

# Blockchain Foundations

DRAFT OF JANUARY 27, 2023

*Dionysis Zindros*

Athens and Stanford

June 2021 - January 2023

# Contents

<b>1</b>	<b>A Big, Bad World</b>	<b>7</b>
1.1	The Nature of Money . . . . .	7
1.2	The Adversary . . . . .	10
1.3	Game-Based Security . . . . .	11
1.4	The Network . . . . .	18
<b>2</b>	<b>Cryptographic Primitives</b>	<b>23</b>
2.1	Hash Functions . . . . .	23
2.2	Signatures . . . . .	30
<b>3</b>	<b>The Transaction</b>	<b>36</b>
3.1	Coins . . . . .	36
3.2	Multiple Outputs . . . . .	37
3.3	Multiple Inputs . . . . .	38
3.4	The Conservation Law . . . . .	39
3.5	Outpoints . . . . .	39
3.6	The UTXO Set . . . . .	40
3.7	Transaction Signatures . . . . .	41
3.8	Transaction Creation . . . . .	43
3.9	Transaction Format . . . . .	43
3.10	Transaction Validation . . . . .	46
<b>4</b>	<b>Blocks</b>	<b>48</b>
4.1	The Network Delay . . . . .	48
4.2	The Double Spend . . . . .	49
4.3	Simple Ideas Don't Work . . . . .	50
4.4	Ledgers . . . . .	52
4.5	Rare Events . . . . .	54
4.6	Proof-of-Work . . . . .	54
4.7	The Block . . . . .	56
4.8	The Mempool . . . . .	57
4.9	Chain of Blocks . . . . .	58
4.10	Genesis . . . . .	59
4.11	Mining . . . . .	60

<b>5</b>	<b>The Chain</b>	<b>63</b>
5.1	The Target . . . . .	63
5.2	The Arrow of Time . . . . .	64
5.3	The Stochastic Nature of Work . . . . .	67
5.4	The Honest Majority Assumption . . . . .	69
5.5	Coinbase Transactions . . . . .	69
5.6	Block Validation . . . . .	72
5.7	Maintaining the Mempool . . . . .	73
<b>6</b>	<b>Chain Attacks DRAFT</b>	<b>75</b>
6.1	Comparing Transactions and Blocks . . . . .	75
6.2	Review of Ledger Virtues . . . . .	75
6.3	Chain Virtues . . . . .	75
6.3.1	Virtues . . . . .	75
6.3.2	Mechanics of Chain Divergence and Convergence . . . . .	76
6.4	Network Attacks . . . . .	76
6.4.1	Nakamoto Attack . . . . .	77
6.4.2	Fan-out Attack . . . . .	78
6.4.3	Majority Adversary Attack . . . . .	78
<b>7</b>	<b>The Selfish Miner DRAFT</b>	<b>80</b>
7.1	Recap of Chain Virtues . . . . .	80
7.1.1	Common Prefix . . . . .	80
7.1.2	Chain Quality . . . . .	80
7.1.3	Chain Growth . . . . .	81
7.2	Censorship Attack . . . . .	81
7.3	Attacks Under Dishonest Majority . . . . .	81
7.4	Healing From Attacks . . . . .	82
7.5	Selfish Mining . . . . .	82
7.6	What Value of $T$ to Choose? . . . . .	84
<b>8</b>	<b>Economics DRAFT</b>	<b>87</b>
8.1	Our Variables . . . . .	87
8.2	Some Bitcoin Statistics . . . . .	87
8.3	Mining . . . . .	88
8.3.1	Parties . . . . .	88
8.3.2	The parameter $\Delta$ and Block Size Limit . . . . .	89
8.3.3	Including a transaction . . . . .	89
8.3.4	Block Reward . . . . .	90
8.4	Variable Mining Difficulty . . . . .	90
8.4.1	Definitions . . . . .	91
8.5	Mining Pools . . . . .	91
8.5.1	How Pools Work . . . . .	92
8.6	Wallets . . . . .	92
8.6.1	Mining and Wallets . . . . .	92
8.6.2	HD Wallets . . . . .	93

<b>9</b>	<b>Accounts DRAFT</b>	<b>95</b>
9.1	Accounts Model . . . . .	95
9.1.1	Accounts Model Compared with UTXO Model . . . . .	95
9.1.2	Accounts Model Replay Attack . . . . .	96
9.2	State Machine Replication . . . . .	97
9.3	Light Clients . . . . .	98
9.3.1	Storage Efficiency: Merkle Trees . . . . .	98
9.3.2	Data Structure: Merkle Tree . . . . .	98
9.3.3	Security of Merkle Trees . . . . .	99
<b>10</b>	<b>Light Clients DRAFT</b>	<b>102</b>
10.1	Motivation . . . . .	102
10.2	Definition . . . . .	102
10.3	Header Chains . . . . .	102
10.3.1	Block Validation . . . . .	103
10.3.2	Benefits . . . . .	103
10.4	Making Payments . . . . .	104
10.5	Block Header Validation . . . . .	104
10.5.1	Transaction Validation . . . . .	104
10.5.2	Local Chain Security . . . . .	104
10.5.3	Privacy . . . . .	105
10.5.4	Full Node Ramifications . . . . .	105
10.6	Light Miners and the Quick Bootstrap Protocol . . . . .	105
10.6.1	Mining as a Light Client . . . . .	105
10.6.2	Quick Bootstrap Protocol . . . . .	106
10.6.3	Light Miner Security . . . . .	106
<b>11</b>	<b>Security in Earnest I DRAFT</b>	<b>108</b>
11.1	Random Oracle . . . . .	108
11.2	Synchrony . . . . .	109
11.3	The Simulation Environment . . . . .	109
11.3.1	A Simplification: Quantize Time . . . . .	110
11.3.2	Rushing Adversary . . . . .	110
11.3.3	Sybil Adversary & Non-Eclipsing Assumption . . . . .	111
11.4	Random Oracle Model . . . . .	111
11.5	Honest Party Algorithm . . . . .	112
11.6	Proof-of-Work . . . . .	113
11.7	Longest Chain . . . . .	114
11.8	Validating a block . . . . .	114
11.9	Chain Virtues . . . . .	115
11.10	Pairing Lemma . . . . .	115
11.11	Honest Majority Assumption $(n, t, \delta)$ . . . . .	116
11.12	Further Reading . . . . .	116
<b>12</b>	<b>Security in Earnest II DRAFT</b>	<b>117</b>
12.1	Safety and Liveness . . . . .	117
12.1.1	Defining Safety and Liveness . . . . .	117
12.1.2	Common Prefix Implies Safety . . . . .	117
12.1.3	Chain Quality and Chain Growth Imply Liveness . . . . .	118
12.2	Proving Chain Growth, Chain Quality, and Common Prefix . . . . .	118

12.2.1 Chain Growth Lemma . . . . .	119
12.2.2 Proving Common Prefix and Chain Growth . . . . .	119
<b>13 Security in Earnest III DRAFT</b>	<b>126</b>
13.1 Recap of last lecture: Security in Earnest (II) . . . . .	126
13.2 Typicality implies $Z(S) < Y(S)$ . . . . .	126
13.2.1 Derivation of Upper Bound of $\mathbb{E}[Z(S)]$ . . . . .	126
13.2.2 Combining it to get $Z(S) < Y(S)$ . . . . .	127
13.3 Proof of Chain Growth . . . . .	128
13.4 Proof of Common Prefix . . . . .	128
13.5 Note about Tradeoffs with $\epsilon$ and $f$ . . . . .	131
<b>Bibliography</b>	<b>132</b>
<b>Index</b>	<b>134</b>
<b>List of Symbols</b>	<b>136</b>

# Preface

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

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

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

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

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

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

---

<sup>1</sup>The misspelling is intentional, as it makes it easier to Google.

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

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

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

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

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

# Chapter 1

## A Big, Bad World

### 1.1 The Nature of Money

Before money, there was debt [10]. 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 [11].

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* [12]. 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 [18].



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-enforceable*. Prevention is preferable to treatment.

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

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

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

## 1.2 The Adversary

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

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

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

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

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

---

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

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

## The Cryptographic Model

Following the cryptographic tradition, and highlighting our computer science methodology, our protocols are structured upon three pillars [13]:

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 and 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  $\Pi$ . 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 [21] with additional access to a random tape. In practice, when thinking about these machines, we simply think of them as regular computer programs (in, say, Python, C++ or TypeScript) in which we have access to a random number generator, which we assume produces fresh, uniform and completely fair randomness every time it is called.

### 1.3 Game-Based Security

We will soon give detailed and rigorous definitions of what security properties we want our protocols to achieve. These will be our design goals. Once we have clearly

articulated these goals, we will attempt to formally prove that our protocols attain the desired properties.

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

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

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

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

In its foundations, our security is analyzed with a *security parameter*: The parameter  $\kappa$ . 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 catastrophe 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  $\kappa$  that we are interested in. The adversarial source code must be the same for all values of  $\kappa$  (we say that we are interested in *uniform* adversaries).

The adversary and honest party run in polynomial time in the security parameter  $\kappa$ . 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  $\kappa$ , 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  $\kappa$ , we must issue the call to the adversary with an input of size  $\kappa$ .

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  $\kappa$  times. Because  $|1^\kappa| = \kappa$ , the length of the input is  $\kappa$ , 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  $\kappa$ , this is sufficient for our purposes, and we do not need to pass the adversary the extra argument  $1^\kappa$ . You can think of the argument  $1^\kappa$  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  $\Pi$  and the code of the adversary  $\mathcal{A}$ . It is invoked with the security parameter  $\kappa$  (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  $\kappa$  to run, as well as some additional arguments that will be defined by the particular protocol. It then runs the adversary, giving her polynomial time in  $\kappa$  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 diagrammatically in Figure 1.1.

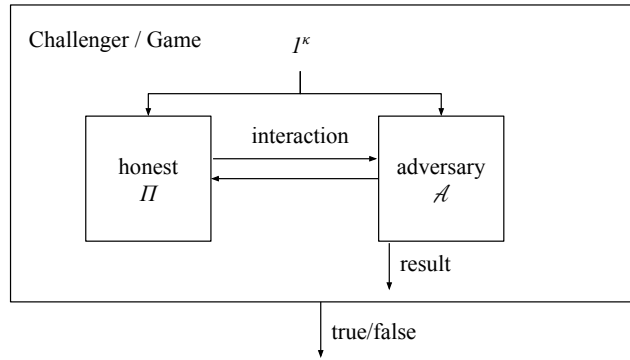


Figure 1.1: A game-based security definition shown diagrammatically. The challenger invokes both the honest party and the adversary before deciding whether the adversary was successful.

---

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

---

```

1: function MY-GAMEΠ, A(κ)
2:   ▷ Invoke the honest party with poly κ time and more arguments
3:   ... Π(1κ ...)
4:   ▷ Invoke the adversary with poly κ time and more arguments
5:   result ← A(1κ ...)
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

```

---

## Negligibility

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

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

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

What happens if an adversary attempts to perform multiple guesses for the secret key? Each of these guesses has a probability of success amounting to  $\frac{1}{2^\kappa}$ . Since the adversary has polynomial time  $p(\kappa)$ , the number of guesses she can perform must be polynomial, too. What is the probability that *at least one of these guesses* is correct? We can apply a *union bound* [17] to find this.

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

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

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

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

If we have one adversary who can succeed with some probability  $Pr_A$ , then a different adversary  $A'$  can succeed with probability bounded by  $p(\kappa)Pr_A$  for

any polynomial  $p$ . This is known as *amplification*. We wish to define a class of probability functions that we deem *acceptable* probabilities of failure. Clearly, if an adversary has a *constant* probability of success (such as 0.5) that does not depend on the security parameter  $\kappa$ , this is *not* acceptable, as we want  $\kappa$  to be our *tuning knob* of how secure our protocol will be. We will deem *acceptable* the class of functions denoting a probability which is not amplifiable to a constant by this manner.

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

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

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

We choose to accept negligible functions exactly because the probability of failure cannot be amplified in this manner. If an adversary  $\mathcal{A}$  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 [13]. Beyond the above argument pertaining to the polynomiality of adversaries, negligible functions are easy to work with because they observe certain *closure* properties. In particular, multiplying a negligible function with a polynomial yields a negligible function. As constants are polynomials, scaling a negligible function by a constant yields a negligible function too. Of course, multiplying something negligible with something negligible keeps it negligible, and taking any constant power of a negligible function keeps it negligible.

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

## Definitions of Security

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



computer science. This proof style is recent and has only appeared within the last 50 years.

The ideal protocol  $\Pi$  satisfies security against all adversaries:

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

However, this is an unattainable goal. To see this, consider the case where the honest party generates a secret of length  $\kappa$  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  $\kappa$ .

A security definition will look like this:

**Definition 2** (Security). *A protocol  $\Pi$  is secure with respect to game  $\text{Game}$  if there exists a negligible function  $\text{negl}(\kappa)$  such that*

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

We are using probability notation here because the execution of the challenger with the same honest protocol  $\Pi$  and against the same adversary  $\mathcal{A}$  and using a fixed security parameter  $\kappa$  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  $\Pi$  and  $\mathcal{A}$  and  $\kappa$ , there is a certain probability that the challenger will report 0, and a certain probability that the challenger will report 1.

Fixing  $\Pi$  and  $\mathcal{A}$ , but leaving  $\kappa$  to take any value, we obtain different probabilities for each value of  $\kappa$ . Therefore, the value denoted by  $\Pr[\text{Game}_{\Pi, \mathcal{A}}(\kappa) = 0]$  for a fixed  $\Pi$  and  $\mathcal{A}$  is a function of  $\kappa$ . 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  $\kappa$ ) become smaller than all inverse polynomials.

## The Honest/Adversarial Gap

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

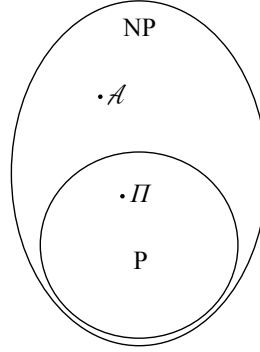


Figure 1.2: In a secure protocol, a successful honest party needs polynomial time, while a successful adversary needs superpolynomial time.

## Proofs of Security

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

The proofs of these theorems will take the form of a *computational reduction*, and they will look roughly as follows, when we are designing a new protocol  $\Pi^*$ :

**Claim.** If protocol  $\Pi^*$  is secure, then protocol  $\Pi$ , built on top of  $\Pi^*$ , is also secure.

**Proof.** Suppose, towards a contradiction, that protocol  $\Pi$  is *insecure*. Then, by the game-based security definition, there must exist a PPT adversary  $\mathcal{A}$  that breaks  $\Pi$  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  $\Pi^*$  within the confines of the challenger for the protocol  $\Pi^*$  (a particular game). The adversary  $\mathcal{A}$  attempts to break the protocol  $\Pi$  within the confines of the challenger for the protocol  $\Pi$  (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  $\Pi$ . 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  $\Pi^*$  was secure, completing the proof.  $\diamond$

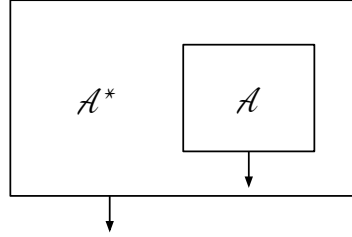


Figure 1.3: A computational reduction between two adversaries. Given an adversary  $\mathcal{A}$  against protocol  $\Pi$ , we construct an adversary  $\mathcal{A}^*$  against a protocol  $\Pi^*$ .

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 [13]. These proofs look like this:

**Proof.** Consider an arbitrary PPT adversary  $\mathcal{A}$  attempting to break the protocol  $\Pi$ . We construct an adversary  $\mathcal{A}^*$  against the protocol  $\Pi^*$  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  $\Pi^*$ , we know that the probability of success of  $\mathcal{A}^*$  against its challenger is negligible. From the relationship between the probabilities of success of  $\mathcal{A}$  and  $\mathcal{A}^*$ , we also deduce that the probability of success of  $\mathcal{A}$  is negligible, completing the proof.  $\diamond$

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

## 1.4 The Network

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

### The Non-Eclipsing Assumption

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

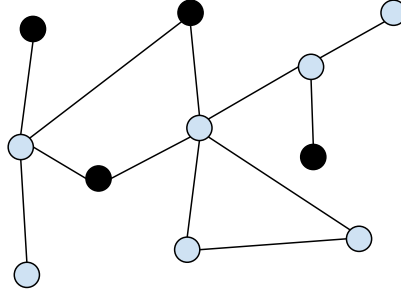


Figure 1.4: The peer-to-peer network. Nodes are shown as circles and connections as lines. The honest nodes are shown in blue, while the adversarial nodes are shown in black.

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

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

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

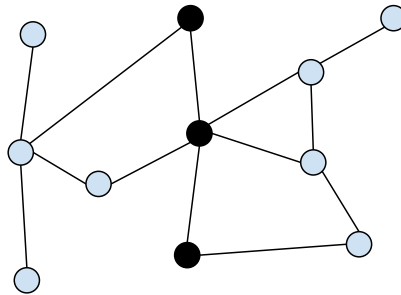


Figure 1.5: An eclipsed peer-to-peer network. Even though every honest party has an honest connection, the network is partitioned into two disconnected components by the adversary.

We are introducing this assumption out of necessity. We cannot hope to build any currency in an eclipsed world. To see why, imagine two completely isolated civilizations, both maintaining their own separate currency. These civilizations,

given a lack of communication between them, cannot hope to be able to deduce who owns how much money in their respective counterpart world.

## The Sybil Attack

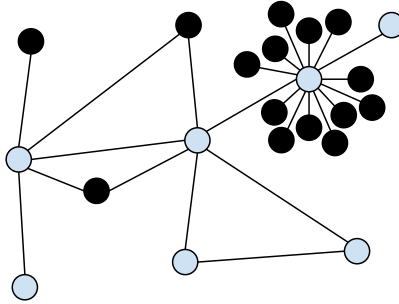


Figure 1.6: A Sybil attacked peer-to-peer network. The non-eclipsing assumption is not violated.

Following our pattern of a powerful adversary, we give the adversary the ability to create as many identities on the network as she desires. This is termed a *Sybil attack* [5]. The adversary may overwhelm an honest party with adversarial connections as illustrated in Figure 1.6.

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

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

## Peer Discovery

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

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

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

## Problems

1.1 Which of the following functions are negligible in  $\kappa$ ?

- a.  $f(\kappa) = 0$
- b.  $f(\kappa) = 1$
- c.  $f(\kappa) = 2^{-128}$
- d.  $f(\kappa) = \frac{2^\kappa}{128}$
- e.  $f(\kappa) = \frac{128}{2^\kappa}$
- f.  $f(\kappa) = \frac{1}{3\kappa^3 + 7\kappa^2 + 12}$
- g.  $f(\kappa) = \frac{\kappa^7}{7^\kappa}$
- h.  $f(\kappa) = \frac{1}{\log \kappa}$
- i.  $f(\kappa) = \frac{1}{\kappa!}$

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

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

1.4 Prove that

- a.  $\text{negl} \cdot \text{negl} = \text{negl}$
- b.  $\text{const} \cdot \text{negl} = \text{negl}$
- c.  $\text{poly} \cdot \text{negl} = \text{negl}$
- d.  $\forall m \in \mathbb{N} : \text{negl}^m = \text{negl}$

## Further Reading

Blockchain science is founded on cryptography. For a great introduction to modern cryptography, consult *Introduction to Modern Cryptography* by Katz and Lindell [13]. 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* [8, 9]. An easier and pleasant to read textbook is Smart's *Cryptography Made Simple* [20].

For a more in-depth treatment of Turing Machines and our computational model, consult Sipser's *Introduction to the Theory of Computation* [19]. 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* [14]. For probability theory, you can use Ross's *A First Course on Probability Theory* [17]. You can read both cover-to-cover, but they also function well as a reference in case you want to look something up.

## Chapter 2

# Cryptographic Primitives

### 2.1 Hash Functions

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

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

#### Collision Resistance

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

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

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



---

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

---

```

1: function COLLISION-SEARCHH( $\kappa$ )
2:   for  $i \leftarrow 0$  to  $2^\kappa$  do
3:     for  $j \leftarrow i + 1$  to  $2^\kappa$  do
4:       if  $H(i) = H(j)$  then
5:         return ( $i, j$ )
6:       end if
7:     end for
8:   end for
9: end function

```

---

Of course, as this function has to run through  $2^{2\kappa}$  combinations, its running time is exponential. The result that hash functions must *necessarily* have collisions stems from the Pigeonhole Principle [14]:

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

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

---

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

---

```

1: function collision-gameH, A( $\kappa$ )
2:    $x_1, x_2 \leftarrow \mathcal{A}(1^\kappa)$ 
3:   return  $H_\kappa(x_1) = H_\kappa(x_2) \wedge x_1 \neq x_2$ 
4: end function

```

---

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

## Gossiping with Hashes

Collision resistance ensures that, if we are given a hash of something, we cannot later be given something else that hashes to the same value. Each object's hash is uniquely identified by its content. We say that objects are *content addressable* by their hashes. This makes hashes suitable for use as identifiers of objects as we exchange them on the network. When gossiping about objects, instead of sending a whole object to each of our peers, we can optimize the process by advertising the ownership of an object through its hash. The hash of an object used in this manner is called the *objectid*. If the peer already knows about this object, they can ignore our advertisement. If the peer has not seen the object before, they can request the

---

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

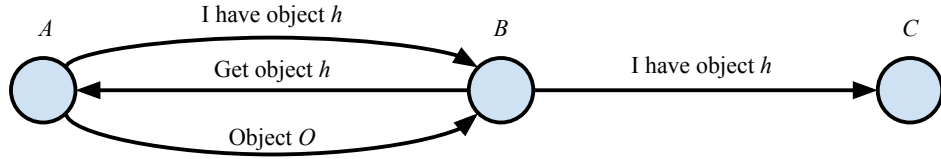


Figure 2.1: Gossiping via hashing. The hash function  $H$  is used to give content-addressable hash  $h$  to object  $O$ .

object through its objectid. Only at this point, we send the full object to the peer. Upon receiving the object, the peer can verify that it is indeed the requested object by hashing it and comparing it to the stored objectid. Towards this purpose, each node must maintain a set of known objectids for quick lookup.

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

1. Node  $A$  first becomes aware of a new object  $O$ , either by receiving it from a peer, or by generating it locally. Object  $O$  has objectid  $h = H(O)$ , where the input to the hash function is a string-encoded version of the object.
2.  $A$  advertises its knowledge of  $O$  by sending a message indicating *I have an object with objectid  $h$*  to its peer  $B$ .
3.  $B$  receives the objectid  $h$  and checks against its database whether it has already seen this object. Suppose that it has not. At this point,  $B$  sends to  $A$  a message requesting the contents of the object with objectid  $h$ .
4.  $A$  sends to  $B$  the object  $O$ . Upon receiving  $O$ , the node  $B$  can verify that  $h = H(O)$ , where  $h$  is the requested objectid. This ensures that  $A$  sent the correct object to  $B$ .
5. In turn,  $B$  advertises to *its* peers that it now knows of an object with objectid  $h$ . It sends node  $C$  a message indicating this.
6. At this point, if  $C$  has already received  $O$  from  $A$ , it will not request the object from  $B$ . This is how the propagation in the gossiping algorithm stops.

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

## Preimage resistance

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

---

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

---

```

1: function preimage-game $_{H,\mathcal{A}}(\kappa)$ 
2:    $x \xleftarrow{\$} \{0,1\}^\kappa$ 
3:    $y \leftarrow H_\kappa(x)$ 
4:    $x' \leftarrow \mathcal{A}(y)$ 
5:   return  $H_\kappa(x') = y$ 
6: end function

```

---



---

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

---

```

1: function 2PRE $_{\mathcal{A},H}(\kappa)$ :
2:    $x_1 \xleftarrow{\$} \{0,1\}^{2\kappa+1}$ 
3:    $x_2 \leftarrow \mathcal{A}(x_1)$ 
4:   return  $x_1 \neq x_2 \wedge H_\kappa(x_1) = H_\kappa(x_2)$ 
5: end function

```

---

performing an exhaustive search as illustrated in Algorithm 4, but this will take exponential time.

---

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

---

```

1: function PREIMAGE-SEARCH $_H(h)$ 
2:    $\text{ctr} \leftarrow 0$ 
3:   while true do
4:     if  $H(\text{ctr}) = h$  then
5:       return  $\text{ctr}$ 
6:     end if
7:      $\text{ctr} \leftarrow \text{ctr} + 1$ 
8:   end while
9: end function

```

---

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

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

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

---

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

---

```

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

```

---

is defined through the following game.

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

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

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

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

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

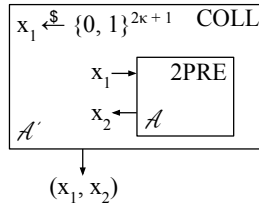


Figure 2.2: A visualization of Theorem 3.

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

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

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

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

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

This is captured in the following theorem:

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

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

---

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

---

```

1: function  $\mathcal{A}'(x_1)$ :
2:    $y \leftarrow H(x_1)$ 
3:    $x_2 \leftarrow \mathcal{A}(y)$ 
4:   return  $x_2$ 
5: end function

```

---

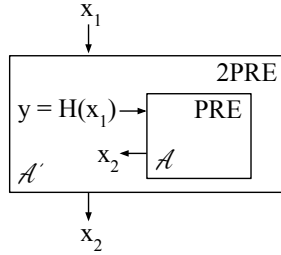


Figure 2.3: The reduction in the proof of Theorem 4.

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

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

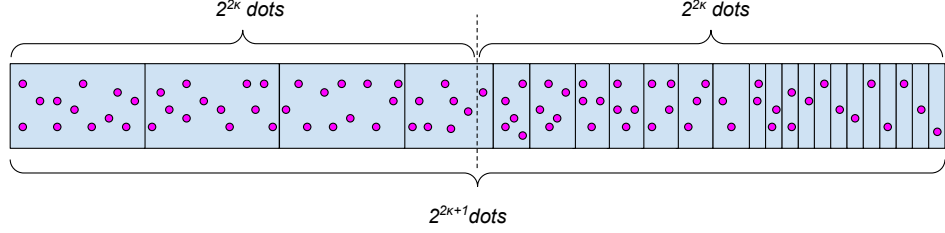


Figure 2.4: A visualization of the counting argument in Theorem 4.

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

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

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

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

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

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

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

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

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

## Hash Security

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

**Definition 5** (Secure Hash Function). *A hash function  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$  is secure if there is a negligible function  $\text{negl}$  such that*

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

## Applied Hashes

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

2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

## 2.2 Signatures

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

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

## Public Key Cryptography

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

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

## Unforgeability

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

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

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

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

**Definition 6** (Signature Correctness). *Consider a signature scheme  $(\text{Gen}, \text{Sig}, \text{Ver})$ . The scheme is correct if for any key pair  $(pk, sk)$  generated by invoking  $\text{Gen}$ , and for all messages  $m$ , it holds that  $\text{Ver}(pk, m, \text{Sig}(sk, m)) = \text{true}$ .*



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

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

---

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

---

```

1: function existential-forgery-game $_{Gen, Sig, Ver, \mathcal{A}}(\kappa)$ 
2:    $(pk, sk) \leftarrow Gen(1^\kappa)$ 
3:    $M \leftarrow \emptyset$ 
4:   function  $\mathcal{O}(m)$ 
5:      $M \leftarrow M \cup \{m\}$ 
6:     return  $Sig(sk, m)$ 
7:   end function
8:    $m, \sigma \leftarrow \mathcal{A}^\mathcal{O}(pk)$ 
9:   return  $Ver(pk, \sigma, m) \wedge m \notin M$ 
10: end function
```

---

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

the adversary is expected to produce a signature  $\sigma$  on a message  $m$  that was not queried to the oracle  $\mathcal{O}$  (the adversary can trivially succeed in providing a signature for messages queried to the oracle). If the message and signature provided by the adversary pass the *Ver* check using the public key  $pk$ , the adversary is deemed successful.

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

**Definition 7** (Secure Signature Schemes). *A signature scheme  $(\text{Gen}, \text{Sig}, \text{Ver})$  is called secure if there exists a negligible function  $\text{negl}$  such that*

$$\forall PPTA : \Pr[\text{existential-forgery}_{\text{Gen}, \text{Sig}, \text{Ver}, A}(\kappa) = 1] < \text{negl}(\kappa)$$

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

## Applied Signatures

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

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

10b4b0f158afb93e3fd6111b564ad4c4054ae9a142362d8d9e05a9f2d64444530

Here is the respective private key:

7aa064fb575c861d5af00febf08c1c31620d5a70094c4bcb11cb2720630ee98a

Here is a signature generated with the above private key:

c538752e628c9ca43b3328f68afc76af40cf68732db00a8c9a885a6d41045b49  
5ef44fb625a6742895d6819a63c254e352537998961a6802687140115811a409

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

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

## Problems

- 2.1 What is an example of a hash function that is not collision resistant, nor preimage resistant, nor 2nd preimage resistant?
- 2.2 The proof of Theorem 4 is a proof with reference to the illustration. Make it rigorous so that it doesn't speak of images, boxes, and dashed lines. Instead, use exact counting formulas and define appropriate notation to represent the boxes as equivalence classes.
- 2.3 In the proof of Theorem 4, the probabilities

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

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

- 2.4 Give a simpler proof of Theorem 4.
- 2.5 Combine Theorems 3 and 4 to directly show that collision resistance implies preimage resistance.
- 2.6 Given a collision resistant hash function  $H$ , construct a preimage and 2nd preimage resistant hash function  $G$  which is not collision resistant. Prove these properties of  $G$ .
- 2.7 Given a collision resistant hash function  $H$ , make a collision resistant hash function  $G$  whose first bit is reliably predictable and prove that it is collision resistant. Prove these properties of  $G$ .
- 2.8 Construct a correct but insecure signature scheme.
- 2.9 Let  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$  be a collision-resistant hash function and define  $G(x) = H(H(x))$ . Show that  $G$  is a collision-resistant hash function.
- 2.10 Given a collision resistant hash function  $H$ , construct a preimage and 2nd preimage resistant hash function  $G$  which is not collision resistant. Prove these properties.

## Further Reading

Any cryptography book contains more information about hash functions and signature schemes. Our treatment here was superficial (as we did not, for example, treat *keyed* hash functions). Read the security definitions in *Modern Cryptography* [13] pertaining to *collision resistance*, *pre-image resistance* and *second pre-image resistance*. Read the security definitions for *existential unforgeability*. In the same book, you can find constructions for signature schemes using various methods, including some details on elliptic curves. Other good books that review these topics and talk about *how* to build a hash function or a signature scheme are *Introduction to Modern Cryptography* [13], *A Graduate Course in Applied Cryptography* [4], or *Foundations of Cryptography* [8, 9].

In our examples, we considered a  $(2\kappa + 1)$ -bit message space as an illustrative example. For a complete and nuanced cryptographic treatment of arbitrary-sized message spaces, which is beyond the scope of this book, refer to the seminal paper by Rogaway and Shrimpton [16] that formalized and proved these notions.

## Chapter 3

# The Transaction

### 3.1 Coins

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

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

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

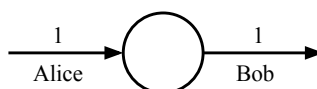


Figure 3.1: A transaction paying 1 unit of money from *Alice* to *Bob*.

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

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

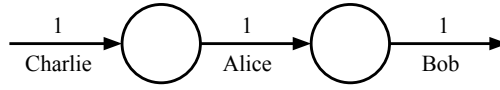


Figure 3.2: *Alice* pays 1 unit of money to *Bob*. She received this money from *Charlie*.

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

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



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

## 3.2 Multiple Outputs

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

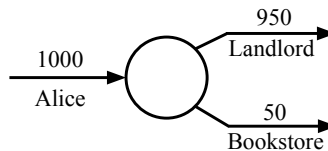


Figure 3.4: Alice uses her salary, the incoming edge, in a single transaction to pay for both her rent and a book in two different outgoing edges.

An outgoing edge can either be spent (if it is connected to another transaction)

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

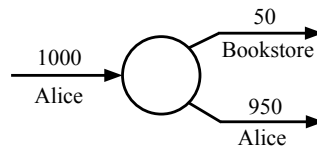


Figure 3.5: Alice uses her salary, the incoming edge, in a single transaction to pay for a book (top output edge). She uses the rest of her money to pay the *change* back to herself (bottom output edge).

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

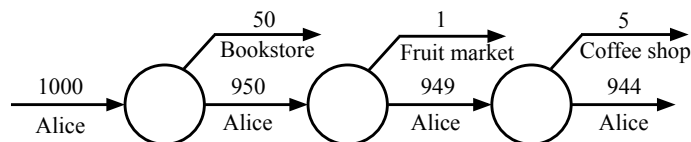


Figure 3.6: Alice uses her salary, the left-most edge, in a series of transactions, always giving change back to herself. The bottom-right edge is her remaining salary.

### 3.3 Multiple Inputs

It is also possible to *combine* multiple inputs into a single transaction to make a larger payment. For example, Alice can use two of her salaries, each of which resides in a different transaction output, to make a down payment for the house she

is buying. This is illustrated in Figure 3.7. This transaction consumes two outputs and produces one new output.

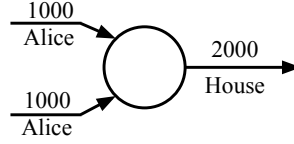


Figure 3.7: Alice combines two of her salaries (left, incoming edges) to pay for her house (right, outgoing edge).

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

### 3.4 The Conservation Law

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

**Definition 8** (Conservation Law). *Given a transaction  $tx$ , we say that it obeys the Conservation Law if*

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

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

### 3.5 Outpoints

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

Since an input of a transaction is always spending a previous output, the input can just be a reference to a previous output. To reference an output, we need



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

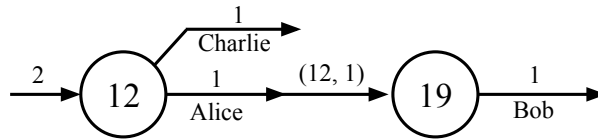


Figure 3.8: The transaction number 19 has a single input that spends the output number 1 (to Alice) of transaction number 12. The output number 0 of transaction number 12 (to Charlie) is still unspent. Here, we have highlighted the outpoint pair (12, 1) that connects the input of transaction 19 with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

### 3.6 The UTXO Set

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

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

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

after the transaction that it spends from. This sequence of transactions, ordered in this consistent manner, is known as a *transaction ledger*. In Figure 3.9, transactions are labelled in one possible consistent order. As new transactions are added to our graph, they can also be appended to the transaction ledger while maintaining consistency.

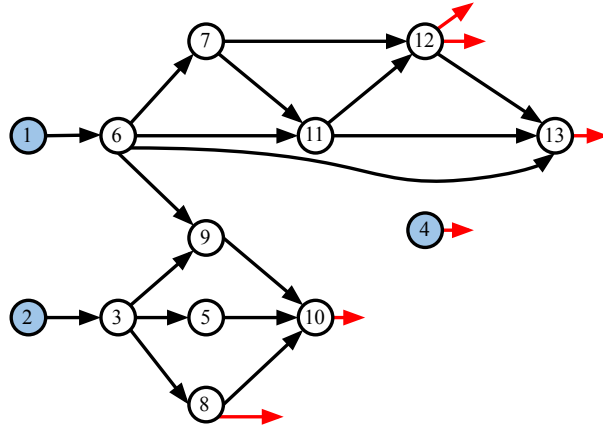


Figure 3.9: A transaction graph with 3 coinbase and 10 non-coinbase transactions. Coinbase transactions are shaded blue. There are 23 outputs, of which 6 belong to the UTXO set. The UTXO set is shaded red. Transaction number 8 contains both a spent and an unspent transaction output.

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

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

### 3.7 Transaction Signatures

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

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

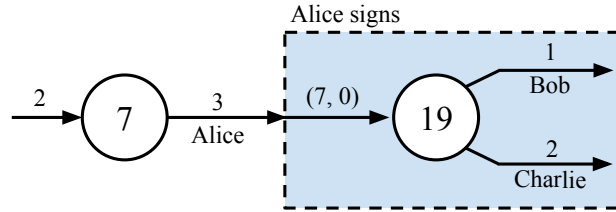


Figure 3.10: Alice creates a new transaction, transaction 19, by which she pays Bob 1 unit and Charlie 2 units. The transaction data include Bob's public key, Charlie's public key, the outgoing amounts 1 and 2, and the outputpoint  $(7, 0)$ . These must all be signed by Alice's secret key.

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

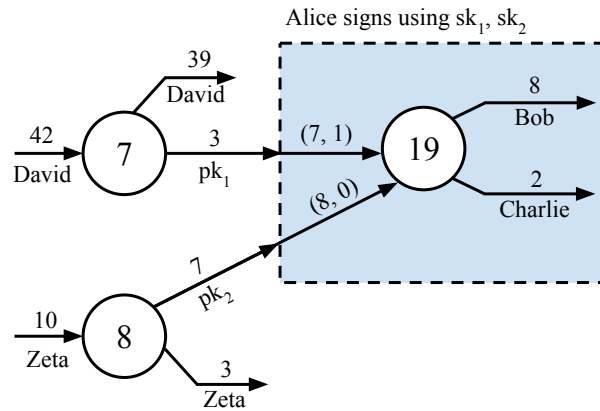


Figure 3.11: Alice creates a new transaction, transaction 19, by which she pays Bob 8 units and Charlie 2 units. The transaction data include Bob's public key, Charlie's public key, the outgoing amounts 1 and 2, and two outputpoints,  $(7, 1)$  and  $(8, 0)$ . All of these data must be signed twice: Once using Alice's  $sk_1$  secret key, and once using Alice's  $sk_2$  secret key, giving two different signatures  $\sigma_1$  and  $\sigma_2$  on the same data.

If Alice wishes to create a transaction with multiple inputs she controls, she must provide a signature corresponding to each of the inputs. Consider the example illustrated in Figure 3.11. Here, Alice wishes to spend two outputs that she owns. The first output has been paid to Alice's public key  $pk_1$ . The second output has

been paid to Alice's public key  $pk_2$ . Alice controls the respective secret keys  $sk_1$  and  $sk_2$ . Alice creates a new transaction, transaction 19, containing the desired inputs and outputs. In the inputs, she places two outpoints pointing to the two outputs she wishes to spend. The new transaction data, including the inputs and outputs, must all be signed twice: First, using Alice's  $sk_1$  (and verifiable using  $pk_1$ ), and secondly using Alice's  $sk_2$  (and verifiable using  $pk_2$ ). This will yield two different signatures  $\sigma_1$  (created using  $sk_1$ ) and  $\sigma_2$  (created using  $sk_2$ ). Both of these signatures verify on the *same* data, but using different public keys. A signature for the outpoint connected to each of the transaction's inputs must accompany the transaction whenever it is broadcast. A transaction is accompanied by as many signatures as it has inputs.

### 3.8 Transaction Creation

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

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

### 3.9 Transaction Format

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

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

An example transaction object looks like this:

```
{
  inputs: [
    {
      outpoint: (
```

```

        "cc6a88afaca94fec238258e3665d64cd
        de592e3ea13f151eca37d5e6589cd169",
        0
    )
},
{
    outpoint: (
        "3a648c42b90af46b9bba7ae723451002
        aa53baba187020051e0c32112bf458a0",
        3
    )
}
],
outputs: [
    {
        pk: "36dec8c46741efdab93a77f8fc75acec
        6e290b1ce0f280e04753db5c864a6469",
        amount: 5012900000000
    },
    {
        pk: "f71c1e7478459d90764d59dd9cb0ea9f
        62406d2c1412dea7c6de0c0355183066",
        amount: 3170000000000
    },
    {
        pk: "d9169b9601f3121f316f44e4d1bd0ecc
        d6d07df4d152fdcbf59a210e7c77e467"
        amount: 1350000000000
    }
]
}

```

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

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

```

{
    inputs: [
        {
            outpoint: (
                "cc6a88afaca94fec238258e3665d64cd
                de592e3ea13f151eca37d5e6589cd169",
                0
            ),
            sig:

```

```

        "680b733e57690024d18025603f6df238
        b6f181c2de76b96961a90c1f0fc0d1e2
        42795ddd48f06e6ac2760579459b6b98
        ef77d91c4278dd0a9fea9c91a57eae08"
    },
    {
        outpoint: (
            "3a648c42b90af46b9bba7ae723451002
            aa53baba187020051e0c32112bf458a0",
            3
        ),
        sig:
            "13f1eb894921c7535ac51aac874187ac
            993b0dcf7b3b439b15cf408dbf66db2b
            7739605ef9ab0bc79be7957a9e15ef0b
            e0dd92524c8881cc3fd3742c621e8c0b"
    }
],
outputs: [
    {
        pk: "36dec8c46741efdab93a77f8fc75acec
        6e290b1ce0f280e04753db5c864a6469",
        amount: 5012900000000
    },
    {
        pk: "f71c1e7478459d90764d59dd9cb0ea9f
        62406d2c1412dea7c6de0c0355183066",
        amount: 3170000000000
    },
    {
        pk: "d9169b9601f3121f316f44e4d1bd0ecc
        d6d07df4d152fdcbf59a210e7c77e467"
        amount: 1350000000000
    }
]
}

```

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

### 3.10 Transaction Validation

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

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

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

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

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

It's possible that *this outputpoint was never added to the UTXO set in the first place*. This can occur for two different reasons. The first reason is malicious. The adversary is creating a transaction spending from a non-existing outputpoint. This transaction must be rejected. The second reason is benign, and it is a *race condition*. If Alice pays Charlie in one transaction tx<sub>1</sub> and then Charlie pays David in another transaction tx<sub>2</sub>, which spends from tx<sub>1</sub>, then tx<sub>1</sub> and tx<sub>2</sub> will be broadcast in this order. However, the verifier may receive them on the network in a different order than they were sent. He can see tx<sub>2</sub> first, and tx<sub>1</sub> only later. If a verifier sees tx<sub>2</sub> first, then he cannot verify this transaction before he has seen tx<sub>1</sub>. After all, he doesn't have the necessary public key to verify the respective signature, and

he doesn't have the necessary amounts to verify the Conservation Law. He must necessarily *reject*  $\text{tx}_2$ . It is not the responsibility of the verifier to hold onto  $\text{tx}_2$  until  $\text{tx}_1$  is received, because he cannot know if such a  $\text{tx}_1$  exists in the first place. For all it knows, an adversary could be attempting to spend a non-existent outpoint.

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

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

## Problems

TBD

## Further Reading

The UTXO model was first put forth in the context of Bitcoin by Satoshi Nakamoto. Satoshi introduced blockchains, and his paper, *Bitcoin: A Peer-to-Peer Electronic Cash System* [15] is the seminal paper that spoke about them for the first time. Consider it mandatory reading. It is an easy and short paper that includes details about the UTXO model that we explored in this chapter, but also blocks, chains, and SPV proofs that we will explore in the next chapters. The denomination *satoshi* in Bitcoin is named after Satoshi Nakamoto. Satoshi was a pseudonymous Japanese man who created both the Bitcoin paper and the first Bitcoin implementation in C++ during the years 2008-2009. After two short years of participation in the community, he disappeared mysteriously, never to be heard from again. His identity remains a mystery.

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



# Chapter 4

## Blocks

### 4.1 The Network Delay

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

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

**Definition 9** (Network Delay). *The network delay parameter  $\Delta$  measures the maximum time it takes for a message to travel from one honest party to every other honest party on the network.*

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

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

## 4.2 The Double Spend

Let's try to understand the double spending problem a little more carefully. Eve receives 1 unit of money from Alice through a transaction  $\text{tx}_1$  as illustrated in Figure 4.1. The transaction  $\text{tx}_1$  was created honestly by Alice and has one input of 1 unit coming from Alice, and one output of 1 unit paying Eve's public key. Eve now creates two transactions: The first transaction,  $\text{tx}_2$  consumes the single output of  $\text{tx}_1$  and pays 1 unit back to Alice. The second transaction,  $\text{tx}'_2$  also consumes the single output of  $\text{tx}_1$  and pays 1 unit, this time to Eve.

Suppose that two other parties, Charlie and Dave, have already seen  $\text{tx}_1$  on the network, but have not yet received either of  $\text{tx}_2$  or  $\text{tx}'_2$ . If Charlie receives  $\text{tx}_2$ , he will accept this transaction as valid. The transaction's input contains an outpoint that points to an element of the UTXO set in his view, since the output of  $\text{tx}_1$  has not been previously spent. Furthermore, the transaction contains a valid signature by Eve created with the correct secret key, and it satisfies the Conservation Law. Upon accepting  $\text{tx}_2$ , Charlie updates his UTXO set, removing the output of  $\text{tx}_1$  and adding the output of  $\text{tx}_2$  to it. If Charlie now receives  $\text{tx}'_2$ , he will reject this transaction, as it is spending from an output that is not in the UTXO set in his view.

On the contrary, Dave receives  $\text{tx}'_2$  first, and  $\text{tx}_2$  after. When Dave receives  $\text{tx}'_2$ , he considers this a valid transaction, because it is spending from the UTXO set in his view. Dave, contrary to Charlie, believes that the output of  $\text{tx}_1$  is still in the UTXO set. Dave then updates his UTXO set, removing the output of  $\text{tx}_1$  and adding the output of  $\text{tx}'_2$  to it.

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

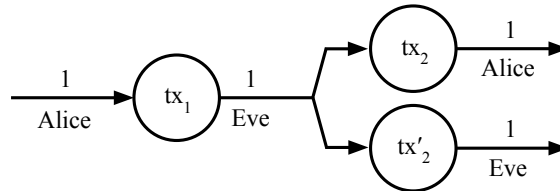


Figure 4.1: A double spend transaction. Alice paid Eve 1 unit through  $\text{tx}_1$ , but Eve spent it in both  $\text{tx}_2$  and in  $\text{tx}'_2$ , which have different recipients.

### 4.3 Simple Ideas Don't Work

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

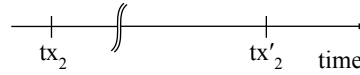


Figure 4.2: The first idea is unviable because it allows the adversary to retroactively invalidate an earlier transaction much later. Here,  $tx'_2$  is initially withheld, but broadcast much later, causing an invalidation to the earlier  $tx_2$  transaction.

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

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

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

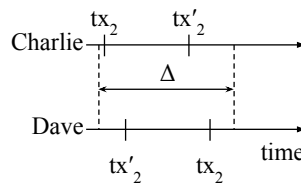


Figure 4.3: The second idea is unviable because parties have views in disagreement about transaction order. Here, Charlie believes  $tx_2$  precedes  $tx'_2$ , whereas Dave believes  $tx'_2$  precedes  $tx_2$ .

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

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

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

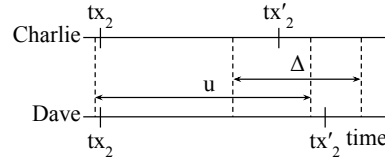


Figure 4.4: The third idea is unviable because parties disagree about whether  $\text{tx}_2$  and  $\text{tx}'_2$  have arrived within time  $u$ , and follow different policies. Here, Charlie rejects both transactions, whereas Dave accepts  $\text{tx}_2$  and rejects  $\text{tx}'_2$ .

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

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

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

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

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

## 4.4 Ledgers

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

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

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

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

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

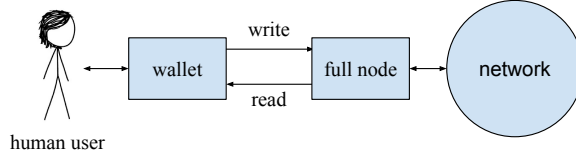


Figure 4.5: The human user interacts with the wallet. The wallet interacts with the full node by invoking its *write* and *read* functionalities. The full node interacts with the network.

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

**Definition 10** (Ledger). *A ledger of an honest party  $P$  reported at time  $r$ , denoted  $L_r^P$  is a finite sequence of transactions returned when the honest party  $P$  invokes the read functionality of its honest protocol  $\Pi$ .*

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

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

**Definition 11** (Safety (informal)). *For any two honest parties, their reported ledgers at any point in time are equal.*

**Definition 12** (Liveness (informal)). *If an honest party writes a transaction into its ledger, then this transaction appears in the ledgers of all honest parties “soon”.*

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

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

**Live but not safe.** A live but not safe protocol acts as follows. Each honest party begins with an empty initial ledger. Whenever the *write* functionality is

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

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

**Definition 13** (Security). *A protocol is called secure if it produces ledgers that are both safe and live.*

## 4.5 Rare Events

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

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

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

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

## 4.6 Proof-of-Work

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

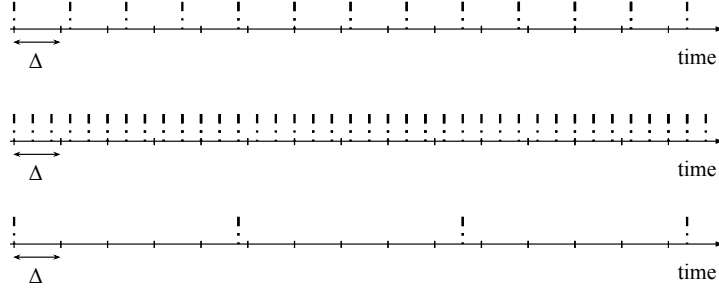


Figure 4.6: Positioning tickets in time. Tickets spread apart by slightly more than  $\Delta$  (top) achieve good safety and liveness. Tickets spread apart less than  $\Delta$  (middle) can cause safety violations. Tickets spread apart much more than  $\Delta$  (bottom) cause liveness to deteriorate.

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

$$H(B) \leq T$$

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

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

---

**Algorithm 10** The proof-of-work algorithm.

---

```

1: function POWH,T
2:   ctr  $\xleftarrow{\$}$   $\{0, 1\}^\kappa$ 
3:   while true do
4:      $B \leftarrow \text{ctr}$ 
5:     if  $H(B) \leq T$  then
6:       return B
7:     end if
8:     ctr  $\leftarrow \text{ctr} + 1$ 
9:   end while
10: end function

```

---



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

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

## 4.7 The Block

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

We've solved one problem, but have introduced another: If we require one ticket per transaction for the adversary, then we must do the same for the honest parties. This means that at most one transaction can be executed every  $\Delta$ . If the honest party wants to issue multiple transactions, this will take a long time, and liveness will deteriorate. In order to solve this, we will *bundle* transactions together into a sequence  $\bar{x} = (\text{tx}_1, \text{tx}_2, \dots, \text{tx}_n)$ , and place this within the ticket, setting  $B = \bar{x} \parallel \text{ctr}$ . Instead of each ticket being associated with *one* transaction, each ticket is associated

with a sequence of transactions  $\bar{x}$ . Such a ticket is known as a *block*. The sequence of transactions  $\bar{x}$  is also known as the *block payload*. Now each honest party can issue as many transactions as he wants in one go, as long as he is able to mine one block. In practice, to save on communication,  $\bar{x}$  may be a list of transaction ids instead of the literal transactions themselves. We'll see more ways to optimize communication when we talk about Light Clients in Chapter ???. From now on, we will no longer speak of tickets, but will speak of blocks, even if a block contains just one transaction.

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

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

## 4.8 The Mempool

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

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

**Definition 14** (Mempool). *The mempool  $\bar{x}$  of an honest party  $P$  at time  $r$  is the sequence of transactions that have been received and validated, but have not yet been*

*included in a block.*

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

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

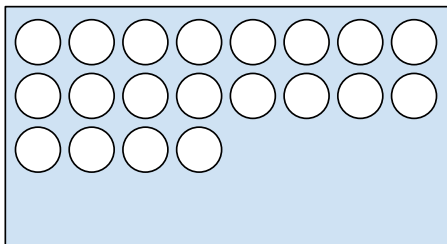


Figure 4.7: A block containing ordered transactions. We will draw a block using a rectangle and a transaction using a circle.

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

## 4.9 Chain of Blocks

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

At this point, we have introduced blocks that function as tickets to allow for the spaced-out broadcasting of transactions. However, our system is not yet safe. There's a problem: While the proof-of-work process ensures that the adversary gets a block in regular spaced out intervals, the adversary is not guaranteed to use these block at the time of issuance. Instead, the adversary could *withhold* a couple of blocks and broadcast them all later in close succession, and in particular closer than  $\Delta$  apart in time. This defeats the purpose we set to achieve. The

attack stems from the ability of the adversary to use blocks that are stale and have been set aside for long. We can resolve the issue by requiring each block to be *fresh*. We can do this by having the block include, in addition to its transactions, a pointer to a previous recent block. This pointer, denoted by  $s$ , is the blockid of the previous block known to the miner, and is known as the *previd*. Our modified block format is now  $B = s \parallel \bar{x} \parallel \text{ctr}$ . Note that, similarly to before, the adversary cannot retroactively change  $s$  after mining a block, because this will invalidate the proof-of-work. This helps with freshness: If a block  $B$  is old and includes a pointer  $s$  to an even older block, this  $s$  cannot be retroactively changed to point to a newer block to make  $B$  appear fresh. The final proof-of-work algorithm illustrated in Algorithm 11. accepts both  $s$  and  $\bar{x}$  as parameters and tries to find a  $\text{ctr}$  that satisfies the proof-of-work equation.

---

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

---

```

1: function POWH,T( $s, \bar{x}$ )
2:    $\text{ctr} \xleftarrow{\$} \{0, 1\}^\kappa$ 
3:   while true do
4:      $B \leftarrow s \parallel \bar{x} \parallel \text{ctr}$ 
5:     if  $H(B) \leq T$  then
6:       return  $B$ 
7:     end if
8:      $\text{ctr} \leftarrow \text{ctr} + 1$ 
9:   end while
10: end function

```

---

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

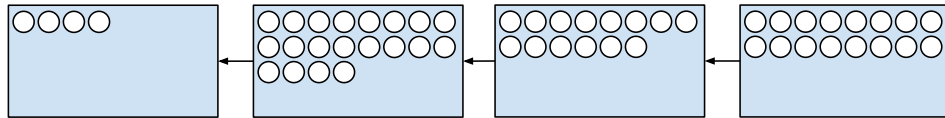


Figure 4.8: A chain of blocks. The blocks were mined from left to right.

## 4.10 Genesis

We've added pointers between blocks to ensure freshness, and each block attests about its freshness by a pointer to a recent block. But what about the freshness of the first block on the chain? That first block cannot have a pointer to a parent block. How do we ensure its freshness? In particular, how do we prevent an adversary from

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

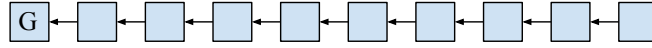


Figure 4.9: A blockchain beginning from genesis.

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

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

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

We will make all honest parties check that all blockchains begin with  $\mathcal{G}$  to avoid premining.

## 4.11 Mining

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

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

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

## Problems

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

## Further Reading

Proof-of-Work was invented long before blockchains, in 1992, by Cynthia Dwork and Moni Naor in their seminal paper *Pricing via Processing or Combatting Junk Mail* [6] published in CRYPTO '92. Satoshi Nakamoto invented the blockchain by using proof-of-work as a building block.



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

## Chapter 5

# The Chain

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

### 5.1 The Target

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

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

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

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

As protocol designers, we need to set  $T$  correctly so that blocks are produced at a rate smaller than once per  $\Delta$ . For this, we need to have an estimate of how



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

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

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

In a nutshell, the more computational power there is on the network, the larger we must make the difficulty. Also, the larger the network delay, the larger we must make the difficulty.

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

## 5.2 The Arrow of Time

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

In order to be able to speak about chains more precisely, let us introduce a bit of notation. A chain  $\mathcal{C}$  is a finite sequence of blocks  $\mathcal{C} = (B_0, B_1, B_2, \dots, B_{n-1})$  ordered chronologically. The chain *length*  $|\mathcal{C}|$  is the number of blocks in the chain. We address these blocks by their zero-based index, so  $\mathcal{C}[0]$  is the first (and oldest) block on the chain,  $\mathcal{C}[1]$  is the second, and so on. The first block among a complete chain is the genesis block, so  $\mathcal{C}[0] = \mathcal{G}$ . We use negative indexing to address blocks from the end of the chain, so  $\mathcal{C}[-1]$  is the last (and most recent) block,  $\mathcal{C}[-2]$  is the penultimate block, and so on. The block  $\mathcal{C}[-1]$  is called the *tip*. The index of a block within its chain is called the block's *height*, so genesis has height 0

and the tip has height  $|\mathcal{C}| - 1$ . It will also be useful to speak about *chunks* of a chain, continuous portions of the chain. We'll denote by  $\mathcal{C}[i:j]$  the portion of the chain starting at block with index  $i$  (inclusive) and ending at block with index  $j$  (exclusive). For example,  $(B_0, B_1, B_2)[0:2] = (B_0, B_1)$ . These  $i$  and  $j$  can also be negative, again meaning indexing from the end. Omitting  $i$  means starting from the beginning, while omitting  $j$  means going until the end of the chain. For example  $(B_0, B_1, B_2, B_3, B_4, B_5)[-2:] = (B_4, B_5)$ .

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

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

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

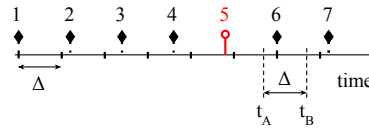


Figure 5.1: a series of successful honest (black solid diamonds) and adversarial (red hollow circles) queries regularly spaced apart more than  $\delta$ .

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

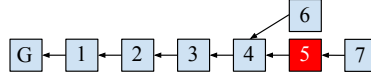


Figure 5.2: a *blocktree* instead of a *blockchain* arising out of the timeline of figure 5.1.

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

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

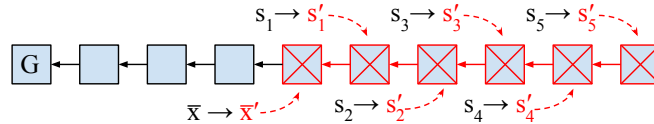


Figure 5.3: Changing just one small part of a block's content causes a cascading effect invalidating the proof-of-work of the whole chain inheriting from that block.

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

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

**Longest Chain Rule.** Among all chains on the network, adopt the *longest* chain (the one with the most blocks) as your *canonical chain*.

If there are multiple competing chains of the same length, choose any chain arbitrarily.

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

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

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

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

Once a party has adopted a canonical chain, reporting a ledger of transactions to implement the *read* operation of a full node is straightforward: Given an adopted chain  $\mathcal{C} = (B_0, B_1, \dots, B_{n-1})$ , the ledger reported is

$$L = B_0.\bar{x} \parallel B_1.\bar{x} \parallel \dots \parallel B_{n-1}.\bar{x}.$$

Here, we use the notation  $B.\bar{x}$  to denote the  $\bar{x}$  component of block  $B$ . The ledger reading rule tells us that we take the transactions in each block of the adopted chain (which are already ordered sequences of transactions) and concatenate them into one big list of transactions in the order mandated by the chain.

### 5.3 The Stochastic Nature of Work

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

In Figure 5.5, we illustrate a blocktree that can arise out of the timeline of queries in Figure 5.4. Even though *all* successful queries were honest, the honest parties

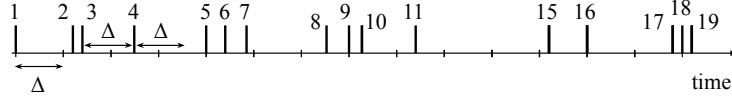


Figure 5.4: The stochastic nature of work makes honest successful queries irregular.

who used the longest chain rule built a blocktree and not a single blockchain. At the end of the execution, the honest parties are split between the canonical chains they have adopted: Some parties have adopted the chain ending on block 17, some have adopted the chain ending on block 18, and some have adopted the chain on block 19, as their canonical chain. If these three blocks contain double spends, the different honest parties could be reporting different ledgers. This happened even though the adversary was not mining any blocks. This is a loss of safety.

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

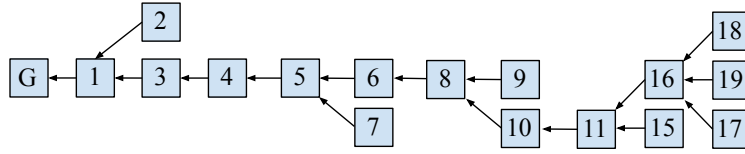


Figure 5.5: A blocktree arising out of the timeline of honest-only queries of Figure 5.4.

Let's go back to Figure 5.5. In this picture, we notice that there really is only one longest chain in the blocktree, except with some extra blocks here and there. This longest chain begins from genesis and goes on until block 16. Here and there on the sidelines of the longest chain, we see a few blocks popping up such as block 2 and block 7. These smaller branches alongside the longest chain are called *temporary forks* and can be one or more blocks long, but typically these temporary forks will be quite short.

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

**Definition 15** (Convergence Opportunity (informal)). *An honest successful query is called a convergence opportunity if it is  $\Delta$  separated in time from all other honest successful queries.*

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

## 5.4 The Honest Majority Assumption

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

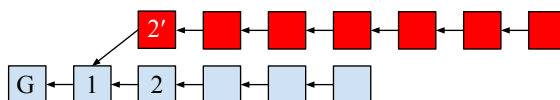


Figure 5.6: An adversary with dishonest majority (top red chain) can outpace the honest parties (bottom blue chain).

This situation is pretty bad. It causes a *loss of safety*, as some of the honest parties have switched from their ledgers to the adversary's ledger, causing disagreement. It also causes a *loss of liveness*, because the adversary may choose not to include any honest transactions. However, if the adversary only has a minority of mining power, even though these attacks can start small, they cannot grow for much longer, as the adversary will be left behind in the dust by the honest parties who will be mining more quickly. We'll explore these attacks much more deeply in Chapter 6, and we'll formalize them in Chapter 11. However, it should already be clear that we must require that the adversary controls only a minority of the compute of the network. We will call this the *honest majority assumption*.

**Definition 16** (Honest Majority Assumption). *The honest majority assumption mandates that the adversary controls less compute than the honest parties:*

$$t < n - t$$

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

## 5.5 Coinbase Transactions

Now that we have completed the design of our longest chain protocol, we have finally solved the problem of how money is *transferred* in a publicly verifiable and decentralized manner. However, we never tackled the problem of how money is *created* in a decentralized manner. If there is no central authority to issue money, how can we create new money, and who receives any new money that is created? With the invention of blocks, there is a natural choice for this. Since blocks are generated in regular intervals, we can use the block to inject the money supply with new money in a manner that maintains controlled scarcity. This is where the

*coinbase* transaction comes into play. Recall that a coinbase transaction has no inputs and a single output with a certain value coming out of it, as illustrated in Figure 5.7. It is the only type of transaction that does not respect the Conservation Law. Our rule says that every block must have *at most one* coinbase transaction, and that transaction must appear *first* in the list of transactions of the block, if at all (different blockchain systems may have slightly different details on what rules they enforce on the syntax of coinbase transactions). The public key in the output of the coinbase transaction is freely chosen by the miner. Naturally, because this is free money, the miner will typically make that public key be their own, so that they can reap the benefits of mining.



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

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

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

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

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

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

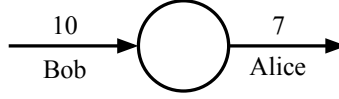


Figure 5.8: A transaction with an input of 10 and an output of 7 pays a fee of 3 units that go to the coinbase output.

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

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

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

To verify a coinbase transaction, we follow these steps:

1. Check that it is the only one in that block.
2. Check that it is the first transaction in the block.
3. Check that it has no input and exactly one output.
4. Check that its output has a value which is less than or equal to the sum of the block reward and the difference between the input and output amounts of all *other* transactions in the block.
5. Check that the coinbase is not spent in the same block.

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

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

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



## 5.6 Block Validation

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

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

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

The validation of transactions  $\bar{x}$  in a block works as follows. For each block  $B$ , we remember a UTXO set which we associate with the state  $\text{st}(B)$  of the world *after* the particular block has been adopted. We also define a *genesis state*  $\text{st}_0$ , the state of the world *before* any transaction ever took place. The genesis state, which is where the UTXO set begins its lifetime, is defined to be the empty set.

To validate the transactions in a block  $B = s \parallel \bar{x} \parallel \text{ctr}$ , we take the state  $\text{st}_{B'}$  associated with its parent block  $B'$  whose blockid is  $s = H(B')$ , or  $\text{st}_0$  if  $B = \mathcal{G}$ . We then attempt to apply each of the transactions in the block, in the order they are defined in  $\bar{x}$ , updating the UTXO set as we go along by consuming and producing elements in the set, and validating every transaction along the way, in the manner we already discussed in Chapter 3. If at any point the transaction validation fails, the whole block is rejected, even if just one transaction was invalid. If all transaction validations succeeded, we assign the state  $\text{st}_B$  after our block to be the UTXO set we arrived at after this process is completed. This is illustrated in Figure ?? . In this example,  $B.\bar{x} = (\text{tx}_1, \text{tx}_2, \text{tx}_3, \text{tx}_4)$ .

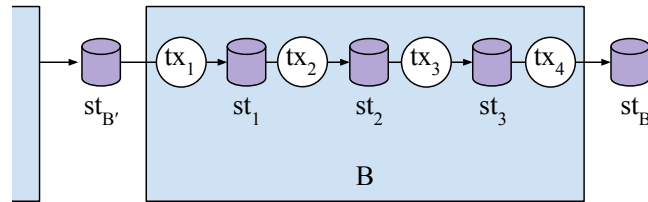


Figure 5.9: Validating the  $\bar{x}$  part of block  $B$ . The intermediate states are shown as purple cylinders. The state  $\text{st}_{B'}$  comes from a block which has already been validated (shown partially on the left).

While we will sometimes say that *chains* are exchanged on the network, it is really *blocks* that are sent over the network. Due to the fact that each block contains a reference to its parent, a block uniquely identifies the chain of which it is a tip. When a party wishes to send a chain to another party, it is sufficient that the tip is sent over. The other party can then request the ancestor blocks if he doesn't already have them. In practical protocols, sometimes these requests for objects are

bundled together in batches to improve communication complexity. For example, all the transactions of one block can be downloaded in one go instead of requesting each of them independently. Validating a chain equates to validating its tip, as this requires validating the parent block recursively. The genesis block is hard-coded into every full node and is considered valid by definition. This is where the recursion stops.

## 5.7 Maintaining the Mempool

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

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

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

This algorithm will be made clearer with an example. Consider the situation illustrated in Figure 5.10, and suppose our party has adopted the chain whose tip

is  $B'_2$  (bottom) and has a mempool of  $\bar{x}'$ . Suddenly, block  $B_3$  arrives. The party downloads the chunk  $(B_1, B_2, B_3)$  and validates it. As before, this validation begins by looking at the state of the latest common ancestor  $B$ . This has the effect that all the transactions in  $B'_1$  and  $B'_2$  are undone prior to applying the transactions in  $B_1$ . First, the transactions in  $B_1$  are applied and  $\text{st}(B_1)$  is calculated. Then  $B_2$  is applied and  $\text{st}(B_2)$  is calculated. Lastly,  $B_3$  is applied and  $\text{st}(B_3)$  is calculated. If at any point a transaction cannot be applied, the whole block containing it is rejected.

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

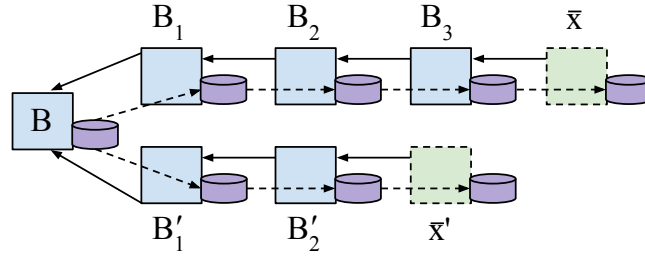


Figure 5.10: The state and mempool calculation in the case of a reorg. Ancestry relations are shown in solid arrows, whereas state updates are shown in dashed arrows. Blocks are depicted as blue solid boxes; states are depicted as purple cylinders; mempools are depicted as dashed green boxes.

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

## Chapter 6

# Chain Attacks DRAFT

### 6.1 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

### 6.2 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  $\Delta$ , but must satisfy this property upon reaching convergence opportunities (more on this in Section 2.2).

### 6.3 Chain Virtues

#### 6.3.1 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).



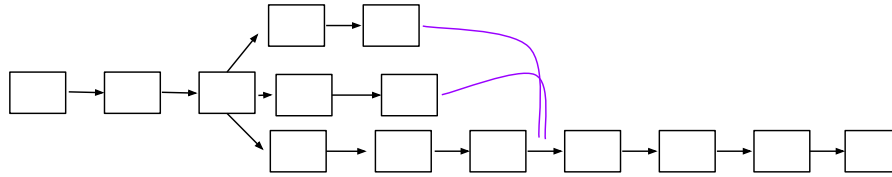


Figure 6.3: In this scenario the chain diverges after the third block. This is because there were three blocks mined at about the same time. There is a convergence opportunity on the sixth block on the bottom chain. At this point honest nodes will switch over since they see the bottom chain is the new longest chain. The purple curved lines represent the honest parties switching over and adopting the new longest chain.

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.

#### 6.4.1 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 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 than 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.

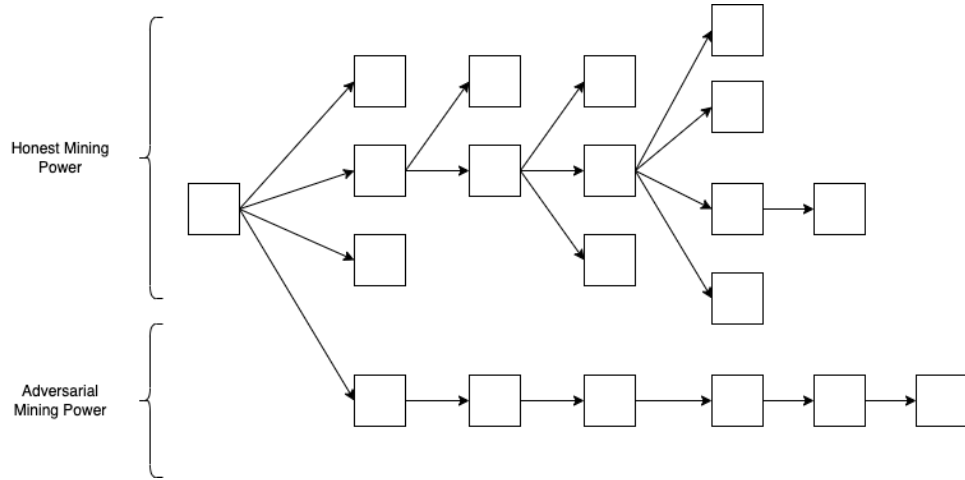


Figure 6.4: A fan-out attack where an adversary who does not hold majority hashing power can still produce the longest chain due to honest mining power being wasted. Displays a drawback of having infrequent convergence opportunities.

### 6.4.2 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.

### 6.4.3 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 Selfish Miner 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  $\mu$
3. Chain growth with parameter  $\tau$

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  $\mu$  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  $\tau$  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.

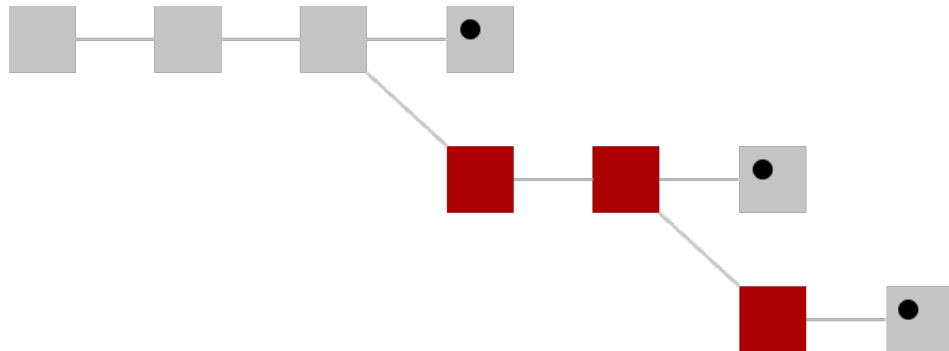


Figure 7.1: Depiction of a censorship attack. Grey blocks are mined by honest miners. Red blocks are valid blocks mined by the adversary. The black dot represents the transaction  $tx$  that the adversary is trying to censor.

## 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  $\tau$  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  $\delta$ , 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  $\mu$ , 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:

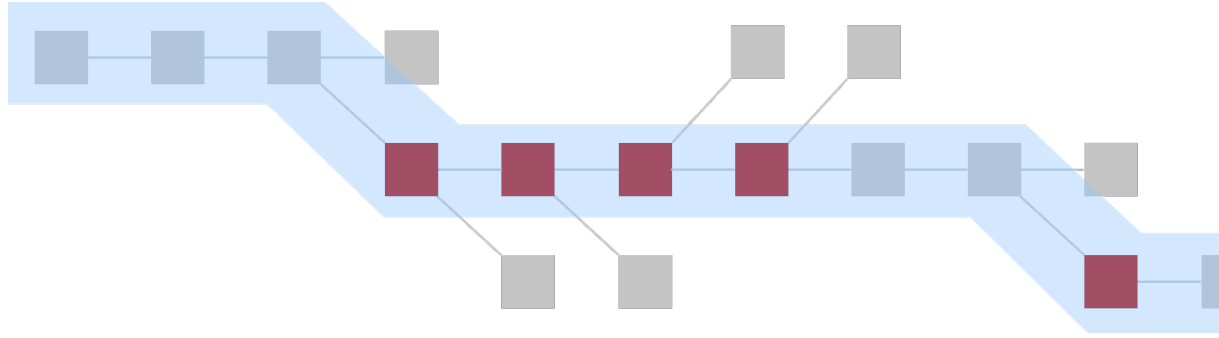


Figure 7.2: In this example, suppose that  $n = 17$ ,  $t = 5$  and  $\frac{n-t}{n} \approx 0.71$ . There are 5 adversarial blocks and 12 honest blocks mined which are proportional to the compute powers. In the longest chain here (highlighted in blue), there are 5 adversarial blocks and 11 total blocks, 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  $\mu$  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 produced 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()

```

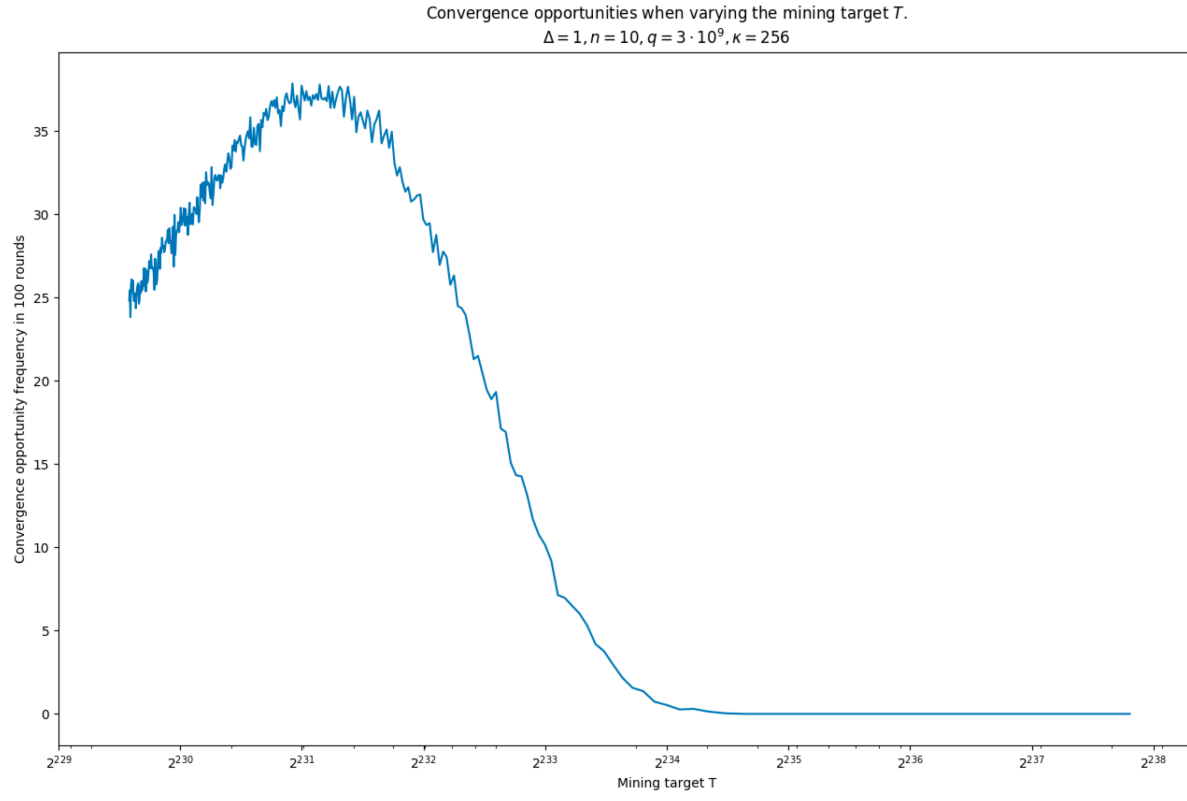


Figure 7.3: Plot of convergence opportunity frequency versus target  $T$

## Chapter 8

# Economics DRAFT

### 8.1 Our Variables

- $\kappa$ : 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
- $\mu$ : Chain Quality
- $\tau$ : Chain Growth

### 8.2 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 8.1. This can be denoted as:

$$q \cdot (n - t) \approx 2^{67} \text{ Hz.}$$

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  $q \approx 20 \text{ GHz}$ . Today, we also use specialized mining power, with the best machines (ASICs) achieving  $q \approx 200 \text{ THz}$ . To see the hash power of the ASIC machine, you can refer to this website: [www.asicmine.com](http://www.asicmine.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:



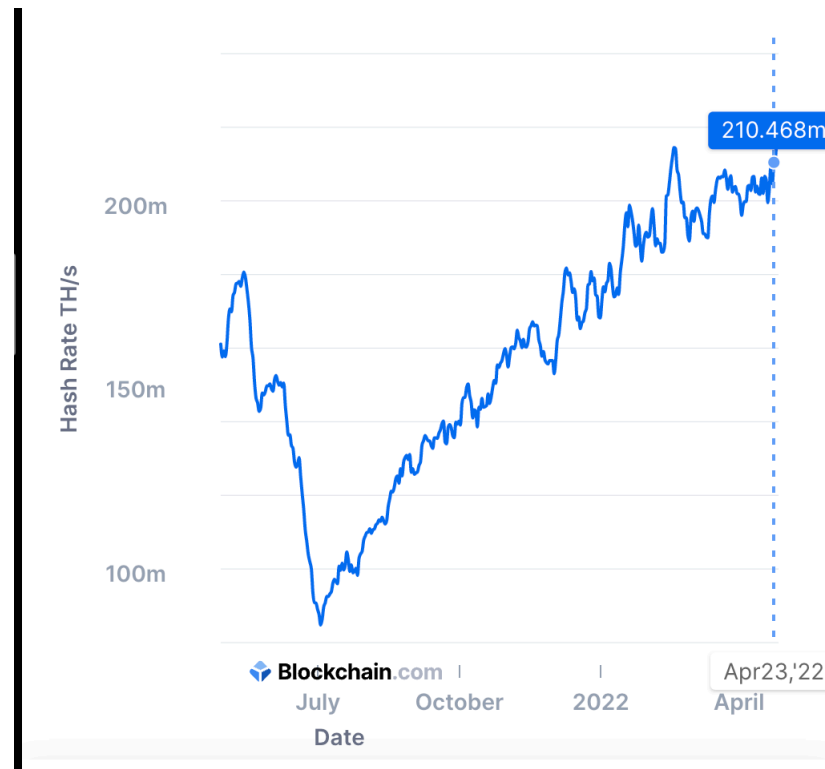


Figure 8.1: Hash Rate of Bitcoin from April 2021 to April 2022 [?]

- The number of transaction spikes in the weekdays and troughs 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.

## 8.3 Mining

### 8.3.1 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.

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

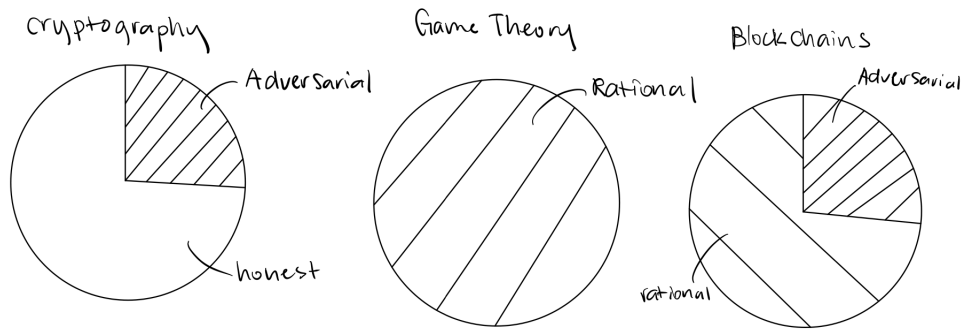


Figure 8.2: Types of Parties in Cryptography, Game Theory, and Blockchains

### 8.3.2 The parameter $\Delta$ and Block Size Limit

Previously, we defined  $\Delta$  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  $\Delta$  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.

### 8.3.3 Including a transaction

Now that we have a constraint on the block size, the rational miner should adopt some strategies to maximize profit. Different 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  $\rightarrow$  include all transactions to the new block

Case 2: Mempool does not fit  $\rightarrow$  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.

### 8.3.4 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 8.3, 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.

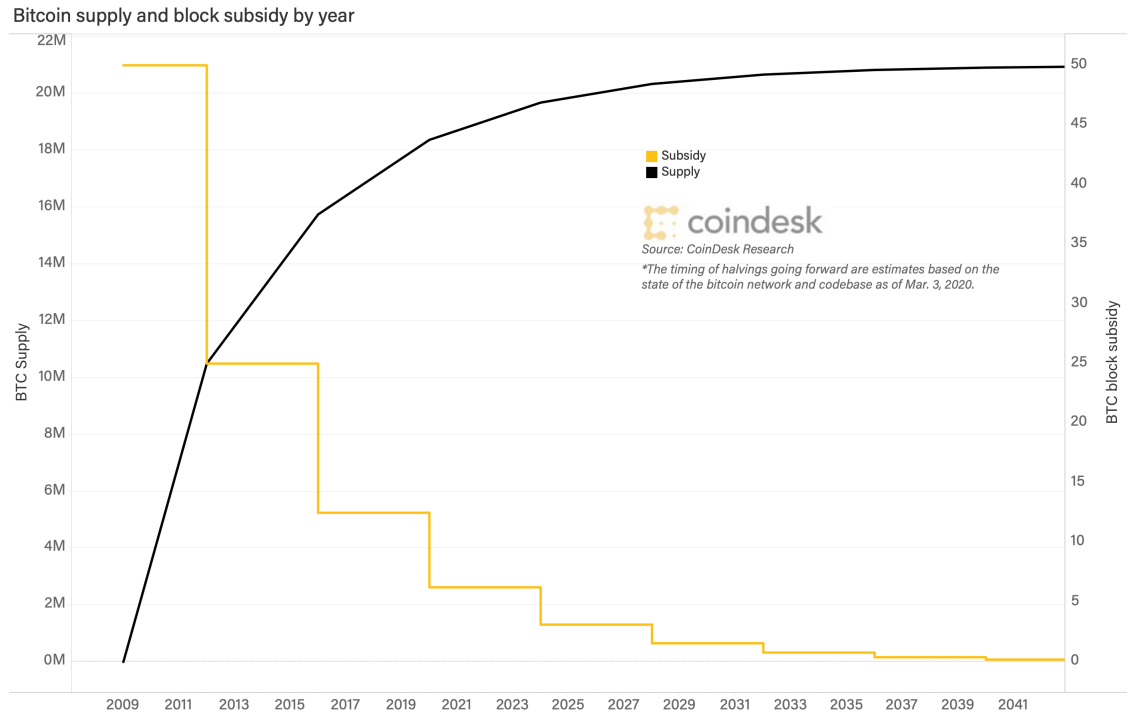


Figure 8.3: Reward Halving for Bitcoin Supply [?]

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.

## 8.4 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  $\Delta$ . To keep a desired block production rate, we want to scale the difficulty appropriately as the hash power of the network increases.

### 8.4.1 Definitions

**Definition 17.** Let  $f$  be the probability of getting an honest block in one unit of time. Then,

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

where  $(1 - p)^{q(n-t)}$  is the probability that every honest party failed.

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

We split the chain into sections  $m$  blocks long.

**Definition 19.** 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

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

### 8.5.1 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 21.** *The light PoW equation provides a target that is significantly easier, called a light block share:*

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

where  $\xi$  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 paid 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.

## 8.6 Wallets

### 8.6.1 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 8.4. The higher-fee transaction replaces the older one in the mempool.

#### 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.

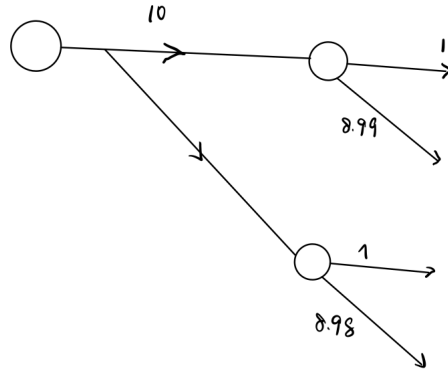


Figure 8.4: An example of a honest, rational party not being able to send a transaction and creating another replace by fee transaction

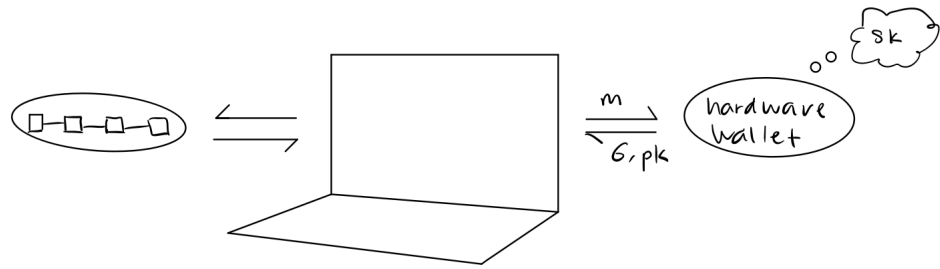


Figure 8.5: Illustration of the interactions between the hardware wallet and the computer

### 8.6.2 HD Wallets

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

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.

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.

$$\begin{aligned} H(\text{ctr} \parallel \text{seed}) &\longrightarrow \text{new sk} \\ H(1 \parallel \text{seed}) &\longrightarrow \text{new sk}_0 \\ H(2 \parallel \text{seed}) &\longrightarrow \text{new sk}_1 \\ &\dots \end{aligned}$$



Figure 8.6: An Example of a Hardware Wallet [?]

# Chapter 9

## Accounts DRAFT

### 9.1 Accounts Model

#### 9.1.1 Accounts Model Compared with UTXO Model

Recall the previous UTXO model: we store a set of unspent transaction outputs (UTXOs). When a transaction occurs, UTXOs corresponding to the transaction's inputs are removed and UTXOs corresponding to the transaction's outputs are added into the UTXO set to produce a new UTXO set, as shown in Figure 9.1.

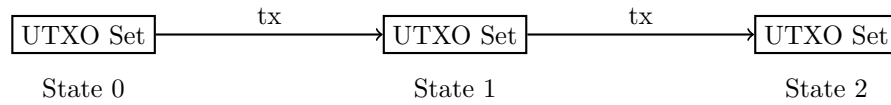


Figure 9.1: State transitions in the UTXO model

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 ( $\sigma$ ).

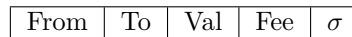


Figure 9.2: Structure of a transaction in the accounts model

For the accounts model, the state is maintained by accounts (public keys) and balances, as shown in Figure 9.3 and Figure 9.4.

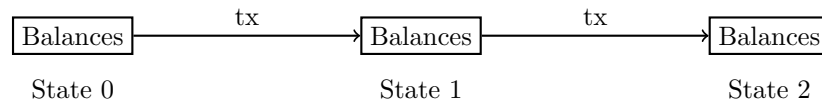


Figure 9.3: State transitions in the accounts model

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 9.4: Balance State

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

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

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

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

The transaction validation process of UTXO model is

- Check  $\sigma$
- Check conservation
- Check inputs are in  $st$

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

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

The transaction validation process of the accounts model is

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

### 9.1.2 Accounts Model Replay Attack

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

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

From	To	Val	Fee	<b>nonce</b>	$\sigma$
------	----	-----	-----	--------------	----------

Figure 9.5: 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 9.5.

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 9.6.

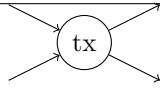
	UTXO	Accounts						
Real System	Bitcoin	Ethereum						
Transaction $tx$		<table><tr><td>From</td><td>To</td><td>Val</td><td>Fee</td><td><b>nonce</b></td><td><math>\sigma</math></td></tr></table>	From	To	Val	Fee	<b>nonce</b>	$\sigma$
From	To	Val	Fee	<b>nonce</b>	$\sigma$			
Transistion $\delta$	Remove consumed outputs and add produced outputs	<div>Update balances</div> $st'[from] := st[from] - value$ $st'[to] := st[to] + value$						
Validation	Signature, Law of Conservation, Inputs exist in $st$ .	Signature, Sufficient balance, Nonce unique.						
Genesis State	$\emptyset$	$\{\}$						

Figure 9.6: Side by Side Comparison of Two Models

## 9.2 State Machine Replication

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

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

## 9.3 Light Clients

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

### 9.3.1 Storage Efficiency: Merkle Trees

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

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

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

### 9.3.2 Data Structure: Merkle Tree

Files are split into  $n$  data chunks.

$$D : D[0], D[1], \dots, D[n-1]$$

A binary tree of depth  $\mu$  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)  $h_e$ . When a data chunk is requested, the server sends the data chunk, along with every sibling hash value to the clients as shown in Figure 9.7. For example, when data chunk at index  $j$  is requested, the server sends  $D[j]$ ,  $\pi_0$ ,  $\pi_1$ ,  $\pi_2$ , and  $\pi_3$  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  $h_\epsilon$ .

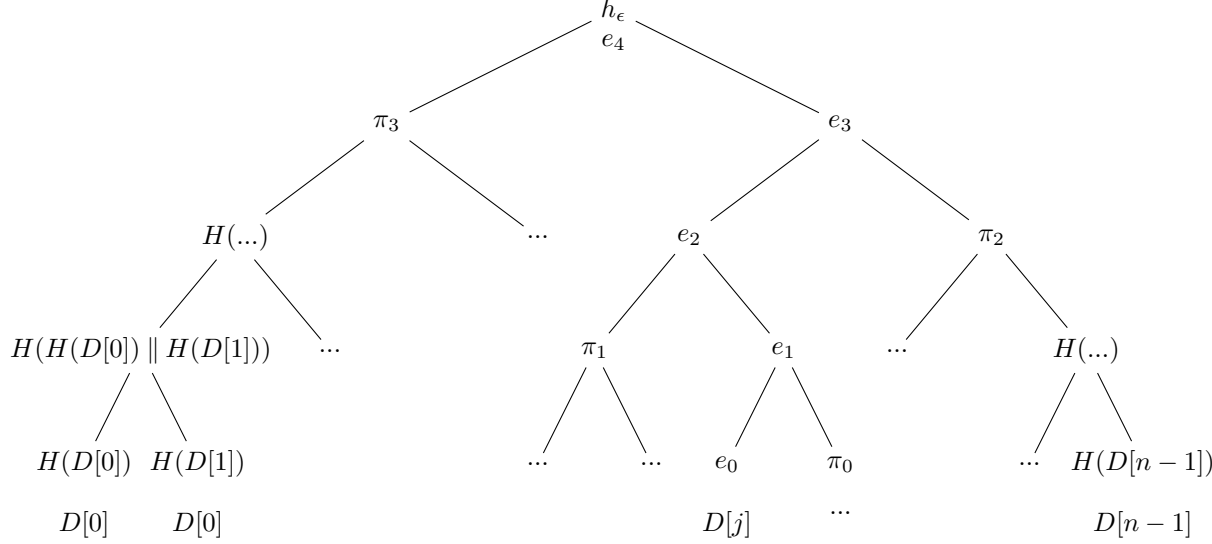


Figure 9.7: Merkle Tree

With this data structure, the data transferred is a list of  $\pi$  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

$$\text{compress}(D) \rightarrow h_\epsilon, \quad (9.4)$$

$$\text{prove}(D, j) \rightarrow \pi, \text{ and} \quad (9.5)$$

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

The correctness of the Merkle tree is specified as:

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

### 9.3.3 Security of Merkle Trees

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

$\text{MERKLE}_{\mathcal{A}}(\kappa) :$

$D, \pi, j, d \leftarrow \mathcal{A}(1^\kappa)$

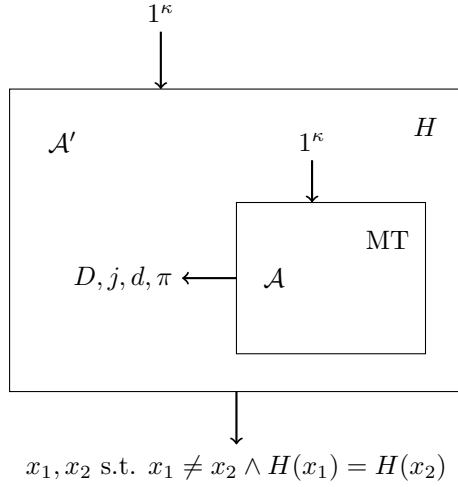
return  $\text{verify}(\text{compress}(D), d, j, \pi) \wedge d \neq D[j]$

Our goal is to prove that

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

**Theorem 5.** *Let  $H$  be a collision-resistant hash function. Then Merkle trees constructed with  $H$  are MT-secure.*

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



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

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

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

Given that  $\mathcal{A}$  succeeds, the returned list of  $\pi$  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 ( $h_\epsilon = e_{\text{top}}$ ) and the hash of the data chunk is different ( $e_0 \neq h_0$ ). (If not,  $\mathcal{A}'$  has already found a collision because different data chunks have the same hash.) Therefore, there must exist a node, some level  $k$  in the tree, such that  $e_k = h_k$  but its children  $e_{k-1}$  or  $\pi_{k-1}$  not equal to the expected  $h$  values. (These must exist because roots are the same, but leaves are different). In this case, we have two different inputs that hash to the same value.

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

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

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

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



# Chapter 10

## Light Clients DRAFT

### 10.1 Motivation

Our current model has three main scalability limitations: storage requirements (full nodes store the entire chain,  $\sim 1\text{TB}$  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  $\sim 100\text{MB}$  of storage, will only download transactions pertinent to our own address, and will not validate all transactions in the transaction graph.

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

### 10.2 Definition

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

### 10.3 Header Chains

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

Instead of a chain of blocks, each containing a vector of transactions, header chains are chains of block headers. Block headers are of the format:  $s||x||\text{ctr}$ , where  $s$  refers to the hash of the previous block header,  $x$  refers to the root of the merkle tree of all transactions in  $\bar{x}$  (the transactions contained in the block), and  $\text{ctr}$  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  $\pi$  from their full node peers for the inclusion of pertinent transactions in the chain, which no PPT adversary can create for a transaction which was not included. Therefore, header chains provide the same guarantees with regards verification of transaction inclusion.

### 10.3.1 Block Validation

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

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

### 10.3.2 Benefits

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

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

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

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



## 10.4 Making Payments

A core function of the light client is interacting with the block chain to make and receive payments concerning only itself. Thus, light clients interact with a 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.

## 10.5 Block Header Validation

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

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

### 10.5.1 Transaction Validation

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

### 10.5.2 Local Chain Security

Common Prefix is still guaranteed for honest-majority networks (specifically, networks where a majority of the mining power is honest), as the PoW requirements on header chains ensure that each header in the chain is a successful query. As a

result, finding a valid block header is just as difficult as finding a valid block, so no minority adversary can outpace the honest nodes in the network. However, there are some security risks for the light client in the event of a dishonest majority.

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

### 10.5.3 Privacy

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

### 10.5.4 Full Node Ramifications

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

## 10.6 Light Miners and the Quick Bootstrap Protocol

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

### 10.6.1 Mining as a Light Client

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

However, mining as a light client is complicated by light clients' inability to validate transactions. The necessary information for validating transactions is the transaction graph of the entire blockchain, or more specifically the UTXO set after execution of the most recent block. Normally, the UTXO set is computed in the process of validating the full transaction graph of the blockchain. But, to allow for light clients (or nodes which do not download the full transaction history of the chain) to mine, we modify our block header structure to be  $s||x||ctr||st$ . We define

$st$  as the merkle root of the merkle tree made of all UTXOs in the UTXO set after executing the block, which we will call  $\bar{st}$ . That is,  $st$  is the merkle root of the merkle tree containing all necessary information for validating transactions which come after the execution of the current block.

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

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

### 10.6.2 Quick Bootstrap Protocol

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

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

The protocol is as follows:

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

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

### 10.6.3 Light Miner Security

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

miner accepts only valid longest chains, by starting with the end of the commonly valid portion of the chain.

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

# Chapter 11

## Security in Earnest I DRAFT

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

### 11.1 Random Oracle

In Chapter 4, we argued that the hash function behaves like a random oracle and that  $\Pr[H(B) \leq T] = p = \frac{T}{2^\kappa}$ . Let us now make the random oracle model more precise. We want the hash function to return a uniformly randomly chosen  $\kappa$ -bit value whenever it is invoked with a fresh input. However, in order for it to be a function, we want it to return the *same* value when it is invoked again with the same input. The random oracle is a shared functionality among all honest parties: If two different parties invoke it, the oracle must answer consistently. This is illustrated in Algorithm 12. This consistency between different parties and multiple queries is necessary so that if a party successfully mines a block, every other party will be able to verify that this mining was successful.

---

**Algorithm 12** The Random Oracle model.

---

```
1:  $\mathcal{T} \leftarrow \{\}$ 
2: function  $H(B)$ 
3:   if  $B \notin \mathcal{T}$  then
4:      $y \xleftarrow{\$} \{0, 1\}^\kappa$ 
5:      $\mathcal{T}[B] \leftarrow y$ 
6:   end if
7:   return  $\mathcal{T}[B]$ 
8: end function
```

---

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

## 11.2 Synchrony

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

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

## 11.3 The Simulation Environment

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

---

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

---

```

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:      $P_i \leftarrow \text{new } \Pi^{H_{\kappa, i}}(\mathcal{G})$ 
6:   end for
7:    $A \leftarrow \text{new } \mathcal{A}^{H_{\kappa, 0}}(\mathcal{G}, n, t)$  ▷ 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  $r < \text{poly}(\kappa)$  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
17:       $M \leftarrow M \cup \{P_i.\text{execute}^H(\overline{M}[i])\}$  ▷ Adversary collects all messages
18:    end for
19:     $Q \leftarrow tq$  ▷ Max number of Adversarial oracle queries
20:     $\overline{M} \leftarrow A.\text{execute}^H(M)$  ▷ 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:         $\text{assert}(m \in \overline{M}[i])$  ▷ Non-eclipsing assumption
24:      end for
25:    end for
26:  end while
27: end function

```

---

### 11.3.1 A Simplification: Quantize Time

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

### 11.3.2 Rushing Adversary

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

### 11.3.3 Sybil Adversary & Non-Eclipsing Assumption

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

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

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

## 11.4 Random Oracle Model

In the simulation algorithm, for both the honest parties and adversarial parties we write `executeH` (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:  $\mathcal{T} \leftarrow \{\}$  ▷ Initiate Cache
3:  $Q \leftarrow 0$  ▷  $q$  for honest parties,  $qt$  for adversary
4: function  $H_\kappa(x)$ 
5:   if  $x \notin \mathcal{T}$  then ▷ First time being queried
6:     if  $Q = 0$  then ▷ Out of Queries
7:       return  $\perp$ 
8:     end if
9:      $Q \leftarrow Q + 1$ 
10:     $\mathcal{T}[x] \xleftarrow{\$} \{0,1\}^\kappa$  ▷ Sample uniformly at random from output space and
    store in Cache
11:  end if
12:  return  $\mathcal{T}[x]$  ▷ Return value from Cache
13: end function
```

---

## 11.5 Honest Party Algorithm

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

The first algorithm is a constructor.

The second algorithm is used to simulate every honest party that mines during each round of the protocol. In this simulation, every honest party follows the longest chain rule, at the beginning of each round they adopt the longest, valid chain (line 8). If they learn about a new chain that is longer than 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( $1^\kappa$ )
8:    $\tilde{\mathcal{C}} \leftarrow \text{maxvalid}(\mathcal{C}, \bar{M}[i])$  ▷ 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  $B \in \mathcal{C}$  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
```

---

## 11.6 Proof-of-Work

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

---

**Algorithm 16** The Proof-of-Work discovery algorithm

---

```
1: function POWH,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
```

---

## 11.7 Longest Chain

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

---

**Algorithm 17** The maxvalid algorithm

---

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

---

## 11.8 Validating a block

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

---

**Algorithm 18** The validate algorithm

---

```
1: function VALIDATEG,δ(·)(C)
2:   if C[0] ≠ G then                                     ▷ Check that first block is Genesis
3:     return false
4:   end if
5:   st ← st0                                              ▷ Start at Genesis state
6:   h ← H(C[0])
7:   st ← δ*(st, C[0].x)
8:   for B ∈ C[1:] do                                       ▷ Iterate for every block in the chain
9:     (s, x, ctr) ← B
10:    if H(B) > T ∨ s ≠ h then                               ▷ PoW check and Ancestry check
11:      return false
12:    end if
13:    st ← δ*(st, B.x)                                       ▷ Application Layer: update UTXO & validate
    transactions
14:    if st = ⊥ then
15:      return false                                         ▷ Invalid state transition
16:    end if
17:    h ← H(B)
18:  end for
19:  return true
20: end function
```

---

## 11.9 Chain Virtues

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

1. **Common Prefix** ( $\kappa$ ).  $\forall$  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** ( $\mu, \ell$ ).  $\forall$  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  $\mu$ .
3. **Chain Growth** ( $\tau, s$ ).  $\forall$  honest parties  $P$  and  $\forall r_1, r_2$  with adopted chain  $C_1$  at round  $r_1$  and adopted chain  $C_2$  at round  $r_2 \geq r_1 + s$ , it holds that  $|C_2| \geq |C_1| + \tau s$ .

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

### 11.10 Pairing Lemma

**Lemma 6.** *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.

### 11.11 Honest Majority Assumption $(n, t, \delta)$

We will now give a new definition of the honest majority assumption by introducing the honest advantage parameter  $\delta$ . 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  $t < (1 - \delta)(n - t)$ .

### 11.12 Further Reading

Even though Satoshi Nakamoto developed the first blockchain and wrote the paper about it, he did not prove that blockchains are secure against *all* adversaries within that paper. Instead, he showed that Bitcoin is secure against the *specific* Nakamoto adversary which we studied in the previous chapters. At a later time, Juan Garay, Aggelos Kiayias, and Nikos Leonardos wrote the *Bitcoin Backbone* paper [7]. The Bitcoin Backbone paper formalizes and proves that blockchains that use Proof-of-Work are secure.

# Chapter 12

## Security in Earnest II DRAFT

### 12.1 Safety and Liveness

#### 12.1.1 Defining Safety and Liveness

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

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

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

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

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

#### 12.1.2 Common Prefix Implies Safety

**Theorem 7.** *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  $\text{tx}$  to appear in the ledger of an honest party,  $\text{tx}$  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.  $\square$

### 12.1.3 Chain Quality and Chain Growth Imply Liveness

**Theorem 8.** *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  $\ell$  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  $u = \max(\frac{\ell+k}{\tau}, s)$   $\square$

## 12.2 Proving Chain Growth, Chain Quality, and Common Prefix

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

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

Over an interval of consecutive rounds  $S$ :

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

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

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

### 12.2.1 Chain Growth Lemma

**Lemma 9** (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  $l + \sum_{i=r}^{r'-1} X_i$  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  $|C| = l = l + \sum_{i=r}^{r'-1} X_i$ .

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

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

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

### 12.2.2 Proving Common Prefix and Chain Growth

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



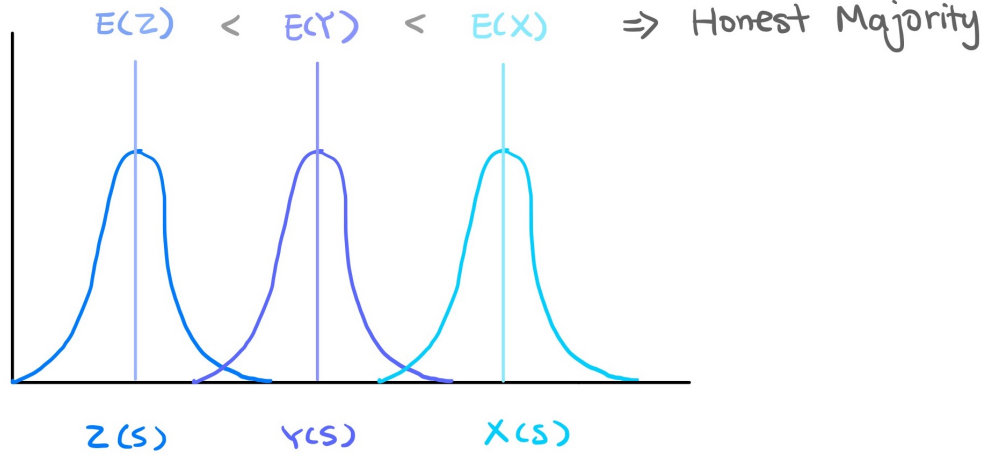


Figure 12.1: Illustration of the probability density functions for  $X(S)$ ,  $Y(S)$ , and  $Z(S)$ . Note that  $\mathbb{E}[Z(S)] < \mathbb{E}[Y(S)] < \mathbb{E}[X(S)]$

But if we want common prefix to hold except with negligible probability, it is not sufficient that the *expectations* are in this order. It is not sufficient that *on average* the adversary will have fewer successful queries than the honest parties will have convergence opportunities. We want something stronger: except with negligible probability, in all large enough intervals of consecutive slots  $S$ , the adversary will have fewer successful queries than the honest parties will have convergence opportunities, i.e.  $\Pr[\forall S: Z(S) < Y(S)] \geq 1 - \text{negl}(\kappa)$ . 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  $\Pr[Y(S) < Z(S)] = \text{negl}(\kappa)$ , we will use a probability tool called the Chernoff Bound and introduce the concept of *typicality*.

### Chernoff Bound

We will not prove this theorem here, but intuitively, the Chernoff Bound says this: if we flip a fair coin  $n$  times, where heads = 1 and tails = 0, then the more tosses we have, the less likely their sum is going to deviate from the expected value of their sum, which is  $0.5 \times n$ . As the number of trials increases, the probability that the sum is off the expectation by a certain *percentage* of error approaches 0 (this probability is represented by the shaded regions in Figure 12.2). More formally, the Chernoff Bound states:

**Theorem 10.** Consider random variables  $X_i$ ,  $i \in [n]$  s.t.  $X_i \stackrel{i.i.d.}{\sim} \text{Bernoulli}(p)$  and  $X = \sum_{i=1}^n X_i$ . For any  $\epsilon > 0$ ,

$$\Pr[X \leq (1 - \epsilon)\mu] \leq e^{-\Omega(n\epsilon^2\mu)}, \quad (12.1)$$

$$\Pr[X \geq (1 + \epsilon)\mu] \leq e^{-\Omega(n\epsilon^2\mu)}. \quad (12.2)$$

For more details about the Chernoff bound and its proof, a good reference is [?]. (This  $\mu$  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 22** ( $(\epsilon, \lambda)$ -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 11.**  $(\epsilon, \lambda)$ -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  $\epsilon$  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.  $\square$

Note that we need the set  $S$  to be of size at least  $\lambda$  to allow the variables to be concentrated within a Chernoff error  $\epsilon$ , because Chernoff bound requires a large enough number of trials to be invoked.

For reasons that we will see soon, we will choose  $\epsilon$  and  $f$  so that  $3f + 3\epsilon \leq \delta$ . This is called the *balancing equation*. Here,  $\delta$  is the honest advantage, i.e.,  $t < (1 - \delta)(n - t)$ .

Now remember our goal is to prove that the expectations of  $X$ ,  $Y$ , and  $Z$  must be sufficiently separated so that  $Y(S) > Z(S)$  except with negligible probability, i.e. we want to show that the lower bound of  $Y(S)$  is larger than the upper bound of  $Z(S)$ . Typicality gives us bounds on  $X(S), Y(S), Z(S)$  but they are with respect to  $\mathbb{E}[X(S)], \mathbb{E}[Y(S)], \mathbb{E}[Z(S)]$ , so we need to find bounds for these expectations respectively. Since the random variables  $X_r$  are independent and identically distributed for different  $r$ ,  $\mathbb{E}[X(S)] = |S|\mathbb{E}[X_r]$ . Similarly,  $\mathbb{E}[Y(S)] = |S|\mathbb{E}[Y_r]$  and  $\mathbb{E}[Z(S)] = |S|\mathbb{E}[Z_r]$ . So, we only need to compare  $\mathbb{E}[X_r], \mathbb{E}[Y_r]$  and  $\mathbb{E}[Z_r]$ .

What do we know about  $\mathbb{E}[X_r]$ ?

$$\begin{aligned}\mathbb{E}[X_r] &= f = \Pr[\text{at least one successful honest query in round } r] \\ &= 1 - \Pr[\text{no honest query succeeds in round } r] \\ &= 1 - (1 - p)^{q(n-t)} \\ &< pq(n - t).\end{aligned}\tag{12.3}$$

The last step above comes from Bernoulli's inequality, i.e.,  $(1 + x)^a > 1 + ax, \forall x, a \in \mathbb{R}, x > -1, x \neq 0, a > 1$ . We can also show that

$$\frac{f}{1 - f} = \frac{1 - (1 - p)^{q(n-t)}}{(1 - p)^{q(n-t)}} = (1 - p)^{-q(n-t)} - 1 > (1 + p)^{q(n-t)} - 1 > pq(n - t).\tag{12.4}$$

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

$$(1 - f)pq(n - t) < \mathbb{E}[X_r] < pq(n - t).\tag{12.5}$$

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

$$\mathbb{E}[Y_r] \geq q(n - t)p(1 - p)^{q(n-t)-1} > pq(n - t)[1 - pq(n - t)] \geq f(1 - f).\tag{12.6}$$

The first inequality is obtained by assuming that all honest parties make all  $q$  queries even after a successful one and summing over all queries the probability  $p(1 - p)^{q(n-t)-1}$  that it is the only successful one. The second inequality uses  $(1 - p)^{q(n-t)-1} > (1 - p)^{q(n-t)}$  and uses Bernoulli's inequality again. The third inequality holds because  $f(1 - f)$  is an increasing function for  $f \in (0, \frac{1}{2})$ .

Since the adversary is allowed at most  $qt$  queries in each round and each query is successful with probability  $p$ , we also have

$$\mathbb{E}[Z_r] = pqt.\tag{12.7}$$

Even though we want the expectations of  $X(S), Y(S)$ , and  $Z(S)$  to be sufficiently separated so that  $Y(S) > Z(S)$  except with negligible probability, we cannot separate them arbitrarily far. This is because the distance between  $\mathbb{E}[Z]$  and  $\mathbb{E}[X]$

is determined by the advantage held by the majority in the honest majority assumption, i.e.,  $t \leq (1 - \delta)(n - t)$  where  $3f + 3\epsilon \leq \delta$ . To see this,

$$\mathbb{E}[Z_r] = pqt = \frac{t}{n-t} \cdot pq(n-t) < \frac{t}{n-t} \cdot \frac{f}{1-f} < \left(1 + \frac{\delta}{2}\right) \cdot f \cdot \frac{t}{n-t}. \quad (12.8)$$

Here, we have used the inequality  $\frac{f}{1-f} > pq(n-t)$  proved in (??), and another inequality  $\frac{1}{1-f} < 1 + \frac{\delta}{2}$ . To prove the second inequality, we know that  $f < \frac{\delta}{3}$  because  $3f + 3\epsilon \leq \delta$  and  $\epsilon > 0$ . So, we need to show that  $\frac{1}{1-\delta/3} < 1 + \frac{\delta}{2}$  which can be verified as follows:

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

which is true because  $0 < \delta < 1$ .

The honest advantage  $\delta$  (which is the distance between  $|S|pqt$  and  $|S|pq(n-t)$ , scaled by  $|S|pq(n-t)$ ) is composed of the distance between  $\mathbb{E}[Z(S)] = |S|pqt$  and  $\mathbb{E}[Y(S)]$ , distance between  $\mathbb{E}[Y(S)]$  and  $\mathbb{E}[X(S)]$ , and distance between  $\mathbb{E}[X(S)]$  and  $|S|pq(n-t)$ . We allocate this total possible distance by following balancing equation:  $3\epsilon + 3f \leq \delta$ . Let's break this inequality down for an intuitive understanding along with Figure 12.2.

- $3\epsilon$  is the relative distance we allocate between  $\mathbb{E}[Z(S)]$  and  $\mathbb{E}[Y(S)]$ , where  $\epsilon$  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  $\epsilon$  to the right of  $\mathbb{E}[Z(S)]$ ,  $\epsilon$  to the left of  $\mathbb{E}[Y(S)]$ , and another  $\epsilon$  in between as a buffer, which gives us  $3\epsilon$ .
- 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  $\lambda$  and  $\epsilon$ . The smaller we require our Chernoff error  $\epsilon$  to be, the longer time  $\lambda$  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  $\lambda f$ ). Recall from the previous lectures that we denoted by  $\kappa$  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  $\epsilon$  and  $\lambda$ . Therefore, given a particular  $\kappa$  (for example  $\kappa = 256$ ) and particular values for  $\epsilon$  and  $f$ , we need to set  $\lambda$  such that  $\kappa \approx \epsilon^2 \lambda f$ . In a nutshell, the larger we make  $\epsilon$ , the less time  $\lambda$  we will need to wait for confirmation.

So, we want both  $\epsilon$  to be large (to achieve fast confirmation and a small  $\lambda$ ), 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  $\delta$  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  $\delta$ , we want to have the fastest possible blockchain, yet maintain security. Splitting equally between  $\epsilon$  and  $f$ , we can choose  $\epsilon = f = \delta/6$ . The takehome lesson is that, to withstand a powerful adversary (small  $\delta$ ), we need to wait a sufficient amount of time for transactions to be confirmed. You cannot have both quick confirmation and good security!

In the next chapter, we will use the tools developed today to show that in typical executions,  $Y(S) > Z(S)$  for all intervals of slots  $S$  with  $|S| \geq \lambda$ . This will help us to prove that Common Prefix and Chain Growth hold in typical executions.

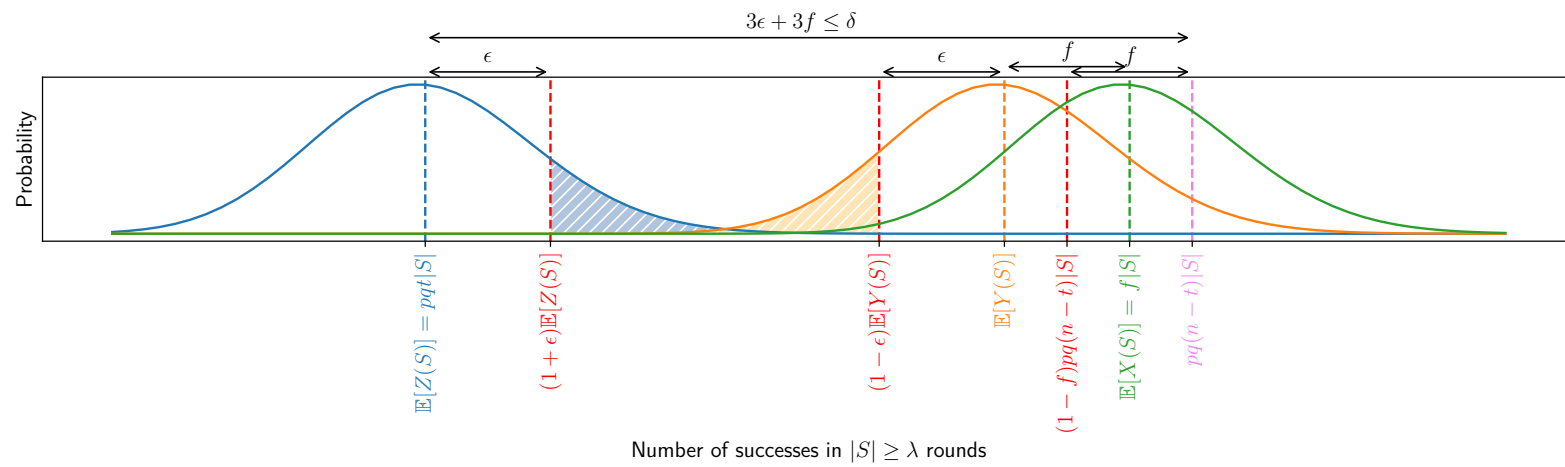


Figure 12.2: The distribution of the random variables  $X$ ,  $Y$ , and  $Z$  in the proof-of-work longest chain protocol.

## Chapter 13

# Security in Earnest III DRAFT

### 13.1 Recap of last lecture: Security in Earnest (II)

We proved several bounds in the previous lecture:

$$(1 - f)pq(n - t) < f = \mathbb{E}[X_r] = 1 - (1 - p)^{q(n-t)} < pq(n - t) \quad (13.1)$$

$$\mathbb{E}[Y_r] \geq q(n - t)p(1 - p)^{q(n-t-1)} > pq(n - t)(1 - pq(n - t)) \geq f(1 - f) > \left(1 - \frac{\delta}{3}\right) f \quad (13.2)$$

$$(1 - \epsilon) \mathbb{E}[X(S)] < X(S) < (1 + \epsilon) \mathbb{E}[X(S)] \quad (13.3)$$

$$(1 - \epsilon) \mathbb{E}[Y(S)] < Y(S) \quad (13.4)$$

$$Z(S) < (1 + \epsilon) \mathbb{E}[Z(S)] \quad (13.5)$$

First,  $X_r$  indicates whether or not round  $r$  was successful, i.e. had at least one successful honest query. Bound 13.1 shows a lower and upper bound for  $\mathbb{E}[X_r]$ . Second,  $Y_r$  indicates whether or not round  $r$  was a convergence opportunity. Bound 13.2 gives us a lower bound on  $\mathbb{E}[Y_r]$ . Bounds 13.3, 13.4, and 13.5 are Chernoff bounds on  $X(S)$ ,  $Y(S)$ , and  $Z(S)$ , respectively where  $S$  is a set of consecutive rounds.

Today, we will be covering Security in Earnest (III). It is highly recommended to read the Bitcoin Backbone paper.

### 13.2 Typicality implies $Z(S) < Y(S)$

We want to prove that in a typical execution, we have the property that  $Z(S) < Y(S)$ . This will help us prove that the Common Prefix property holds.

#### 13.2.1 Derivation of Upper Bound of $\mathbb{E}[Z(S)]$

First, we want to derive an upper bound of  $\mathbb{E}[Z(S)]$ , which will help us achieve our result.  $Z_r$  counts the number of successful queries for the adversary in round  $r$ . It is a sum of  $qt$  repeated independent Bernoulli trials, each with a  $p$  chance of succeeding, so  $\mathbb{E}[Z_r] = pqt$ . We can rewrite  $pqt$  as  $\frac{t}{n-t}pq(n-t)$ . Now, if we apply bound 13.1 we derived from last class and the fact that  $\frac{1}{1-f} < 1 + \delta/2$  with  $f = \delta/6$

we have that

$$\mathbb{E}[Z_r] = \frac{t}{n-t} pq(n-t) \quad (13.6)$$

$$< \frac{t}{n-t} \cdot \frac{f}{1-f} \quad (13.7)$$

$$< \left(1 + \frac{\delta}{2}\right) f \frac{t}{n-t}. \quad (13.8)$$

### 13.2.2 Combining it to get $Z(S) < Y(S)$

We can use bound 13.2, which gives a lower bound on  $\mathbb{E}[Y_r]$ . We have that:

$$Y(S) = (1 - \epsilon) \mathbb{E}[Y(S)] \quad (13.9)$$

$$= (1 - \epsilon) \mathbb{E}[Y_r] |S| \quad (13.10)$$

$$> (1 - \epsilon) f(1 - f) |S| \quad (13.11)$$

$$> \left(1 - \frac{\delta}{3}\right) f |S| \quad (13.12)$$

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

$$(1 - \epsilon)(1 - f) > 1 - \frac{\delta}{3} \quad (13.13)$$

$$\Leftrightarrow \left(1 - \frac{\delta}{6}\right) \left(1 - \frac{\delta}{6}\right) > 1 - \frac{\delta}{3} \quad (13.14)$$

$$\Leftrightarrow \frac{2\delta}{6} + \frac{\delta^2}{36} > -\frac{\delta}{3} \quad (13.15)$$

$$\Leftrightarrow \frac{\delta^2}{36} > 0 \quad (13.16)$$

$$\Leftrightarrow \delta > 0. \quad (13.17)$$

From the Chernoff bound 13.5 and upper bound 13.8, we also have that:

$$Z(S) < (1 + \epsilon) \mathbb{E}[Z(S)] \quad (13.18)$$

$$= (1 + \epsilon) \mathbb{E}[Z_r] |S| \quad (13.19)$$

$$< (1 + \epsilon) \frac{t}{n-t} \cdot \frac{f}{1-f} |S| \quad (13.20)$$

$$< \frac{t}{n-t} \cdot \frac{f}{1-f} |S| + \epsilon \frac{t}{n-t} \cdot \frac{1}{1-f} f |S| \quad (13.21)$$

$$< \frac{t}{n-t} \cdot \frac{f}{1-f} |S| + \epsilon f |S| \quad (13.22)$$

$$\leq \left(1 - \frac{2\delta}{3}\right) f |S|. \quad (13.23)$$

To prove inequality 13.22, note that from the balancing equation we have  $f \leq \frac{\delta}{3}$ .



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

$$\frac{t}{n-t} \cdot \frac{1}{1-f} < 1 \quad (13.24)$$

$$\Leftrightarrow \frac{1-\delta}{1-f} < 1 \quad (13.25)$$

$$\Leftrightarrow 1-\delta < 1 - \frac{\delta}{3}. \quad (13.26)$$

To prove inequality 13.23, we again use our choice of values  $f = \epsilon = \frac{\delta}{6}$ . Then,

$$\frac{t}{n-t} \cdot \frac{1}{1-f} + \epsilon < 1 - \frac{2\delta}{3} \quad (13.27)$$

$$\Leftrightarrow \frac{1-\delta}{1-f} + \epsilon < 1 - \frac{2\delta}{3} \quad (13.28)$$

$$\Leftrightarrow \frac{1-\delta}{1-\delta/6} + \frac{\delta}{6} < 1 - \frac{2\delta}{3} \quad (13.29)$$

$$\Leftrightarrow 1 - \delta + \frac{\delta}{6} - \frac{\delta^2}{36} < 1 - \frac{\delta}{6} - \frac{2\delta}{3} + \frac{2\delta^2}{18} \quad (13.30)$$

$$\Leftrightarrow -\frac{5\delta}{6} - \frac{\delta^2}{36} < \frac{5\delta}{6} + \frac{\delta^2}{9} \quad (13.31)$$

$$\Leftrightarrow -\frac{\delta^2}{36} < \frac{\delta^2}{9} \quad (13.32)$$

Combining results 13.23 and 13.12 together gives us:

$$Z(S) < \left(1 - \frac{2\delta}{3}\right) f|S| < \left(1 - \frac{\delta}{3}\right) f|S| < Y(S). \quad (13.33)$$

### 13.3 Proof of Chain Growth

Recall chain growth lemma from the previous lecture.

**Lemma 12** (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 13** (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  $\tau = (1 - \epsilon)f$ .  $\square$

### 13.4 Proof of Common Prefix

In order for the common prefix property to be violated, the adversary must have had a separate successful query for every convergence opportunity (although, not

necessarily during the same round). Any convergence opportunity without a matching adversarial success would lead to convergence among the honest parties. In a nutshell, it must hold that  $Z(S) > Y(S)$  over  $S$  where  $|S| > \lambda$ , in order for no convergence to happen. Our plan for this proof is as follows. We will show this by contradiction. Suppose that  $\text{CP}(k)$  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  $\lambda$  to be at least  $2f$ .

**Lemma 14** (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

$$X(S) + Z(S) < (1 + \epsilon) \mathbb{E}[X(S)] + \left(1 - \frac{2\delta}{3}\right) f|S| \quad (13.34)$$

$$= (1 + \epsilon)f|S| + \left(1 - \frac{2\delta}{3}\right) f|S| \quad (13.35)$$

$$= \left(2 + \epsilon - \frac{2\delta}{3}\right) f|S| \quad (13.36)$$

$$\leq (2 - 2f)f|S| \quad (13.37)$$

$$\leq (2 - 2f)f \cdot \left(\frac{k}{2f} + 2\right) \quad (13.38)$$

$$= (1 - f)(k + 4f) \quad (13.39)$$

$$< k \quad (13.40)$$

The last inequality holds for  $k \geq 4$  (this follows from  $\lambda \geq 2/f$ ). To prove inequal-

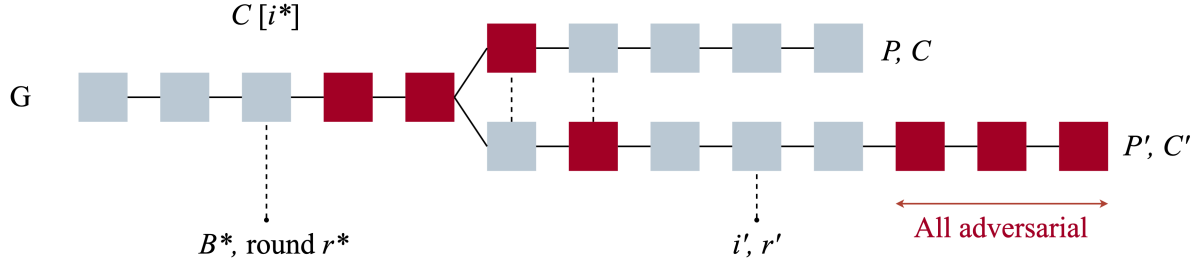


Figure 13.1: A common prefix violation. Honestly computed blocks are shown in gray, while adversarially computed blocks are shown in red. At the end of round  $r$ , party  $P$  has adopted chain  $C$  and party  $P'$  has adopted chain  $C'$ .  $B^*$  is the most recent honestly produced block in the common prefix of  $C, C'$ . This is genesis if all other blocks in the common prefix are adversarial.  $r'$  is the round with the last convergence opportunity. By the Pairing Lemma, convergence opportunities in rounds after the fork must be matched with a successful adversarial query. For example,  $C[i' - 3]$  is an adversarially produced block is matched with  $C'[i' - 3]$ , and similarly for  $C'[i' - 2], C[i' - 2]$ . It is also possible to have multiple honestly produced blocks in a round after the fork without any convergence opportunities, as  $C[i' - 1], C'[i' - 1]$  show.

ity 13.37, we use the balancing equation  $3f + 3\epsilon < \delta$  as follows:

$$2 + \epsilon - \frac{2\delta}{3} \leq 2 - 2f \quad (13.41)$$

$$\Leftrightarrow 3\epsilon - 2\delta \leq -6f \quad (13.42)$$

$$\Leftrightarrow 3f + \frac{3}{2}\epsilon \leq \delta \quad (13.43)$$

$$\Leftrightarrow 3f + \frac{3}{2}\epsilon \leq 3f + 3\epsilon \quad (13.44)$$

Since we wanted at least  $k$  blocks, inequality 13.40 is a contradiction.  $\square$

Recall also the Pairing Lemma from the last lecture.

**Lemma 15** (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 16** (Common Prefix). *A typical execution satisfies Common Prefix with  $k = 2\lambda f$ .*

*Proof.* Assume towards contradiction that there is a  $CP(k)$  violation, illustrated in Figure 13.1. We have two forks  $C, C'$  where if we remove the last  $k$  blocks from each of them, they are not the same chain. Let  $S = \{r^*, \dots, r\}$ , 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.  $\square$

## 13.5 Note about Tradeoffs with $\epsilon$ and $f$

Let us discuss the relationship between a concrete  $\epsilon$  and  $\lambda$  obtained by the Chernoff bound. Larger  $\epsilon$  allows for smaller  $\lambda$ . 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  $\kappa$  (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  $\lambda$  to be large enough to account for the small  $\epsilon^2$ .

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  $\epsilon$  and  $f$  large, because we like to have quick confirmation (small  $\lambda$ ) and fast chain growth (large  $f$ ). However, we cannot make both of them large, as we are bounded by our honest advantage  $\delta$ .

The verdict is that, in a setting with a powerful adversary (small  $\delta$ ), we must have a slow chain, either producing blocks slowly, or with the requirement to wait many blocks for confirmation, in order to ensure security. A fast chain with fast confirmation won't cut it.

# Bibliography

- [1] Developer guide - bitcoin. Available at: <https://bitcoin.org/en/developer-guide>.
- [2] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. "O'Reilly Media, Inc.", 2014.
- [3] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.
- [4] D. Boneh and V. Shoup. A graduate course in applied cryptography. *Draft 0.5*, 2020.
- [5] J. R. Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.
- [6] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.
- [7] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In E. Oswald and M. Fischlin, editors, *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9057 of *LNCS*, pages 281–310. Springer, Apr 2015.
- [8] O. Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge university press, 2007.
- [9] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge university press, 2009.
- [10] D. Graeber. *Debt: The First 5,000 Years*. Melville House, 2014.
- [11] G. Ingham. Money is a social relation. In S. Fleetwood, editor, *Critical realism in economics: Development and debate*, pages 104–105. Routledge, 1998.
- [12] W. S. Jevons. *Money and the Mechanism of Exchange*. H.S. King & Co., 1875.
- [13] Y. Lindell and J. Katz. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2014.
- [14] C. L. Liu. *Elements of Discrete Mathematics*. McGraw-Hill, 2 edition, 1985.
- [15] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Available at: <https://bitcoin.org/bitcoin.pdf>, 2008.

- [16] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004.
- [17] S. M. Ross. *A First Course in Probability*. Pearson Boston, MA, 10 edition, 2019.
- [18] G. Simmel. *The Philosophy of Money*. 1900.
- [19] M. Sipser et al. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.
- [20] N. Smart. *Cryptography Made Simple*. Springer, 2016.
- [21] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

# Index

- 2nd Preimage Resistance, 27
- Address, 37
- Adversary, 10
- Amplification, 15
- Arrow of Time, 66
- Bad Event, 12
- Block, 57
- Block Reward, 70
- Blockchain, 59
- Blocktree, 65
- Bootstrapping, 20
- Canonical Chain, 66
- Change, 38
- Coin, 36
- Coinbase Maturation, 71
- Collision Resistance, 24
- Confirmation, 58
- Content Addressable, 24
- Convergence Opportunity, 68
- Cypherpunks, 8
- Difficulty, 56
- Discrete Logarithm Problem, 33
- Double Spend, 49
- ECDSA, 33
- ed25519, 33
- Elliptic Curve, 33
- Environment, 109
- Existential Unforgeability, 33
- Fees, 70
- Full node, 52
- Genesis, 60
- Genesis State, 72
- Gossip, 18
- Hash Function, 23
- Hash function, 23
- Honest Majority Assumption, 69
- Honest parties, 10
- Identity, 30
- Index, 134
- Kerckhoff's Principle, 10
- Ledger, 41
- Longest Chain Rule, 66
- Mempool, 57
- Mempool State, 73
- Mining, 56
- Negligible function, 15
- Network Delay, 48
- Nonce, 56
- Outpoint, 40
- Party, 64
- Peer Discovery, 20
- Pigeonhole Principle, 24
- PPT, 11
- Preimage Resistance, 26
- Premining, 60
- previd, 59
- Private Key, 30
- Proof-of-Work, 55
- Pseudonymity, 30
- Public key, 30
- Random Oracle, 108
- Reorg, 67
- Rushing Adversary, 110
- secp256k1, 33
- Secret Key, 30

- Security, 54
- Security parameter, 12
- Signature, 30
- Successful query, 63
- Sybil Attack, 20
- Synchrony Assumption, 109
  
- Target, 55
- Temporary Fork, 68
- Transaction, 36
- Transaction graph, 40
- Transaction input, 36
- Transaction output, 36
- Trusted Third Party, 8
- txid, 39
  
- UTXO, 37
- UTXO set, 40



# List of Symbols

$t$	adversarial parties 64
$\mathcal{A}$	adversary 12
$q$	compute 64
$\Delta$	network delay 48
$\mathcal{Z}$	environment 109
$\mathcal{G}$	genesis 60
$H$	hash function 23
$\Pi$	honest protocol 12
$\text{ctr}$	nonce 56
$pk$	public key 31
$sk$	secret key 31
$\kappa$	security parameter 12
$\text{Sig}$	signature function 31
$\sigma$	signature 31
$p$	successful query probability 63
$T$	target 55
$n$	total parties 64
$\text{Ver}$	signature verification 31