since each block header is much smaller than an entire block, there are also computational benefits for both light and full nodes, as well as the network as a whole. This is because the computational complexity of calculating $H(B)$ is $O(|B|)$. As block headers are much smaller than blocks, this provides significant computational benefits in verifying PoW for both light and full nodes. This also allows light nodes to verify PoW without verifying the validity of the transactions in $\bar{x}$, which is left to the full nodes to compute. Finally, as $H(B)$ takes $O(3\kappa)$ rather than $O(|\bar{x}|)$ computational power to compute, full nodes are no longer incentivized to mine smaller blocks, which would have previously increased their hash rate.

In addition to these benefits, as light nodes only need to request the block header chain and a subset of all transactions, rather than the entire block chain and transaction graph, header chains allow light nodes to make significantly fewer requests to their full node peers, and for those requests to result in much smaller and shorter responses.

10.4 Making Payments

A core function of the light client is interacting with the block chain to make and receive payments concerning only itself. Thus, light clients interact with a subgraph of the network and rely on full nodes to receive the UTXOs belonging to it. A light client can then use these UTXOs to pay other addresses, sign such a transaction, and broadcast it to the network for validation and inclusion in the chain.

To build this subgraph of UTXOs, light clients must be able to make requests to full node peers for all UTXOs in the transaction graph that are owned by a given address (in this case, the address-of-interest of the light client itself). Furthermore, they must request a merkle proof from these peers for the inclusion of these transactions in the chain, which we know cannot be counterfeited by a PPT adversary. This reliance on full node peers to provide the transaction sub-graph poses no security risks under the non-eclipsing assumption, as an honest node will provide all transactions-of-interest to a requesting peer.

10.5 Block Header Validation

Light clients do not validate blocks. Instead, they validate only the block headers, and follow only valid header chains, without ever downloading the block itself. As a result, for a light client to verify and validate an incoming block header, the validation steps are a further modification (compared to the full node) of the usual process. Specifically, light clients must do the following.

1. download the new block header
2. validate the header PoW and ancestry
3. request the subset of transactions relating to their address (in case any new pertinent transactions are in this new block)
4. request proofs of inclusion for any of these relevant transactions which are in the new block (older transactions should already have been verified).

10.5.1 Transaction Validation

While light nodes can download header chains and verify each header (PoW and ancestry checks), they cannot validate transactions constituting the block corresponding to that block header. This is because light nodes are unaware of the entire transaction graph, and more specifically the current UTXO set. Without the UTXO set, a light node is unable to determine whether a transaction's inputs are valid. Furthermore, light nodes do not download transactions which are not pertinent to them, which means they are also unaware of the validity of said transactions. We will now see how an adversary might exploit this fact.

10.5.2 Local Chain Security

Common Prefix is still guaranteed for honest-majority networks (specifically, networks where a majority of the mining power is honest), as the PoW requirements on header chains ensure that each header in the chain is a successful query. As a result, finding a valid block header is just as difficult as finding a valid block, so no minority adversary can outpace the honest nodes in the network. However, there are some security risks for the light client in the event of a dishonest majority.

While no PPT adversary can create a proof π that some transaction is included in the chain when it is not, a dishonest *majority* might create an invalid chain (containing double spends, invalid coinbase transactions, etc.) which grows longer than the longest honest chain. While no full node will accept the adversary's invalid chain, light clients validate only PoW and ancestry of the block header, and therefore would accept such an invalid chain. Preventing such an adversarial majority attack would require verifying the entire transaction graph. However, since light nodes do not contribute blocks to the chain, there are no security concerns for the network at large in this situation, and any invalid transactions submitted by the compromised light client would not be validated nor included in the competing honest chain. So long as the adversary's majority is transient, the light client's risks are also transient.

10.5.3 Privacy

As a light clients requests transactions relating only to its own address, it must reveal its public key to the full node(s) from whom it is receiving the transactions. This is a privacy compromise made for the sake of efficiency, although there are some ways to mitigate this risk (e.g. bloom filters, as used in the Bitcoin network).

10.5.4 Full Node Ramifications

On the topic of light clients making requests to their full node peers, honest full nodes must be able to provide all transactions which relate to the requested address. Therefore, they must store some type of mapping from public key addresses to sets of pertinent transactions, adding some complexity to the full nodes' protocol implementation.

10.6 Light Miners and the Quick Bootstrap Protocol

We will now explore whether light nodes can be miners in the network, and what modifications to our protocol are necessary for enabling this.

10.6.1 Mining as a Light Client

A naive approach to light client mining would be to bootstrap as normal, and then accept transactions into a mempool before including them in a block and broadcasting that block to the network.

However, mining as a light client is complicated by light clients' inability to validate transactions. The necessary information for validating transactions is the transaction graph of the entire blockchain, or more specifically the UTXO set after execution of the most recent block. Normally, the UTXO set is computed in the process of validating the full transaction graph of the blockchain. But, to allow for light clients (or nodes which do not download the full transaction history of the chain) to mine, we modify our block header structure to be $s||x||ctr||st$. We define *st* as the merkle root of the merkle tree made of all UTXOs in the UTXO set after executing the block, which we will call $\bar{st}$. That is, *st* is the merkle root of the merkle tree containing all necessary information for validating transactions which come after the execution of the current block.

This state commitment *st* should also be validated by full nodes, in the process of validating the block body $\bar{x}$. This can be done either by downloading the full state $\bar{st}$ from a peer and comparing with the result from one's own execution, or by executing the transactions $\bar{x}$ and then calculating a merkle root based on the resulting UTXO set, which is compared with *st*.

We will use this new block header definition in outlining our quick bootstrap protocol for light miners.

10.6.2 Quick Bootstrap Protocol

We can construct a quick bootstrap protocol by making use of the state commitment *st* in each block header. This commitment in a given header can be used to download and verify the UTXO set after the execution of the corresponding block. By relying on *st*, a light miner can avoid downloading $\bar{x}$ for all blocks which were mined before they joined the network, while still being able to validate transactions based off of the most recent state.

Therefore, our quick bootstrap protocol differs from the regular full node bootstrap protocol in that it does not have the light miner validate the entirety of the transaction graph, but only the portions of the graph which are included after the light miner boots. Further, only transactions included in blocks after the light miner boots need to be executed, greatly simplifying the boot process.

The protocol is as follows:

1. download and validate the header chain
2. download the state $\bar{st}$ of the chain tip and validate the merkle tree
3. accept transactions into the mempool and form valid transactions into a block (validated using the state $\bar{st}$)
4. mine the header of this block off of the chain tip of the longest chain
5. when new blocks are announced, validate as normal (download the block header, validate, download $\bar{x}$, validate, etc.).

We can see that this protocol does not require the downloading of any portion of the transactions $\bar{x}$ of any block already included in the chain at the time of booting, in keeping with our usage of "light" with respect to light clients.

10.6.3 Light Miner Security

Since light miners do not validate any transactions that were on the blockchain before they booted, they are liable to begin mining off of an invalid chain tip, potentially contributing their hashing power to the adversary's chain. We rely upon the Common Prefix guarantee of the chain to avoid this, by starting with block $C[-k]$ to begin our full validation of the blockchain—where we accept $C[-k]$ as being the tip of a valid chain—and validating the k most recent blocks (downloading and validating $\bar{x}$) of the longest chain which follow. This ensures that the light miner accepts only valid longest chains, by starting with the end of the commonly valid portion of the chain.

Properties of Full Node, Full Miner, and Light Node			
Properties	Miner	Full Node	Light Node
Download Header	✓	✓	✓
Download Body	✓	✓	
Create Blocks	✓		
PoW Check	✓	✓	✓
Check Tx Validity	✓	✓	
Size	$\sim$ 1TB	$\sim$ 1TB	$\sim$ 100MB
Honest Majority	$\frac{n-t}{n}$		

Chapter 11

Security in Earnest I DRAFT

We have spent the previous chapters developing our intuition about blockchain systems and why they work. In this chapter and the next couple of chapters, we will formalize the notion of security and prove that blockchains are secure. We start by making a few things more precise, beginning with our hash function and the notion of time. Once we have specified these, we will describe the environment that the parties operate in. This will make precise the concept of a Sybil adversary and the non-eclipsing assumption, among other things. Next, we will write out the protocol of the honest parties as exact pseudocode. We will move on to formally prove the three chain virtues we explored intuitively: Chain Growth, Common Prefix, and Chain Quality. Finally, we will formally show that ledger virtues follow from chain virtues.

11.1 Random Oracle

In Chapter 4, we argued that the hash function behaves like a random oracle and that $Pr[H(B) \leq T] = p = \frac{T}{2^k}$. Let us now make the random oracle model more precise. We want the hash function to return a uniformly randomly chosen κ-bit value whenever it is invoked with a fresh input. However, in order for it to be a function, we want it to return the *same* value when it is invoked again with the same input. The random oracle is a shared functionality among all honest parties: If two different parties invoke it, the oracle must answer consistently. This is illustrated in Algorithm 15. This consistency between different parties and multiple queries is necessary so that if a party successfully mines a block, every other party will be able to verify that this mining was successful.

Algorithm 15 The Random Oracle model.

```
$\mathcal{T} \leftarrow \{\}$
function $H(B)$
    if $B \notin \mathcal{T}$ then
        $y \xleftarrow{\$} \{0,1\}^\kappa$
        $\mathcal{T}[B] \leftarrow y$
    end if
    return $\mathcal{T}[B]$
end function
```

The oracle keeps track of the randomnesses it has generated in a dictionary $\mathcal{T}$. If the key B queried has been queried before, the random oracle returns the cached value $\mathcal{T}[B]$. Otherwise, the oracle generates a uniformly random κ-bit string, caches it in the $\mathcal{T}$ dictionary for future use, and returns it.

11.2 Synchrony

Until now, we considered time as continuous. We will now follow a synchronous model where time is broken up into rounds, each round lasting a discrete time Δ. Additionally, we will consider the network delay to be precisely Δ. This implies that any message broadcast by an honest party at some point during the round r will be received by **every honest party** at the start of round $r + 1$, all at exactly the same moment.

This model synchronizes the arrival of messages such that they all arrive at the boundary of each round, and lets us discretize time by eliminating complexities of network distance and speed. Furthermore, the execution of our network entities is simplified to a "lockstep execution" model, where we can easily predict when any node will receive a given message, which will be precisely one round after it was broadcast.

11.3 The Simulation Environment

In order to rigorously define properties of our blockchain and give security proofs, we need to precisely define how we will simulate our environment: the setup and how each round happens.

Algorithm 16 The environment and network model running for a polynomial number of rounds $\mathsf{poly}(\kappa)$.

```
1:  $r \leftarrow 0$
2:  function $\mathcal{Z}^{n,t}_{\Pi,\mathcal{A}}(1^\kappa)$
3:      $\mathcal{G} \xleftarrow{\$} \{0,1\}^\kappa$                                   ▷ Genesis block
4:      for $i \leftarrow 1$ to $n-t$ do                              ▷ Boot stateful honest parties
5:          $P_i \leftarrow$ new $\Pi^{H_{\kappa,i}}(\mathcal{G})$
6:      end for
7:      $A \leftarrow$new $\mathcal{A}^{H_{\kappa,0}}(\mathcal{G},n,t)$                     ▷ Boot stateful adversary
8:      $\overline{M} \leftarrow []$                                           ▷ 2D array of messages
9:      for $i \leftarrow 1$ to $n-t$ do
10:         $\overline{M}[i] \leftarrow []$                       ▷ Each honest party has an array of messages
11:     end for
12:     while $r < \mathsf{poly}(\kappa)$ do                                      ▷ Number of rounds
13:         $r \leftarrow r+1$
14:         $M \leftarrow \emptyset$
15:         for $i \leftarrow 1$ to $n-t$ do                      ▷ Execute honest party $i$ for round $r$
16:             $Q \leftarrow q$            ▷ Maximum number of oracle queries per honest party
    (Section 2)
17:             $M \leftarrow M \cup \{P_i.\mathsf{execute}^H(\overline{M}[i])\}$       ▷ Adversary collects all messages
18:         end for
19:         $Q \leftarrow tq$                          ▷ Max number of Adversarial oracle queries
20:         $\overline{M} \leftarrow A.\mathsf{execute}^H(M)$                   ▷ Execute rushing adversary for round $r$
21:         for $m \in M$ do                          ▷ Ensure all parties will receive message $m$
22:             for $i \leftarrow 1$ to $n-t$ do
23:                 assert($m \in \overline{M}[i]$)                          ▷ Non-eclipsing assumption
24:             end for
25:         end for
26:     end while
27: end function
```

11.3.1 A Simplification: Quantize Time

Notice that we are running the simulation in rounds (line 12). In the real world, time is continuous, however by breaking down the simulation into short rounds, it makes it much easier to define and prove security properties of our blockchain. Furthermore, it makes it easier to define the properties of our adversary as seen below.

11.3.2 Rushing Adversary

In this environment, we are assuming a Rushing Adversary. This is because every round (lines 12-26), we first simulate the honest parties independently - they do not see the messages produced by each other that round - (lines 15-19), collect all the messages on the gossip network (line 17), then run the adversary with all the gossiped messages (line 20).

11.3.3 Sybil Adversary & Non-Eclipsing Assumption

The adversary sees the messages gossiped by each honest party before the other honest parties. The adversary then has the power to manipulate what messages the honest parties will see next round in the following way

1. The adversary can inject new messages
2. The adversary can reorder messages
3. The adversary can introduce disagreement
4. The adversary cannot censor messages (lines 20, 21, 22)

The fourth point is due to the Non-Eclipsing Assumption: since there is a path of honest parties between any two honest parties, and each honest party follows the algorithm detailed in section 3, we know that an honestly produced message will be propagated to all honest parties on the next round.

11.4 Random Oracle Model

In the simulation algorithm, for both the honest parties and adversarial parties we write $\mathsf{execute}^H$ (lines 17, 20). This means that we model the hash function as a random oracle and give both the honest and adversarial parties Black-Box access to the oracle. This means that for any ”new” input, the output is queried uniformly at random from the output space (line 10) and returned. Furthermore, when an input is queried for the first time, it is stored in a cache (stored in $\mathcal{T}$), therefore if the same input is later queried, the value is looked up in the cache and returned (line 12). Black-Box access means that the parties do not have access to the cache or the random sampling algorithm. They can only submit a query x and receive a response $\mathcal{T}[x]$.
Secondly, in order to model the hash rate, we give each party a maximum number of oracle queries per round. Each honest party receives q queries, and the adversary receives qt queries. (line 3, 9).

Algorithm 17 The Hash Function in the Random Oracle Model

```
$r \leftarrow 0$
$\mathcal{T} \leftarrow \{\}$                          ▷ Initiate Cache
$Q \leftarrow 0$                                        ▷ $q$ for honest parties, $qt$ for adversary
function $H_\kappa(x)$
    if $x \notin \mathcal{T}$ then                      ▷ First time being queried
        if $Q = 0$ then                                 ▷ Out of Queries
            return $\perp$
        end if
        $Q \leftarrow Q - 1$
        $\mathcal{T}[x] \xleftarrow{\$} \{0,1\}^\kappa$    ▷ Sample uniformly at random from output space and store in Cache
    end if
    return $\mathcal{T}[x]$                             ▷ Return value from Cache
end function
```

11.5 Honest Party Algorithm

Below is a class of algorithms belonging to an honest party.

The first algorithm is a constructor.

The second algorithm is used to simulate every honest party that mines during each round of the protocol. In this simulation, every honest party follows the longest chain rule, at the beginning of each round they adopt the longest, valid chain (line 8). If they learn about a new chain that is longer then their current one, they gossip it (line 9,10). The honest party then tries to mine a block using the transactions in their mempool (line 12, 13). If a block is successfully mined, the honest party will gossip it.

The third algorithm is used to extract all transactions from a blockchain. This is useful when validating a new chain as we need to check all transactions starting from the genesis state to ensure that there are no invalid transactions and to maintain an up-to-date UTXO set.

Algorithm 18 The honest party

```
$\mathcal{G}$
$\mathcal{C} \leftarrow [\,]$
function Constructor($\mathcal{G}'$)
    $\mathcal{G} \leftarrow \mathcal{G}'$                                   ▷ Select Genesis Block
    $\mathcal{C} \leftarrow [\mathcal{G}]$                                  ▷ Add Genesis Block to start of chain
    $\mathsf{round} \leftarrow 1$
end function
function Execute($1^\kappa$)
    $M \leftarrow \emptyset$
    $\tilde{\mathcal{C}} \leftarrow \text{maxvalid}(C, \bar{M}[i])$         ▷ Adopt Longest Chain in the network
    if $\tilde{\mathcal{C}} \neq \mathcal{C}$ then
        $M \leftarrow \{\tilde{\mathcal{C}}\}$
        $\mathcal{C} \leftarrow \tilde{\mathcal{C}}$
    end if
    $x \leftarrow$ Input()                                                  ▷ Take all transactions in mempool
    $B \leftarrow \mathsf{PoW}(x, \tilde{\mathcal{C}})$
    if $B \neq \perp$ then                                                  ▷ Successful Mining
        $\mathcal{C} \leftarrow \mathcal{C} \parallel B$                    ▷ Add block to current longest chain
        $M \leftarrow M \cup \{\mathcal{C}\}$
    end if
    $\mathsf{round} \leftarrow \mathsf{round}{+}1$
    return $M$
end function
function Read
    $x \leftarrow [\,]$                                                     ▷ Instantiate transactions
    for $B \in \mathcal{C}$ do
        $x \leftarrow x \parallel \mathcal{C}.x$                            ▷ Extract all transactions from each block in the chain
    end for
    return $x$
end function
```

11.6 Proof-of-Work

The algorithm below is run by miners to find a new block. Notice that all parties (adversarial and honest) have a maximum number of q queries per round. This is to model the hash rate of parties. Furthermore, we construct a block as the concatenation of the previous block s, the transactions x, and the nonce ctr. For a block to be mined successfully, we require that $H(B) \leq T$, where T is the mining target. Due to the size of the space $\{0,1\}^\kappa$, the probability of two parties mining with the same nonce is negligible, therefore we may assume that there are no “nonce collisions”.

Algorithm 19 The Proof-of-Work discovery algorithm

```
function $\textsc{pow}_{H,T,q}(x, s)$
    $ctr \xleftarrow{\$} \{0,1\}^{\kappa}$                    ▷ Randomly sample Nonce
    for $i \leftarrow 1$ to $q$ do                ▷ Number of available queries per party
        $B \leftarrow s||x||ctr$                                  ▷ Create block
        if $H(B) \leq T$ then                              ▷ Successful Mining
            return $B$
        end if
        $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$
    end for
    return $\perp$                                        ▷ Unsuccessful Mining
end function
```

11.7 Longest Chain

This algorithm is run by honest nodes in order to adopt the longest chain each round. Since every honest node abides to the longest chain rule, the conditions are required for a chain to be adopted: the chain is valid and the chain is strictly longer (line 4). This algorithm is called in line 2 of the honest party algorithm: it will loop through every chain it received through the gossip network, check its validity and check that it is longer than the currently adopted chain.

Algorithm 20 The maxvalid algorithm

```
1: function $\textsc{maxvalid}_{\mathcal{G},\delta(\cdot)}(\overline{C})$
2:     $C_{\mathsf{max}} \leftarrow [\mathcal{G}]$                      ▷ Start with current adopted chain
3:     for $C \in \overline{C}$ do      ▷ Iterate for every chain received through gossip network
4:         if $\mathsf{validate}_{\mathcal{G},\delta(\cdot)}(C) \wedge |C| > |C_{\mathsf{max}}|$ then       ▷ Longest Chain Rule
5:             $C_{\mathsf{max}} \leftarrow C$
6:         end if
7:     end for
8:     return $C_{\mathsf{max}}$
9: end function
```

11.8 Validating a block

This algorithm is used to validate a block, it is called in line 4 of the longest chain algorithm run by honest parties. The algorithm first checks that the Genesis block the chain is correct (line 2). Then for every block in the chain, the algorithm will update the UTXO, checking that each transaction is valid (lines 13-16). The algorithm will also check the PoW for each block and check that the block is in the correct format of $s||x||ctr$ (lines 9-12)

Algorithm 21 The validate algorithm

```
function VALIDATE$_{\mathcal{G},\delta(\cdot)}(C)$
    if $C[0] \neq \mathcal{G}$ then                    ▷ Check that first block is Genesis
        return false
    end if
    st ← st$_0$                                         ▷ Start at Genesis state
    $h \leftarrow H(C[0])$
    st ← $\delta^*(st, C[0].x)$
    for $B \in C[1{:}]$ do                          ▷ Iterate for every block in the chain
        $(s, x, ctr) \leftarrow B$
        if $H(B) > T \lor s \neq h$ then            ▷ PoW check and Ancestry check
            return false
        end if
        st ← $\delta^*(st, B.x)$        ▷ Application Layer: update UTXO & validate transactions
        if $st = \bot$ then
            return false                                ▷ Invalid state transition
        end if
        $h \leftarrow H(B)$
    end for
    return true
end function
```

11.9 Chain Virtues

Equipped with this new rigorous definition of the environment, our assumptions and the algorithm ran by the honest party, we can now mathematically define the Chain Virtues, introduced earlier in the lectures.

1. **Common Prefix (κ).** $\forall$ honest parties P_1, P_2 adopting chains C_1, C_2 at any rounds $r_1 \leq r_2$ respectively, Common Prefix property $C_1[: -\kappa] \leq C_2$ holds.
2. **Chain Quality (μ, ℓ).** $\forall$ honest party P with adopted chain C, $\forall i$ any chunk $C[i : i + \ell]$ of length $\ell > 0$ has a ratio of honest blocks μ.
3. **Chain Growth (τ, s).** $\forall$ honest parties P and $\forall r_1, r_2$ with adopted chain C_1 at round r_1 and adopted chain C_2 at round $r_2 \geq r_1 + s$, it holds that $|C_2| \geq |C_1| + \tau s$.

We define a round during which one or more honest party found block as a **successful round (r).** A round has a **convergence opportunity(r)** if only one honest party found a block irrespective of adversarial parties.

11.10 Pairing Lemma

Lemma 10. *Let $B = C[i]$ for some chain C s.t. B was computed by an honest party P during a convergence opportunity r. Then for any block B' at position i of some other chain C', if $B \neq B'$, then B' was adversarially computed.*

Proof. Suppose for contradiction that B' was mined on a round r'. For the sake of contradiction, assume that B' was honestly computed. Thus, we need to analyze three following cases:

1. Case 1: $r = r'$. This is not possible as round r was a convergence opportunity.
2. Case 2: $r < r'$. This is not possible as due to the longest chain rule, after round r, everybody will have adopted a chain of at least i blocks, so honest parties would not accept B'.
3. Case 3: $r > r'$. This is not possible, same as above but this time honest parties wouldn't adopt block B.

So we have a contradiction, thus, B' must have been adversarially mined. □

From the above, we note that, if the adversary wants to displace block B, she has to pay for it by mining B'. Therefore, if the adversary does not mine a block, then the convergence opportunity will be a true honest convergence.

11.11 Honest Majority Assumption (n, t, δ)

We will now give a new definition of the honest majority assumption by introducing the honest advantage parameter δ . We will see in the next lecture that we need this parameter in order the chain virtues hold for Bitcoin. We say that the honest majority assumption holds if $t < (1 - \delta)(n - t)$.

Problems

11.1 Go back to Lemma 6. Restate it and prove it formally in the model developed in this chapter.

11.12 Further Reading

Even though Satoshi Nakamoto developed the first blockchain and wrote the paper about it, he did not prove that blockchains are secure against *all* adversaries within that paper. Instead, he showed that Bitcoin is secure against the *specific* Nakamoto adversary which we studied in the previous chapters. At a later time, Juan Garay, Aggelos Kiayias, and Nikos Leonardos wrote the *Bitcoin Backbone* paper [14]. The Bitcoin Backbone paper formalizes and proves that blockchains that use Proof-of-Work are secure. This is the model we have presented in this chapter, with minor modifications.

Chapter 12

Security in Earnest II DRAFT

12.1 Safety and Liveness

12.1.1 Defining Safety and Liveness

Now that we have formally defined the synchronous model underlying our blockchain (in the previous lecture), we now rigorously define the security properties that a blockchain-based ledger must satisfy, notably, *safety* and *liveness*.

A ledger achieves safety when all honest parties have views of the ledger that are consistent with one another. More precisely, any transaction included in one party's ledger at a specific round is also included at the same position in all other parties' ledgers at all later rounds. A ledger achieves liveness if whenever honest parties attempt to add a transaction to the ledger, it gets added to all parties' ledgers within u rounds.

- **Safety:** For all honest parties P_1, P_2, and rounds $r_1 < r_2$, $\forall i \in [|L_{r_1}^{P_1}|]$, a transaction reported at $L_{r_1}^{P_1}[i]$ also appears at $L_{r_2}^{P_2}[i]$.
- **Liveness(u):** If all honest parties attempt to inject a transaction tx at rounds $r, ..., r+u$, then for all honest parties P, tx will appear in L_{r+u}^{P}.

Note that u should be defined so that it is at least large enough that an honest party successfully creates and broadcasts a block in u rounds, and that block gets buried under k blocks.

Now that safety and liveness are defined, we will prove how satisfying the chain virtues common prefix, chain quality, and chain growth implies that safety and liveness hold.

12.1.2 Common Prefix Implies Safety

Theorem 11. *If the longest chain protocol satisfies $CP(k)$, then the resulting ledger is safe.*

Proof. Let C_1 be the view of party P_1 at round r_1, and C_2 be the view of P_2 at round r_2; where $r_1 < r_2$. Because $CP(k)$ is satisfied, the condition $C_1[: -k] \preceq C_2$ must hold. The honest protocol states that for a transaction tx to appear in the ledger of an honest party, tx must be buried under k blocks in the honest party's chain. Therefore, we know that $L_{r_1}^{P_1}$ is made up of the transactions in $C_1[: -k]$, because $C_1[:-k]$ is buried k blocks deep from the longest chain tip. We know that $C_1[:-k] \preceq C_2$ due to common prefix. Moreover, each block in $C_1[:-k]$ must be buried at least k blocks deep in C_2 because the honest node P_1 must have broadcast C_1 in or before round r_1 and due to the longest chain rule, $|C_2| \geq |C_1|$. Hence, $L_{r_1}^{P_1}$ is a prefix of $L_{r_2}^{P_2}$. This implies that all transactions in $L_{r_1}^{P_1}$ must also be included in $L_{r_2}^{P_2}$ at the same positions, and hence safety holds. □

12.1.3 Chain Quality and Chain Growth Imply Liveness

Theorem 12. *If the protocol satisfies $CQ(\mu, \ell)$ and $CG(\tau, s)$ then the ledger satisfies liveness with $u = \max(\frac{\ell+k}{\tau}, s)$.*

Proof. Due to the Chain Quality assumption $\text{CQ}(\mu, \ell)$, at least one block out of ℓ consecutive blocks in a chain will be honestly mined if $\mu\ell \geq 1$. Moreover, we require that for a block to be included in a ledger, it must be buried under k blocks. Therefore, we know that an honestly mined block is included in the ledger if we wait for the honestly adopted chains to grow by $\ell + k$ blocks. Because $u\tau$ is the minimum growth of the honestly adopted chains in u rounds, therefore, if we want liveness to hold with u, then we require that $u\tau \geq \ell + k$. However, in order to invoke Chain Growth at all, we need to wait at least s rounds, therefore, the above only holds if $u \geq s$ also holds.

Observe that we require both $u \geq s$ and $u\tau \geq \ell + k$ in order to guarantee that an honest block is included in the ledger, therefore liveness must hold with $u = \max(\frac{\ell+k}{\tau}, s)$ □

12.2 Proving Chain Growth, Chain Quality, and Common Prefix

Let $X_r \in \{0,1\}, Y_r \in \{0,1\}, Z_{r,j} \in \{0,1\}$, and $Z_r = \sum_{j=1}^{tq} Z_{r,j}$ be random variables to model the events happening at each round r of the blockchain execution. These quantities will become helpful when we relate them to each other.

- $X_r \in \{0,1\}$ denotes whether round r was successful. $X_r = 1$ if at least one honest party has mined a block at round r, and $X_r = 0$ if otherwise.
- $Y_r \in \{0,1\}$ denotes whether round r was a convergence opportunity. $Y_r = 1$ if round r is a convergence opportunity and $Y_r = 0$ if otherwise.
- $Z_{r,j} \in \{0,1\}$ denotes whether the j^{th} query of the adversary $\mathcal{A}$ was successful at round r. $Z_{r,j} = 1$ if the j^{th} query of the adversary at round r is successful, and $Z_{r,j} = 0$ if otherwise.
- $Z_r = \sum_{j=1}^{tq} Z_{r,j}$ denotes the number of successful queries by the adversary during round r.

Over an interval of consecutive rounds S:

- $X(S) = \sum_{r\in S} X_r$, i.e. number of successful rounds in the interval S
- $Y(S) = \sum_{r\in S} Y_r$, i.e. number of convergence opportunities in the interval S

- $Z(S) = \sum_{r \in S} Z_r$, i.e. number of successful adversarial queries in the interval S

Note that X_r is not the total number of successful queries at round r, just whether there is *at least one* successful honest query. Similarly, $X(S)$ counts the number of honestly successful rounds in the interval S, not the number of successful honest queries as there could be multiple successful honest queries in once round.

12.2.1 Chain Growth Lemma

Lemma 13 (Chain Growth Lemma)**.** *Suppose that at round r, an honest party P has a chain of length l. Then by round $r' \geq r$, every honest party has adopted a chain of length at least $l + \sum_{i=r}^{r'-1} X_i$.*

Since X_r only indicates whether there is at least one successful honest query, we do not overestimate chain length by counting multiple honest parties mining blocks that are forks of each other. Also note that the sum $\sum_{i=r}^{r-1} X_i$ is defined to be 0.

Proof. We will prove by induction that all parties have chain lengths at round r' of at least $l + \sum_{i=r}^{r'-1}$ for all values of $r' \geq r$.

We will perform a proof by induction on r'. In the base case $r' = r$, if an honest party has a chain C of length l at round r, then that party broadcast C at a round earlier than r. It follows that every honest party will receive C by round r, and therefore adopts a chain of length at least $|C| = l = l + \sum_{i=r}^{r'-1}$.

For $r' > r$, suppose $C_{r'}$ is the chain adopted by an honest party. For the inductive step, suppose $|C_{r+j}| \geq l + \sum_{i=r}^{r+j-1} X_i$. Consider the following two cases:

1. Case 1: $X_{r+j} = 0$. Due to the longest chain rule, $|C_{r+j+1}|$ must be at least as long as $|C_{r+j}|$, and $\sum_{i=r}^{r+j} X_i = \sum_{i=r}^{r+j-1} X_i$, therefore, it is clear that $|C_{r+j+1}| \geq |C_{r+j}| \geq l + \sum_{i=r}^{r+j+1} X_i$
2. Case 2: $X_{r+j} = 1$. At round $r + j$ all parties have adopted chains of length $|C_{r+j}|$, so due to the longest chain rule, the honest party must have mined a chain of at least length $|C_{r+j}| + 1$ at round $r + j$. Therefore, $|C_{r+j+1}| = |C_{r+j}| + 1 \geq l + \sum_{i=r}^{r+j} X_i$.

We see that through induction, our statement holds for all j. Note that this proof requires that $P(X_r = 1) \neq 0$, for there to be Chain Growth.

□

12.2.2 Proving Common Prefix and Chain Growth

First, let's consider the relations between the expectations of X, Y, Z under the honest majority assumption. It is clear that $\mathbb{E}[X(S)] > \mathbb{E}[Y(S)]$, because an honestly successful round is not necessarily a convergence opportunity. Moreover, we expect that $\mathbb{E}[X(S)] > \mathbb{E}[Z(S)]$ due to the honest majority assumption because less computing power implies fewer expected successful queries (this will be formally proven later). Therefore, under the honest majority assumption, we would expect that $\mathbb{E}[Z(S)] < \mathbb{E}[Y(S)] < \mathbb{E}[X(S)]$. At this point, it is not obvious that $\mathbb{E}[Z(S)] < \mathbb{E}[Y(S)]$ but we will prove this later in this lecture.

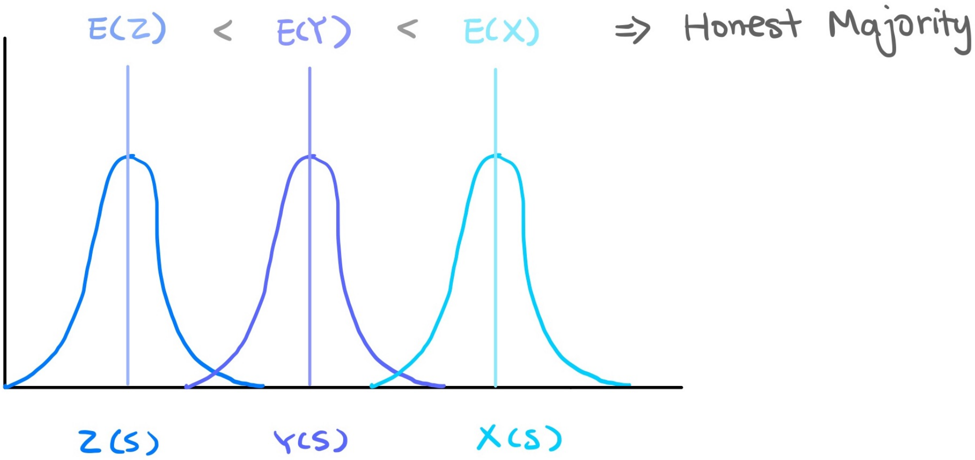


Figure 12.1: Illustration of the probability density functions for $X(S)$, $Y(S)$, and $Z(S)$. Note that $\mathbb{E}[Z(S)] < \mathbb{E}[Y(S)] < \mathbb{E}[X(S)]$

But if we want common prefix to hold except with negligible probability, it is not sufficient that the *expectations* are in this order. It is not sufficient that *on average* the adversary will have fewer successful queries than the honest parties will have convergence opportunities. We want something stronger: except with negligible probability, in all large enough intervals of consecutive slots S, the adversary will have fewer successful queries than the honest parties will have convergence opportunities, i.e. $\Pr[\forall S\colon Z(S) < Y(S)] \geq 1 - \mathrm{negl}(\kappa)$. In other words, the expectations of $X(S)$, $Y(S)$, and $Z(S)$ must be sufficiently separated so that the probability of $Y(S) \leq Z(S)$ is negligible.

In order to prove $\Pr[Y(S) < Z(S)] = \mathrm{negl}(\kappa)$, we will use a probability tool called the Chernoff Bound and introduce the concept of *typicality*.

Chernoff Bound

We will not prove this theorem here, but intuitively, the Chernoff Bound says this: if we flip a fair coin n times, where heads = 1 and tails = 0, then the more tosses we have, the less likely their sum is going to deviate from the expected value of their sum, which is $0.5 \times n$. As the number of trials increases, the probability that the sum is off the expectation by a certain *percentage* of error approaches 0 (this probability is represented by the shaded regions in Figure 12.2). More formally, the Chernoff Bound states:

Theorem 14. *Consider random variables X_i, $i \in [n]$ s.t. $X_i \overset{i.i.d.}{\sim} \mathrm{Bernoulli}(p)$ and $X = \sum_{i=1}^{n} X_i$. Let $\mu = \mathbb{E}[X] = n\mathbb{E}[X_i]$. For any $0 < \epsilon < 1$,*

$$\Pr[X \leq (1-\epsilon)\mu] \leq e^{-\Omega(n\epsilon^2\mu)}, \tag{12.1}$$

$$\Pr[X \geq (1+\epsilon)\mu] \leq e^{-\Omega(n\epsilon^2\mu)}. \tag{12.2}$$

For more details about the Chernoff bound and its proof, a good reference is [9]. (This μ is internal to the Chernoff bound and is not the chain quality parameter.)

Typicality

Ultimately, we want to prove that common prefix and chain growth are satisfied except with negligible probability. In order to simplify our analysis of probabilities, we introduce the concept of "typicality". Typical executions will be defined such that executions will be typical except with negligible probability. If we show that typical executions uphold chain growth and common prefix, then we will have proven that chain growth and common prefix hold except with negligible probability.

Definition 26 ((ϵ, λ)-Typical executions)**.** *An execution is typical if for all sets of consecutive rounds S with $|S| \geq \lambda$:*

1. $(1-\epsilon)E[X(S)] < X(S) < (1+\epsilon)E[X(S)]$
2. $(1-\epsilon)E[Y(S)] < Y(S)$
3. $Z(S) < E[Z(S)] + \epsilon E[X(S)]$

In addition, there are no collisions or predictions in the Random Oracle.

Theorem 15. (ϵ, λ)*-Typical executions occur with probability at least* $1 - \text{negl}(\lambda)$.

Proof. We can show that each of the three properties hold with $(1 - \text{negl}(\lambda))$ probability, where $n = |S| \geq \lambda$.

1. Directly by Chernoff bound, this is true with probability $\geq 1 - e^{-\Omega(n\epsilon^2)}$.
2. By the left-sided Chernoff bound, we have that $(1-\epsilon)E[Y(S)] < Y(S)$ is true with probability $\geq 1 - e^{-\Omega(n\epsilon^2)}$.
3. By the right-sided Chernoff bound, we have that $Z(S) < (1+\epsilon)E[Z(S)] = E[Z(S)] + \epsilon E[Z(S)]$ is true with probability $\geq 1 - e^{-\Omega(n\epsilon^2)}$. By the honest majority assumption, we expect $E[Z(S)] < E[X(S)]$ because less computing power implies fewer expected successful queries (this will be proven momentarily). Since ϵ is positive, $\epsilon E[Z(S)] < \epsilon E[X(S)]$. All together we have $Z(S) < (1+\epsilon)E[Z(S)] = E[Z(S)] + \epsilon E[Z(S)] < E[Z(S)] + \epsilon E[X(S)]$.
4. Since the Random Oracle randomly samples each output from $\{0,1\}^\kappa$, the probability that it samples the same output for two different inputs is $\text{negl}(\kappa)$. Similarly, the probability that the adversary can predict the output for an input it has not queried before is $\text{negl}(\kappa)$. The probability that a collision or prediction occurs over a polynomial time execution is also $\text{negl}(\kappa)$. (We will choose $\kappa = \Omega(\lambda)$).

Note that the reason we can use the Chernoff Bound is because we are using the random oracle model, which gives us that each query to the random oracle being successful is an independent Bernoulli random variable, hence the random sequences X, Y, and Z are independent across time (X_r and Y_r are not independent, but X_r and $X_{r'}$ for $r \neq r'$ are). This is stronger than just a collision-resistant hash function. □

Note that we need the set S to be of size at least λ to allow the variables to be concentrated within a Chernoff error ϵ, because Chernoff bound requires a large enough number of trials to be invoked.

For reasons that we will see soon, we will choose ϵ and f so that $3f + 3\epsilon \leq \delta$. This is called the *balancing equation.* Here, δ is the honest advantage, i.e., $t < (1-\delta)(n-t)$.

Now remember our goal is to prove that the expectations of X, Y, and Z must be sufficiently separated so that $Y(S) > Z(S)$ except with negligible probability, i.e. we want to show that the lower bound of $Y(S)$ is larger than the upper bound of $Z(S)$. Typicality gives us bounds on $X(S), Y(S), Z(S)$ but they are with respect to $\mathbb{E}[X(S)], \mathbb{E}[Y(S)], \mathbb{E}[Z(S)]$, so we need to find bounds for these expectations respectively. Since the random variables X_r are independent and identically distributed for different r, $\mathbb{E}[X(S)] = |S|\mathbb{E}[X_r]$. Similarly, $\mathbb{E}[Y(S)] = |S|\mathbb{E}[Y_r]$ and $\mathbb{E}[Z(S)] = |S|\mathbb{E}[Z_r]$. So, we only need to compare $\mathbb{E}[X_r]$, $\mathbb{E}[Y_r]$ and $\mathbb{E}[Z_r]$.

What do we know about $\mathbb{E}[X_r]$?

$$\begin{aligned}\mathbb{E}[X_r] = f &= \Pr[\text{at least one successful honest query in round } r]\\ &= 1 - \Pr[\text{no honest query succeeds in round } r]\\ &= 1-(1-p)^{q(n-t)}\\ &< pq(n-t). \end{aligned} \tag{12.3}$$

The last step above comes from Bernoulli's inequality, i.e, $(1+x)^a > 1 + ax, \forall x, a \in \mathbb{R}, x > -1, x \neq 0, a > 1$. We can also show that

$$\frac{f}{1-f} = \frac{1-(1-p)^{q(n-t)}}{(1-p)^{q(n-t)}} = (1-p)^{-q(n-t)} - 1 > (1+p)^{q(n-t)} - 1 > pq(n-t). \tag{12.4}$$

Therefore, we can sandwich the value of $\mathbb{E}[X_r]$ by its lower and upper bounds.

$$(1-f)pq(n-t) < \mathbb{E}[X_r] < pq(n-t). \tag{12.5}$$

For $\mathbb{E}[Y_r]$, we have

$$\mathbb{E}[Y_r] \geq q(n-t)p(1-p)^{q(n-t)-1} > pq(n-t)[1-pq(n-t)] \geq f(1-f). \tag{12.6}$$

The first inequality is obtained by assuming that all honest parties make all q queries even after a successful one and summing over all queries the probability $p(1-p)^{q(n-t)-1}$ that it is the only successful one. The second inequality uses $(1-p)^{q(n-t)-1} > (1-p)^{q(n-t)}$ and uses Bernoulli's inequality again. The third inequality holds because $f(1-f)$ is an increasing function for $f \in (0, \frac{1}{2})$.

Since the adversary is allowed at most qt queries in each round and each query is successful with probability p, we also have

$$\mathbb{E}[Z_r] = pqt. \tag{12.7}$$

Even though we want the expectations of $X(S)$, $Y(S)$, and $Z(S)$ to be sufficiently separated so that $Y(S) > Z(S)$ except with negligible probability, we cannot separate them arbitrarily far. This is because the distance between $\mathbb{E}[Z]$ and $\mathbb{E}[X]$

is determined by the advantage held by the majority in the honest majority assumption, i.e., $t \leq (1-\delta)(n-t)$ where $3f + 3\epsilon \leq \delta$. To see this,

$$\mathbb{E}[Z_r] = pqt = \frac{t}{n-t} \cdot pq(n-t) < \frac{t}{n-t} \cdot \frac{f}{1-f} < \left(1 + \frac{\delta}{2}\right) \cdot f \cdot \frac{t}{n-t}. \tag{12.8}$$

Here, we have used the inequality $\frac{f}{1-f} > pq(n-t)$ proved in (**??**), and another inequality $\frac{1}{1-f} < 1 + \frac{\delta}{2}$. To prove the second inequality, we know that $f < \frac{\delta}{3}$ because $3f + 3\epsilon \leq \delta$ and $\epsilon > 0$. So, we need to show that $\frac{1}{1-\delta/3} < 1 + \frac{\delta}{2}$ which can be verified as follows:

$$\begin{aligned} & 1 < \left(1 - \frac{\delta}{3}\right)\left(1 + \frac{\delta}{2}\right) \\ \iff & 1 < 1 + \frac{\delta}{2} - \frac{\delta}{3} - \frac{\delta^2}{6} \\ \iff & 0 < \frac{\delta}{6} - \frac{\delta^2}{6} \\ \iff & 0 < \frac{\delta}{6}(1-\delta) \end{aligned}$$

which is true because $0 < \delta < 1$.

The honest advantage δ (which is the distance between $|S|pqt$ and $|S|pq(n-t)$, scaled by $|S|pq(n-t)$) is composed of the distance between $\mathbb{E}[Z(S)] = |S|pqt$ and $\mathbb{E}[Y(S)]$, distance between $\mathbb{E}[Y(S)]$ and $\mathbb{E}[X(S)]$, and distance between $\mathbb{E}[X(S)]$ and $|S|pq(n-t)$. We allocate this total possible distance by following balancing equation: $3\epsilon + 3f \leq \delta$. Let's break this inequality down for an intuitive understanding along with Figure 12.2.

- 3ϵ is the relative distance we allocate between $\mathbb{E}[Z(S)]$ and $\mathbb{E}[Y(S)]$, where ϵ is the Chernoff error. By Chernoff, we have that $Z(S)$ should not exceed $(1+\epsilon)\mathbb{E}[Z(S)]$ with more than negligible probability, and likewise $Y(S)$ should not go below $(1-\epsilon)\mathbb{E}[Y(S)]$. Therefore, if we leave some buffer distance between $(1+\epsilon)\mathbb{E}[Z(S)]$ and $(1-\epsilon)\mathbb{E}[Y(S)]$ we should have that $Y(S) > Z(S)$ except with negligible probability. So we secure ϵ to the right of $\mathbb{E}[Z(S)]$, ϵ to the left of $\mathbb{E}[Y(S)]$, and another ϵ in between as a buffer, which gives us 3ϵ.

- The probability of an honestly successful round is $f = \mathbb{E}[X_r]$. Recall that we have calculated that $\mathbb{E}[Y_r] \geq f(1-f)$ in (**??**), so $\mathbb{E}[Y(S)]$ is at most f relative distance away from $\mathbb{E}[X(S)]$. Following similar logic as above, we secure f to the right of $\mathbb{E}[Y(S)]$ and f to the left of $\mathbb{E}[X(S)]$.

- Finally, $\mathbb{E}[X(S)]$ is at most f relative distance away from $|S|pq(n-t)$. We get this from (**??**).

Note the relationship between λ and ϵ. The smaller we require our Chernoff error ϵ to be, the longer time λ we need to wait for this concentration to occur. If we look at the Chernoff bound equations, the probability that our variables *fail* to be nicely concentrated is approximately $e^{-\Omega(\epsilon^2 \lambda f)}$ (since the expectations of $X(S)$, $Y(S)$, $Z(S)$ are proportional to λf). Recall from the previous lectures that we denoted by κ our security parameter, and this indicated the *bits* in our accepted probability of *failure*. Our accepted probability of failure was then at most in the order of $2^{-\kappa}$. We want to achieve the same thing with our choice of ϵ and λ. Therefore, given a particular κ (for example $\kappa = 256$) and particular values for ϵ and f, we need to set λ such that $\kappa \approx \epsilon^2 \lambda f$. In a nutshell, the larger we make ϵ, the less time λ we will need to wait for confirmation.

So, we want both ϵ to be large (to achieve fast confirmation and a small λ), but also f to be large (to achieve good chain growth with a fast block production rate). However, we cannot make both of them arbitrarily large, as they have to satisfy the balancing equation: $3f + 3\epsilon \leq \delta$. The value δ is not a parameter we can change, but is given to us from the threat model and adversarial assumptions: It is telling us what sort of adversary we are able to withstand. In the end, for a given δ, we want to have the fastest possible blockchain, yet maintain security. Splitting equally between ϵ and f, we can choose $\epsilon = f = \delta/6$. The takehome lesson is that, to withstand a powerful adversary (small δ), we need to wait a sufficient amount of time for transactions to be confirmed. You cannot have both quick confirmation and good security!

In the next chapter, we will use the tools developed today to show that in typical executions, $Y(S) > Z(S)$ for all intervals of slots S with $|S| \geq \lambda$. This will help us to prove that Common Prefix and Chain Growth hold in typical executions.

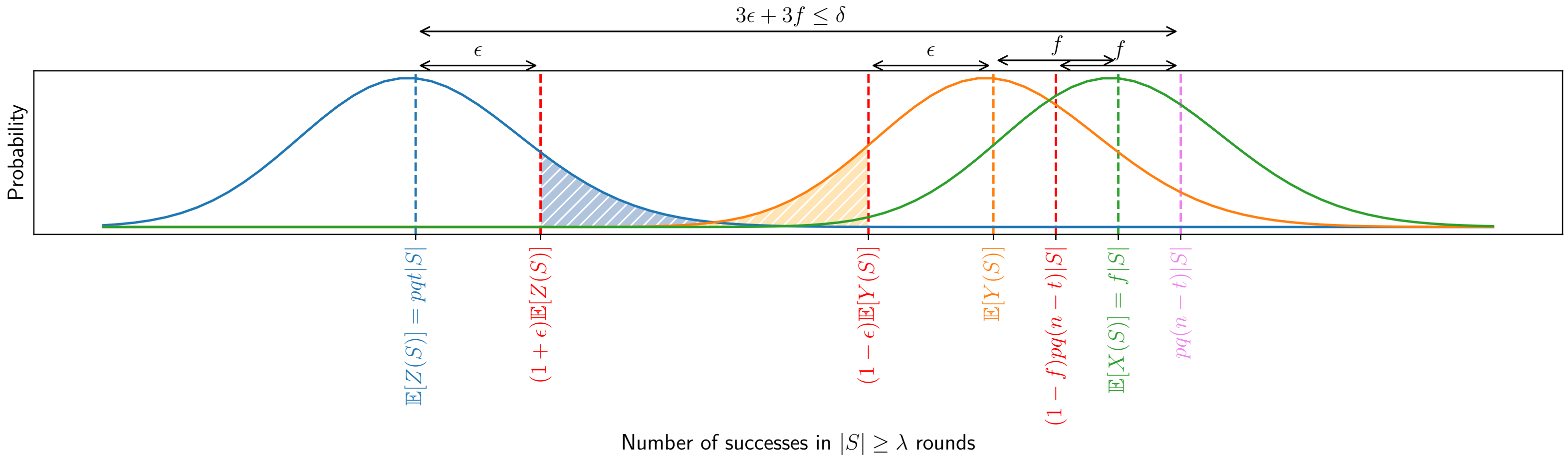


Figure 12.2: The distribution of the random variables X, Y, and Z in the proof-of-work longest chain protocol.

Chapter 13

Security in Earnest III DRAFT

13.1 Recap of last lecture: Security in Earnest (II)

We proved several bounds in the previous lecture:

$$(1-f)pq(n-t) < f = \mathbb{E}[X_r] = 1-(1-p)^{q(n-t)} < pq(n-t) \tag{13.1}$$

$$\mathbb{E}[Y_r] \geq q(n-t)p(1-p)^{q(n-t-1)} > pq(n-t)(1-pq(n-t)) \geq f(1-f) > \left(1-\frac{\delta}{3}\right)f \tag{13.2}$$

$$(1-\epsilon)\,\mathbb{E}[X(S)] < X(S) < (1+\epsilon)\,\mathbb{E}[X(S)] \tag{13.3}$$

$$(1-\epsilon)\,\mathbb{E}[Y(S)] < Y(S) \tag{13.4}$$

$$Z(S) < (1+\epsilon)\,\mathbb{E}[Z(S)] \tag{13.5}$$

First, X_r indicates whether or not round r was successful, i.e. had at least one successful honest query. Bound 13.1 shows a lower and upper bound for $\mathbb{E}[X_r]$. Second, Y_r indicates whether or not round r was a convergence opportunity. Bound 13.2 gives us a lower bound on $\mathbb{E}[Y_r]$. Bounds 13.3, 13.4, and 13.5 are Chernoff bounds on $X(S)$, $Y(S)$, and $Z(S)$, respectively where S is a set of consecutive rounds.

Today, we will be covering Security in Earnest (III). It is highly recommended to read the Bitcoin Backbone paper.

13.2 Typicality implies $Z(S) < Y(S)$

We want to prove that in a typical execution, we have the property that $Z(S) < Y(S)$. This will help us prove that the Common Prefix property holds.

13.2.1 Derivation of Upper Bound of $\mathbb{E}[Z(S)]$

First, we want to derive an upper bound of $\mathbb{E}[Z(S)]$, which will help us achieve our result. Z_r counts the number of successful queries for the adversary in round r. It is a sum of qt repeated independent Bernoulli trials, each with a p chance of succeeding, so $\mathbb{E}[Z_r] = pqt$. We can rewrite pqt as $\frac{t}{n-t}pq(n-t)$. Now, if we apply bound 13.1 we derived from last class and the fact that $\frac{1}{1-f} < 1+\delta/2$ with $f = \delta/6$

we have that

$$\begin{aligned}\mathbb{E}[Z_r] &= \frac{t}{n-t}pq(n-t) \quad (13.6)\\ &< \frac{t}{n-t}\cdot\frac{f}{1-f} \quad (13.7)\\ &< \left(1+\frac{\delta}{2}\right)f\frac{t}{n-t}. \quad (13.8)\end{aligned}$$

13.2.2 Combining it to get $Z(S) < Y(S)$

We can use bound 13.2, which gives a lower bound on $\mathbb{E}[Y_r]$. We have that:

$$\begin{aligned}Y(S) &\geq (1-\epsilon)\,\mathbb{E}[Y(S)] \quad (13.9)\\ &= (1-\epsilon)\,\mathbb{E}[Y_r]|S| \quad (13.10)\\ &> (1-\epsilon)f(1-f)|S| \quad (13.11)\\ &> \left(1-\frac{\delta}{3}\right)f|S| \quad (13.12)\end{aligned}$$

The last inequality is derived as follows by setting $f=\epsilon=\frac{\delta}{6}$:

$$\begin{aligned}(1-\epsilon)(1-f) &> 1-\frac{\delta}{3} \quad (13.13)\\ \Leftarrow \left(1-\frac{\delta}{6}\right)\left(1-\frac{\delta}{6}\right) &> 1-\frac{\delta}{3} \quad (13.14)\\ \Leftarrow -\frac{2\delta}{6}+\frac{\delta^2}{36} &> -\frac{\delta}{3} \quad (13.15)\\ \Leftarrow \frac{\delta^2}{36} &> 0 \quad (13.16)\\ \Leftarrow \delta &> 0. \quad (13.17)\end{aligned}$$

From the Chernoff bound 13.5 and upper bound 13.8, we also have that:

$$\begin{aligned}Z(S) &< (1+\epsilon)\,\mathbb{E}[Z(S)] \quad (13.18)\\ &= (1+\epsilon)\,\mathbb{E}[Z_r]|S| \quad (13.19)\\ &< (1+\epsilon)\frac{t}{n-t}\cdot\frac{f}{1-f}|S| \quad (13.20)\\ &< \frac{t}{n-t}\cdot\frac{f}{1-f}|S| + \epsilon\frac{t}{n-t}\cdot\frac{1}{1-f}f|S| \quad (13.21)\\ &< \frac{t}{n-t}\cdot\frac{f}{1-f}|S| + \epsilon f|S| \quad (13.22)\\ &\leq \left(1-\frac{2\delta}{3}\right)f|S|. \quad (13.23)\end{aligned}$$

To prove inequality 13.22, note that from the balancing equation we have $f \leq \frac{\delta}{3}$.

It suffices to show that $\frac{t}{n-t} \cdot \frac{1}{1-f} < 1$:

$$\frac{t}{n-t} \cdot \frac{1}{1-f} < 1 \tag{13.24}$$
$$\Leftarrow \frac{1-\delta}{1-f} < 1 \tag{13.25}$$
$$\Leftarrow 1-\delta < 1-\frac{\delta}{3}. \tag{13.26}$$

To prove inequality 13.23, we again use our choice of values $f = \epsilon = \frac{\delta}{6}$. Then,

$$\frac{t}{n-t} \cdot \frac{1}{1-f} + \epsilon < 1 - \frac{2\delta}{3} \tag{13.27}$$
$$\Leftarrow \frac{1-\delta}{1-f} + \epsilon < 1 - \frac{2\delta}{3} \tag{13.28}$$
$$\Leftarrow \frac{1-\delta}{1-\delta/6} + \frac{\delta}{6} < 1 - \frac{2\delta}{3} \tag{13.29}$$
$$\Leftarrow 1-\delta+\frac{\delta}{6}-\frac{\delta^2}{36} < 1-\frac{\delta}{6}-\frac{2\delta}{3}+\frac{2\delta^2}{18} \tag{13.30}$$
$$\Leftarrow -\frac{5\delta}{6}-\frac{\delta^2}{36} < -\frac{5\delta}{6}+\frac{\delta^2}{9} \tag{13.31}$$
$$\Leftarrow -\frac{\delta^2}{36} < \frac{\delta^2}{9} \tag{13.32}$$

Combining results 13.23 and 13.12 together gives us:

$$Z(S) < \left(1-\frac{2\delta}{3}\right) f|S| < \left(1-\frac{\delta}{3}\right) f|S| < Y(S). \tag{13.33}$$

13.3 Proof of Chain Growth

Recall chain growth lemma from the previous lecture.

Lemma 16 (Chain Growth Lemma)**.** *Suppose that at round r, an honest party P has a chain of length l. Then by round $r' \geq r$, every honest party has adopted a chain of length at least $l + \sum_{i=r}^{r'-1} X_i$.*

Now we are equipped with all the necessary tools to prove our first chain virtue.

Theorem 17 (Chain Growth)**.** *In a typical execution, Chain Growth is attained with $\tau = (1-\epsilon)f, s \geq \lambda$.*

Proof. For rounds S, such that $|S| \geq \lambda, X(S) > (1-\epsilon)f|S|$ with overwhelming probability. Invoking the growth chain lemma, it is deduced that the chain grows by at least $(1-\epsilon)f\lambda$. Therefore, the chain velocity is $\tau = (1-\epsilon)f$. □

13.4 Proof of Common Prefix

In order for the common prefix property to be violated, the adversary must have had a separate successful query for every convergence opportunity (although, not necessarily during the same round). Any convergence opportunity without a matching adversarial success would lead to convergence among the honest parties. In a nutshell, it must hold that $Z(S) > Y(S)$ over S where $|S| > \lambda$, in order for no convergence to happen. Our plan for this proof is as follows. We will show this by contradiction. Suppose that $\mathrm{CP}(k)$ is violated. Then,

1. We show that it takes a long time to produce these k blocks, so $|S| \geq \lambda$.
2. We use the pairing lemma: every convergence opportunity is paired to an adverserially successful query, so $Y(S) \leq Z(S)$.
3. Lastly, we use our result for typical executions that we can apply as S is large: $Z(S) < Y(S)$, which contradicts the previous point.

The last two points are already proven, so we are only missing the first point to prove Common Prefix. To do so, we prove the following lemma. We will choose λ to be at least $2f$.

Lemma 18 (Patience Lemma)**.** *In typical executions, any $k \geq 2\lambda f$ blocks have been computed in at least $\frac{k}{2f}$ rounds.*

Proof. Let S' be the set of consecutive rounds during which these k blocks were computed. Towards a contradiction, assume that $|S'| < \frac{k}{2f}$. The idea is to apply typicality to the set of rounds S' and to show that, within that small set of rounds, all these k blocks could not possibly have been computed. However, we cannot directly use the set of rounds S', as it is not long enough to apply typicality. We will expand S' to a larger set of rounds S where typicality is applicable, so we need to set $|S| \geq \lambda$. We will do this by including more rounds from the future into S. We will then show that, even in this larger S, it is impossible that k blocks were computed. So let S be the set of rounds extending S' such that $|S| = \lceil \frac{k}{2f} \rceil + 1 \leq \frac{k}{2f} + 2$. We can now apply typicality to S.

The number of blocks that were computed during S is at most $X(S) + Z(S)$ (i.e., they were computed by either the honest parties or the adversary).

We have that

$$\begin{aligned}
X(S) + Z(S) &< (1+\epsilon)\,\mathbb{E}[X(S)] + \left(1 - \frac{2\delta}{3}\right) f|S| && (13.34)\\
&= (1+\epsilon) f|S| + \left(1 - \frac{2\delta}{3}\right) f|S| && (13.35)\\
&= \left(2 + \epsilon - \frac{2\delta}{3}\right) f|S| && (13.36)\\
&\leq (2 - 2f) f|S| && (13.37)\\
&\leq (2 - 2f) f \cdot \left(\frac{k}{2f} + 2\right) && (13.38)\\
&= (1 - f)(k + 4f) && (13.39)\\
&< k && (13.40)
\end{aligned}$$

The last inequality holds for $k \geq 4$ (this follows from $\lambda \geq 2/f$). To prove inequal-

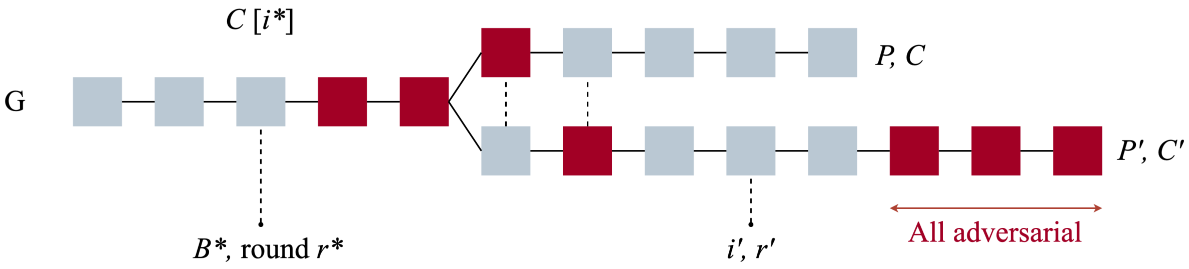


Figure 13.1: A common prefix violation. Honestly computed blocks are shown in gray, while adversarially computed blocks are shown in red. At the end of round r, party P has adopted chain C and party P' has adopted chain C'. $B^\star$ is the most recent honestly produced block in the common prefix of C, C'. This is genesis if all other blocks in the common prefix are adversarial. r' is the round with the last convergence opportunity. By the Pairing Lemma, convergence opportunities in rounds after the fork must be matched with a successful adversarial query. For example, $C[i'-3]$ is an adversarially produced block is matched with $C'[i'-3]$, and similarly for $C'[i'-2], C[i'-2]$. It is also possible to have multiple honestly produced blocks in a round after the fork without any convergence opportunities, as $C[i'-1], C'[i'-1]$ show.

ity 13.37, we use the balancing equation $3f + 3\epsilon < \delta$ as follows:

$$2 + \epsilon - \frac{2\delta}{3} \leq 2 - 2f \tag{13.41}$$

$$\Leftarrow 3\epsilon - 2\delta \leq -6f \tag{13.42}$$

$$\Leftarrow 3f + \frac{3}{2}\epsilon \leq \delta \tag{13.43}$$

$$\Leftarrow 3f + \frac{3}{2}\epsilon \leq 3f + 3\epsilon \tag{13.44}$$

Since we wanted at least k blocks, inequality 13.40 is a contradiction. □

Recall also the Pairing Lemma from the last lecture.

Lemma 19 (Pairing Lemma)**.** *Consider a block $C[i]$ produced during a convergence opportunity. If $C'[i] \neq C[i]$, then $C'[i]$ was adversarially computed.*

We are now ready to prove our second chain virtue.

Theorem 20 (Common Prefix)**.** *A typical execution satisfies Common Prefix with $k = 2\lambda f$.*

Proof. Assume towards contradiction that there is a CP(k) violation, illustrated in Figure 13.1. We have two forks C, C' where if we remove the last k blocks from each of them, they are not the same chain. Let $S = \{r^*, ..., r\}$, where r^* is the round in which the most recent honestly mined block $C[i^*] = B^*$ in the common prefix of C, C' was produced. After r^*, all honest parties will be mining on chains at least i^* long.

We claim that $Z(S) \geq Y(S)$. Let J be set of the heights of blocks B, where B was produced during a convergence opportunity in S. Let r' be the last convergence opportunity in S, in which block B' was computed at height i'.

We distinguish three cases for the heights of the convergence opportunities within S.

Case 1: The blocks between B^* and the fork point are adversarial (by the definition of r^*).

Case 2: Any blocks of height larger than i' must be adversarial. To see this, observe that, if there was an honestly produced block with height more than i', then, during round r', the honest party would not have mined at height $i' - 1$. So, all the blocks that exist at height larger than the shorter chain between C and C' must also be adversarial.

Case 3: For the blocks that were produced in the same heights in these two forks, we can apply the Pairing Lemma. We conclude that, since there are two different chains with blocks at this height, one of them must be adversarial.

Thus, overall, we have matched every convergence opportunity with an adversarially successful query. We conclude that $Z(S) \geq Y(S)$.

We have k blocks that were produced in S, so by the Patience Lemma, we have $|S| \geq \lambda$, which gives us typicality. However, typicality states that $Z(S) < Y(S)$, a direct contradiction. □

13.5 Note about Tradeoffs with ϵ and f

Let us discuss the relationship between a concrete ϵ and λ obtained by the Chernoff bound. Larger ϵ allows for smaller λ. Let us explore why. The bound on the probability of failure given to us by the Chernoff bound is in the order of $e^{-\epsilon^2 \lambda f}$. Our acceptable probability of failure must be very small and is determined by the security parameter κ (recall that typically $\kappa = 256$ and our acceptable probability of failure is $2^{-\kappa}$). Therefore, solving $\kappa = -\epsilon^2 \lambda f$, then we must set λ to be large enough to account for the small ϵ^2.

One of the bounds we enforce is:

$$3\epsilon + 3f < \delta$$

Larger f gives larger chain growth since it is easier to produce successful queries. Overall, we want to make both ϵ and f large, because we like to have quick confirmation (small λ) and fast chain growth (large f). However, we cannot make both of them large, as we are bounded by our honest advantage δ.

The verdict is that, in a setting with a powerful adversary (small δ), we must have a slow chain, either producing blocks slowly, or with the requirement to wait many blocks for confirmation, in order to ensure security. A fast chain with fast confirmation won't cut it.

Bibliography

[1] Developer guide - bitcoin. Available at: `https://bitcoin.org/en/developer-guide`.

[2] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies.* " O'Reilly Media, Inc.", 2014.

[3] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps.* O'reilly Media, 2018.

[4] M. Arapinis, A. Gkaniatsou, D. Karakostas, and A. Kiayias. A Formal Treatment of Hardware Wallets. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*, pages 426–445. Springer, 2019.

[5] K. Babel, P. Daian, M. Kelkar, and A. Juels. Clockwork Finance: Automated Analysis of Economic Security in Smart Contracts. *arXiv preprint arXiv:2109.04347*, 2021.

[6] L. Bahack. Theoretical Bitcoin Attacks with less than Half of the Computational Power. *arXiv preprint arXiv:1312.7013*, 2013.

[7] D. Boneh and V. Shoup. A graduate course in applied cryptography. *Draft 0.5*, 2020.

[8] M. Carlsten, H. Kalodner, S. M. Weinberg, and A. Narayanan. On the Instability of Bitcoin Without the Block Reward. In *Proceedings of the 2016 acm sigsac conference on computer and communications security*, pages 154–167, 2016.

[9] H. Chernoff et al. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[10] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

[11] A. Dembo, S. Kannan, E. N. Tas, D. Tse, P. Viswanath, X. Wang, and O. Zeitouni. Everything is a Race and Nakamoto Always Wins. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 859–878, 2020.

[12] J. R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[13] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[14] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In E. Oswald and M. Fischlin, editors, *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9057 of *LNCS*, pages 281–310. Springer, Apr 2015.

[15] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In J. Katz and H. Shacham, editors, *Annual International Cryptology Conference*, volume 10401 of *LNCS*, pages 291–323. Springer, Aug 2017.

[16] O. Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools.* Cambridge university press, 2007.

[17] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge university press, 2009.

[18] D. Graeber. *Debt: The First 5,000 Years.* Melville House, 2014.

[19] G. Ingham. Money is a social relation. In S. Fleetwood, editor, *Critical realism in economics: Development and debate*, pages 104–105. Routledge, 1998.

[20] W. S. Jevons. *Money and the Mechanism of Exchange.* H.S. King & Co., 1875.

[21] K. Karantias. Sok: A taxonomy of cryptocurrency wallets. Technical report, IACR Cryptology ePrint Archive, 2020: 868, 2020.

[22] A. Kiayias and G. Panagiotakos. Speed-security tradeoffs in blockchain protocols. *Cryptology ePrint Archive*, 2015.

[23] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. pages 382–401. ACM, 1982.

[24] Y. Lindell and J. Katz. *Introduction to Modern Cryptography.* Chapman and Hall/CRC, 2014.

[25] C. L. Liu. *Elements of Discrete Mathematics.* McGraw-Hill, 2 edition, 1985.

[26] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Available at: `https://bitcoin.org/bitcoin.pdf`, 2008.

[27] M. Palatinus and P. Rusnak. BIP 0044: Multi-Account Hierarchy for Deterministic Wallets. Available at: `https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki`, Apr 2014.

[28] M. Palatinus, P. Rusnak, A. Voisine, and S. Bowe. BIP 0039: Mnemonic code for generating deterministic keys. Available at: `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`, Sep 2013.

[29] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004.

[30] S. M. Ross. *A First Course in Probability.* Pearson Boston, MA, 10 edition, 2019.

[31] G. Simmel. *The Philosophy of Money.* 1900.

[32] M. Sipser et al. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[33] N. Smart. *Cryptography Made Simple.* Springer, 2016.

[34] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[35] P. Wuille. BIP 0032: Hierarchical Deterministic Wallets. Available at: `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`, Feb 2012.

Index

List of Symbols

t	adversarial parties 64
$\mathcal{A}$	adversary 12
$B_{\max}$	block size limit 106
μ	chain quality 88
τ	chain velocity 88
k	common prefix 84
q	compute 64
Δ	network delay 48
$\mathcal{Z}$	environment 131
m	epoch duration 114
Gen	generation function 32
$\mathcal{G}$	genesis 60
H	hash function 23
Π	honest protocol 12
ctr	nonce 56
pk	public key 31
sk	secret key 31
κ	security parameter 12
Sig	signature function 32
σ	signature 31
p	successful query probability 63
T	target 55
n	total parties 64
score	transaction score 107
Ver	signature verification 32