Blockchain Foundations

Dionysis Zindros

Athens and Stanford

June 2021 - January 2023

Contents

Chapter 1

A Big, Bad World

1.1 The Nature of Money

Before money, there was debt [4]. 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 [5].

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* [6]. 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 [10].

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 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 decentralized 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.

---

[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.

The Cryptographic Model

Following the cryptographic tradition, and highlighting our computer science methodology, our protocols are structured upon three pillars [7]:

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 [13] 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.

Negligibility

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).

We will keep the κ parameter free throughout our analysis, but we will demand that the probability of failure drops close to *exponentially*, or more precisely *superpolynomially* as κ increases linearly. Let us make this definition precise. If a function $f(\kappa)$ denotes the probability of failure, we want f to be approximately on the order of $\frac{1}{2^\kappa}$ or less. Any probability in the form of $\frac{1}{\kappa^3 + \kappa + 9}$, or any other inverse polynomial is unacceptable, as the probability does not drop fast enough with respect to κ. Certainly any positive *constant* probability such as 0.0001 is unacceptable. We want our function to be *eventually 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 $\kappa > \kappa_0$:*

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

Let us give some motivation about our choice of the class of *negligible* functions as our acceptable probability of failure. Suppose that a PPT adversary $\mathcal{A}$ succeeds in breaking our protocol with some non-negligible probability $p(\kappa)$. This means that $p(\kappa)$ is equal to some inverse polynomial in κ. But now consider a different adversary $\mathcal{A}'$ that operates by running the adversary $\mathcal{A}$ multiple times. In particular, $\mathcal{A}'$ simulates the execution of $\mathcal{A}$ as many as $\frac{1}{p(\kappa)}$ times. If the success of $\mathcal{A}$ is independent in all of these runs, this causes the probability of $\mathcal{A}'$ to become quite close to 1. Additionally, the adversary $\mathcal{A}'$ still runs in polynomial time, because it takes time proportional to the time needed to simulate $\mathcal{A}$ (which is a polynomial due to $\mathcal{A}$ being a PPT) and also proportional to $\frac{1}{p(\kappa)}$ (which is a polynomial because $p(\kappa)$ is non-negligible).

Therefore, any inverse polynomial probability of failure would be unacceptable. 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 [7]. 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.

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

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. Our 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 their 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*. 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.

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.
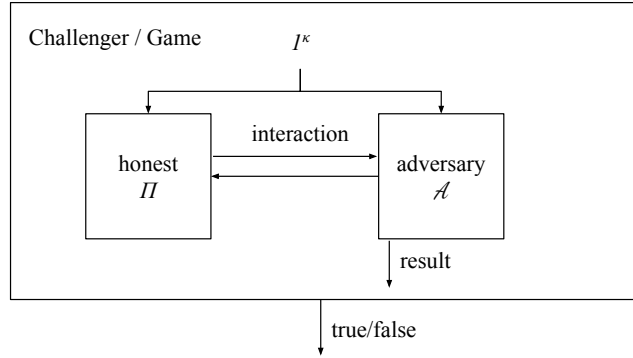

Figure 1.1: A game-based security definition shown diagramatically. The challenger invokes both the honest party and the adversary before deciding whether the adversary was successful.

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

1: **function** MY-GAME$_{\Pi,\mathcal{A}}(\kappa)$
2:     ▷ *Invoke the honest party with poly κ time and more arguments*
3:     $\cdots \Pi(1^\kappa \cdots)$
4:     ▷ *Invoke the adversary with poly κ time and more arguments*
5:     result $\leftarrow \mathcal{A}(1^\kappa \cdots)$
6:     ▷ *Evaluate whether the adversary has been successful*
7:     **if** result indicates adversarial success **then**
8:         **return** true
9:     **else**
10:         **return** false
11:     **end if**
12: **end function**

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*

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

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.

Breaking by Guessing

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* [9] 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 at most $\frac{p(\kappa)}{2^\kappa}$, which remains negligible, because $\mathsf{poly} \cdot \mathsf{negl} = \mathsf{negl}$. This reinforces our reason for adopting the class of negligible functions as our acceptable probability of failure.

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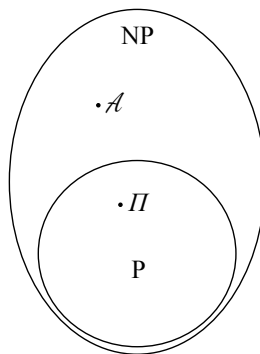
Figure 1.2: In a secure protocol, a successful honest party needs polynomial time, while a successful adversary needs superpolynomial time.

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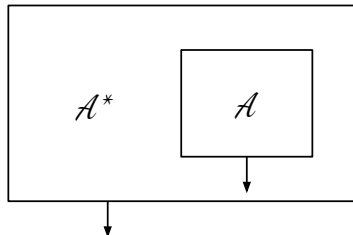
Figure 1.3: A computational reduction between two adversaries. Given an adversary $\mathcal{A}$ against protocol Π, we construct an adversary $\mathcal{A}^*$ against a protocol Π^*.

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 [7]. 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.
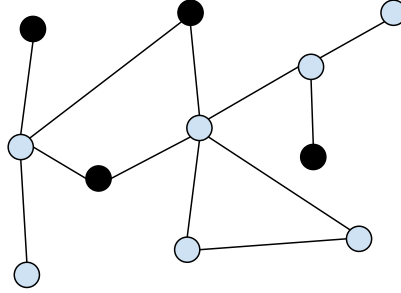
Figure 1.4: The peer-to-peer network. Nodes are shown as circles and connections as lines. The honest nodes are shown in blue, while the adversarial nodes are shown in black.

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.

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.

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

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* [1]. The adversary may overwhelm an honest party with adversarial connections as illustrated in Figure 1.6.
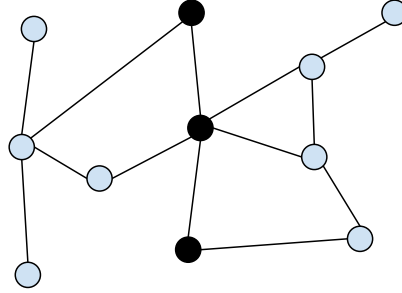
Figure 1.5: An eclipsed peer-to-peer network. Even though every honest party has an honest connection, the network is partitioned into two disconnected components by the adversary.
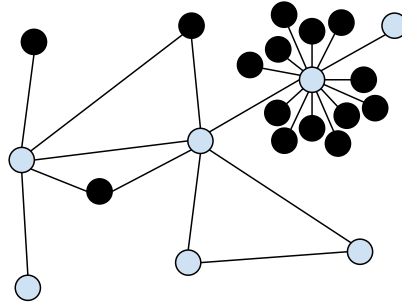
Figure 1.6: A Sybil attacked peer-to-peer network. The non-eclipsing assumption is not violated.

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

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

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

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 [7]. 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* [2, 3]. An easier and pleasant to read textbook is Smart's *Cryptography Made Simple* [12].

For a more in-depth treatment of Turing Machines and our computational model, consult Sipser's *Introduction to the Theory of Computation* [11]. 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* [8]. For probability theory, you can use Ross's *A First Course on Probability Theory* [9]. 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 DRAFT

2.1 Hash Functions

We use hash functions to represent objects in the network. A hash function is a mathematical function that takes in an arbitrary string input and outputs a unique identifier of that string: $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^\kappa$. In implementation, we use the SHA-256 hash function ($\kappa = 256$), meaning that the outputted address is of length 256.

Gossiping strategy:

1. Upon learning of an object, advertise that you have it by its objectid, which is the hash of the object (meaning that we send it to all peers).

2. Upon seeing an objectid, request it if you don't have it.

Note: It is important that each object has a *unique* identifier (one-to-one), since in the Gossip protocol, instead of sending an entire object, we signal ownership of it by sending its hash.

However, since $|\kappa|$ may be $\leq *$, by the Pigeonhole Principle, $\exists x_1, x_2 : H(x_1) = H(x_2)$. In other words, collisions exist because the input space is larger than the output space.

However, while collisions exist, they are hard to find which we will show.

To find a collision without polynomial time restriction, we can use a brute-force method:

The three properties we want to achieve are as follows:

Algorithm 3 Collision Game

1: **function** COLL$_{\mathcal{A}}(\kappa)$:
2:     $x, x' \leftarrow \mathcal{A}(1^\kappa)$ ▷ Invoke adversary for both inputs
3:         **return** $x! = x' \wedge H(x) = H(x')$
4: **end function**

In Alg 3, we invoke the adversary for both inputs and return a collision. The adversary is successful if she returns $x \neq x'$ such that $H(x) = H(x')$.

Algorithm 2 Brute force collision finding
___
1: **function** BRUTE FORCE COLLISION FINDING(1^κ)
2:     **for** $i \in \{0, \ldots, 2^\kappa\}$ **do**
3:         **for** $j \in \{i+1, \ldots, 2^\kappa\}$ **do**
4:             **if** $H(i) = H(j)$ **then**
5:                 **return** (i, j)
6:             **end if**
7:         **end for**
8:     **end for**
9: **end function**
___

Algorithm 4 2nd-preimage Game
___
1: **function** 2PRE$_\mathcal{A}(\kappa)$:   ▷
2:     $x_1 \leftarrow R\{0, 1\}^{2\kappa+1}$ ▷ Given x_1, adversary finds second input
3:     $x_2 \leftarrow \mathcal{A}(x_1)$
4:     **return** $x_1 != x_2 \wedge H(x_1) = H(x_2)$
5: **end function**
___

In Alg 4, the adversary is given a randomly sampled $2\kappa + 1$ bit 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, i.e. $H(x_1) = H(x_2)$.

Algorithm 5 Preimage Game
___
1: **function** PRE$_\mathcal{A}(\kappa)$:   ▷
2:     $x \leftarrow R\{0, 1\}^{2\kappa+1}$
3:     $y = H(x)$
4:     $x' \leftarrow \mathcal{A}(y)$ ▷ Give $\mathcal{A}$ output y
5:     **return** $H(x') = y$
6: **end function**
___

In Alg 5, the adversary is given the hash of a randomly sampled $2\kappa+1$ bit input and is successful if she can find an x that hashes to the given hash.

A hash function is easy to compute but hard to invert, meaning that $x \rightarrow y$ is easy (polynomially computable) while the inversion of the hash function, $y \rightarrow x$, is hard.

Theorem 2. *If H is collision resistant, then H is 2nd-preimage resistant. More formally, $\forall$ PPT (probabilistic polynomial time) $\mathcal{A}$: $Pr[2\mathsf{PRE}_\mathcal{A}(\kappa) = 1] \leq negl(\kappa)$.*

Proof. For the sake of contradiction, suppose that H is not 2nd-preimage resistant. Thus, there exists some adversary $\mathcal{A}$ such that $Pr[2\mathsf{PRE}_\mathcal{A}(\kappa)] = p$, where p is non-negligible. We construct an adversary $\mathcal{A}'$ against Collision game (Alg. 6).

$Pr[\mathsf{COLL}_{\mathcal{A}'}(\kappa) = 1] = Pr[2\mathsf{PRE}_\mathcal{A}(\kappa) = 1] = \text{non-negligible} \implies Pr[\mathsf{COLL}'_\mathcal{A}(\kappa) = 1]$ is non-negligible. This is a contradiction, hence, H is 2-nd-preimage resistant.

Algorithm 6 $\mathcal{A}'$ adversary

1: **function** $\mathcal{A}'(1^\kappa)$:
2:     $x_1 \leftarrow R\{0,1\}^{2\kappa+1}$
3:     $x_2 \leftarrow \mathcal{A}(x_1)$
4:     **return** x_1, x_2
5: **end function**
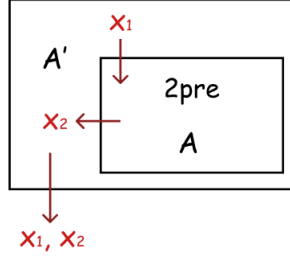
Figure 2.1: 2nd-preimage game visualization: Follows 6 construction, in which $\mathcal{A}'$ produces x_1 and gives x_1 to $\mathcal{A}$. Then, $\mathcal{A}$ provides x_2 and $\mathcal{A}'$ returns x_1, x_2.

$\square$

Theorem 3. *If H is 2nd-preimage resistant, then H is preimage resistant.*

Proof. For the sake of contradiction, suppose that H is not preimage resistant. Then, there exists an adversary against preimage. We construct $\mathcal{A}'$ against 2nd-preimage resistance 7.

Algorithm 7 $\mathcal{A}'$ Construction

1: **function** $\mathcal{A}'(1^\kappa)$:
2:     $x \leftarrow R\{0,1\}^{2\kappa+1}$
3:     $y \leftarrow H(x)$
4:     $x' \leftarrow \mathcal{A}(y)$
5:     **return** $x' \neq x_1 \wedge H(x') = H(x)$
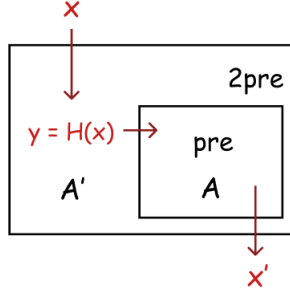6: **end function**

Figure 2.2: Preimage game visualization: Follows 7 construction, in which $\mathcal{A}'$ produces $y = H(x)$ given x. Then, give y to $\mathcal{A}$ to produce x'.

We hope $x' \neq x$. Partition the input space into "boxes" (y: output value) containing dots (x: input value) that all hash to the same value. Sort boxes by their sizes (number of inputs) in ascending order, in which the larger boxes are on the left. The purpose of this is that we want to argue that it's harder to guess a dot in a larger box.

Next, we split into two halves of equal number of inputs ($2^{2\kappa}$ dots each half). It is impossible to have $< 2^\kappa$ dots in a "box" on the left half because then each box on the right half has $< 2^\kappa$. There are $< 2^\kappa$ boxes on the right half, each with $< 2^\kappa$ dots, so the total dots in the right half is $< 2^\kappa \cdot 2^\kappa = 2^{2\kappa} \implies$ there are $< 2^{2\kappa}$ dots. So, all the left-half boxes are "large". The probability that we sampled an input on the left half $= \frac{1}{2}$ (sampling uniformly). The y value of the box of our input will have $\geq 2^\kappa$. The adversary has no better strategy than guessing uniformly at random from the box.

Thus, there holds: $Pr[2\mathsf{PRE}'_{\mathcal{A}}(\kappa)] \geq Pr[\mathsf{PRE}_{\mathcal{A}}(\kappa) = 1](\frac{1}{2})(1 - 2^{-\kappa}) = $ non-negligible * constant * value close to $1 = $ non-negligible. So $Pr[2\mathsf{PRE}'_{\mathcal{A}}(\kappa)]$ is non-negligible. This implies that $2\mathsf{PRE}$ security is broken, in contradiction. □

2.2 Transactions and Signatures

Verifiability of money: If money is created correctly, money has verifiability property such as ownership validation. Here, we discuss a transfer of ownership.

In blockchain networks, a transaction is an entry in local state. Each node keeps a ledger, which is a recorded sequence of transactions ("tx"s). Ledgers are used to verify and update the system state, a record of the balances of all parties in the system, such that consensus across nodes is maintained.

Assume we have parties A, B, C, etc. Each party can have some sort of money and each party will share all financial information. An example of a transaction could be party A saying "Give 5 units to party B." The following is an outline of ledger/transaction procedures:

1. Initial system state: some distribution of money.

2. Issue a tx by broadcasting

3. Upon receiving a tx:

(a) Check validity (source has enough money)

(b) Append to local ledger

4. Read ledger to determine balances (system state)

However, nobody should be able to issue transactions spending money from someone's else balance. Therefore, we need to authenticate transactions. One way to achieve it is by using digital signatures.

Digital signature scheme, Alg. 8, consists of three algorithms: key generation, signing and verifying algorithms.

Algorithm 8 Private and Public Keys: Signature Scheme

1: $(pk, sk) \leftarrow Gen(1^\kappa)$
2: $\sigma \leftarrow Sig(sk, m)$
3: $Ver(pk, m, \sigma) \leftarrow \{\text{True/False}\}$

An important property of digital signature scheme is **correctness**: $\forall m : (pk, sk) \leftarrow Gen(1^\kappa) : Ver(pk, m, Sig(sk, m)) = \text{True}$. In other words, for any message, we should be able to verify using a public key and m such that $Ver() = True$. Transactions are happening between public keys and messages are signed with private key by the author, verified by other users via the author's public key.

For a signature scheme to be secure, it should be impossible for adversary to produce a valid signature without knowing a private key. More formally, a security game for digital signatures is defined as follows in Alg 9:

Algorithm 9 Signatures are unforgable

1: **function** Forgery$_{\mathcal{A}}(\kappa)$
2:      $M \leftarrow \emptyset$
3:      $(pk, sk) \leftarrow Gen(1^\kappa)$
4:      $\sigma, m \leftarrow \mathcal{A}^O(pk)$
5:      **return** $Ver(pk, m, \sigma) \wedge m \notin M$
6: **end function**

We want to allow the adversary to have access to an oracle such that the adversary can choose an adversarial message. We use M to track the set of messages requested by the adversary. We don't want to allow the adversary to simply invoke the oracle 10 function and return a forged messaged outputted by M.

Oracle, that can be called by the adversary is defined in Alg. 10.

Algorithm 10 Signatures are unforgable

1: **function** O(m'):
2:      $\sigma' \leftarrow Sig(sk, m')$
3:      $M \leftarrow M \cup \{m'\}$
4:      **return** σ'
5: **end function**

Chapter 3

UTXO DRAFT

3.1 Introduction

Public verifiability of money consists of two parts:

- Money creation

- Ownership

Focus of this lecture: How to verify ownership when money changes hands?

3.2 UTXO Model

We use a graph to represent transactions and the ownership of coins. Transactions are represented as circles (nodes) with inputs and outputs (edges). Outputs signify receiver(s) of money and inputs signify the sender(s). Both senders and receivers are represented by public keys. An outgoing edge that doesn't connect to a node is referred to as an "unspent" transaction output (UTXO), and is an element of the UTXO set. A payment from public key pk_a to public key pk_b can be visualised in Figure 3.4. Note that nodes in this graph are not related to nodes in the network. The senders and receivers of transactions are not necessarily nodes in the network or block miners.

$(pk_a,\ value)$ $\qquad\qquad$ $(pk_b,\ value)$
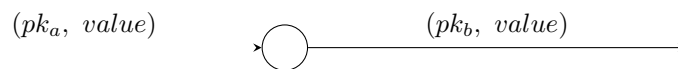
Figure 3.1: Representation of a transaction

Transactions are connected to one another to form a graph, where the output of a prior transaction becomes the input to a new transaction, as shown in Figure 2. This allows the network to verify that there was a prior transaction that gave money to the sender of a new transaction. For now, we will take money creation for granted and assume that there was an initial transaction that created money and gave it to some public key(s) to start with.

A chain of transactions is called a "coin". A coin has a current owner, denoted in the final outgoing output edge which is not connected to another transaction as

26

$(pk_a, \ value)$ $(pk_b, \ value)$ $(pk_c, \ value)$
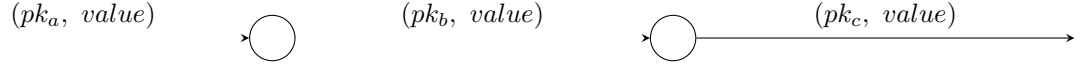
Figure 3.2: The chaining of transactions

input. A coin also has a history of its previous owners. An additional facet of the UTXO model is that all coins are fully spent. In transactions where we don't want to spend the entire coin, we can "spend" any remainder on ourselves (i.e. keep the change) to maintain the "fully spent" requirement. These types of transactions will be visualised by a node with two output edges: one edge will have the public key of the recipient of a payment, and the other will have the public key of the payer, as shown in Figure 3. Transactions can also have multiple different inputs (from different or the same public key(s)). For instance, a user could combine two paychecks to purchase a car and receive change by creating a transaction with 2 inputs (each paycheck) and 2 outputs (car dealer and change). This is shown in Figure 4.

$(pk_b, \ value)$

$(pk_a, \ value + change)$

$(pk_a, \ change)$
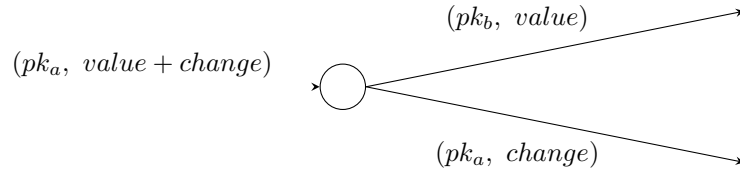
Figure 3.3: Representation of a transaction with two outputs

$(pk_a, \ paycheck_1)$

$(pk_c, \ paycheck_1 + paycheck_2)$

$(pk_b, \ paycheck_2)$
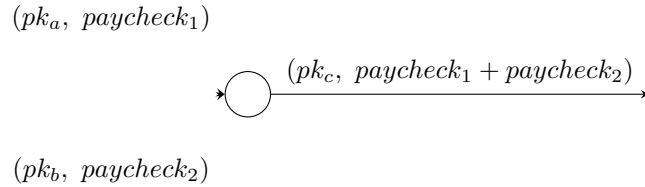
Figure 3.4: Representation of a transaction with two inputs

The UTXO set (Unspent Transactions Output set) is the set of all unspent transaction outputs. The sum of all values in the UTXO set equals the amount of money in existence. The amount of money owned by a user, e.g. Alice, is the sum of all values in the UTXO for which Alice holds the corresponding secret key (sk).

3.2.1 Steps for Getting Paid (Strawman Scheme)

1. $(pk, \ sk) \leftarrow \text{Gen}(1^k)$ – Generate public and secret keys.

2. Send pk to payer (don't need to do this on-chain).

3. Payer creates a transaction that pays the recipient.

4. Payer broadcasts this on the network such that all nodes receive the transaction via the gossip protocol.

5. Validate transaction (check the amount, etc.).

6. Payment is confirmed.

As an example, consider a situation in which Charlie wants to create a transaction that transfers 1 unit to Alice. Charlie requests Alice for her public key. He will then create a transaction. The transaction includes the amount, Alice's public key (pk_A), and a pointer to the UTXO that is being spent. The UTXO must be owned by Charlie, i.e. Charlie knows the secret key (sk_C) corresponding to the public key of the UTXO. Charlie must sign the transaction using his secret key sk_C to prove his ownership of the UTXO and so that the transaction cannot be changed. He then broadcasts the transaction to the network and it is validated (by Alice and other nodes).

For Marabu, the unit of money will be called a *bu*. Because computers typically have trouble representing floating point numbers, for Marabu, all transaction amounts will be represented as integer amounts of a unit. This unit will be 10^{-12} of a bu (a *picabu*) and all transactions will be represented as number of picabu. This is similar to Bitcoin where all transaction amounts are represented in number of Satoshis (1 Satoshi = 10^{-8} bitcoin).

3.2.2 Law of Conservation

The law of conservation requires inputs to strictly equal outputs, while the "weak" law of conservation requires inputs to exceed or equal outputs (money *can* be destroyed).

3.2.3 Format of a transaction

An outpoint defines a single output of a prior transaction and is represented by the transaction id (the hash of a transaction) and the index (because a transaction may have multiple outputs) of the output of the prior transaction. Transactions are represented as an array of inputs and an array of outputs. Each element of the input array is a tuple of an outpoint and a signature of the transaction under the secret key of the public key associated with that outpoint. Each element in the output array is a tuple of a public key and the value to be given to that public key. A transaction is represented like:

```
tx = {inputs: [{outpoint: {txid, index}, signature}],
      outputs: [{public key, value}]}
```

3.2.4 Creating a transaction:

1. Create a transaction with the public key of the receiver(s) and the value in the output(s).

2. Find elements of the UTXO set to constitute the required amount.

3. Sign the entire transaction with the respective secret keys of all UTXOs being spent. While signing the transaction, the signatures are first replaced by `null` to create an unsigned transaction. The unsigned transaction is used as the message for generating the signatures. Then the signatures are put into the transaction to create a signed transaction.

4. Broadcast signed transaction.

When transactions are broadcasted, instead of the entire transaction being broadcasted, just the hash of the transaction is broadcasted (in order to reduce network traffic). If when verifying, a node does not have a transaction with a matching hash, they will request the network for the full transaction.

3.2.5 Validating a transaction:

1. Check that each input point to an existing UTXO

2. Check signature(s)

3. Check weak law of conservation

4. Erase consumed UTXO(s)

5. Create new UTXO(s)

Each validating node maintains a UTXO set. Validation is done with respect to that UTXO set. Steps 4 and 5 above describe how this UTXO set is updated and maintained.

In our implementation of the Marabu node, if a transaction is not valid, a node should not gossip it further, so as to prevent spam on the network.

Chapter 4

Blocks and Chains DRAFT

4.1 Protecting against Double Spend Attacks

4.1.1 Network Delay

To ensure complete security of the network, all nodes must have ledgers that completely agree on the set of transactions that has happened[1]. Otherwise, if two nodes have ledgers that disagree than one another, then there may be disagreements on what transactions are valid and what transactions are not. For instance, if Alice believes that she has 10 dollars left, whereas Bob believes that she only has 3, then if Alice makes a transaction with a value of 5, she will think that her own transaction is valid, whereas Bob will not, leading to a disagreement.

Using the model we have developed in class so far, it is impossible for there to be complete consensus among the ledgers. This is because we assume every network has a network delay. Network delay, or Δ, is defined as the maximum time needed for a message to reach from one honest party to every other honest party. In other words, when a honest party makes a transaction, it may take up to Δ time interval for the transaction to propagate to every other honest party. The existence of network delays allows adversaries to make a double spend attack, which is when an adversary can spend the same UTXO more than once.

4.1.2 Double Spend

The adversary can achieve double spending by signing and broadcasting two different transaction that spends the same UTXO. Let's call these tx_1 and tx_2 (See Figure 1). Now, notice that if it takes a maximum of Δ for a transaction to be broadcasted to every other honest note, then the honest nodes could receive these two transactions in different orders, as long as both transactions are broadcasted within a time span Δ. For example, honest node B could receive tx_1 first and validate that transaction, and then reject the tx_2 that comes later because an outpoint of tx_2 is no longer in the UTXO set. On the other hand, honest node C could receive the transactions in the opposite order, and validate tx_2 whilst rejecting tx_1

---

[1]Interestingly, it is not necessary to require that every node agree on the order of transactions, because there is only one way to draw a construct a UTXO graph out of any valid set of transactions anyway.
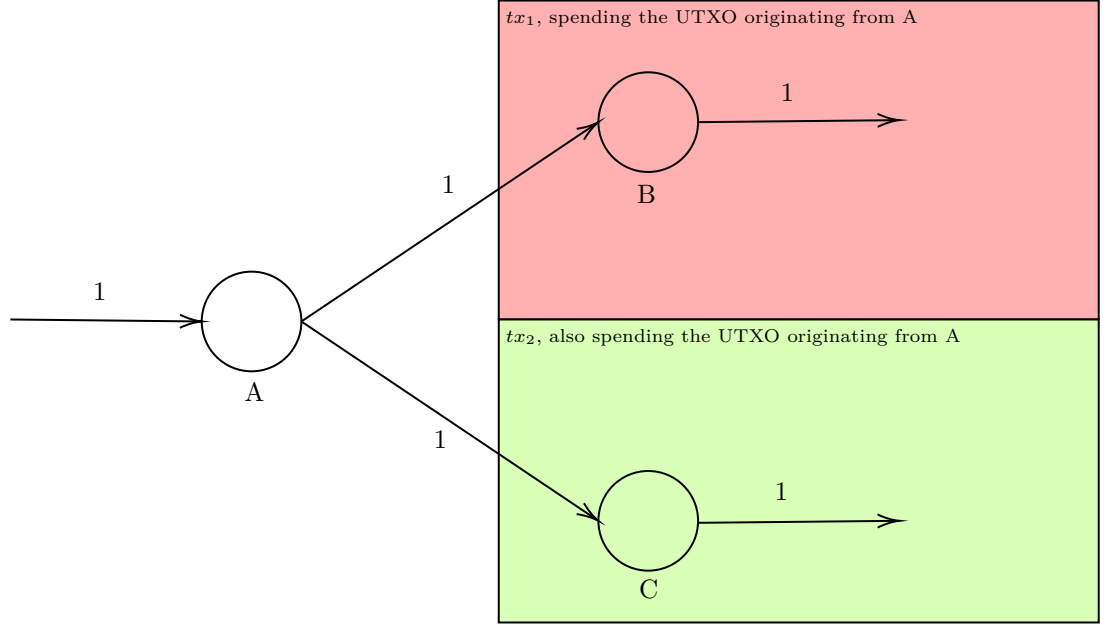
Figure 4.1: Under our current scheme, the honest node cannot distinguish whether tx_1 or tx_2 is valid, if both are broadcasted within the network delay Δ

as invalid. As a result, the ledgers $L_B \neq L_C$, and there will be disagreement about ownership.
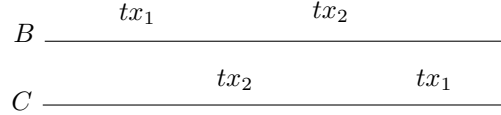
Figure 4.2: A possible configuration of transaction arrival times that make the protocol vulnerable to double spends, assuming both transactions reach both parties within time Δ; B and C disagree whether tx_1 or tx_2 came first, i.e. which one is valid.

What are some simple solutions that could solve this double spend problem? One idea is to just cancel double spends, since no honest node would ever double spend. We would cancel both transactions if double spend occurs. However, consider the scenario where an adversary first buys an item in one transaction, and goes on to double spend the money used in that transaction after every honest node receives the first transaction. When both transactions are later cancelled, the adversary still owns that item, so the seller loses money in that case.

A second idea is to accept the first transaction that is received and cancel the second transaction. However, as previously discussed, different nodes may receive the transactions in different orders if they are broadcasted within the time span Δ. As such, each node will believe that the first transaction they received is valid, and since the order of transactions cannot be guaranteed, the ledgers disagree in this

case as well.

A third idea is to set up a time window. If both transactions are received within the window, then we cancel both transactions. Otherwise, we accept the first transaction and cancel the second transaction that lies outside of the time window. Ideally, the time window would allow all honest nodes to receive the first transaction, before the second transaction is then broadcasted. However, consider the scenario where the adversary broadcast the second transaction tx_2 at time t just before the end of the time window, such that $t + \Delta$ lies outside of the time window. In this case, some nodes would recieve tx_2 within that time window, and other nodes would receive it outside of the time window. Hence, there will be disagreements once again.

It is clear that none of our preliminary ideas work, and to adequately defend against a double spend attack, we need to introduce the idea of a Block.

4.2 Blocks and Blockchains

4.2.1 Virtues of Ledgers

There are two virtues of ledgers that should be satisfied:

- **Safety:** honest parties should agree and achieve consensus.

- **Liveness:** transactions created by honest parties are added to the ledgers of honest parties "soon".

As observed from previous examples, double spends occur because network delays can cause disagreements in the arrival time of different transactions. If somehow transactions could only be created in intervals greater than the network delay parameter Δ, there would be no disagreements because each transaction would have already been broadcasted to all honest parties before the next one gets broadcasted. This gives rise to safety. However, requiring such a delay between transactions means that there will be less transactions being broadcasted every time period, and as such it may be the case that the liveness property is not satisfied. This presents a tradeoff between safety and liveness: as Δ increases, there is less certainty for liveness to be guaranteed.

4.2.2 Blocks

One way to satisfy both the virtues of safety and liveness is to group multiple transactions together and then broadcast them as "blocks", rather than individually. Each block, denoted $\bar{x} = (tx_1, tx_2, \dots)$, contains a sequence of transactions in any order that respects the topological sorting of the corresponding UTXO graph.

In this model, double spends can only occur if the two transactions that creates the double spend exist in 2 different blocks that arrives in different order to different honest parties, similar to the previous example with transactions. If the two double spending transactions appeared within the same block, then there is no valid UTXO graph for the transactions of that block, and so the entire block is invalid. If the two double spending transactions appear in two blocks that are further than Δ apart, then the first transaction would have been validated by all honest parties, and so the second transaction fails to double spend.

Hence, all we need to do to ensure protection against double spending is to ensure that each **block**—rather than each transaction—is rare, which preserves safety while ensuring a higher level of liveness.
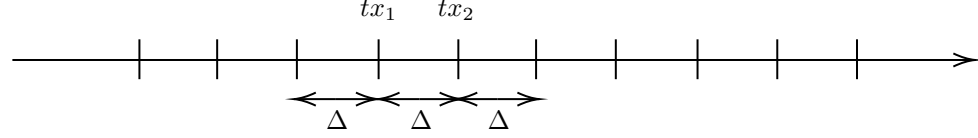

Figure 4.3: In an ideal scenario, each blocks are broadcasted exactly Δ away from each other, so in a double spend we can always accept tx_1 and reject tx_2

4.2.3 Proof of Work and Mining

To ensure that the ability to create blocks is rare (i.e. that one can only be created after Δ has passed since the last block), we devise a scheme to issue rare tickets for creating a block. To do so, we will introduce the concept of Proof of Work.

Definition 5 (Proof-of-Work Equation). $H(B) \leq T$ [2]

Here, T is called the target. In order to solve the Proof-of-Work equation, one would have to find a value of B such that $H(B) \leq T$.

Definition 6 (Random Oracle Model). *Output of H is distributed uniformly at random.*

Under the Random Oracle Assumption, notice that the time it would take to find a bruteforce solution to a Proof-of-Work equation depends on the value of T. When T is larger, it would be easier to find B such that $H(B) \leq T$, and vice versa. In particular, $Pr[H(B) \leq T] = \frac{T}{2^\kappa}$ for any arbitrary B.

In the network, T is fixed and hard-coded for honest parties. The value is chosen such that the time it takes for a node in the network to find a solution to the Proof-of-Work equation is around Δ.

Now, notice that if we allow honest parties to pick whatever B they want, then as soon as the first valid value of B is found, each node can just trivially reuse that value instantaneously, and safety is lost again. As such, in the network, we require that $B = \delta||\bar{x}||ctr$, where $||$ means concatenation and where δ is the hash of the previous block $H(B')$. Both δ and $\bar{x}$ are fixed, and ctr is a number to be found that solves the Proof-of-Work equation. In other words, when nodes are trying to solve the Proof-of-Work equation, they are trying different values of ctr until finding a satisfactory one. The time it takes the first node to find a satisfactory ctr for their block will be around δ. The process of finding a bruteforce solution to the Proof-of-Work equation is called mining.

4.2.4 Blockchains

The previous block hash, δ, is a part of B because we want to ensure "freshness", which means that only blocks that are recently created should be able to be broadcasted. Consider the scenario where an adversary mines multiple blocks, but does

---

[2]This scheme was derived in 1993 by Cynthia Dwork and Moni Naor in the seminal paper "Pricing via Processing or Combatting Junk Mail".

not broadcast them immediately. Then, at a later point in time, the adversary broadcast all of these blocks at the same time. Since there is not a Δ interval between the time these blocks are broadcasted, the problem of double spend arises again. There could be disagreements in the ledgers of the honest nodes. In order to prevent this, we require each block to point to the most recent known block. By including δ as a part of B, we ensure that blocks that were mined beforehand are no longer valid, since the adversary could not have predicted δ before the most recent block was broadcasted. As a result, we can guarantee that blocks will be roughly be broadcasted in regular time intervals. Since blocks are all connected to each other in this way, this chain of block is called the *blockchain*.

4.2.5 Genesis Block

The Genesis Block is the first block in a blockchain. It contains real world data to anchor time, in order to ensure that it is impossible to pre-mine. If the hash of the Genesis Block is known before it is broadcasted, then the adversary could pre-mine multiple blocks beforehand (before the network is even available to the rest of the world). Then, when the Genesis Block is broadcasted, the adversary would be able to broadcast multiple blocks within a short time span. Hence, it is necessary for the Genesis Block to contain real world data to anchor time. For example, Bitcoin genesis block quotes the January 3 2009 The Times headline.
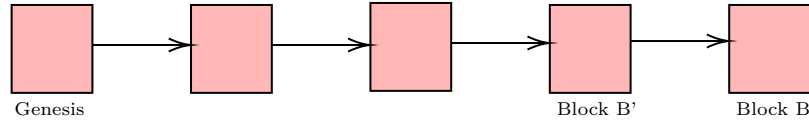
Figure 4.4: To mine block B, the miner runs the Proof of Work Equation with B $= H(B')||\bar{x}||ctr$

4.2.6 Honest Party Block Generation Algorithm

Putting it all together, an honest party (which we can assume to be every honest node in the network for now) follows the following algorithm to create a valid block. First, it collects any amount of transactions from the network and add them to a mempool, which is a set of validated but unconfirmed transactions [3]. Then, it writes $B = \delta||\bar{x}||ctr$, and then solves the Proof-of-Work Equation. In other words, it finds a value of ctr (which is also known as the nonce) such that $H(B) \leq T$. If the honest party is successful, it will broadcast the mined block to the network. Otherwise, if another block has been found and received by the honest party first, it will validate the new block, update it's UTXO by removing all the transactions that were spent in the newest block, and update the freshest block in its memory. As such, B would necessarily change, and the honest party will have to solve a new Proof-of-Work equation.

---

[3]The number of transactions it collects will be formalized later

Chapter 5

Proof-of-Work DRAFT

5.1 Introduction

In this lecture, we will delve deeper into chains, resolve the last few problems with proof of work and the double-spending problem, explain how money is created, and describe how to update the UTXO. By the end of this lecture, while there will still be optimizations to be done, you will have all the necessary knowledge to implement a fully functional and secure blockchain.

5.2 Target Parameter (T)

In the previous lecture we spoke about the necessity of sending messages separated by time intervals, i.e. in the form of rare events. This is done to prevent adversaries from double spending by sending transactions from the same outpoint within a short period of time, thereby causing confusion about which transaction is valid.

The time interval between sending messages should be such that it is greater than the network delay so that transactions are allowed to propagate to every node in the network thus preventing adversaries from double spending.

In order to tie blocks to rare events, the protocol sets a target parameter T such that a block B is valid only if

$$H(B) < T \tag{5.1}$$

5.3 Safety and Liveness

Increasing T leads to a loss of safety. This means that there is an increased likelihood of honest nodes receiving conflicting transactions. Decreasing T leads to a loss of liveness, this is because it takes longer to generate a valid hash that is below the threshold. Safety can be thought of as the honest ledgers being in agreement with each other. Liveness can be thought as the ledgers making progress by adding new transactions.

5.4   Freshness

To produce the hash of the block we run our hashing algorithm on $B = (s, \overline{x}, \mathrm{ctr})$ where s is the hash of the previous block, $\overline{x}$ is the ordered set of transactions that are included in the block and ctr is a randomly chosen nonce. We include the set of transactions in the proof-of-work so that the adversary cannot reuse the same proof-of-work multiple times.

The hash of the previous block is included in the new block we generate so as to maintain freshness. Freshness provides an arrow of time. Since each new block includes the hash of the previous block we can confidently say that the new block was produced after the previous block. This prevents adversaries from premining blocks and adding them when convenient because the adversary must include the hash of the previous block when producing the block that comes after that.

Linking blocks to each other with the help of hashes forms a *blockchain*. It is also known as a hash chain. This makes it impossible to go back and change a block without having to change all the blocks that came after that block as shown in Figure 5.1.
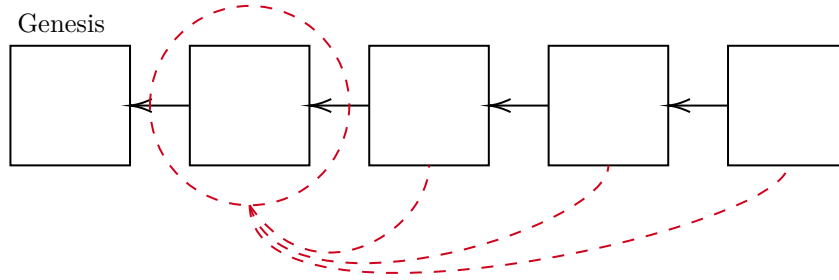
Figure 5.1: Chaining blocks prevents changing the old blocks as this would invalidate the proof-of-work equation for the blocks following it in the chain.

5.5   Determining the Target Parameter

For simplicity, consider that B is randomly chosen in $\{0, 1\}^{\kappa}$. Then,

$$\Pr[H(B) \le T] = \frac{T}{2^{\kappa}} = p \tag{5.2}$$

- p: Probability of successful query

- q: hash power of a single party

- n: number of parties participating in network

Here, we think of one party as one unit of computation. All parties in the network are simultaneously trying to create a block.

The target variable T should be chosen such that the time taken to generate a valid nonce (B) should be greater than the network delay.

$$\frac{1}{pnq} = \Delta \tag{5.3}$$

where Δ is the network delay.

$$T = \frac{2^\kappa}{\Delta nq} \tag{5.4}$$

Here, qn is the total hashing power of the network. We can see that as qn increases T should decrease. This means that it should become harder for each party to generate a valid nonce. Increasing the network delay will also similarly require T to be decreased.

5.6 Accounting for Stochastic Nature of Proof of Work

Ideally we would want that all blocks are produced with the same delay as decided by the equation above. However, the Proof-of-Work algorithm is stochastic. Thus based on our above calculation, the expected time to produce a block will be Δ but in practice there may be instances where blocks are produced in quick succession without the required time delay. This takes us back to square one in terms of thinking about how to prevent double spends. Figure 5.2 shows convergence opportunities (i.e. periods of time where an event happened spaced out by more than Δ from both the previous and next event) and conflicting events (i.e. periods of time where two events happened in a time interval shorter than Δ) produced during the mining process.
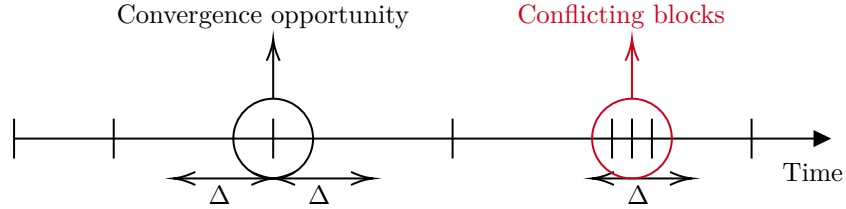
Figure 5.2: Block production during operation is a stochastic process

5.7 Honest Majority Assumption

Thus when we have a convergence opportunity, i.e. block produced with the stipulated time delay we want all the honest nodes to agree that a particular block is the freshest. We need some sort of voting scheme for the honest blocks to accept the latest block as the freshest one. We cannot have each node vote once on which block is the freshest as the adversary could carry out a Sybil attack.

Instead, we will have the honest node add blocks to the longest chain. The assumption here is that a majority of the computational power is controlled by honest parties. Due to this, the length of the chain mined by honest blocks will always be greater than the length of the chain mined by the adversary. As a result, new honest blocks will always be added to the longest chain. Figure 5.3 shows the
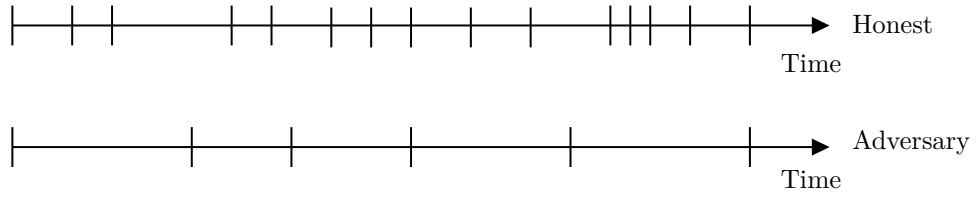
Figure 5.3: Block production of honest nodes vs. the adversary

block production of honest nodes and the adversary. In order to start building off of the longest chain, we still require convergence events, i.e. blocks separated by a specified time delay because otherwise we would not know which chain to build upon (Figure 5.4). Thus we now have a blockchain that all the honest nodes can agree upon.

In order to have a mathematical guarantee that the blockchain converges, the honest majority assumption stated below has to be upheld.

$$t < n - t \qquad (5.5)$$

where t is the computational power of the adversary and n.
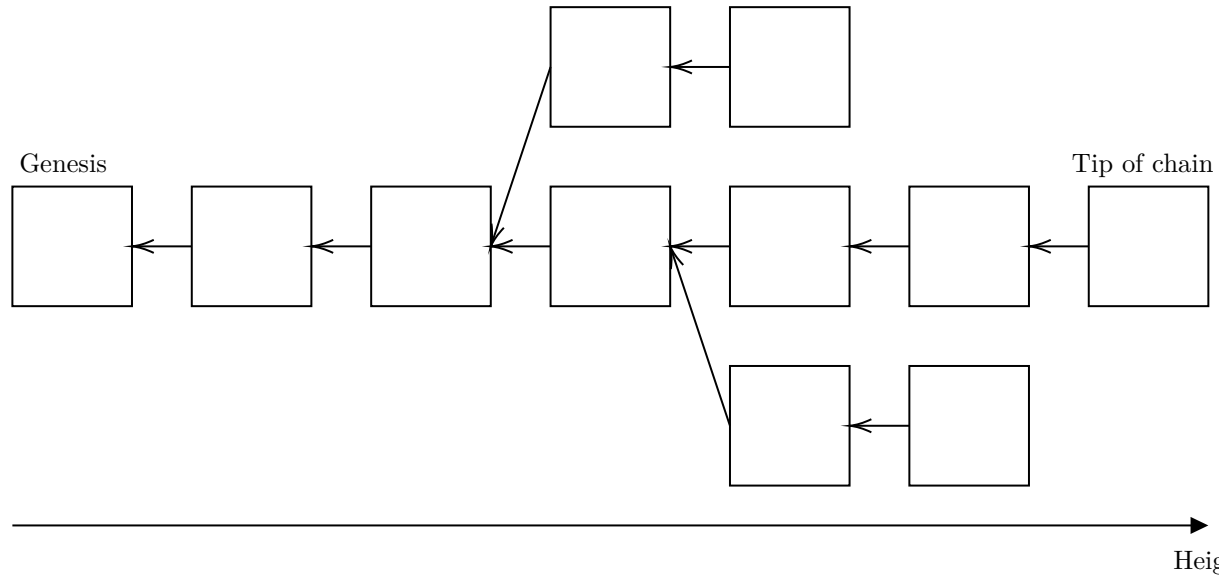
Figure 5.4: Block trees and the longest chain rule. There may be temporary forks because of conflicting blocks. But, the occurrence of convergence opportunities ensures that eventually, there will be a unique tip of the longest chain.

5.8   Coinbase Transactions

While we have successfully implemented a way to validate ownership of money in a decentralized manner, we have yet to see how to tackle the creation of money.

To do so, we introduce the notion of a "coinbase transaction". This transaction is uniquely present in every block. The miner creates this transaction, and can reward any public key they want with an amount less than or equal to a fixed amount f (the block reward) determined by the network plus transaction fees (the fee reward). These fees correspond to the difference between inputs and outputs in each transaction confirmed in the block. Due to the weak law of conservation, recall that there can be a difference between the sum of the inputs and the sum of the outputs in each transaction. This difference is paid out as fees. In other words, the miner can create a transaction with no input and one output with an amount a respecting the following equation:

$$a \leq f + \sum_{i \in \text{input}} i.v - \sum_{o \in \text{ouput}} o.v \tag{5.6}$$

where the sum is over all transactions in the block. In our Marabu protocol, we use $f = 50$ bu.

Finally, to avoid multiple identical transactions that would have the same hash as a result of having the same amount and same public key output, an additional *height* parameter is included in the coinbase transaction. This parameter corresponds to the number of blocks that this block is away from genesis.

5.9   Verifying Objects

5.9.1   Verifying a Block

Upon receiving a block, a honest node should:

1. Verify the proof of work (and do this first to avoid spam attacks).

2. Verify the parent block recursively (and if it does not exist locally, request it from the network).

3. Verify the transactions inside the block, including the coinbase transaction (and if it does not have them, request them from the network).

5.9.2   Verifying a Chain

Upon receiving a chain, a honest node should:

1. Verify each block in the chain.

2. Check that it starts with genesis (or is connected to a block known to be connected to genesis).

3. Adopt the longest chain or one of the longest chains.

5.9.3   Verifying a Regular Transaction

See lecture 4

5.9.4   Verifying a Coinbase Transaction

A valid coinbase transaction must:

1. be the only one in that block,

2. be the first transaction in the block,

3. have no input and exactly one output,

4. have a value which is less than or equal to the sum of the fixed block reward and the difference between the input and output amounts of all other transactions in the block,

5. not be spent in the same block.

5.10   Comparing Transactions and Blocks

Now that we understand how blocks and transactions work, it is worth drawing parallels between the two.

	Transaction	Block
Inductive base	Coinbase	Genesis
Inductive hypothesis	Outpoint UTXO	Previous ID (s)
Inductive step	Consuming produced UTXO Signatures Conservation laws	Proof of Work Causality Transactions

5.11   Updating the UTXO

For each received transaction, if the transaction is valid, apply the transaction to the previous UTXO to get the new UTXO.

For each reveived block, apply each transaction of the block to the previous UTXO. If at any point a transaction is invalid, reject the entire block and revert to the previous UTXO.

For all transactions not yet in a block, add them to a mempool with a temporary UTXO which starts as the same UTXO as the current longest chain tip. These transactions are added in the order in which they are received. If the transaction is invalid with respect to the current UTXO, reject it. When mining a block, use all of the transactions in the mempool to create this block.

Upon receiving a block at the tip of the current longest chain, validate the block and update the UTXO. Also update the mempool and temporary UTXO by applying transactions in the block or removing any transaction that is now either invalid or already in the newly received block.

Upon receiving a block which changes the longest chain to another chain, set the UTXO to the fork between the current longest chain and this new chain (by undoing all transactions after the fork in the current chain). Then, go through each block until the tip in the new longest chain and update the UTXO. If at any point, a block is invalid, reject the new chain and revert to the previous chain and previous UTXO. If all blocks after the fork in the new longest chain are valid, update the mempool by starting with the UTXO at the tip of the new longest chain, applying

every valid transaction in the previous longest chain that is not present in the new longest chain, then applying all the still-valid transactions that were previously in the mempool. This process of adopting the longest chain is shown in Figure 5.5.
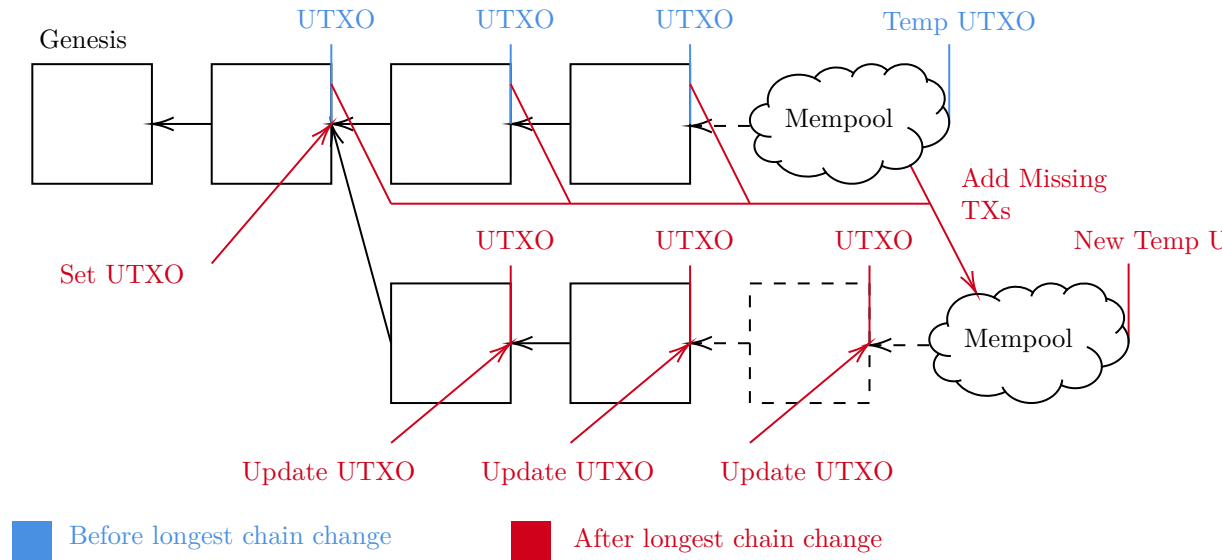
Figure 5.5: Updating the UTXO upon chain change: 1. Set UTXO to the fork. 2. Update UTXO by applying blocks in the new chain. 3. Create a new temporary UTXO starting at the tip of the new chain. 4. Add transactions from the previous chain and previous mempool that are missing in the new chain to the new mempool.

Chapter 6

Chain Attacks DRAFT

6.1 Review of Ledger Virtues

All fundamental use cases and proofs of security of blockchains are dependent on the following virtues being upheld.

- **Liveness**: Honest transactions are included in all honest ledgers "soon".

- **Safety**: For any two honest ledgers L_B and L_D produced a "sufficient time" apart such that L_D is fresher than L_B, L_B must be a prefix of L_D. Note that ledgers may diverge for some time as long as blocks are mined within a network delay Δ, but must satisfy this property upon reaching convergence opportunities (more on this in Section 2.2).

6.2 Chain Virtues

6.2.1 Virtues

The liveness and safety of ledgers directly follow from chain virtues. For this reason, we outline fundamental properties that chains must uphold.

- **Common Prefix**(k): Honest parties agree on their chains with the exception of the last k blocks. This chain virtue will be used to prove ledger safety(Section 1).

- **Chain Quality**(μ): A sufficiently long chunk of the longest chain contains a $\mu > 0$ proportion of honest blocks.

- **Chain Growth**(τ): The chain adopted by honest parties will grow. Chain growth and chain quality will be used to prove the liveness virtue of ledgers(Section 1)

6.2.2 Mechanics of Chain Divergence and Convergence

Chain divergence happens when two or more blocks are mined at approximately the same time. When chain divergence happens, the honest node hash power is split up among multiple different chains. This is because there are multiple longest chains
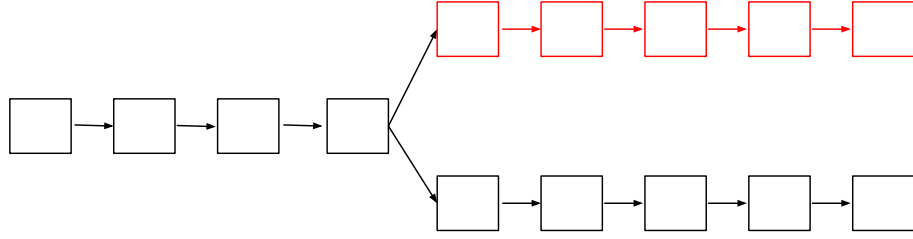
Figure 6.1: These two chains satisfy the common prefix chain virtue with a k of 6. The first 4 blocks of both chains are the honest parties common prefix.
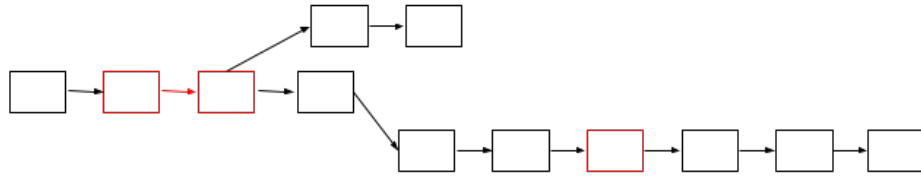
Figure 6.2: In this scenario the longest chain consists of 10 blocks. There are 3 blocks mined by an adversary and 7 honestly mined blocks in the longest chain. Hence, the chain quality μ of this chain is 70%.

that the nodes can choose to extend. If blocks keep getting mined at the same time many chains can grow simultaneously. A convergence opportunity happens when exactly one block is mined with period of silence of duration Δ before and after it. Honest parties move their hash power to extend the longest chain.

6.3 Network Attacks

The attacks discussed in here are based on a strategy that gives rise to a "Nakamoto race".

Chain Quality is defined as the proportion of honestly mined blocks out of all mined blocks in a particular chain. A common conjecture is that Chain Quality can be approximated by the fraction of mining power that honest parties (however, future lectures will reveal that this is not the case).

In terms of chain virtues, the following attacks can break common prefix property and chain quality property if it were able to maintain the longest chain.

6.3.1 Nakamoto Attack

The strategy of Nakamoto attack involves the adversary mining a competing chain in silent, withholding constituent blocks for as long as desired (often when the target adversarial transactions are buried k blocks deep for instant confirmation). Therefore, since honest miners are unaware of that private adversarial chain, they
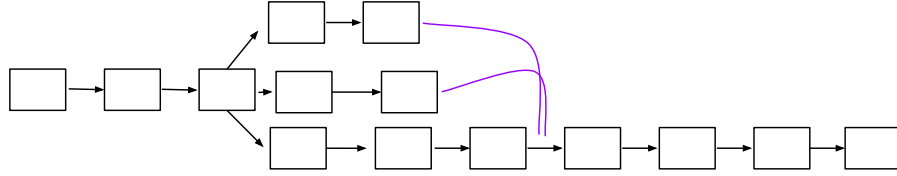
Figure 6.3: In this scenario the chain diverges after the third block. This is because there were three blocks mined at about the same time. There is a convergence opportunity on the sixth block on the bottom chain. At this point honest nodes will switch over since they see the bottom chain is the new longest chain. The purple curved lines represent the honest parties switching over and adopting the new longest chain.

continue extending the longest honest chain. The adversary is, in effect, "racing" to construct a chain longer than the longest honest chain. If the private chain is longer that the honest chain, adversary can release the private chain. In such case, honest participants have to switch and extend the adversarial chain since it is the new longest chain. Thus, this adversarial chain replaces the honest chain. If the honest parties switch chains by reverting more than k blocks, then the adversary has broken the Common Prefix property.

This attack is unlikely to succeed for a minority adversary. Consider the situation where the adversary and the honest parties begin mining at a particular block. Breaking Common Prefix requires that the honest parties mine a chain k blocks long after that common block, while the adversary also mines k or more blocks after the common block. As k is chosen to be a somewhat large value, it will take time for the honest parties to mine k blocks. Given that this span of time will be sufficiently large, the number of adversarially mined blocks will be fewer than the honest convergence opportunities within that timespan. If this is the case, the adversary cannot win.

6.3.2 Fan-out Attack

This type of attack is unique in that it is more due to a fault of the network rather than an exploit from an adversary. Networks are susceptible to fan-out attacks when convergence opportunities are rare.

Infrequent convergence opportunities lead to honest nodes frequently working on competing chains, thus making the chain "fan out" as shown in 6.4. All but one of these competing branches are eventually thrown out, leading to the honest mining power being wasted. An adversary, on the other hand, can coordinate its attack to build on a single withheld chain, winning the race even without having majority hashing power. This is the reason why we do not want fast chains where the target is easily attainable.
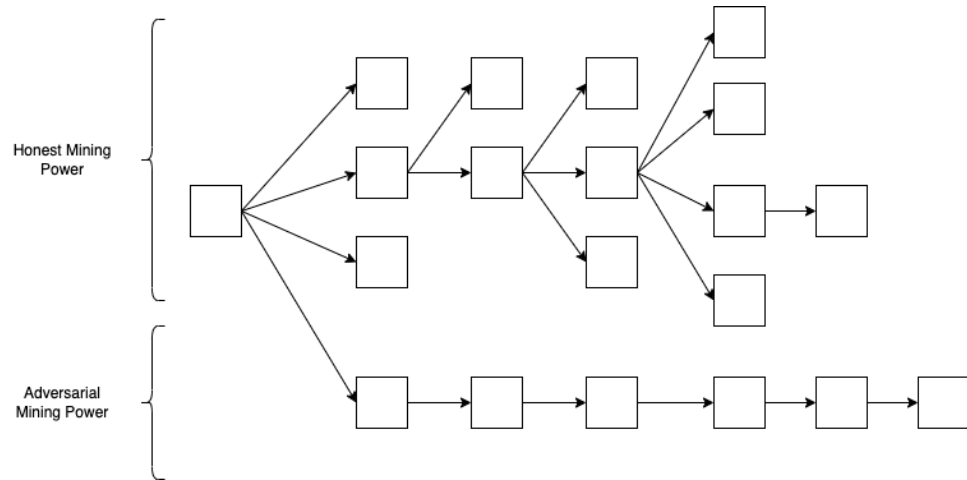
Figure 6.4: A fan-out attack where an adversary who does not hold majority hashing power can still produce the longest chain due to honest mining power being wasted. Displays a drawback of having infrequent convergence opportunities.

6.3.3 Majority Adversary Attack

Now if the adversary held the majority of mining power in a network, all she needs to do is to mine a secret chain, which can be released at any time due to having more compute power. What properties can adversary break if she holds majority of the network hash power?

Clearly, she can violate the Common Prefix property as was mentioned earlier. If honest parties agree on the longest chain containing some block B_m, she mines a secret chain extending from the previous block B_{m-1}. Once the longest chain has k or more blocks after B_m, she releases her secret chain to some of the honest miners. Since the adversarial chain is longer than the honest chain, these honest miners adopt the adversarial chain.

Additionally, an adversary with majority hash power can break Chain Quality as follows. If a block B_m is mined by an honest miner, she creates a new longest chain extending the block B_{m-1}. Thus, the block B_m is "replaced" by a new block mined by the adversary. We will discuss this attack in more details in the next lecture.

However, in such attacks Chain Growth is upheld as long as there are any honest participants mining that force the adversarial chain to grow.

Chapter 7

The Target DRAFT

7.1 Recap of Chain Virtues

There are 3 chain virtues that are relevant to our study:

1. Common prefix with parameter k

2. Chain quality with parameter μ

3. Chain growth with parameter τ

Common prefix has implications for the safety of transactions on chain. Meanwhile chain quality and chain growth affect the liveness of transactions (which refers to how much time it takes for a transaction to be confirmed after it is sent to the network).

7.1.1 Common Prefix

Two or more chains have a common prefix of k if and only if the chains agree on all blocks except the last k blocks at the end of the chains. Chains adopted by different honest nodes satisfy the common prefix property. The probability of violation of this property decreases exponentially as k increases. The greater k is, the more "forgiving" the common prefix property is, and the longer one waits to confirm a transaction. Common prefix property implies safety of the ledger.

7.1.2 Chain Quality

The chain adopted by any honest node contains at least μ fraction of blocks mined by honest nodes.

The number of blocks mined by honest nodes in a chain is important because adoption of a chain by an honest node means it is valid (there are no double spends), but there could be a censorship attack (there are no honestly mined blocks in the chain). This property is required for liveness as adversarially mined blocks may not contain any transactions.

7.1.3 Chain Growth

The chain adopted by an honest node grows at a rate of τ blocks per unit time. This has ramifications for how fast transactions are included in the blockchain, and hence the liveness too.

7.2 Censorship Attack

Many chain attacks fall under the umbrella of a Nakamoto race, where honest parties and the adversary race to extend the length of their respective chains. Here, we will see another kind of attack. We started with the Honest Majority Assumption (henceforth HMA) which means that honest parties have more compute power than the adversary. We will look at a censorship attack that can be carried out by the adversary when she has majority compute power.

In a censorship attack, the adversary tries to prevent a certain transaction from being confirmed. The basic idea is that when she sees a transaction tx in a certain block, she mines at the previous block to prevent that transaction being included in the chain. Since the adversary has majority, she can always win the Nakamoto race and therefore replace every honest block from the longest chain. Therefore, the transaction never enters the longest chain. This attack breaks chain quality. The mechanism is illustrated below.
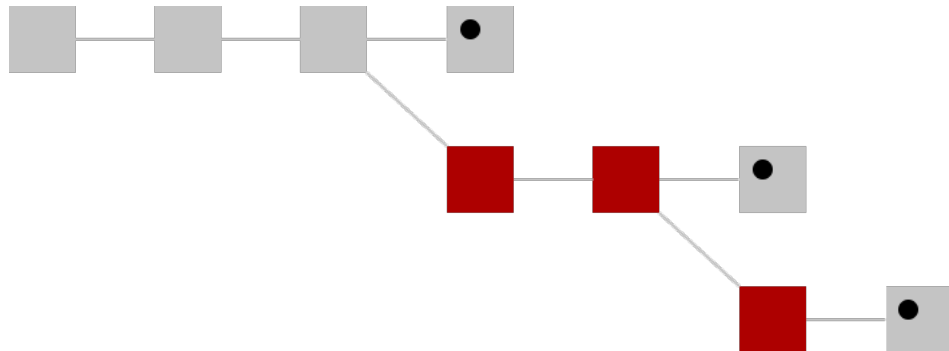
Figure 7.1: Depiction of a censorship attack. Grey blocks are mined by honest miners. Red blocks are valid blocks mined by the adversary. The black dot represents the transaction tx that the adversary is trying to censor.

7.3 Attacks Under Dishonest Majority

When honest majority holds, we have seen that the blockchain satisfies common prefix, chain quality and chain growth. If the adversary has majority of the computing power for some time, she can break common prefix (recall the Nakamoto race attack from the previous lecture). The adversary can also break chain quality (see the censorship attack above). However, chain growth is not broken, although it may be slowed down (i.e. the parameter τ may be reduced), because honest nodes continue to produce blocks on their longest chain.

A majority adversary can revert a transaction (by breaking common prefix). The adversary can also censor transactions (by breaking chain quality). As a consequence, the adversary can break safety or liveness of the ledger. However, the adversary cannot spend coins owned by an honest party because she cannot generate valid transactions for such a signature. The adversary also cannot create more money than what is allowed by the macroeconomic policy, because such blocks would not be accepted by the honest nodes.

7.4 Healing From Attacks

Assume there is a temporary adversarial majority of compute power (TAM). Then the HMA is not respected during some time period δ, and then it is respected again. During the period of TAM, safety and liveness may not hold. The adversary can double spend and can carry out censorship attacks, violating properties like common prefix and chain quality.

After the period of TAM, when HMA is respected again, liveness heals because chain quality is recovered. Chain quality recovers because when HMA is respected, the honest parties will win the Nakamoto race and there will eventually be an honest block that includes previously censored transactions. Also, safety will heal as common prefix heals. Common Prefix recovers because of the reemergence of the HMA, and the presence of convergence opportunities means that the honest nodes will once again converge on their longest chains.

The recovery of HMA presents convergence opportunities. If convergence occurs, then any conflicting transactions which would render a block invalid (i.e. a double spend) are not included in the honestly adopted chain or in the mempool, rather they are dropped.

Since safety and liveness are not guaranteed during TAM, and it takes some time to recover safety and liveness after HMA is recovered, a user should not be using the blockchain during the TAM and a while after (if they are aware of the TAM). This is because safety and liveness cannot be recovered for coins affected during the TAM. For example, if a car dealer delivered a car in exchange for a transaction on the blockchain during TAM, and the adversary reverted that transaction after receiving the car, that money cannot be recovered by the car dealer.

7.5 Selfish Mining

We now ask the question: Is there a lower bound on the chain quality μ, and if there is, what is it? It is intuitive to guess that the lower bound is roughly the percentage of hashing power that is honest. In other words, our conjecture is that:

$$\mu \geq \frac{n-t}{n} \tag{7.1}$$

We will see now that this is false.
Consider the selfish miner $\mathcal{A}$, who does the following:

1. Adopt the longest chain

2. Mine in secret extending the last block of the longest chain

3. When an honest block is found:

(a) If $\mathcal{A}$ has a secret block, broadcast it.

(b) Otherwise, adopt the honest tip.

Then repeat from Step 2.

Here, the adversary $\mathcal{A}$ is also a "rushing adversary", which is one who sees honestly broadcasted messages prior to everybody else. As such, since $\mathcal{A}$ can broadcast her own block before the honest block gets to the rest of the honest miners, the honest miners will adopt $\mathcal{A}$'s block and start building the next block off of that. The honest block that $\mathcal{A}$ had originally seen is now wasted effort. If this continues, the block tree may look something like this:
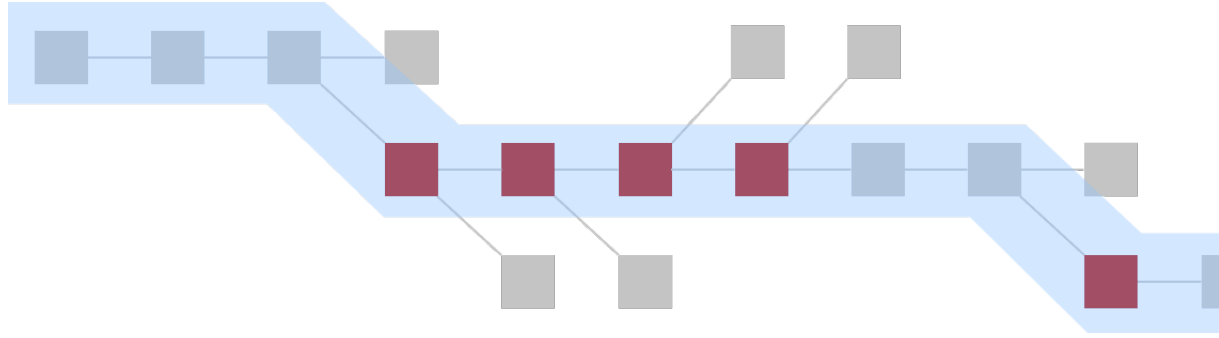
Figure 7.2: In this example, suppose that $n = 17$, $t = 5$ and $\frac{n-t}{n} \approx 0.71$. There are 5 adversarial blocks and 12 honest blocks mined which are proportional to the compute powers. In the longest chain here (highlighted in blue), there are 5 adversarial blocks and 11 total blocks, so we can see that $\mu \approx 0.55$. Thus, we can see that our earlier conjecture ((**??**)) is false.

Due to the wasted efforts of the honest blocks, we can start to see that μ is not necessarily equal to the fraction of honest mining power. None of the adversarial blocks are wasted in this example, but many of the honest ones are. The goal of the selfish mining attack is for every single adversarial block to be contained in the longest chain.

Below is the simulation in typescript for selfish mining. If you run it yourself, you'll see that chain quality is on average less than 0.1, even though adversarial power is 0.49.

```
const ADVERSARIAL_POWER = 0.49
const BLOCK_LIMIT = 500
const MONTE_CARLO = 10

function simulate() {
  let honestBlocks = 1
  let adversaryHeadStart = 0
  let chainLength = 1

  while (chainLength < BLOCK_LIMIT) {
    if (Math.random() < ADVERSARIAL_POWER) {
      // adversary got a block
      ++adversaryHeadStart
    }
    else {
      // honest got a block
      if (adversaryHeadStart > 0) {
        --adversaryHeadStart
      }
      else {
        ++honestBlocks
      }
      ++chainLength
    }
  }
  return honestBlocks / BLOCK_LIMIT
}

let sumQuality = 0

for (let i = 0; i < MONTE_CARLO; ++i) {
  sumQuality += simulate()
}
console.log(sumQuality / MONTE_CARLO)
```

In conclusion, an adversary does not need much mining power in order to incur heavy damage on chain quality.

7.6   What Value of T to Choose?

Last class, we discussed how neither a high T nor a low T necessarily allows for the optimal convergence opportunity frequency. With a high T, blocks are produces more frequently, but they're so frequent that convergence opportunities are extremely rare. With an extremely low T, nearly every block is a convergence opportunity, but the blocks are so infrequent that the absolute frequency of convergence opportunities is low. So, what is the optimal T, you wonder? Below is the code for simulating convergence opportunity frequency with varying T, followed by the accompanying generated plot.

```
import random
import matplotlib.pyplot as plt
import numpy as np

MONTE_CARLO_REPEAT = 30
TIME_INTERVAL = 100

def simulate(eta, Delta):
  convergence_opportunities = 0
  t = 0
  prev_interarrival_time = 2 * Delta
  while t < TIME_INTERVAL:
    interarrival_time = random.expovariate(1/eta)
    if interarrival_time > Delta:
      convergence_opportunities += 1
    t += interarrival_time
    prev_interarrival_time = interarrival_time
  return convergence_opportunities

def monte_carlo(eta, Delta):
  convergence_sum = 0
  for i in range(MONTE_CARLO_REPEAT):
    convergence_sum += simulate(eta, Delta)
  return convergence_sum / MONTE_CARLO_REPEAT

Delta = 1
x = []
y = []
kappa = 256
min_T_exp = 229
max_T_exp = 239
n = 10
q = 3000000
min_eta = 0.01
max_eta = 3.0
eta_step = 0.01
# eta is the expected block interarrival time
for i, eta in enumerate(np.arange(min_eta, max_eta, eta_step)):
  T = 1 / (n * q * eta)
  x.append(T)
  y.append(monte_carlo(eta, Delta))

plt.xscale('log')
plt.xlim((2**(min_T_exp - kappa), 2**(max_T_exp - kappa)))
plt.xticks(
  [2**x for x in range(min_T_exp - kappa, max_T_exp + 1 - kappa)],
  ['$2^{' + str(x) + '}$' for x in range(min_T_exp, max_T_exp + 1)]
)
plt.plot(x, y)
plt.xlabel('Mining target $T$')
plt.ylabel(f'Convergence opportunity frequency in ${TIME_INTERVAL}$ rounds')

plt.title(f'Convergence opportunities when varying the mining target $T$.
\n$\Delta = {Delta}, n = {n}, q = 3 \cdot 10^9, \kappa = {kappa}$')
plt.show()
```

51
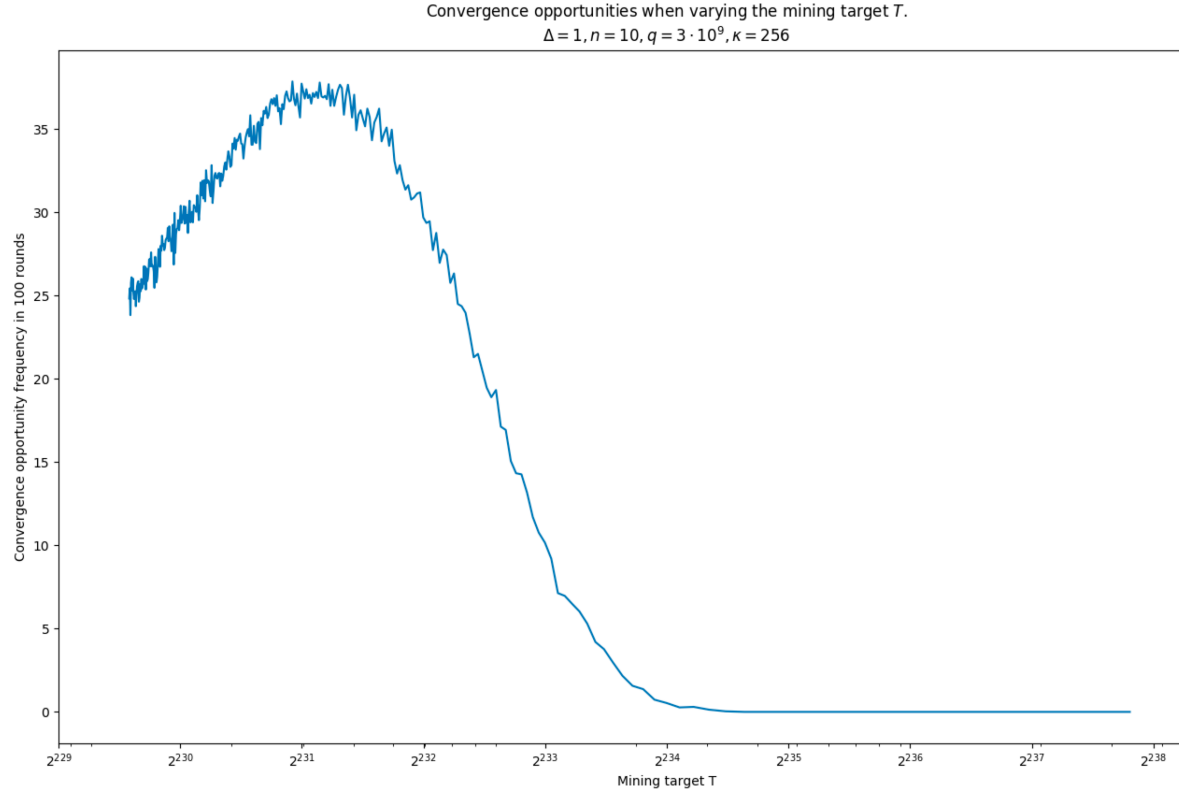
Figure 7.3: Plot of convergence opportunity frequency versus target T

7.7 Our Variables

- κ: security parameter
- $\mathcal{A}$: adversary
- H: hash
- T: target
- p: prob of successful query
- n: total parties
- t: adversarial parties
- q: hash rate
- k: common prefix
- μ: Chain Quality
- τ: Chain Growth

7.8 Some Bitcoin Statistics

We can take a look at the blockchain statistics for Bitcoin on the website: https://www.blockchain.com/charts/hash-rate. Today, the hash rate of the bitcoin network is 210.48m TH/s, measured in terahertz per second, as shown in Figure 7.4. This can be denoted as:

$$q \cdot (n - t) \approx 2^{67} \text{ Hz}.$$
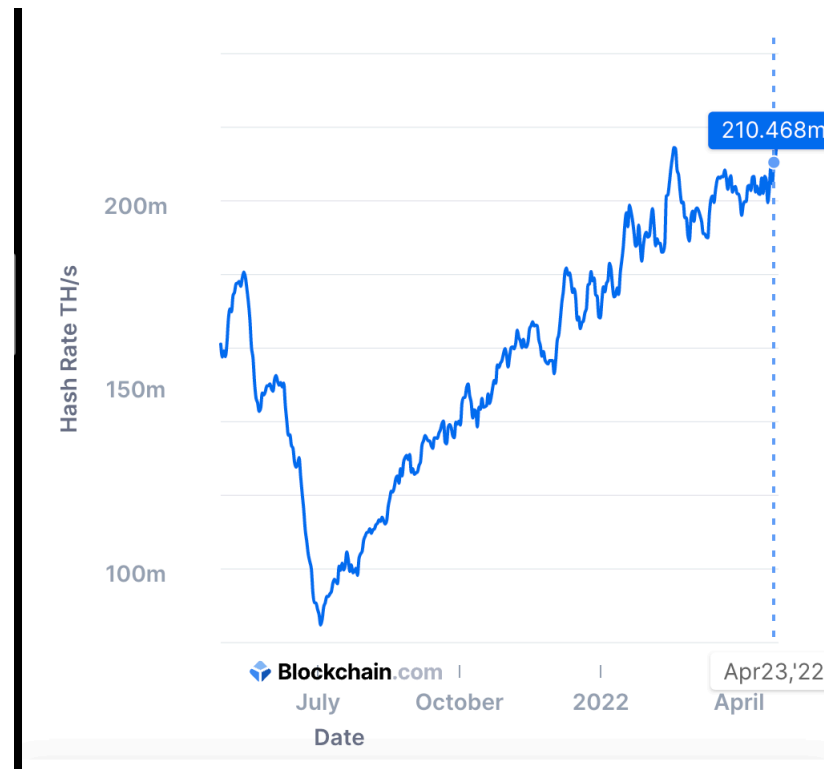
Figure 7.4: Hash Rate of Bitcoin from April 2021 to April 2022[?]

We can estimate the value of q (the hash power of 1 party) on a real computer. On a laptop, $q \approx 100\text{MHz}$. On a GPU, we can raise this to $q \approx 20$ GHz. Today, we also use specialized mining power, with the best machines (ASICs) achieving $q \approx 200$ THz. To see the hash power of the ASIC machine, you can refer to this website: www.asicmine rvalue.com

We can also use the website to examine the number of transactions that are confirmed in any time frame. Here are some other observations:

- The number of transaction spikes in the weekdays and toughs on the weekends

- The number of UTXOs grew significantly in the past year

- The mempool size is more erratic

Overall, observe that many network activity values depend on human factors.

7.9 Mining

7.9.1 Parties

In the world of blockchain, there are parties that are not honest. We would like at least the majority of parties to be honest for our blockchain system to work well, so to encourage honesty it should be disadvantageous to be an adversarial party. In addition, we make the assumption that the members of the honest majority generally act rationally with respect to what is most beneficial economically. Our goal is that for the honest parties to maximize their profits, they should behave in a predictable, rational manner.
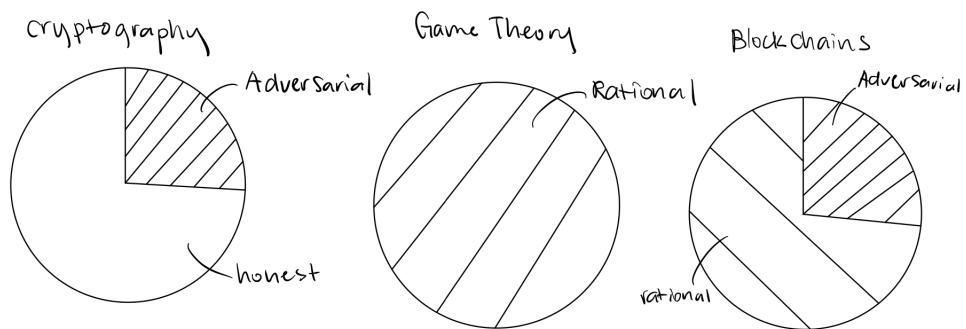
Figure 7.5: Types of Parties in Cryptography, Game Theory, and Blockchains

However, for blockchain, we assume that the users consist of adversarial and rational parties.

7.9.2 The parameter Δ and Block Size Limit

Previously, we defined Δ as the parameter for the network delay, which is also the desired parameter for the average time of block mining. From the previous lectures, we may assume that Δ is a constant. However, in practice, as more transactions gets placed into a block, the size of the block increases, so the time it takes for the entire network download the block also increases. Since we would like network delay to be lower than a small fixed value, in practice blockchains have a block size limit specifying the maximum size in bytes of a block.

7.9.3 Including a transaction

Now that we have a constraint on the block size, the rational miner should adopt some strategies to maximize profit. Difference combinations of transactions in a block will give different rewards, so a rational miner should choose carefully which transactions to include in a block. When putting together a new block to mine, a miner will encounter one of two cases:

> Case 1: Mempool fits in block $\longrightarrow$ include all transactions to the new block

Case 2: Mempool does not fit $\longrightarrow$ sort transactions by their fee-per-byte and include the top transactions until the block reaches the size limit

Case 1 is easy, but Case 2 can be reduced to the knapsack problem, which is NP-hard. Furthermore, transaction ordering is affected by extra constraints: we have to order the transactions in a block such that if a transaction spends the output of another transaction in the mempool, it must appear after the transaction it spends in the block. The actual way transactions are selected are implementation details which are decided by each miner. In any case, since transaction fees provide extra reward, a rational miner would not mine an empty block.

Since miners prefer to mine more profitable transactions (i.e. those with higher fees), the chosen transaction fee would determine the confirmation time for a transaction. If a wallet wants their transaction to be confirmed faster, they would set a higher fee to incentivize miners to include the transaction into their blocks. If the wallet pays a lower fee, the transaction may take longer to be included, or never be included.

7.9.4 Block Reward

Previously, we defined the coinbase value as:

$$\text{Coinbase value} = \text{block reward} + \text{fees}$$

In the Marabu protocol, the block reward is fixed. However, in Bitcoin and many other real-world cryptocurrencies, the block reward decreases over time. Bitcoin implements a macro-economic policy called "reward halving", shown in Figure 7.6, in which the block reward is halved every 4 years. The sum is finite, thus the supply of bitcoin is finite, at around 21 million total.

Other implementations of cryptocurrencies, such as Monero, have also chosen a smooth emissions where the change of the block reward is continuous, and the sum still converges to a constant.

7.10 Variable Mining Difficulty

The Marabu protocol has a constant difficulty parameter, and thus a constant target T. However, this creates a problem since the hash power of the network is constantly changing. Therefore, when the hash power increases, the rate of block production could be less than Δ. To keep a desired block production rate, we want to scale the difficulty appropriately as the hash power of the network increases.

7.10.1 Definitions

Definition 7. *Let f be the probability of getting an honest block in one unit of time. Then,*

$$f \approx p \cdot q \cdot (n - t)$$
$$f = 1 - (1 - p)^{q(n-t)}$$

where $(1 - p)^{q(n-t)}$ is the probability that every honest party failed.
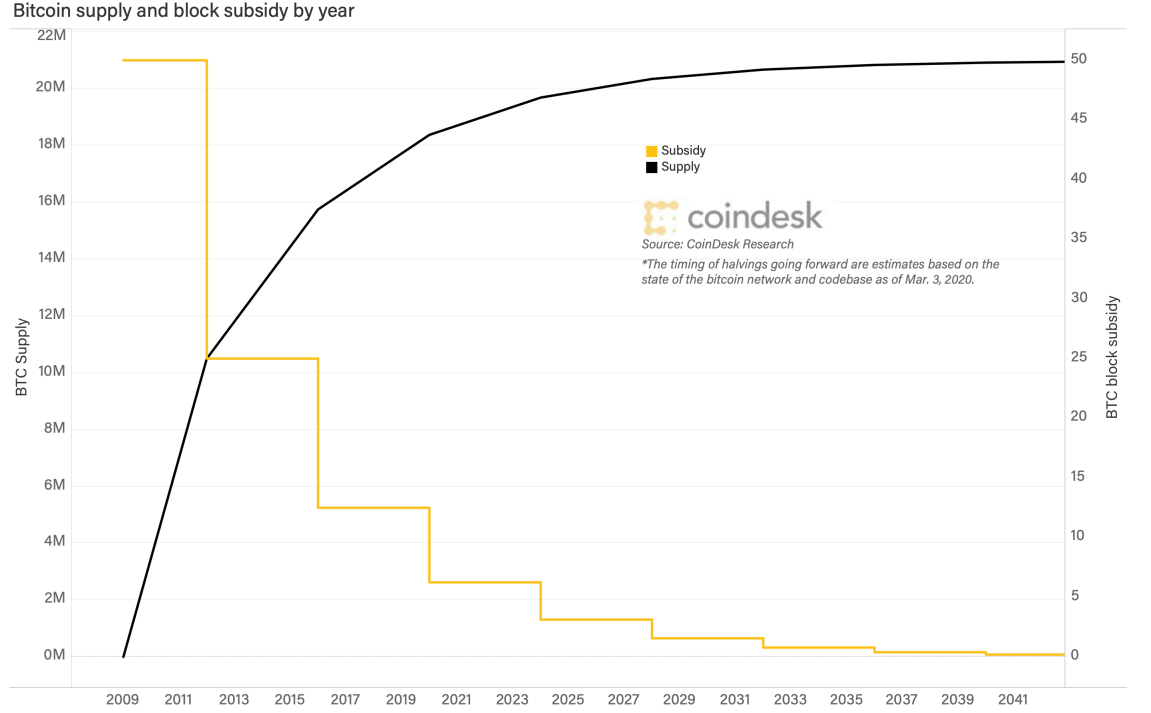
Figure 7.6: Reward Halving for Bitcoin Supply[**?**]

In Bitcoin, where 1 block is produced approximately every 10 minutes, we have $f \approx 1/600$ seconds. For small p, we have

$$(1 - p)^{q(n-t)} \approx 1 - qp(n - t)$$

Definition 8. *Let $\eta = \frac{1}{f}$ be the expected block production duration.*

We split the chain into sections m blocks long.

Definition 9. *Let an epoch be a section that is m blocks long, where m is a constant.*

Given the target for epoch $j - 1$, denoted T_{j-1}, we wish to find the target for the next epoch T_j. The desired epoch duration is $m \cdot \eta$, the number of blocks times the expected production rate, but the actual duration is $t_2 - t_1$ where t_1 and t_2 gives the mining times of the first and last blocks of the epoch $j - 1$, respectively. Then we recalculate the target via

$$T_j = T_{j-1} \frac{t_2 - t_1}{m\eta}$$

We reach the following conclusion:

If actual time $<$ desired $\longrightarrow$ target decreased, difficulty increased.
If actual time $>$ desired $\longrightarrow$ target increased, difficulty decreased

7.11 Mining Pools

The probability of an individual miner successfully mining a block (and earning $200k for the reward) is low. This reward has a high expectation, but it also has high variance. However, the miners want a consistent return, with the same expectation but a lower variance. To do this, miners combine together to form a *pool*.

Definition 10. *A pool is a collaboration of miners. If any one miner succeeds, then they share the profit with other miners.*

7.11.1 How Pools Work

The pool operator, a trusted party, generates a key pair (pk, sk) and shares the public key pk with all miners. The participants mine the block, in which the coinbase transaction goes to the public key pk of the pool operator. Then, the operator distributes profits to the miners.

Now the pool operator must verify that the miners are actually mining. They could achieve this by setting up a light PoW verification.

Definition 11. *The light PoW equation provides a target that is significantly easier, called a light block share:*

$$H(B) \leq 2^{\xi} T$$

where ξ denotes a constant that scales the target.

The participants would send the light PoW block to the operator once they have mined a block. The operator validates that:

1. The light block share satisfied the light PoW equation

2. The coinbase pays the operator

Finally, after the block is mined, the operator distributes profits in proportion to the shares reported. An adversarial miner can only get payed if they submit the valid light block share. Additionally, the miner cannot change a valid block's public key to their own address because this will change the hash. Finally, an adversary would want to share a found block because they would get rewarded as part of the pool.

7.12 Wallets

7.12.1 Mining and Wallets

While miners wish to maximize fees to increase their rewards, wallets wish to minimize fees to decrease the price of transactions. The fee-per-byte is fixed by the user, so one way to lower the transaction fee is to minimize the size in bytes of the transaction. In case a user miscalculates the fee-per-byte and gives a value that is too low, an honest user can submit the same transaction but with a higher fee. This is an "honest" double spend called a *replace by fee*, as shown in Figure 7.7. The higher-fee transaction replaces the older one in the mempool.
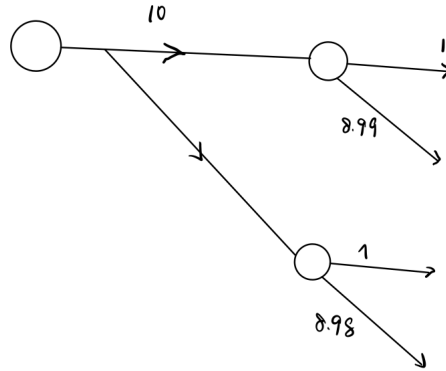
Figure 7.7: An example of a honest, rational party not being able to send a transaction and creating another replace by fee transaction

Types of Wallets

Wallets can be "hot" or "cold". Hot wallets are online, so they are easily available to use but less secure. Cold wallets are stored offline, such as in a hardware wallet or written down on a piece of paper. The hardware wallet could be plugged into a computer. The computer would store the transaction information, while the wallet generates the public keys, secret keys, and the signature without the secret keys leaving the device. They are more secure but tend to be harder to operate.
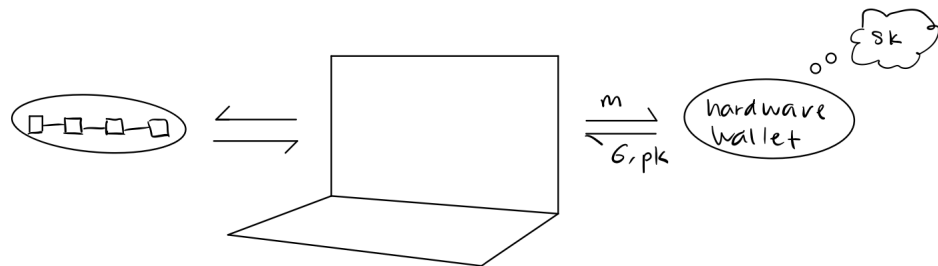
Figure 7.8: Illustration of the interactions between the hardware wallet and the computer

7.12.2 HD Wallets

For a wallet, we want to generate public and secret key pairs $(\text{sk}, \text{pk}) \longleftarrow \text{Gen}(1^\kappa)$.

To do this, one approach is to start with a seed that is randomly generated (such as a series of words), then hash the seed with a counter. Note that we cannot use human-generated random words, such as "I love my dog", because it could be easily stolen.

Figure 7.9: An Example of a Hardware Wallet[**?**]

One commonly used approach is the following. Given a randomly generated *seed*, we can concatenate it with a counter and hash the concatenation to achieve a new secret key. Then, from the secret key, we can generate a public key.

$$H(\text{ctr}\|\text{seed}) \longrightarrow \text{new sk}$$
$$H(1\|\text{seed}) \longrightarrow \text{new sk}_0$$
$$H(2\|\text{seed}) \longrightarrow \text{new sk}_1$$
$$\cdots$$

Chapter 8

Accounts DRAFT

8.1 Accounts Model

8.1.1 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 8.1.
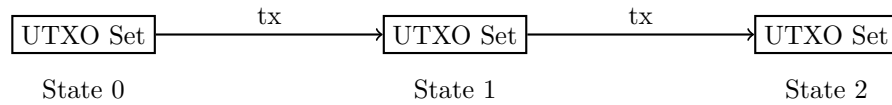
Figure 8.1: State transitions in the UTXO model

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).
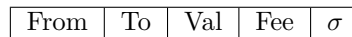
Figure 8.2: Structure of a transaction in the accounts model

For the accounts model, the state is maintained by accounts (public keys) and balances, as shown in Figure 8.3 and Figure 8.4.
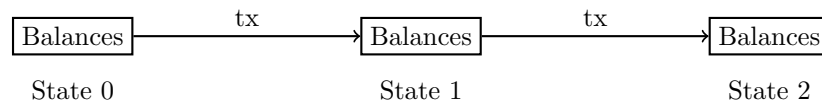
Figure 8.3: State transitions in the accounts model

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:

Balances State	
Alice	5 bu
Bob	100 bu
Dionysis	1 bu

Figure 8.4: Balance State

$$\delta(st, tx) = \begin{cases} st' & \text{if } tx \text{ valid w.r.t. } st \\ \perp & \text{otherwise} \end{cases} \tag{8.1}$$

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

$$\delta_{UTXO}(st, tx) = \begin{cases} st \setminus tx_{in} \cup tx_{out} & \text{if } tx \text{ valid w.r.t. } st \\ \perp & \text{otherwise} \end{cases} \tag{8.2}$$

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

- Check inputs are in st

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

$$\delta_{acc}(st, tx) = \begin{cases} st' & \text{where } st'[tx.from] = st[tx.from] - tx.value, \\ & st'[tx.to] = st[tx.to] + tx.value, \text{ if } tx \text{ valid w.r.t.} \\ & st \\ \perp & \text{otherwise} \end{cases} \tag{8.3}$$

The transaction validation process of the accounts model is

- Check σ

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

8.1.2 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.

From	To	Val	Fee	**nonce**	σ

Figure 8.5: Structure of tx in accounts model

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 8.5.

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 8.6.
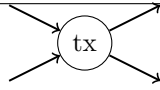
	UTXO	Accounts
Real System	Bitcoin	Ethereum
Transaction tx		From | To | Val | Fee | **nonce** | σ
Transistion δ	Remove consumed outputs and add produced outputs	Update balances $st'[from] := st[from] - value$ $st'[to] := st[to] + value$
Validation	Signature, Law of Conservation, Inputs exist in st.	Signature, Sufficient balance, Nonce unique.
Genesis State	$\emptyset$	$\{\}$

Figure 8.6: Side by Side Comparison of Two Models

8.2   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.

8.3 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.

8.3.1 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.

8.3.2 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.

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

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 8.7. 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.
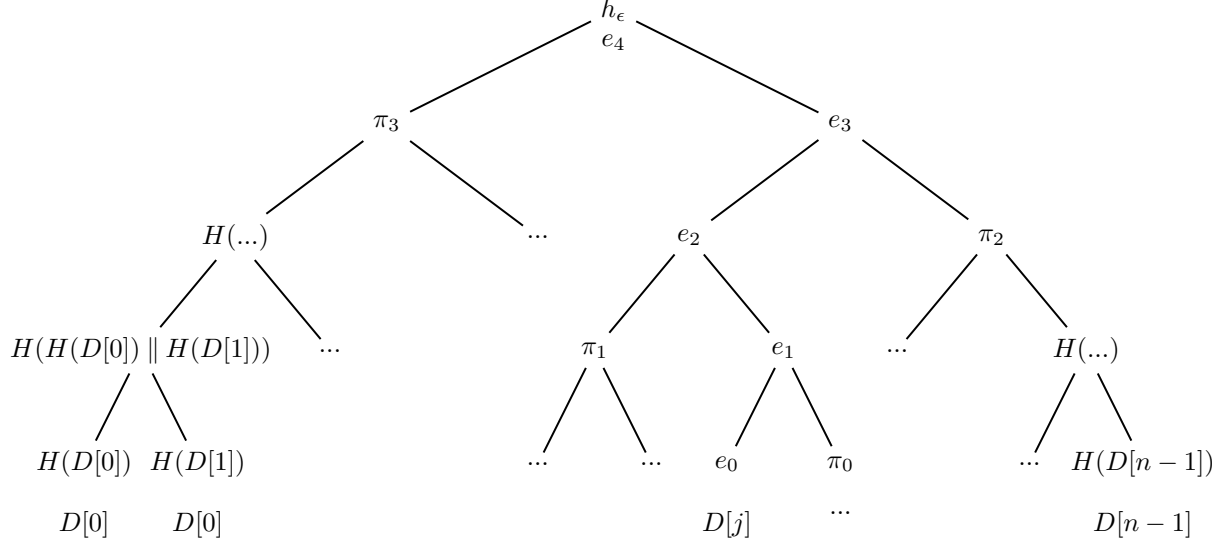
Figure 8.7: Merkle Tree

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

$$\text{compress}(D) \to h_\epsilon, \tag{8.4}$$

$$\text{prove}(D, j) \to \pi, \text{ and} \tag{8.5}$$

$$\text{verify}(h_\epsilon, d, j, \pi) \to \begin{cases} \text{true} & \text{if valid} \\ \text{false} & \text{otherwise} \end{cases}. \tag{8.6}$$

The correctness of the Merkle tree is specified as:

$$\forall D, \forall j, \text{verify}(\text{compress}(D), D[j], j, \text{prove}(D, j)) = \text{true} \tag{8.7}$$

8.3.3 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.

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

Our goal is to prove that

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

Theorem 4. *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.

$$x_1, x_2 \text{ s.t. } x_1 \ne x_2 \wedge H(x_1) = H(x_2)$$
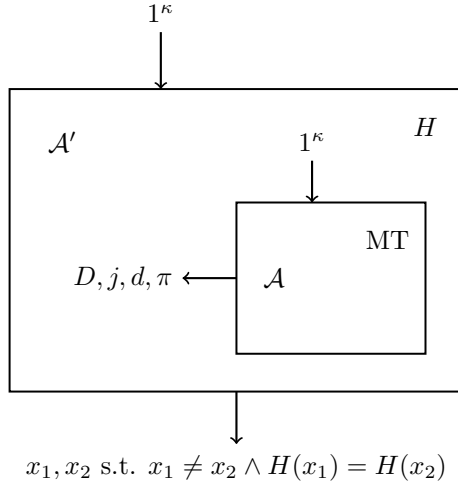
We use e for the hash value calculated by the client, h for the expected hash value in the correct Merkle tree, and π for the hash values returned by the server.

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

$\mathcal{A}'$ 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 ($e_0 \ne h_0$). (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 $x_1 \ne x_2 \wedge H(x_1) = H(x_2)$.

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 9

Light Clients DRAFT

9.1 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.

9.2 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.

9.3 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.

9.3.1 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.

9.3.2 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.

9.4 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 sub-graph 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.

9.5 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).

9.5.1 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.

9.5.2 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.

9.5.3 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).

9.5.4 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.

9.6 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.

9.6.1 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.

9.6.2 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.

9.6.3 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	$\sim 1\text{TB}$	$\sim 1\text{TB}$	$\sim 100\text{MB}$
Honest Majority	$\frac{n-t}{n}$		

Chapter 10

Security in Earnest I DRAFT

10.1 Random Oracle

So far, we have defined our random oracle such that $Pr[H(B) \leq T] = p = \frac{T}{2^k}$. The guarantees of this definition are actually stronger than collision resistance of a hash function, but we will ignore that for now. Instead, we will focus on making our random oracle more concrete.

10.1.1 Naive Random Oracle

A first pass at the implementation of our random oracle would be the following algorithm.

Algorithm 11 Naive Random Oracle
---
    procedure H(B)
        $y \xleftarrow{\$} \{0,1\}^\kappa$
        return y
    end procedure

While this satisfies the statistical properties of our random oracle, this algorithm is not a hash function, and is not consistent, i.e. it does not return the same output if it is called with the same input. This inconsistency renders it useless for our purposes, so we modify it in the next section to better represent our true random oracle.

10.1.2 Our Random Oracle

Our modification of the random oracle above involves the inclusion of a state $\mathcal{T}$.

We see that this achieves consistency, i.e. $H(x)$ returns the same output every time across different calls with the same input x, for any x. However, our storage symbol $\mathcal{T}$ must be consistent over all parties in the network, and therefore does not represent a local mapping, but rather turns our oracle into a so-called "network function".

Algorithm 12 Random Oracle

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

10.2 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 let's us discretize time by eliminating complexities of network distance and speed. Further, 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.

10.3 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 13 The environment and network model running for a polynomial number of rounds $\mathsf{poly}(\kappa)$.

1: $r \leftarrow 0$
2: **function** $\mathcal{Z}_{\Pi,\mathcal{A}}^{n,t}(1^\kappa)$
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:            $\mathsf{assert}(m \in \overline{M}[i])$ ▷ Non-eclipsing assumption
24:          **end for**
25:        **end for**
26:      **end while**
27: **end function**

10.3.1   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.

10.3.2   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).

10.3.3 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.

10.4 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 14 The Hash Function in the Random Oracle Model

1: $r \leftarrow 0$
2: $\mathcal{T} \leftarrow \{\}$ $\triangleright$ Initiate Cache
3: $Q \leftarrow 0$ $\triangleright$ q for honest parties, qt for adversary
4: **function** $H_\kappa(x)$
5:    **if** $x \notin \mathcal{T}$ **then** $\triangleright$ First time being queried
6:      **if** $Q = 0$ **then** $\triangleright$ Out of Queries
7:        **return** $\perp$
8:      **end if**
9:      $Q \leftarrow Q - 1$
10:      $\mathcal{T}[x] \xleftarrow{\$} \{0,1\}^\kappa$ $\triangleright$ Sample uniformly at random from output space and store in Cache
11:    **end if**
12:    **return** $\mathcal{T}[x]$ $\triangleright$ Return value from Cache
13: **end function**

10.5 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 15 The honest party

1: $\mathcal{G} \leftarrow \epsilon$
2: **function** CONSTRUCTOR($\mathcal{G}'$)
3:     $\mathcal{G} \leftarrow \mathcal{G}'$ ▷ Select Genesis Block
4:     $\mathcal{C} \leftarrow [\mathcal{G}]$ ▷ Add Genesis Block to start of chain
5:     round $\leftarrow 1$
6: **end function**
7: **function** EXECUTE(1^κ)
8:     $\tilde{\mathcal{C}} \leftarrow \text{maxvalid}(C, \bar{M}[i])$ ▷ Adopt Longest Chain in the network
9:     **if** $\tilde{\mathcal{C}} \neq \mathcal{C}$ **then**
10:         BROADCAST($\tilde{\mathcal{C}}$)   ▷ Gossip Protocol
11:     **end if**
12:     $x \leftarrow \text{INPUT}()$ ▷ Take all transactions in mempool
13:     $B \leftarrow \text{PoW}(x, \tilde{\mathcal{C}})$
14:     **if** $B \neq \perp$ **then**   ▷ Successful Mining
15:         $\mathcal{C} \leftarrow \tilde{\mathcal{C}}||B$ ▷ Add block to current longest chain
16:         BROADCAST($\mathcal{C}$)   ▷ Gossip protocol
17:     **end if**
18:     round $\leftarrow$ round+1
19: **end function**
20: **function** READ
21:     $x \leftarrow \epsilon$ ▷ Instantiate transactions
22:     **for** $B \in C$ **do**
23:         $x \leftarrow x||C.x$ ▷ Extract all transactions from each block in the chain
24:     **end for**
25:     **return** x
26: **end function**

10.6   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 16 The Proof-of-Work discovery algorithm

1: **function** POW$_{H,T,q}(x, s)$
2:     $ctr \xleftarrow{\$} \{0, 1\}^\kappa$ ▷ Randomly sample Nonce
3:     **for** $i \leftarrow 1$ to q **do**   ▷ Number of available queries per party
4:         $B \leftarrow s||x||ctr$ ▷ Create block
5:         **if** $H(B) \leq T$ **then**   ▷ Successful Mining
6:             **return** B
7:         **end if**
8:         ctr $\leftarrow$ ctr $+1$
9:     **end for**
10:    **return** $\perp$ ▷ Unsuccessful Mining
11: **end function**

10.7 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 17 The maxvalid algorithm

1: **function** 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** 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_{max}
9: **end function**

10.8 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 18 The validate algorithm

1: **function** VALIDATE$_{\mathcal{G},\delta(\cdot)}(C)$
2:     **if** $C[0] \neq \mathcal{G}$ **then**   ▷ Check that first block is Genesis
3:         **return** false
4:     **end if**
5:     $st \leftarrow st_0$ ▷ Start at Genesis state
6:     $h \leftarrow H(C[0])$
7:     $st \leftarrow \delta^*(st, C[0].x)$
8:     **for** $B \in C[1:]$ **do**   ▷ Iterate for every block in the chain
9:         $(s, x, ctr) \leftarrow B$
10:         **if** $H(B) > T \vee s \neq h$ **then**   ▷ PoW check and Ancestry check
11:             **return** false
12:         **end if**
13:         $st \leftarrow \delta^*(st, B.x)$ ▷ Application Layer: update UTXO & validate transactions
14:         **if** $st = \bot$ **then**
15:             **return** false   ▷ Invalid state transition
16:         **end if**
17:         $h \leftarrow H(B)$
18:     **end for**
19:     **return** true
20: **end function**

10.9 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.

10.10 Pairing Lemma

Lemma 5. *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. $\square$

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.

10.11   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 $t < (1 - \delta)(n - t)$.

Chapter 11

Security in Earnest II DRAFT

11.1 Safety and Liveness

11.1.1 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.

11.1.2 Common Prefix Implies Safety

Theorem 6. *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. $\qquad\square$

11.1.3 Chain Quality and Chain Growth Imply Liveness

Theorem 7. *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 $CQ(\mu, \ell)$, 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 $u = \max(\frac{\ell+k}{\tau}, s)$ $\qquad\square$

11.2 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 jth query of the adversary $\mathcal{A}$ was successful at round r. $Z_{r,j} = 1$ if the jth 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.

11.2.1 Chain Growth Lemma

Lemma 8 (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.

$\square$

11.2.2 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.
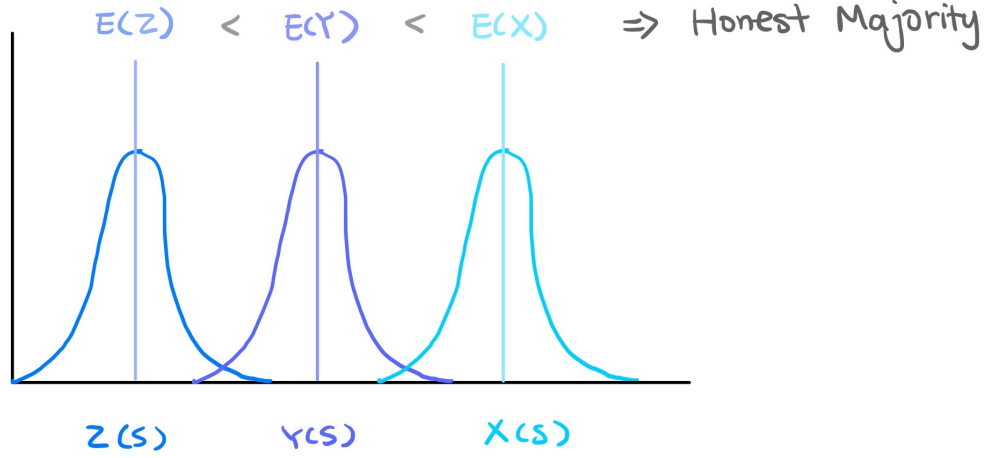
Figure 11.1: Illustration of the probability density functions for $X(S)$, $Y(S)$, and $Z(S)$. Note that $\mathbb{E}[Z(S)] < \mathbb{E}[Y(S)] < \mathbb{E}[X(S)]$

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 11.2). More formally, the Chernoff Bound states:

Theorem 9. *Consider random variables X_i, $i \in [n]$ s.t. $X_i \overset{i.i.d.}{\sim} \mathrm{Bernoulli}(p)$ and $X = \sum_{i=1}^{n} X_i$. For any $\epsilon > 0$,*

$$\Pr[X \leq (1 - \epsilon)\mu] \leq e^{-\Omega(n\epsilon^2\mu)}, \tag{11.1}$$

$$\Pr[X \geq (1 + \epsilon)\mu] \leq e^{-\Omega(n\epsilon^2\mu)}. \tag{11.2}$$

For more details about the Chernoff bound and its proof, a good reference is [**?**]. (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 12 ((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 10. (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 $\mathbb{E}[X_r]$?

$$\begin{aligned} \mathbb{E}[X_r] = f &= \Pr[\text{at least one successful honest query in round } r] \\ &= 1 - \Pr[\text{no honest query succeeds in round } r] \\ &= 1 - (1-p)^{q(n-t)} \\ &< pq(n-t). \end{aligned} \tag{11.3}$$

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

$$\frac{f}{1-f} = \frac{1 - (1-p)^{q(n-t)}}{(1-p)^{q(n-t)}} = (1-p)^{-q(n-t)} - 1 > (1+p)^{q(n-t)} - 1 > pq(n-t). \tag{11.4}$$

Therefore, we can sandwich the value of $\mathbb{E}[X_r]$ by its lower and upper bounds.

$$(1-f)pq(n-t) < \mathbb{E}[X_r] < pq(n-t). \tag{11.5}$$

For $\mathbb{E}[Y_r]$, we have

$$\mathbb{E}[Y_r] \geq q(n-t)p(1-p)^{q(n-t)-1} > pq(n-t)[1 - pq(n-t)] \geq f(1-f). \tag{11.6}$$

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

$$\mathbb{E}[Z_r] = pqt. \tag{11.7}$$

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,

$$\mathbb{E}[Z_r] = pqt = \frac{t}{n-t} \cdot pq(n-t) < \frac{t}{n-t} \cdot \frac{f}{1-f} < \left(1 + \frac{\delta}{2}\right) \cdot f \cdot \frac{t}{n-t}. \quad (11.8)$$

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:

$$1 < \left(1 - \frac{\delta}{3}\right)\left(1 + \frac{\delta}{2}\right)$$
$$\iff 1 < 1 + \frac{\delta}{2} - \frac{\delta}{3} - \frac{\delta^2}{6}$$
$$\iff 0 < \frac{\delta}{6} - \frac{\delta^2}{6}$$
$$\iff 0 < \frac{\delta}{6}(1 - \delta)$$

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 11.2.

- 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 class, 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.
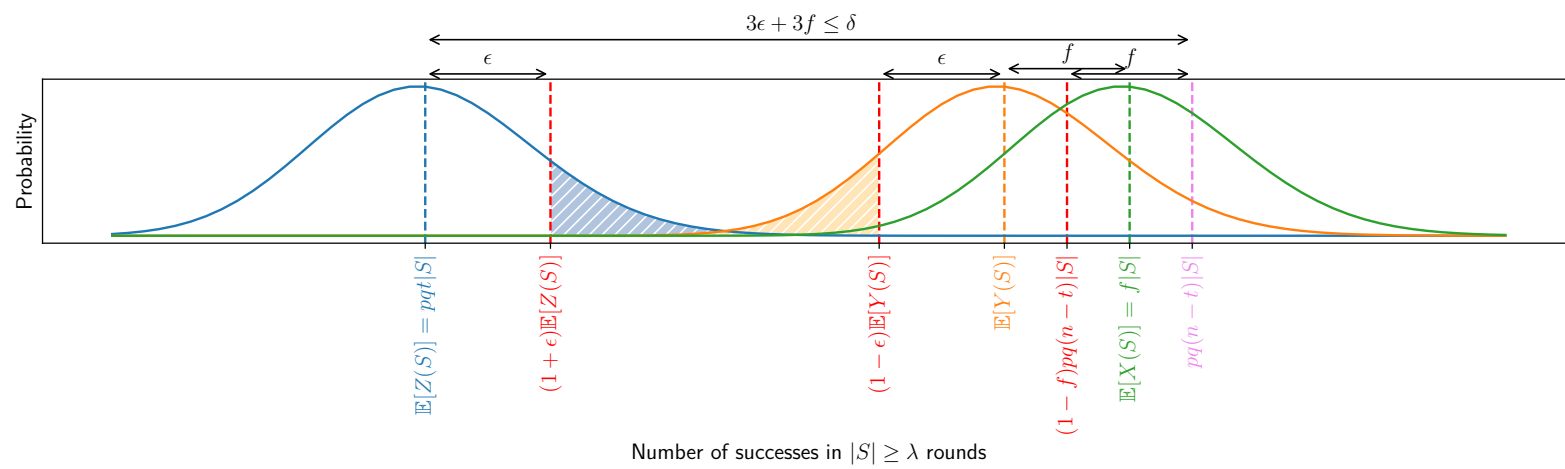
Figure 11.2: The distribution of the random variables X, Y, and Z in the proof-of-work longest chain protocol.

Chapter 12

Security in Earnest III DRAFT

12.1   Recap of last lecture: Security in Earnest (II)

We proved several bounds in the previous lecture:

$$(1-f)pq(n-t) < f = \mathbb{E}[X_r] = 1 - (1-p)^{q(n-t)} < pq(n-t) \quad (12.1)$$

$$\mathbb{E}[Y_r] \geq q(n-t)p(1-p)^{q(n-t-1)} > pq(n-t)(1-pq(n-t)) \geq f(1-f) > \left(1 - \frac{\delta}{3}\right)f \tag{12.2}$$

$$(1-\epsilon)\mathbb{E}[X(S)] < X(S) < (1+\epsilon)\mathbb{E}[X(S)] \tag{12.3}$$

$$(1-\epsilon)\mathbb{E}[Y(S)] < Y(S) \tag{12.4}$$

$$Z(S) < (1+\epsilon)\mathbb{E}[Z(S)] \tag{12.5}$$

First, X_r indicates whether or not round r was successful, i.e. had at least one successful honest query. Bound 12.1 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 12.2 gives us a lower bound on $\mathbb{E}[Y_r]$. Bounds 12.3, 12.4, and 12.5 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.

12.2   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.

12.2.1   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 12.1 we derived from last class and the fact that $\frac{1}{1-f} < 1+\delta/2$ with $f = \delta/6$

we have that

$$\mathbb{E}[Z_r] = \frac{t}{n-t}pq(n-t) \tag{12.6}$$

$$< \frac{t}{n-t} \cdot \frac{f}{1-f} \tag{12.7}$$

$$< \left(1 + \frac{\delta}{2}\right) f \frac{t}{n-t}. \tag{12.8}$$

12.2.2 Combining it to get $Z(S) < Y(S)$

We can use bound 12.2, which gives a lower bound on $\mathbb{E}[Y_r]$. We have that:

$$Y(S) = (1-\epsilon)\,\mathbb{E}[Y(S)] \tag{12.9}$$

$$= (1-\epsilon)\,\mathbb{E}[Y_r]|S| \tag{12.10}$$

$$> (1-\epsilon)f(1-f)|S| \tag{12.11}$$

$$> \left(1 - \frac{\delta}{3}\right)f|S| \tag{12.12}$$

The last inequality is derived as follows by setting $f = \epsilon = \frac{\delta}{6}$:

$$(1-\epsilon)(1-f) > 1 - \frac{\delta}{3} \tag{12.13}$$

$$\Leftarrow \left(1 - \frac{\delta}{6}\right)\left(1 - \frac{\delta}{6}\right) > 1 - \frac{\delta}{3} \tag{12.14}$$

$$\Leftarrow \frac{2\delta}{6} + \frac{\delta^2}{36} > -\frac{\delta}{3} \tag{12.15}$$

$$\Leftarrow \frac{\delta^2}{36} > 0 \tag{12.16}$$

$$\Leftarrow \delta > 0. \tag{12.17}$$

From the Chernoff bound 12.5 and upper bound 12.8, we also have that:

$$Z(S) < (1+\epsilon)\,\mathbb{E}[Z(S)] \tag{12.18}$$

$$= (1+\epsilon)\,\mathbb{E}[Z_r]|S| \tag{12.19}$$

$$< (1+\epsilon)\frac{t}{n-t} \cdot \frac{f}{1-f}|S| \tag{12.20}$$

$$< \frac{t}{n-t} \cdot \frac{f}{1-f}|S| + \epsilon\frac{t}{n-t} \cdot \frac{1}{1-f}f|S| \tag{12.21}$$

$$< \frac{t}{n-t} \cdot \frac{f}{1-f}|S| + \epsilon f|S| \tag{12.22}$$

$$\leq \left(1 - \frac{2\delta}{3}\right)f|S|. \tag{12.23}$$

To prove inequality 12.22, note that from the balancing equation we have $f \leq \frac{\delta}{3}$.

It suffices to show that $\frac{t}{n-t} \cdot \frac{1}{1-f} < 1$:

$$\frac{t}{n-t} \cdot \frac{1}{1-f} < 1 \tag{12.24}$$

$$\Leftarrow \frac{1-\delta}{1-f} < 1 \tag{12.25}$$

$$\Leftarrow 1 - \delta < 1 - \frac{\delta}{3}. \tag{12.26}$$

To prove inequality 12.23, we again use our choice of values $f = \epsilon = \frac{\delta}{6}$. Then,

$$\frac{t}{n-t} \cdot \frac{1}{1-f} + \epsilon < 1 - \frac{2\delta}{3} \tag{12.27}$$

$$\Leftarrow \frac{1-\delta}{1-f} + \epsilon < 1 - \frac{2\delta}{3} \tag{12.28}$$

$$\Leftarrow \frac{1-\delta}{1-\delta/6} + \frac{\delta}{6} < 1 - \frac{2\delta}{3} \tag{12.29}$$

$$\Leftarrow 1 - \delta + \frac{\delta}{6} - \frac{\delta^2}{36} < 1 - \frac{\delta}{6} - \frac{2\delta}{3} + \frac{2\delta^2}{18} \tag{12.30}$$

$$\Leftarrow -\frac{5\delta}{6} - \frac{\delta^2}{36} < \frac{5\delta}{6} + \frac{\delta^2}{9} \tag{12.31}$$

$$\Leftarrow -\frac{\delta^2}{36} < \frac{\delta^2}{9} \tag{12.32}$$

Combining results 12.23 and 12.12 together gives us:

$$Z(S) < \left(1 - \frac{2\delta}{3}\right) f|S| < \left(1 - \frac{\delta}{3}\right) f|S| < Y(S). \tag{12.33}$$

12.3 Proof of Chain Growth

Recall chain growth lemma from the previous lecture.

Lemma 11 (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 12 (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 $\tau = (1-\epsilon)f$. $\qquad \square$

12.4 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 13 (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

$$X(S) + Z(S) < (1 + \epsilon)\, \mathbb{E}[X(S)] + \left(1 - \frac{2\delta}{3}\right) f|S| \tag{12.34}$$

$$= (1 + \epsilon)f|S| + \left(1 - \frac{2\delta}{3}\right) f|S| \tag{12.35}$$

$$= \left(2 + \epsilon - \frac{2\delta}{3}\right) f|S| \tag{12.36}$$

$$\leq (2 - 2f)f|S| \tag{12.37}$$

$$\leq (2 - 2f)f \cdot \left(\frac{k}{2f} + 2\right) \tag{12.38}$$

$$= (1 - f)(k + 4f) \tag{12.39}$$

$$< k \tag{12.40}$$

The last inequality holds for $k \geq 4$ (this follows from $\lambda \geq 2/f$). To prove inequal-
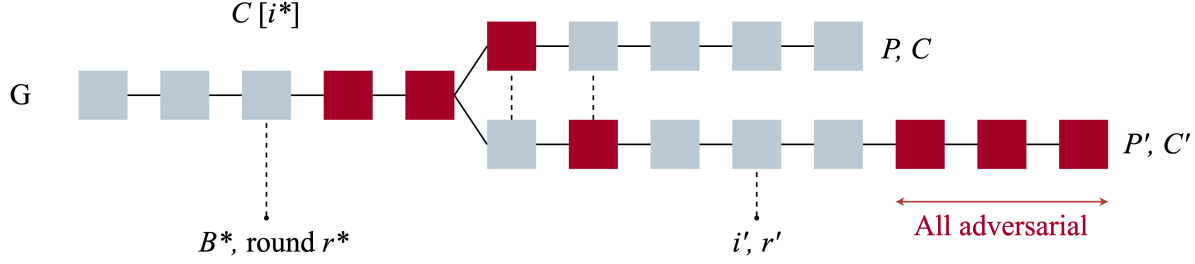
Figure 12.1: A common prefix violation. Honestly computed blocks are shown in gray, while adversarially computed blocks are shown in red. At the end of round r, party P has adopted chain C and party P' has adopted chain C'. $B^\star$ is the most recent honestly produced block in the common prefix of 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 12.37, we use the balancing equation $3f + 3\epsilon < \delta$ as follows:

$$2 + \epsilon - \frac{2\delta}{3} \leq 2 - 2f \tag{12.41}$$

$$\Leftarrow 3\epsilon - 2\delta \leq -6f \tag{12.42}$$

$$\Leftarrow 3f + \frac{3}{2}\epsilon \leq \delta \tag{12.43}$$

$$\Leftarrow 3f + \frac{3}{2}\epsilon \leq 3f + 3\epsilon \tag{12.44}$$

Since we wanted at least k blocks, inequality 12.40 is a contradiction. $\qquad\square$

Recall also the Pairing Lemma from the last lecture.

Lemma 14 (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 15 (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 12.1. 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. $\qquad\square$

12.5   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] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[2] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools.* Cambridge university press, 2007.

[3] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge university press, 2009.

[4] David Graeber. *Debt: The First 5,000 Years.* Melville House, 2014.

[5] Geoffrey Ingham. Money is a social relation. In Steve Fleetwood, editor, *Critical realism in economics: Development and debate*, pages 104–105. Routledge, 1998.

[6] William Stanley Jevons. *Money and the Mechanism of Exchange.* H.S. King & Co., 1875.

[7] Yehuda Lindell and Jonathan Katz. *Introduction to Modern Cryptography.* Chapman and Hall/CRC, 2014.

[8] C. L. Liu. *Elements of Discrete Mathematics.* McGraw-Hill, 2 edition, 1985.

[9] Sheldon M Ross. *A First Course in Probability.* Pearson Boston, MA, 10 edition, 2019.

[10] Georg Simmel. *The Philosophy of Money.* 1900.

[11] Michael Sipser et al. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[12] Nigel Smart. *Cryptography Made Simple.* Springer, 2016.

[13] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

Index