Formally verifying the security properties of a proof-of-stake blockchain protocol

Kawin Worrasangasilpa

UNIVERSITY OF
CAMBRIDGE

University of Cambridge
Computer Laboratory
Wolfson College

This thesis is submitted for the degree of
Doctor of Philosophy

April, 2021

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Formally verifying the security properties of a proof-of-stake blockchain protocol

Kawin Worrasangasilpa

Abstract

In 2009, Bitcoin was brought into existence, as the first real-world application of blockchain protocols, by its pseudonymous and mysterious creator, Satoshi Nakamoto. It was presented as a form of cryptocurrency, has become widely known and been recognised as the first successful E-cash since the introduction of the E-cash idea in 1983. Many more crytocurrencies including Ethereum and Tether have emerged following the success of Bitcoin.

Bitcoin is secure, meaning that it satisfies persistence and liveness. Without the need of a trusted third party, it appears to prevent double-spending attacks. Bitcoin's security is obtained by the use of cryptographically secure chains of blocks for time-stamping (hence the name 'blockchain') and a technique, often called Nakamoto consensus, combined from the longest-chain rule and Proof-of-Work (PoW). Briefly, it allows and encourages all parties to participate in picking the longest chain in the system and solving a cryptographically difficult puzzle to declare the next block of transactions for that chain. PoW is sometimes referred to as a lottery system.

PoW requires a majority of the computational power to be honest and it consumes a gigantic amount of energy, so it is not scalable. In the light of this problem, Proof-of-Stake (PoS) was suggested to replace PoW. PoS-based blockchain protocols, instead of using computational power, use in-system currency to agree on a new block to be added, but keep mostly everything else the same. Kiayias et al. was the first to propose a provably secure PoS-based blockchain protocol: Ouroboros. Ouroboros' security guarantees, persistence and liveness, can be verified through proving that Ouroboros satisfies three elementary properties for blockchains as proposed by Gary et al.: Common Prefix, Chain Growth, and Chain Quality.

In this project, we attempt to formalise, in Isabelle/HOL, the combinatorial analysis used to prove that Ouroboros satisfies common prefix with near certainty. We cover the case of a static stake protocol under a few assumptions: the network is synchronous, the majority of the stake is honest, and the stake transfer between executions does not effect the lottery system.

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Professor Lawrence Paulson. For almost a decade, he has been patiently and passionately guiding me towards the field I did not have expertise in. Without his help, laying down the ground work for this project would be tremendously more difficult than it was. Above all his academic excellence, he is also supportive and understanding. Instead of pushing for the highest productivity, he always asked me to take care of myself physically and mentally. Every time I felt stuck, or felt like I did not progress enough or was good enough to succeed in my research, he always had a way to reassure and prevent me from being too difficult on myself. Thank you for your kindness and for having faith in me until the end.

This project would not have been possible without the funding from IOHK. I am wholeheartedly grateful for all people at IOHK that made this happen, especially Charles Hoskinson, who has always believed in me and valued my work.

Many people have helped me gain technical skills necessary to finishing this project. To name some, I would like to thank Wenda Li for helping with basics in using Isabelle in the first year, Manuel Eberl for suggesting your then unpublished Isabelle library for my work, Andreas Lochbihler and Sébastien Gouëzel for helping with formalising conditional expectations and super-martingales in Isabelle, Lars Brünjes for being the first person to explain me proof-of-stake protocols elaborately, Aggelos Kiayias and Bernardo David for answering all of my questions towards the Ouroboros project, Alexander Russell for discussing with me your complicated and exhaustive proofs, Tanut (Nash) Treetanthiploet and Leran Cai for shaking the rust off of my probability theory knowledge, and for many more I cannot possibly list them all here.

Life in Cambridge would have been quite lonely if not for other Thai friends who accompanied me in the first few years I was here. Studying very far from home is tough, and you guys shared similar ambitions to mine. Thank you Baitong, Pavi, Nash, Nae, Rainbow, Pond, Numtip, Mild, Proud, Ta, and many others for making it enjoyable, and I hoped we reunite soon.

For countless number of times, dancing tango has helped me de-stress while I was tired and exhausted from PhD work, but not only that, it also found me a welcoming community and friends. Thank you tango friends, both in Cambridge and afar, for all warm hugs I have been given.

A supportive family is such a privilege I have been thankful for my entire life.

Dear Mom and Dad, above all financial supports you two have provided, I thank you for everything I have learned from you, for time you two have devoted raising me and my brothers, for always bringing me back up when I fell, and for being a thousand times happier than me when I succeeded.

Lastly, I would like to thank my wife, Yokk, for endless emotional supports and always having my back, for taking care of me when I was sick, for giving me your honest thoughts compassionately, and for being my best friend.

Contents

List of Figures

Chapter 1

Introduction

The use of a cryptographically secure chain of blocks was introduced in 1991 by Haber and Stornetta [1]. The main idea they proposed is to time-stamp digital information in a way that a user maintains their privacy from untrustworthy service. At that time, it was not suggested that the protocol is to be used for any specific kind of documents.

In 2009, the first real-world application of this protocol, Bitcoin, came into existence after its pseudonymous and mysterious creator, Satoshi Nakamoto, presented it in the form of cryptocurrency [2]. Without the need of a trusted third party but claimed to prevent double-spending attacks, Bitcoin has been widely known and can be recognised as the first successful E-cash since the introduction of the E-cash idea in 1983 [3]. The protocol is called the blockchain protocol.

Many more crytocurrencies have emerged following the birth of Bitcoin. These include Ethereum [4] and Ripple [5]. Furthermore, blockchain protocols have been applied to many different applications such as smart contracts[1], supply chains [7], and digital voting, due to their security for decentralised systems.

In order for the protocol to be secure —in this case, secure from double-spending— it requires enough people participating in a certain security procedure in an honest

[1]The idea firstly introduced by Szabo in 1990s [6], but we will leave details in this chapter and discuss again in chapter 6

way. We will later on refer to this group of people as honest participants. We need to note that it is not required for all Bitcoin users to participate in securing the protocol, so when we later on refer to users, we do not assume their participation in the security procedure. Therefore, there must be incentives for people to participate in the protocol in a way that acting honestly is the most reasonable act.

In Bitcoin and many other cryptocurrencies, people are encouraged to spend their computational power to provide a proof of work. More specifically, if one party spends more computational power than others do, they are more likely to get rewarded. The reward is in the form of in-system currency. This concept is usually compared to gold mining in a way that people spending their computational power are compared to gold "miners" and having a chance to get in-system currency is compared to having a chance to find gold from mining. Therefore, we refer to people who spend computational power trying to provide proof of work as "miners" and the process as "mining". This process is called the proof-of-work (PoW) protocol, and the blockchains that use it are known as PoW-based blockchain protocols.

However, PoW-based protocols have some inevitable drawbacks due to their inefficiency. Because of the steep growth of Bitcoin and other similar PoW-based cryptocurrencies, more and more miners are trying to mine for coins using their computational power, resulting in an enormous amount of wasted energy. The electrical power used for Bitcoin was estimated to be equal to the energy consumption of one small country, and it has even been called an "environmental disaster" [8].

In the light of this problem, there are ideas of replacing the energy used with other resources. Proof-of-space, for example, is where each participant is forced to store files and requested to ensure they really store them [9, 10]; there is a blockchain protocol based on this approach called Spacemint [11]. The proof-of-activity [12] is another method which employs the idea of PoW enhanced with the use of the stake—a certain amount of in-system currencies—held by participants called stakeholders. Nevertheless, these mechanisms still need electrical power to process the proof. Lastly and of specific concern to this project, the pure proof-of-

stake (PoS) is where participants who hold higher stakes have a higher probability to get rewarded [13]. We also call participants in these protocols stakeholders. Examples of the PoS-based blockchain protocols are Ouroboros [14], Algorand [15], and SnowWhite [16].

This project aims to formalise Ouroboros[2] security guarantees through Isabelle, a generic theorem prover. Ouroboros is claimed by Kiayias et al. to be the first provably secure PoS blockchain protocol. Particularly, we focus on working towards formalising the claims that Ouroboros is a stable and robust transaction ledger. We also discuss how the results are related or of any use to other similar PoS-based protocols. The project is funded by IOHK, the company who invented Ouroboros; nevertheless, using a theorem prover is an unbiased way to verify its statements.

1.1 Blockchain

Blockchain was originally invented to solve problems of digital time-stamping of documents. With an important document, for example, patents, it is important to be able to certify when it was created or modified and especially to be confident that this timestamp cannot be easily altered.

One can think of a naive way to solve this problem which is a "digital safety-deposit box". The "digital safety-deposit box" works simply as follows: a client asks a time-stamping service (TSS) to time-stamp their document, the TSS then makes a copy of it with the time it was submitted, the TSS returns the original copy to the client, and when getting challenged to check the time of the document, the TSS uses their copy to compare and validate if the document is changed from the time the TSS got it.

However, many concerns arise from the "digital safety-deposit box" as follows: 1) privacy because other parties who have access to the TSS can look at the copy the TSS retains; 2) bandwidth and storage because the longer the document, the more

[2] As there are many versions of Ouroboros developed in the past 4 years, the original version of Ouroboros is often referred to as Ouroboros classic, which we sometimes use in this thesis when we compare it to other versions.

space taken to store a copy; 3) incompetence because data or time-stamp can be corrupted, lost while transferring from the client to the TSS, or the TSS can fail to keep the data; and 4) trust, the most complicated of all four, because the TSS might be untrustworthy.

Stuart Haber and W. Scott Stornetta proposed a way to solve these four concerns, which developed into the blockchain. In brief, they used a cryptographically secure collision-free hash (a function h such that is computationally difficult to find a distinguish pair of x and x' that $h(x) = h(x')$) to encode a whole document into a fixed-length data and send to the TSS a hash value of the said document instead. This solves already the first two issues: privacy and storage.

The incompetence of the TSS can then be solved using a signature scheme. This scheme is for the TSS to build a package by signing a pair of the hash valued received from the client and the receiving time, so the client and others knows the TSS has really processed their request before sending it back to the client. This package can be used as a time-stamp. Regarding the concern about trust, the TSS, instead of only signing the hash data of the document with the time it was received, packs this information with the identification number of the client and the linking information from the previous document. The TSS then sends out, as a time-stamp, this signed certificate back to the client with the next client identification number when the next request has been processed. This method performs a chain of blocks (of hashed information and time-stamp), each of which can be queried by any documents' owners who use the service from any blocks in the chain because each block has linking information for the previous and the next block.

1.1.1 Proof-of-Work blockchain protocols

Proof-of-work was not specifically developed for the blockchain protocol. It is a method used to prevent spamming in a broader sense. Invented in 1993 by Dwork and Naor[17], the concept of proof-of-work is to have people solve problems requiring the expenditure of enormous computational power to solve but proving trivial to verify the answer.

However, from the proof-of-work presented in hashcash[3], Satoshi could give life to Bitcoin by adapting its concept and combining it with the blockchain structure. In this subsection, we will outline the mechanism of Bitcoin, the first PoW-based blockchain protocol, as an example of how blockchain protocols use proof-of-work to maintain their security. For Bitcoin, documents that need time-stamping are transactions of in-system "coins". Bitcoin is a peer-to-peer protocol that eliminates the need for a trusted third-party by letting all users in the system have their views of transactions in the form of blockchains. The important point that needs to be mentioned is each user has their own chains which can be different from others', but eventually the system will work to mitigate those differences to the minimum. The brief steps of the Bitcoin protocol are outlined below.

First, when new transactions from any parties broadcasting them are added collectively enough to a certain amount, they will be grouped into one block; this step is performed individually by each participant, so blocks containing the same group of transactions can be in different orders, or transactions in the block formed around the same time can have different transactions as there is a transmission delay.

Then, each participant will try to provide its proof of work, a number that has its hash value starting with the required number of zeros. Next, the ones who have finished their proof of work will extend their local views of chains by a block containing the mentioned collected transactions, a link to the previous block, and their proof of work. They will then broadcast the new chains to all other parties. Third, all parties will select the longest valid chain to be their new local chains.

To understand why all these steps help prevent double-spending attacks, as shown in figure 1.1, we need to explain how a double-spending attack is performed in the blockchain protocol. First, an adversary transfers in-system currency to another user; this money transfer then goes into a transaction and is eventually broadcast to all users from the steps above. Next, the receiver sees that its local chain contains the information about this money transfer, and acts accordingly, such as, paying the adversary in another currency, or spending this amount of money they just earned. Lastly, the adversary then tries to remove that transaction from all

[3]http://www.hashcash.org

local chains and creates a new transaction by transferring this amount of money to another user (hence the name double-spending) on a separate chain that is longer than the other chains that all the users have as their local chains.

From the last step described above, adversarial participants aiming to commit double-spending attacks have to work hard, in terms of using more computational power than acting honestly. This work becomes even harder as more blocks get declared extending their transactions' blocks in the same chain.

In Bitcoin, the protocol incentivises users to provide the proof of work by rewarding the fastest users who solve puzzles, so enough honest parties would participate in solving puzzles to extend the blockchain: we call them miners, who are compared to gold miners since they do work to get themselves coins. It was claimed by Satoshi Nakamoto that the protocol stays stable, or cryptographically secure from double-spending attacks, if there is an honest majority [2]; however, there was an analysis of different kinds of attacks apart from double-spending ones, such as a selfish-mining attack [18], showing that a majority alone may not be enough.

1.1.2 Proof-of-stake blockchain protocols

Proof-of-stake, unlike proof-of-work, is a term used specifically for cryptocurrencies as a means to mitigate the main drawback of the proof-of-work method in many cryptocurrencies following the popularity of Bitcoin, that is to resolve the issue of the immense amount of energy spent just to keep these cryptocurrencies functioning securely. Discussed for the first time in 2011, the first cryptocurrency to adopt the proof-of-stake concept was created by King and Nadal in 2012.

The main idea of this model of the blockchain protocol is to change the incentive given in the PoW-based protocol from spending computational power to holding in-system currencies [19]. They mentioned that "Philosophically speaking, money is a form of 'proof-of-work' in the past and thus should be able to substitute proof-of-work all by itself." After that, there have been many different attempts and methods in implementing the PoS-based blockchain protocols [12, 13, 19, 20, 21], which might be classified into two groups as follows.

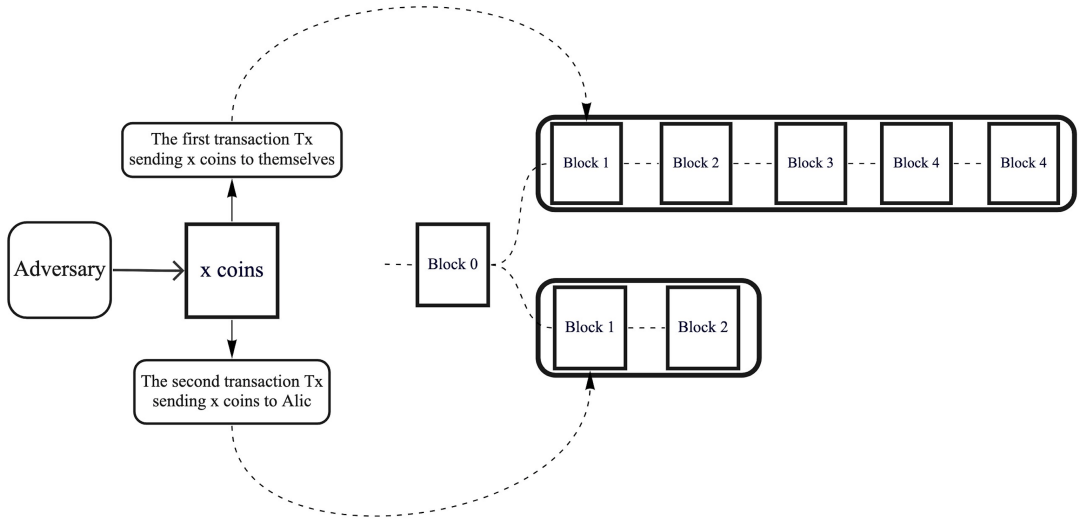


Figure 1.1: When adversary has more hash power than other miners' hash power, they can generate a private chain (the top chain) that is longer than another chain and the network adopts the top chain and abandons the other (the bottom chain).

1.1.2.1 Block generator randomisation

This version of the PoS-based protocol uses stake, users' holdings of in-system currencies, solely with a randomisation process to pick a new block generator. Examples of protocols that fall into this category are Algorand [15], SnowWhite [16], and Ouroboros [14]. For Ouroboros, which is the central theme of our project, the protocol works as follows. The protocol manages in slots, fixed periods of time, rounds, and the fixed number of slots. The smallest in-system coins are assigned with unique numbers. Then in each slot, there is an algorithm randomising a number; assuming this algorithm is not biased, the system then picks an in-system coin when its unique number matches that randomised number. A user who holds that coin is then assigned to be a block declarer. Next, the block declarer acts the same as the user who has finished "proof-of-work" in the PoW-based protocol extending their local chain with the newly declared block and broadcasting this new chain. All users, or ones that receive it, collect this chain and choose the longer one between chains in their views and the newly received one. This method can lead to different views of users only if a malicious user is picked to declare a

block, so they can broadcast a number of variations of chains and hence can affect other users' views.

1.1.2.2 Coin-age based models

Coin-age models also use the amount of stake users hold as well, but they also combine it with the time for which individual users hold that stake. More specifically, for example in Peercoin [19], the protocol is designed to replace Bitcoin's proof-of-work with proof-of-stake. Hash is still used but with a limited search space and not with a fixed answer as in Bitcoin. Instead, the required amount of work that one needs to prove changes proportionally to the coin-age. Therefore, the greater the coin-age one has, the easier they can provide proof-of-work.

1.2 Security properties of the PoS-based blockchain protocols

When we discuss the security of the blockchain protocol, we refer to the specific properties it should have. These properties are persistence and liveness, and they are not only used for the blockchain protocol but more generally for distributed ledgers. Persistence is defined as if there is a user claiming a certain transaction x as stable, all other users will claim the same when asked. Liveness is defined as if an honest user creates a transaction, eventually this transaction will be stable. If a distributed ledger has these two properties, it is said to be robust.

1.2.1 Persistence and liveness of the PoS-based blockchain protocols

From the definitions, persistence and liveness depend on how we define "stability". For any blockchain protocol, a stable transaction means a transaction that is in a block that has a certain number of blocks on top of it in the longest chain. To prove that a blockchain protocol satisfies persistence and liveness, ones can prove that the blockchain protocol satisfies these following properties instead:

- **Common Prefix (CP)** Considering any two local chains possessed by different honest users in different slots, if the chain that is accepted in a more recent slot is cut by a fixed number of blocks from the tail (this number indicates how strong the common prefix property is), the resulting chain will be the prefix of the other chain.
- **Chain Quality (CQ)** Considering any local chain of an honest user's and then any portion of fixed length, there is a fixed upper bound ratio of blocks created by adversarial users.
- **Chain Growth (CG)** Considering any two local chains of different honest users in different slots, the difference in the length of two chains is greater or equal to the product of the number of slots between the two slots and the fixed number.

1.2.2 Reducing common prefix with forkable strings

From 1.2.1, we explain that common prefix is concerned with the views of two honest users in a specific way: cutting a fixed-length chain of blocks from the tail of the more recent chain leaves a prefix of the other chain. To formalise this property, the notion of forkable strings is introduced[4]. To understand what forkable strings are, characteristic strings need to be defined first. The n-length characteristic strings are strings associated with a fixed period of time where there are n slots of time for each block to be created in PoS protocol, so there can be no more than n blocks created in one chain in this time period. Characteristic consist of 0 and 1 to indicate, for each slot of time, if the slot leader[5] is honest, 0, or adversarial, 1, to the blockchain. Forkable strings are, to put it simply, characteristic strings that have enough adversarial slots so that the common prefix property is not met. Forkable strings are useful because we can then reduce the complexity of the problems considering many chains of blocks and find a pair that breaks common prefix property.

[4]Forkable strings will be explained more elaborately in section 2.2
[5]User with granted right to declare a new block

1.3 Objectives

This project aims towards formalising Ouroboros' security guarantees that the protocol satisfies persistence and liveness. We capture the arguments from the most simplified setting of Ouroboros only. More specifically, we are to formally prove that the static stake version of Ouroboros satisfies that the forkable strings rarely happen; then we use this result, as explained in 1.2.2, to formalise the Common Prefix (CP) property.

Despite the fact that there are some probability theory materials in the proofs regarding forkable strings and the Common Prefix (CP) property, the main focus of this project is discrete structure of the proofs which requires more delicacy on special cases. With this reason, we also leave out the formalisation of the Chain Quality (CQ) and Chain Growth (CG) properties as they only use stochastic processes without combinatorial analysis since they could be done in completely different manners. Therefore, only the discrete structure part of forkable strings and Common Prefix is our scope. The emphasis on formalisation of forkable strings and Common Prefix consists of formalising rose trees, forks, their basic properties, and their specific properties that are used in the future proof of the protocol liveness and persistence.

Although we are working on one specific protocol, Ouroboros, we also aim to compare the protocol with other PoS-based protocols and understand how general our formalisation is. For instance, we will analyse which protocols are suitable for the description by the notion of forkable strings. Significantly, we will compare Ouroboros with other versions of Ouroboros, especially ones that use alternative versions of forkable strings; small changes might be applied to our formalisation to facilitate the formalisations of the security statements of other Ouroboros protocols. It is noteworthy that we neither formalise an implementation of the protocol nor the algorithm of the protocol. We only focus on mathematical theorems claiming that the protocol is secure.

We use the Isabelle/HOL theorem prover to formally verify all work. The tool has two main styles of proof, namely **apply**-style proof and structured proof (or

`Isar`). Almost all lemmas are proved in the structured proof style for the sake of readability, but sometimes the **apply**-style is used in trivial or intermediate proofs.

1.4 Achieving our goals

The full formalisation can be separately viewed as four main modules, all of which we need to achieve. These are (i) introduction to forkable strings, (ii) discrete structures proofs of forkable strings, (iii) probabilistic proofs of forkable strings, and (iv) common prefix.

For this chapter, we will skip the detailed explanation of forkable strings. In brief, forkable strings are a specific data structure used for the first time in the Ouroboros paper to help depict the combined view of all chains possessed and adopted by any parties in the system at any point of time. Without storing information related to transactions, forkable strings focus only on the links between blocks in a simpler fashion making the proofs drastically less messy.

(i) introduction to forkable strings This part focuses on formalising basic definitions and lemmas related to forkable strings. The complications for this module involve developing the usability of our formalisation on one data structure (forks). Many hidden lemmas unnecessary and trivial for pen-and-paper proofs are formalised in this module.

(ii) discrete structure proofs of forkable strings Intermediate variables and properties for the forkable strings are formalised in this module. We call them 'intermediate' because these variables (and properties) do not carry an obvious purpose of what they describe but exist to be used in significant results later on.

(iii) probabilistic proofs of forkable strings In this part, we need to use the Isabelle/HOL probability theory to formalise several random variables and a sequence of independent and identically distributed random variables. The random variables we need to deal with are all discrete, meaning that probability mass functions can describe them, so we mainly use theory `probability_mass_function` in Isabelle/HOL to avoid unnecessary complications. The main goal in this module is to extend the formalisation of margin to formalise a desirable upper bound of the probability that a string is forkable.

(iv) common prefix Using the results in the previous module and the formalisation of how forkability can be linked with the common prefix property, we then formalise the common prefix (CP) property for the Ouroboros protocol.

All Isabelle proof scripts in this project are stored at https://bitbucket.org/wkawin/ouroboros with the theory files named `Ouro` and `Ouro2`. We only separate our code into two files due to its overall large size. The results in our code follow the order of the modules in this section. The formalisation works for the latest version of Isabelle – Isabelle2023.

1.5 Organisation of thesis

This thesis starts with the background of the Ouroboros classic protocol in chapter 2 focusing on the model used to describe the protocol, the assumptions for the model, the definitions of concerned components and the properties that the protocol aims to satisfy. There are different stages of the protocol; we choose to focus on only static stake one, while all the other stages are described only briefly. We also present the background information about the notion of forkable strings in chapter 2. Although this notion was first described in the Ouroboros classic paper [14], we discuss it separately as the link between them is only with the leader selection process of the static stake protocol of Ouroboros.

Then, the formalisation tasks are explained in the order of modules listed in section 1.4. Firstly, we describe in brief on how we formalise the basic elements of the forkable strings notion in chapter 3: By referring to basic elements, we mean the ones we already presented in the background of forkable strings in chapter 2, and as it is the first chapter to present formalisation in Isabelle/HOL, we also devote the first section to explaining Isabelle proof constructions and keywords/commands that will be used throughout the project. Then, the details of the formalisation towards the density of forkable strings theorems are described in chapter 4, where we focus on both formalising an intermediate value and margin a probabilistic proofs. Finally, the formalisation of the common prefix property for Ouroboros is outlined in chapter 5: this chapter mainly discusses the formalisation of intermediate value divergence with a description of how it is linked to the common prefix property.

In chapter 6, we review bodies of research on the proof of security guarantees for different kinds of blockchain protocols. The protocols discussed range from proof-of-work protocols (mainly Bitcoin) to proof-of-stake protocols. Regarding the usage of mathematical models in the security proofs, we pay particular atten-

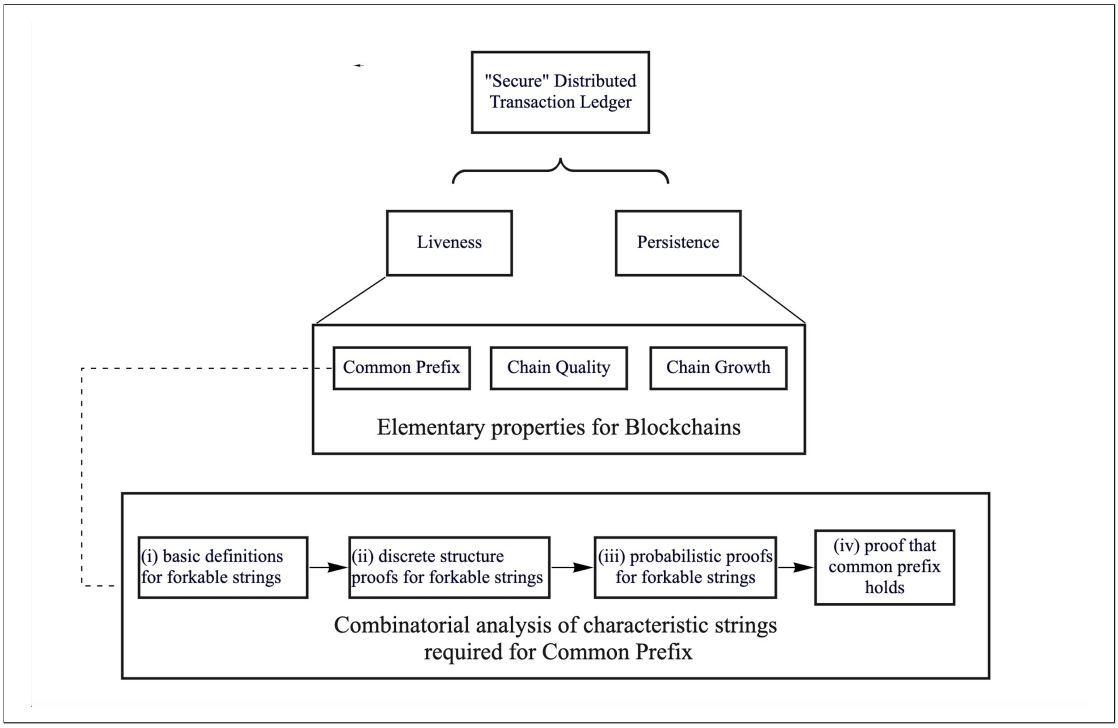


Figure 1.2: Our goals in formalising that the Ouroboros protocol satisfies common prefix through combinatorial analysis of characteristic strings

tion to a variety of forkable string notions which differ from the one we formalise in this project, to identify a potential option to alter our formalisation to be compatible with other lines of work. Also, we include potential more broadened and generalised future work. Chapter 7 concludes both finished tasks and ongoing tasks with the evaluation of the quantity and efficiency of the overall process.

Chapter 2

Background: Ouroboros and forkable strings

This chapter outlines some background body of research about Ouroboros, the protocol we choose to formalise its security guarantees, and forkable strings, the notation which we use to formalise Proof-of-Stake blockchain protocols. It is organised into two sections: the summary of the model of Ouroboros protocol, and the forkable strings. Both sections rely mainly on the Ouroboros paper by Kiayias et al. [14]; however, there are some analyses on theories related to the notion of forkable strings in section 2.2 that refers to results in the work by Russell [22].

2.1 Ouroboros and its security models

Ouroboros is a base protocol for the Cardano blockchain platform, which is claimed to be "the first to be founded on peer-reviewed research and developed through evidence-based methods" [1]. In 2017, the Ouroboros paper was published by Kiayias et al. [14]. There are many versions of the Ouroboros protocol: the main line of work (in order) consists of Ouroboros Praos [23], Ouroboros Genesis [24], and Ouroboros Chronos [25]; the progression in versions is to make more and more realistic protocols working in real-world implementations. The versions with additional features are Ouroboros BFT [26], focusing on a Byzantine fault tolerance consensus, and Ouroboros Crypsinous [27], focusing on preserving privacy of users. [2] The first Ouroboros (or Ouroboros Classic) is claimed in the original paper to

[1] https://cardano.org/ouroboros; Cardano's ADA is one of the leading cryptocurrencies, and it has a market capitalisation around US$ 40 billion (as of April 2021)

[2] Ones can check a brief summary of the development of Ouroboros family in "Ouroboros Tutorial" presentation by Badertscher and Kiayias [28].

be the first proof-of-stake based blockchain protocol with rigorous analysis.

The rigorous analysis of Ouroboros classic is based on the following principles. First, a model is provided to formalise the PoS-based protocol, focusing on persistence and liveness, which represent security properties of the robust transaction ledger [29]. Second, Ouroboros is novel and unique in many ways. For example, it uses the multi-party coin-flipping protocol to establish randomness, and instead of considering the stake of parties in each round to select the leader of the round (or *slot*), it snapshots stakeholders and the stake they hold once every time the fixed number of slots passes; those rounds are together called *epochs*. Third, pen-and-paper proofs were given, showing that the protocol is secure under certain assumptions.

The security guarantees were given in terms of a notion of "forkable strings". Other interesting results include the incentive structure of the protocol, the stake delegation mechanism added to the protocol and certain attack analyses. However, they are not relevant to this project.

It is important to note that Ouroboros only deals with simple security aspects under certain conditions which will be described in this chapter. There are other proof-of-stake based blockchain protocols such as Algorand [15] and SnowWhite [16], which were presented with rigorous analysis as well but with different settings and concerns of security arguments. We will leave them out for now and discuss them later in chapter 6. This chapter will only cover the Ouroboros model and the security guarantees without the proofs, so the contents will feature a summary of "Ouroboros: a provably secure proof-of-stake blockchain protocol" by Kiayias et al. [14] with additional details of relevant algorithms from "The Bitcoin Backbone Protocol: Analysis and Applications" by Garay et al[29].

2.1.1 Model and assumptions

This subsection first summarises a model and assumptions for the Ouroboros protocol before the details of the protocol are described in 2.1.2. It must be acknowledged that the original work of Kiayias et al. [14] omits the details on how they proceeded to implement the protocol to satisfy these assumptions. We cover how timing works in the protocol and functionalities that help reduce complications when describing the steps of the protocol.

2.1.1.1 Synchrony

Time is divided into discrete units called slots. In one slot, at most one ledger block, containing several transactions, is added to the system. Slots are labelled

in the form of sl_r when $r \in \{1, 2, \ldots\}$ and in regard to a publicly-known increasing function of current time. The model is assumed to be synchronous meaning that all parties have access to the same slot at a specific point of time and communication delays are insignificant compared to the length of time a slot lasts. All messages are guaranteed to arrive in the same slot to which they are sent.

2.1.1.2 Security model

The rigorous description of the Ouroboro protocol is provided in the Ouroboros paper, in which there is the ideal functionality F encompassing several functionalities that facilitate the execution of the protocol so that:

1. each user gets their stake assigned at the beginning of the protocol execution
2. signature schemes are handled correctly, so that each user has unique private key and public key[3],
3. messages are sent and received in the same time slot, so there are no network delays in consideration,
4. adversarial nodes are selected,
5. slot leaders are selected according to the leader selection process, 2.6 taking into account how adversarial nodes were selected in (4),
6. the maximum length of time all parties can be offline is controlled, and
7. less than 50% adversarial stake ratios is controlled.

In our project, we assume these functionalities work correctly, as we are only concerned with security under attacks of adversaries while the protocol runs correctly.

2.1.2 Protocol overview

The protocol presents four stages of different adversarial models it can tolerate. Although the four stages are presented below, we focus only on the first and simplest stage setting, the static stake protocol, which can then be used as a basis step for the induction proof for the dynamic stake with the beacon. More specifically, in the Ouroboros paper, detailed proofs of the common prefix, chain quality and chain growth properties (their formal definitions will be given in 2.1.4) of the blockchain protocol were given in the static stake model. This was then

[3]Canetti and Ran [30] employed the paradigm where $\mathcal{F}_{\mathsf{DSIG}}$ was originally proposed and proved that it can be realised by some real signature schemes such as `EUF_CMA`. Therefore, we can regard the use of $\mathcal{F}_{\mathsf{DSIG}}$ as an "idealised" signature scheme and use it in proving the security argument instead of the realistic one.

developed to fulfill the proof that the dynamic stake protocol satisfies persistence and liveness. Therefore, we emphasise the first stage with a brief description of the other three.

All stages of the Ouroboros protocol are parameterised as follows: (i) a security parameter k describing how many blocks on top of each transaction are required to be stable, (ii) ϵ is the difference between the proportion of honest stake and that of adversarial stake over the total stake and (iii) D, L and R, all measured in slots, are the corruption delay imposed on the adversary, the lifetime of the system, and the length of one epoch.

The first stage is called the static stake, where $D = L$, meaning that all corrupt nodes involved with the execution of the protocol are decided before the session starts. At the start of the first slot, the advantage of honest players against adversarial ones, ϵ, is applied to the initial stakeholders. Additionally, the stake transfers do not affect the selection process, which means the players who are not in the set of initial stakeholders cannot be chosen as a slot leader even if they eventually get the stake transferred. The second, third, and forth stages of the protocol are defined differently depending on relations of variables D, L and R.

2.1.3 Static stake protocol

This subsection structurally describes the static stake protocol. First, we define blocks, a blockchain, an epoch, the adversarial stake ratio, and the leader selection process.

Definition 2.1. (Genesis Block). *The genesis block B_0 contains the list of stakeholders identified by their public-keys, their respective stakes (vk_1, s_1), ...,(vk_n, s_n) and auxiliary information ρ.*

Definition 2.2. (Block) *A block B generated at a slot $sl_i \in \{sl_1, \ldots, sl_R\}$ contains the current state $st \in \{0,1\}^\lambda$, data $d \in \{0,1\}^*$, the slot number sl_i and a signature $\sigma = \mathsf{Sign}_{\mathsf{sk}_i}(st, d, sl_i)$ computed under the secret key sk_i corresponding to the stakeholder U_i generating the block.*

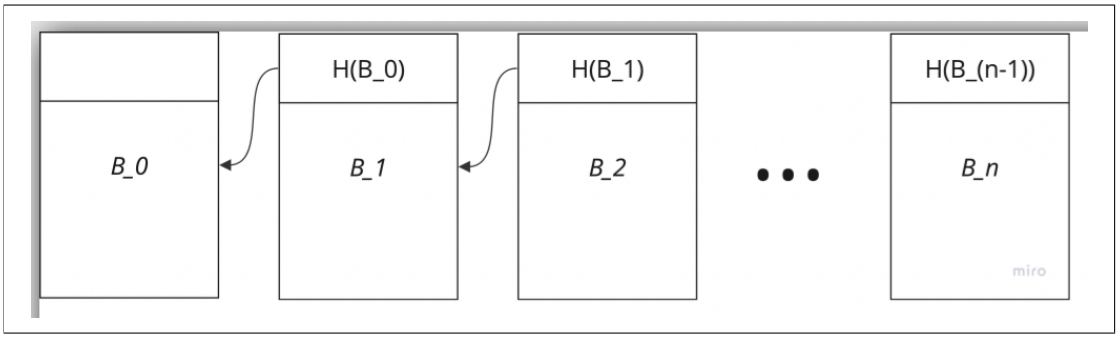


Figure 2.1: A simple blockchain construction as described in defintion 2.3

Definition 2.3. (Blockchain). *A blockchain (or simply chain) relative to the genesis block B_0 is a sequence of blocks $B_1, \ldots, B_n$ associated with a strictly increasing sequence of slots for which the state st_i of B_i is equal to $H(B_{i-1})$, where H is a prescribed collision-resistant hash function. The length of a chain* $\text{len}(\mathcal{C}) = n$ *is its number of blocks. The block B_n is the head of the chain, denoted as* $\text{head}(\mathcal{C})$. *We treat the empty string ϵ as a legal chain and by convention set* $\text{head}(\epsilon) = \epsilon$.

Definition 2.4. (Epoch). *An epoch is a set of R adjacent slots $S = \{sl_1, \ldots, sl_R\}$.*

The value R is a parameter of the protocol mentioned in the last subsection.

Definition 2.5. (Adversarial Stake Ratio). *Let $U_{\mathcal{A}}$ be the set of stakeholders controlled by an adversary $\mathcal{A}$. Then the adversarial stake ratio is defined as*

$$\alpha = \frac{\sum_{j \in U_{\mathcal{A}}} s_j}{\sum_{i=1}^{n} s_i},$$

where n is the total number of stakeholders and s_i is the stake of the stakeholder U_i.

Definition 2.6. (Leader Selection Process). *A leader selection process with respect to stakeholder distribution* $\mathbb{S} = \{(\mathsf{vk}_1, s_1), \ldots, (\mathsf{vk}_n, s_n)\}$, $(\mathcal{D}, \mathsf{F})$ *is a pair consisting of a distribution and a deterministic function such that, when* $\rho \leftarrow \mathcal{D}$ *(*ρ *is a copy of* $\mathcal{D}$*) it holds that for all* $sl_j \in \{sl_1, \ldots, sl_R\}, \mathsf{F}(\mathbb{S}, \rho, sl_j)$ *outputs* $U_i \in \{U_1, \ldots, U_n\}$ *with the probability*

$$p_i = \frac{s_i}{\sum_{k=1}^{n} s_k},$$

where s_i *is the stake held by the stakeholder* U_i *(we call this "weighting by stake"); furthermore, the family of random variables* $\{\mathsf{F}(\mathbb{S}, \rho, sl_j)\}_{j=1}^{R}$ *is independent.*

The static stake protocol is where, as briefly stated in the previous subsection, the stake transfers do not affect the leader selection process. Therefore, from definitions 2.5 and 2.6, the probability of an adversarial stakeholder being picked as a slot leader equals the adversarial stake ratio at the onset of the epoch that slot is in. Moreover, the protocol initialisation has to follow our assumption that there is an advantage ϵ of the honest stake ratio against the adversarial stake ratio. Hence, the adversarial stake ratio is $(1 - \epsilon)/2$. Also, with the condition that the corruption delay is equal to the lifetime of the protocol, adversarial nodes are fixed from the beginning of the execution to the end.

2.1.3.1 Short summary of the protocol π_{iSPoS}

Although real implementations of the blockchain protocol are peer-to-peer, the protocol π_{iSPoS} reduces that complication by having all stakeholders interact with certain ideal functionalities instead. Note that this protocol represents the static stake protocol, meaning that the slot leader process is done beforehand (and each of the slot is given a leader independently of other slots) for all the slots that the execution will cover. The protocol is run by each stakeholder as follows:

- Initialisation — the stakeholder gets assigned a public/secret key and gets updated by functionalities what the current slot is and what its local chain is. If it is now the first slot, sl_1, in an epoch, the functionalities will generate the genesis block (definition 2.1) as the stakeholder's chain. Otherwise, it receives the initial chain $\mathcal{C}$, and sets the local blockchain to $\mathcal{C}$.
- Chain extension — the stakeholder collects all chains from the broadcast of other stakeholders. Then, through communication with functionalities, it verifies, for each chain:
  1. all blocks have the correct linking information, and
  2. each block has the correct signature of its slot leader of the slot in which that block was declared.

Then, it collects all valid chains in the set with its local chain, and picks the longest chain to be its new local chain. If the stakeholder is the current slot leader, it also extends its local chain with a block with transaction information and broadcast this new local chain to all parties.

- Transaction generation — the stakeholder generates a transaction and signs it with the private key. It is broadcast to other stakeholders, and they then put in blocks waiting to extend their chains when they are picked as slot leaders in chain extension step.

As we can see in the chain extension step, the only ones who have the authority to extend the chain before broadcasting are slot leaders. Therefore, the adversary does not have any power in altering other parties' views of chains if it is not selected to be a slot leader. However, the adversary, if selected to be a slot leader, can opt to process differently in a few limited ways: 1) they can behave as honest stakeholders do, 2) they can extend a chain (or chains) with different blocks and broadcast all of them, or 3) not extend a chain.

2.1.4 Transaction ledger properties

The protocol π_{iSPoS} is executed to implement a robust transaction ledger collecting transactions in blocks chained in the order corresponding to when transactions are added. This ledger satisfies, with high probability, persistence and liveness. These are the properties of the robust transaction ledger introduced in the work of Garay et al. [29].

- **Persistence with parameter** $k \in \mathbb{N}$ is achieved if whenever one node reports a transaction tx as *stable*, all other nodes will report the same when queried. Stability in the context of blockchain is parameterised by a certain number k. This number means the transaction is *stable* when there are k blocks on top of the block a concerned transaction is in.
- **Liveness with parameter** $u \in \mathbb{N}$ is achieved if whenever an honest node tries to add a certain transaction tx to a ledger, after u slots have passed, all nodes if responding honestly when queried, will report tx as stable.

In work by Garay et al. [29], there are three elementary properties of the blockchain that can together imply persistence and liveness, which are defined below:

- **Common Prefix (CP); with parameters** $k \in \mathbb{N}$. The chains $\mathcal{C}_1$, $\mathcal{C}_2$ adopted by two honest parties at the onset of the slots $sl_1 < sl_2$ are such that $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$, where $\mathcal{C}^{\lceil k}$ denotes the chain obtained by removing the last k blocks from $\mathcal{C}$, and $\preceq$ denotes the prefix relation.

- **Existential Chain Quality (∃CQ); with parameters** $s \in \mathbb{N}$. Consider the chain $\mathcal{C}$ adopted by an honest party at the onset of a slot at any portion of $\mathcal{C}$ spanning s prior slots; then at least one honestly-generated block appears in this portion.
- **Honest Chain Growth (HCG); with parameters** $\tau \in (0, 1]$, $s \in \mathbb{N}$. Consider the chain $\mathcal{C}$ adopted by an honest party. Let sl_2 be the slot associated with the last block of $\mathcal{C}$ and let sl_1 be a prior slot in which $\mathcal{C}$ has an honestly-generated block. If $sl_2 \geq sl_1 + s$, then the number of blocks appearing in $\mathcal{C}$ after sl_1 is at least τs. The parameter τ is called the speed coefficient.

Also combining HCG with ∃CQ proves the stronger property of chain growth, which is defined as follows.

- **Chain Growth (CG); with parameters** $\tau \in (0, 1]$, $s \in \mathbb{N}$. Consider the chain $\mathcal{C}$ adopted by an honest party at the onset of a slot and any portion of $\mathcal{C}$ spanning s prior slots; then the number of blocks appearing in the portion of the chain is at least τs. The parameter τ is called the speed coefficient.

2.2 Forkable strings

The notion of forkable strings was introduced and claimed to be "a natural and fairly general tool that can be applied as part of a security argument the PoS setting" [14]. More specifically, forkable strings reflect the possibility that the overall views of systems can be corrupted or "forked". The abstraction of forkable strings plays an important role in the combinatorial ananylsis of the common prefix defined in chapter 2. Therefore, this notion omits the details of the signing process and other pieces of information contained in ledger blocks, but it only focuses on labelling slots to blocks and tracking which slots are honest.

In this section, we will outline necessary definitions to understand what forkable strings are and discuss the choice of theorems we formalise. We do not present all the definitions or lemmas related to forkable strings in this chapter, but will state all of the basic definitions and lemmas later along with our formalisation of the forkable strings notion in later chapters.

2.2.1 Basic definitions

Definition 2.7. (Characteristic String). *Fix an execution $\mathcal{E}$ with genesis block B_0, adversary $\mathcal{A}$, and environment $\mathcal{Z}$. Let $S = i+1, ..., i+n$ denote a sequence of slots of length $|S| = n$. The characteristic string $w \in \{0,1\}^n$ of S is defined so that $w_k = 1$ if and only if the adversary controls the slot leader of slot $i+k$. For such a characteristic string $w \in \{0,1\}^*$, we say that the index i is adversarial if $w_i = 1$ and honest otherwise.*

Characteristic strings are the way to simplify the result of the leader selection process as described in definition 2.6 for $\mathcal{F}_{\mathsf{LS}}^{D,\mathsf{F}}[\mathsf{mode}]$ used in protocol π_{iSPoS}. In each slot, either an honest player or an adversarial player is selected to be a slot leader. Therefore, in a fixed number of consecutive slots, we can label honest slot leaders by 0 and adversarial ones by 1.

Definition 2.8. (Fork). *Let $w \in \{0,1\}^n$ and let $H = \{i \mid w_i = 0\}$ denote the set of honest indices. A fork for the string w is a directed, rooted tree $F = (V, E)$ with a labeling $\ell : V \to \{0, 1, \ldots, n\}$ such that*

- *each edge of F is directed away from the root;*
- *the root $r \in V$ is given the label $\ell(r) = 0$;*
- *the labels along any directed path in the tree are strictly increasing;*
- *each honest index $i \in H$ is the label of exactly one vertex of F;*
- *the function $\mathbf{d} : H \to \{1, \ldots, n\}$, defined so that $\mathbf{d}(i)$ is the depth in F of the unique vertex v for which $\ell(v) = i$, is strictly increasing. (Specifically, if $i, j \in H$ and $i < j$, then $\mathbf{d}(i) < \mathbf{d}(j)$.)*

We use $F \vdash w$ to indicate that F is a fork of the string w. A *trivial* fork is a fork with only one vertex, the root.

Forks that are associated with characteristic strings are the only cases that are of our concern. From this point of view, it makes sense why forks are defined this way. We recall from the chain extension step described in 2.1.3.1 that honest participants, when picked as a slot leader, always extend the longest valid chain available in the system with a block and then broadcast the new chain. Therefore, considering a chain diffused from an honest player at a certain slot, this chain is valid meaning that the sequence of blocks in this chain is increasing in terms of the block owners' slot number. Also, it is the longest chain viewed by all honest players. Hence, all previous honest players did not diffuse chains with length longer or equal to this chain since it would contradict the fact that the current honest player did not pick those chains to extend them. It follows that the length of chains diffused by honest participants is increasing, or the blocks created by more

recent honest participants are always deeper from the genesis block.

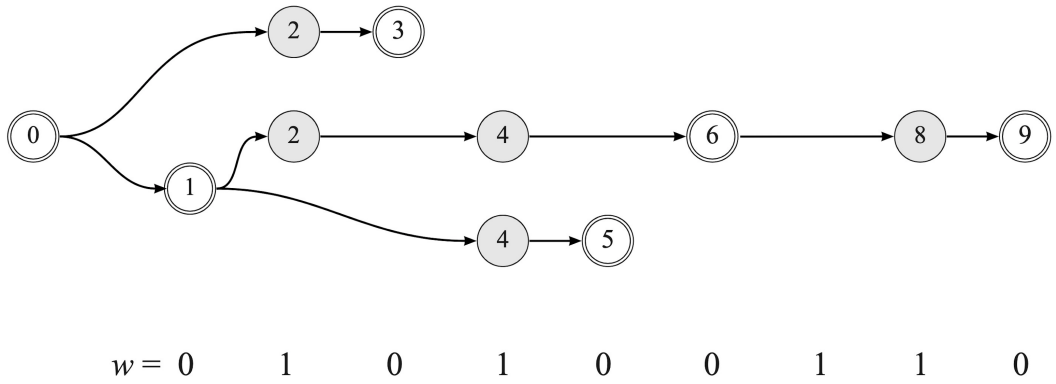


Figure 2.2: This represents fork F of a string $w = 010100110$.[4] We can see that there is exactly one vertex for each honest slot, i.e. slots $0, 1, 3, 5, 6, 9$. Other slots are adversarial, they can have any number of vertices (including 0). Moreover, all honest nodes have strictly-increasing depths.

Definition 2.9. (Tines, depth, and height). *A path in a fork F originating from the root is called a* tine. *For a tine t, we let* length(t) *denote its* length, *equal to the number of edges on the path. For a vertex v, we let* depth(v) *denote the length of the (unique) tine terminating at v. The height of a fork (as usual for a tree) is defined to be the length of the longest tine.*

The definition of tines is simply for us to identify each tine; one should be aware that two different tines can have exactly the same sequence of labels along the paths of those tines. The depth is only used for honest indices (it can be applied for adversarial indices if they are unique in a fork, but there is no usage of it this way in any original proofs.) The height of a fork and the length of a tine are trivial. Each tine represents one unique chain in the network.

Definition 2.10. (Flat forks; the $\sim$ relation). *For two tines t1 and t2 of a fork F, we write $t1 \sim t2$ if they share an edge. Note that $\sim$ is an equivalence relation on the set of nontrivial tines; on the other hand, if t_ϵ denotes the "empty" tine consisting solely of the root vertex, then $t_\epsilon \nsim t$ for any tine t. We say that a fork is flat if it has two tines $t1 \nsim t2$ of length equal to the height of the fork. A string $w \in 0, 1^*$ is said to be forkable if there is a flat fork $F \vdash w$.*

We can restate the situation when two honest parties (not necessarily participants), going offline and coming online again at the same time, have completely different

updates. If we set the characteristic string s of this situation by taking a starting digit to be the slot after these two go offline and the ending digit to be the slot when they come back online, this situation is comparable to a fork F with two edge-disjoint tines with the maximum (and hence the same) length. Thus s is forkable.

2.2.2 The density of forkable strings

In the Ouroboros paper, it was proved that forkable strings are rare by giving an upper bound to the density of forkable strings parameterised by the length of characteristic strings we consider.

Theorem 2.11. *Let $\epsilon \in (0,1)$ and let w be a string drawn from $\{0,1\}^n$ by independently assigning each $w_i = 1$ with probability $(1-\epsilon)/2$. Then*

$$\Pr[w \textit{ is forkable}] = 2^{-\Omega(\sqrt{n})},$$

This result is proved mainly by using Markov chains. The relevant theorems required are the gambler's ruin (see chapter 12 in Grinstead et al. [31]) and the Chernoff bound (see corollary A.1.14 in Alon et al. [32]).

However, better upper bounds for the density of forkable strings are given in the 'Forkable strings are rare' paper [22]. In this paper, two bounds are presented, but we only pick the first one for our presentation here.

Theorem 2.12. *Let $\epsilon \in (0,1)$ and let w be a string drawn from $\{0,1\}^n$ by independently assigning each $w_i = 1$ with probability $(1-\epsilon)/2$. Then*

$$\Pr[w \textit{ is forkable}] = \exp(-2\epsilon^4(1 - O(\epsilon))n).$$

The core ideas of the proof of this result are to calculate the expectations of many random variables, to finally get to the point where we can use Azuma-Hoeffding's inequality (see section 4.16 in Motwani et al. [33] for discussion) to get the bound. In this project, we formalise this result since not only is the proof more straightforward and readable, but it is also more efficient in providing a better upper bound.

Chapter 3

Formalisation of basic elements of forkable strings

The notion of forkable strings as it was introduced in chapter 3 is based on rose trees, a kind of tree in which each vertex can have any number (possibly zero) of successors. Although formalising trees should be considered fundamental, the forkable strings notion has its own details that need to be carefully dealt with. The aim of this chapter is to capture only the formalisations of basic elements of the forkable strings abstraction.

This chapter is also the foundation for chapters 4 and 5, as they keep building on formalisations regarding the formal definition of forks. Some formal definitions and lemmas are significantly different from their pen-and-paper versions for important reasons. Most of the time, we state their original definitions (if they were not stated before this chapter) as well as explain the idea how we differentiate them in the formalised versions.

In addition to these original definitions, there are several formal definitions and lemmas on top of what exists in the pen-and-paper versions. Since they are necessary for the formal proofs later on, not all of them will be listed and explained, but some good examples will be selected to depict the complication of formal proofs compared to the pen-and-paper ones.

We will assign numbers to the definitions and lemmas both from the pen-and-paper versions and the formal versions in Isabelle/HOL, but if there exist those for both versions, we will use the same number. (For consistency, this numbering style is applied throughout the thesis.) Pen-and-paper definitions and proofs are not our contributions but are presented as references to what the Isabelle/HOL code formalises.

We write formal proofs in Isabelle/HOL, so the first section (3.1) of this chapter will outline useful information on Isabelle/HOL, such as symbols used for meta language, different commands for definitions, and proof styles. All the following sections will be devoted to formalisation only.

3.1 Isabelle/HOL

Isabelle, a generic proof assistant, was developed by Paulson and Nipkow [34]. It can specify a variety of logics, such as first-order logic and higher-order logic (HOL). Written in standard ML, there is a small kernel implementing logic and generating theorems. The developments in Isabelle are collected in theory files, and they are in a hierarchy, meaning that one theory can depend on many other theories. A theory file typically contains datatype definitions, functions, theorems, and sometimes axioms (without proofs) by users, which is also true for this project. Children theories are able to use anything asserted, defined, or proven in their mother theories.

3.1.1 Higher order logic in Isabelle

Isabelle/HOL is one of the most used instances of Isabelle. Polymorphism and type constructors are provided and supported by this logic. Gordon's HOL theorem prover [35] originating from Church's paper [36] was an inspiration for Isabelle/HOL devolopment. Through many years of developments, a wide range of theories have been defined; this includes the natural numbers, set theory, inductive definitions, and equivalence relations. Important results ranging from mathematical theory to hardware/software specification have been formally verified in Isabelle/HOL, such as, the verification of security protocols [37] and verification of the type system in Java [38]. Through our presentation, we will use many ASCII symbols to express explicitly what we have written in Isabelle/HOL, some of them that we repeatedly use are shown in figure 3.1.

	$\neg$, not
`==>`	$\Longrightarrow$, implication (meta level)
`->`	$\longrightarrow$, implications (object level)
`=`, $\equiv$	$\equiv$, if and only if
`- A`	$\overline{A}$, set complement

Figure 3.1: ASCII notation for HOL

3.1.2 Proof constructions and styles

There are two ways to construct proofs in Isabelle: backward and forward proofs.

- In backward proofs, we apply tactics to the main goals to get simpler subgoals until they are all resolved. Tactics typically rewrite terms following certain inference rules. Applying tactics can be done by using command **apply** following by the name of the tactic. This is an example of the use of **apply**; this lemma is from natural numbers theory file in the Isabelle/HOL library:

```
lemma mult_eq_1_iff :
"(m * n = Suc 0) = (m = Suc 0 ∧ n = Suc 0)"
  apply (induct m)
   apply simp
  apply (induct n)
   apply auto
  done
```

- In forward proofs, we derive new assumptions from existing ones combining with rules we already have. The proof is complete when we derive an assumption that implies our goal. This can be done by using Isar language which facilitates structured proofs in Isabelle. In this project, we formalise this approach most of the time because structured proofs make the proofs more readable and similar to steps in the original pen-and-paper proofs. It uses clear keywords when proving, rendering long proofs relatively convenient. This lemma is an example of the structured proof.

```
lemma mult_cancel1 : "(k * m = k * n) = (m = n | (k = (0::nat)))"
proof -
  have "k ≠ 0 ⟹ k * m = k * n ⟹ m = n"
  proof (induct n arbitrary: m)
    case 0 then show "m = 0" by simp
    next
    case (Suc n) then show "m = Suc n"
     by (cases m) (simp_all add: eq_commute [of "0"])
  qed
  then show ?thesis by auto
qed
```

3.1.3 Keywords for Isabelle/HOL

In this thesis, we usually present how we define functions and formalise results in Isabelle, but we omit the detailed proof scripts and explain how challenging they

are instead. Therefore, we will only introduce some keywords that help read the formalisation without the names of specific proof methods or tactics.

3.1.3.1 Types and the datatype command

Isabelle/HOL is a typed formalism, with a type system like those of functional programming languages. Many types are defined, with supporting theories. For example, `bool`, `nat`, and `'a list` are types for Boolean, natural numbers, and lists for generic type `'a`.

To introduce a new type, we can use command **datatype**. This command is followed by how we would like to structure the preferred type. For example, datatype `nat` is asserted as follows:

datatype `nat = 0 | Suc nat`

meaning that we define a type for natural numbers inductively by introducing `Suc` and use in a way that n embedded `Suc`'s before 0 is a formalisation of a number n.

The command **datatype** can also use pre-defined types to build a new type. For example, the type of list of values is defined as follows:

datatype `bool_or_nat = B bool | N nat`

The **datatype** `bool_or_nat` can represent either `bool` or `nat` depending on the different type constructors `B` or `N`.

3.1.3.2 Defining a function

There are several different ways of defining functions in Isabelle/HOL. Here, we will focus on three commands:

- **definition** — the command **definition** is used for straightforward functions where there is no recursion involved. Also, it can be used to define 0-ary functions (constants).
- **fun** — the command **fun** can support recursive definitions, and it only allows definition that is obvious for Isabelle to see that its recursion terminates eventually.
- **function** — the command **function** is similar to the command **fun** in terms of supporting recursive definitions, but it requires us to prove the termination of the defined functions separately.

3.1.3.3 Inductive definitions

We can define predicates inductively using the command **inductive** by giving inductive rules. Instead of mapping every possible input of a defined predicate to Boolean values: True, False, the predicate defined will only return true from the rules provided. For example, the predicate `ev` returns `True` when it accepts an even number; otherwise it returns `False`:

```
inductive ev :: "nat ⇒ bool" where
ZeroI : "ev 0" |
Add2I : "ev n ⟹ ev (Suc (Suc n)) "
```

More importantly, the predicates defined this way will come with a unique induction rule, which is called induction on derivations.

3.2 Characteristic strings and forks

Now that we are ready to provide formalisation in Isabelle/HOL, we are prepared to define forks of characteristic strings, described in definitions 2.7 and 3.6, by declaring a **datatype** representing rose trees. From now on we will refer to any rose tree simply as a tree.

Definition 3.1. (Datatype: rose trees)
datatype `rtree = Tree (label: nat) (sucs: "rtree list")`

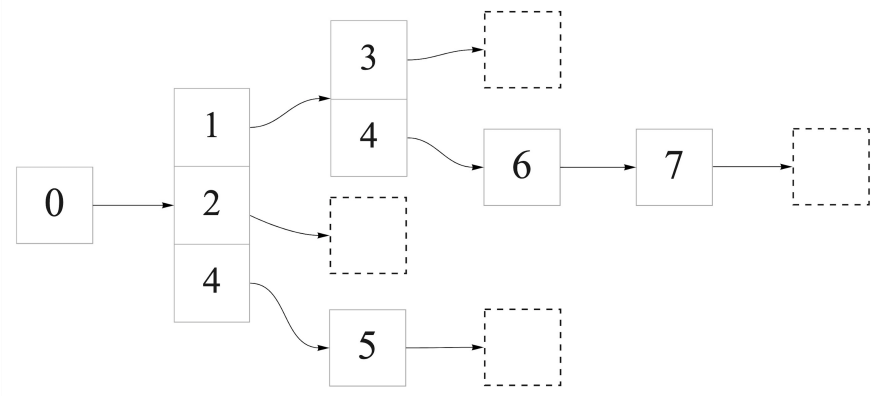


Figure 3.2: A tree representing `Tree 0 [(Tree 1 [Tree 3 [], Tree 4 [Tree 6 [Tree 7 []]]]), (Tree 2 []), (Tree 4 [Tree 5 []])]`. Each column of connected square boxes represents a list of `rtree`, with a dot-line-perimeter square representing an empty list, and each number represents a label of each `rtree`

Next, a few definitions are formalised to be used with **datatype** `rtree` to complete a formal definition for forks:

Definition 3.2. (Set of honest indices)
```
definition honest_indices :: "bool list ⇒ nat set"
  where "honest_indices w ≡ insert 0 (Suc ‘ {n. n < size w ∧ ~ w!n})"
```

The function `honest_indices` is a formalisation of H in Definition 3.6; in our formalisation, we count `0` as an honest index, and it is the only difference between `honest_indices` and H. This can be done without contradictions with any pen-and-paper proofs because only the root is labelled by 0, and its depth is 0.

Definition 3.3. (Functions depths and depth)
```
fun depths :: "rtree ⇒ nat ⇒ nat set"
  where "depths (Tree l Fs) i =
         (if l=i then {0} else Suc ‘ (⋃F ∈ set Fs. depths F i))"

definition depth :: "rtree ⇒ nat ⇒ nat"
  where "depth F i ≡ (THE d. depths F i = {d})"
```

The function `depths` returns a set of all natural numbers which are the depths of vertices in the tree (the first parameter) labelled by a certain number (the second parameter). However, it only works with a strictly increasing tree where for each vertex, its label is greater than its parent's label. The "`THE d`" part of `depth` means a unique existence of `d`, so `depth` is only defined when `depths` returns a singleton set. The function `depth` is meant to be used only with the label of exactly one vertex.

Definition 3.4. (Strictly increasing trees)
```
inductive strictly_inc :: "rtree ⇒ bool"
  where "⟦⋀F. F ∈ set Fs ⟹ l < label F;
          ⋀F. F ∈ set Fs ⟹ strictly_inc F⟧
          ⟹ strictly_inc (Tree l Fs)"
```

`strictly_inc F` is true if `F` is a strictly increasing tree.

Definition 3.5. (Multiset of vertex labels)
```
fun mset_of_tree :: "rtree ⇒ nat multiset"
  where
  "mset_of_tree(Tree l Fs) = {#l#} + (∑x ∈# mset Fs. mset_of_tree x)"
```

Finally, we have `mset_of_tree` which returns a multiset of all labels that appear in an `rtree`. This is additional to the pen-and-paper definition of forks, because

in the formalised version, we need to be able to address how many vertices have a certain label.

Now, we are ready to define forks in two steps. Firstly, we inductively define `fork` as a predicate with two parameters: `rtree` and `nat set` representing a fork and a set of honest indices.

Definition 3.6. (Intermediate function for fork)
```
inductive fork :: "rtree ⇒ nat set ⇒ bool"
  where "⟦label F = 0;
          strictly_inc F;
          ⋀h. h∈H ⟹ count (mset_of_tree F) h = 1;
          ⋀i j. ⟦i∈H; j∈H; i < j⟧ ⟹ depth F i < depth F j⟧
          ⟹ fork F H"
```

Secondly, we instantiate the second parameter of fork by `honest_indices` of a `bool list` value, which represents a characteristic string, to define forks.

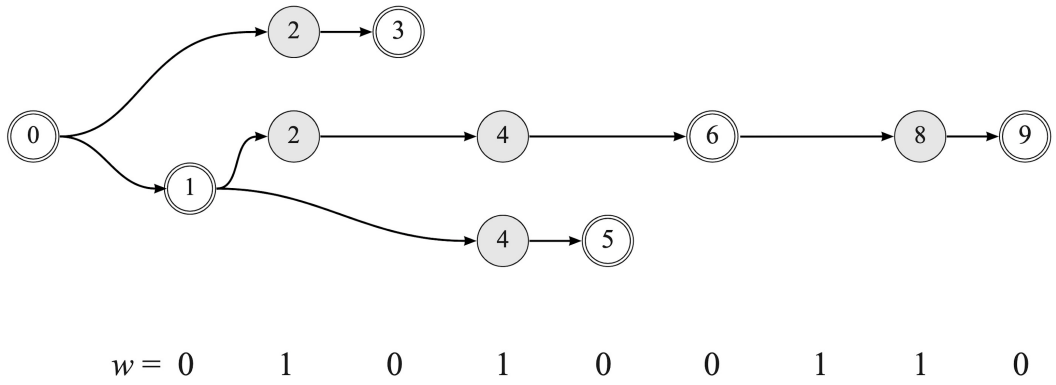
Definition 3.7. (Fork)
```
definition forks :: "rtree ⇒ bool list ⇒ bool" (infix "⊩" 50)
  where "F ⊩ w ≡ fork F (honest_indices w)"
```

It is noteworthy that by formalising trees using `list`, we create a crucial discrepancy between the original and the formalised version of forks, since the original version does not order the tree branches/tines, but the formalised one does. This difference we introduce makes it easier to trace a tine of a tree in the formal proof, but it has some serious disadvantages as we basically create repetitive formalisations of each fork and later on it messes up the definition of fork prefixes. However, we do think that the advantages of using lists for the successors of a vertex instead of other choices outweigh the disadvantages.

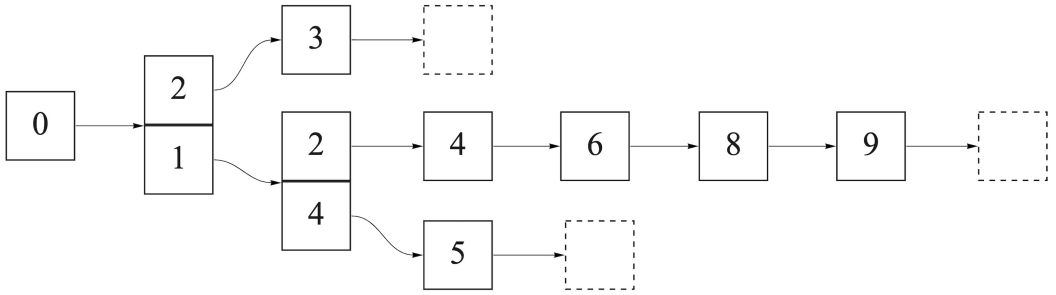
3.3 Tines

Forks are essentially trees, so some definitions related to trees are obvious to formalise. This section emphasises the definition of Tines and their related functions.

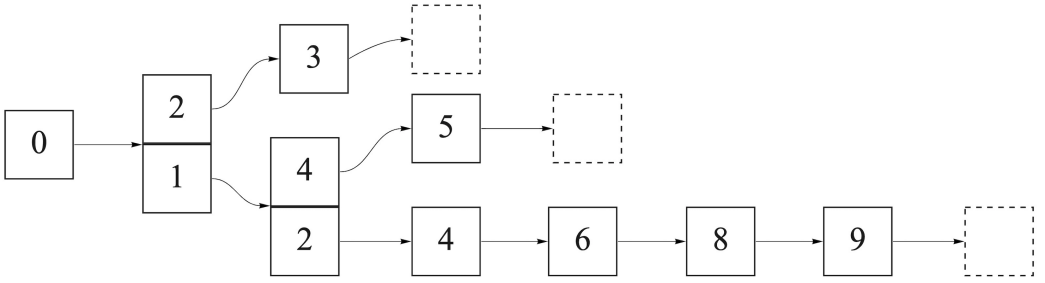
Definition 3.8. (Tines, depth, and height). *A path in a fork F originating at the root is called a tine. For a tine t we let* length(t) *denote its* length*, equal to the number of edges on the path. For a vertex v, we let* depth(v) *denote the length of the (unique) tine terminating at v. The* height *of a fork (as usual for a tree) is defined to be the length of the longest tine.*



(a) Fork F of string $w = 010100110$



(b)



(c)

Figure 3.3: Both `rtrees` in 3.3b and 3.3c are formalisations of fork F shown in 3.3a.

However, we can skip formalisation of the function depth as we already mentioned in the previous section as a necessary part to finish the formalisation of forks.

Definition 3.9. (Tines)

```
inductive tinesP :: "rtree ⇒ nat list ⇒ bool" where
   Sucs: "⟦k < size (sucs F); tinesP (sucs F ! k) t⟧
          ⟹ tinesP F (k # t)"
 | Nil: "tinesP F []"
definition tines :: "rtree ⇒ nat list set"
  where "tines F ≡ {t. tinesP F t}"
```

We start with a formalisation of tines. The function `tinesP` is defined inductively on how `rtree` is defined: this function returns a set of all tines of an `rtree`. The term `tines F t` is true if `t` is a tine (a valid path) in the tree `F`. However, the tines in these definitions are just paths from the root without the information about the labels of the vertices in them.

Definition 3.10. (Tine-tracing function)

```
fun trace_tine :: "rtree ⇒ nat list ⇒ nat list"
  where "trace_tine (Tree l Fs) t =
        (case t of [] ⇒ [l]
      | Cons k t' ⇒ if k < size Fs
                    then l # trace_tine (Fs!k) t' else [l])"
```

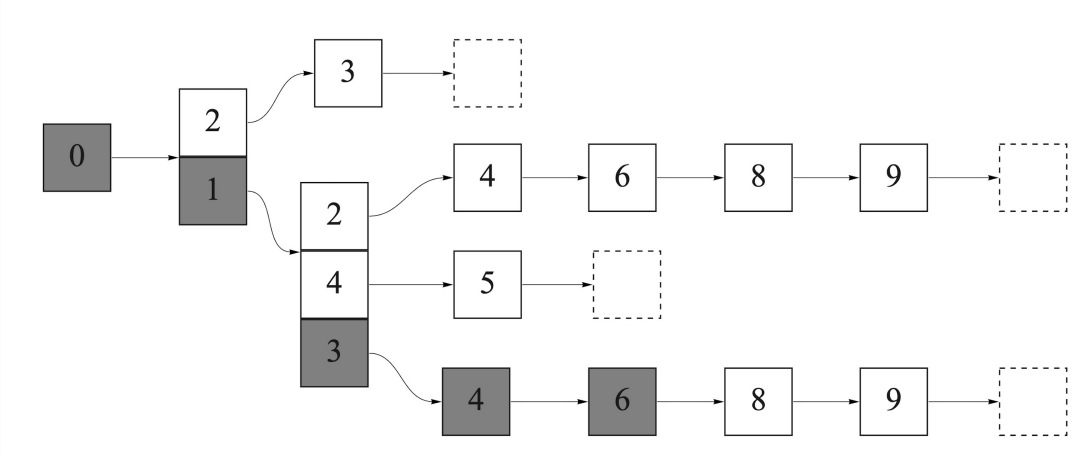


Figure 3.4: Let `F` be the `rtree` shown in this figure, `trace_tine F [1, 2, 0, 0] = [0, 1, 3, 4, 6]`

With `trace_tine` we can trace any tine in a tree and get a list of labels of all vertices in a path including the root's, shown in figure 3.4, so the output is a list

of length equal to the length of the input tine's plus one.

In addition to definition 3.8, we can describe if a tine is honest or adversarial when we know a characteristic string associated with that fork. The terms honest/adversarial tines are informally used in the Ouroboros paper. A tine is honest when it ends with a vertex labelled with an honest index; it is adversarial otherwise.

Definition 3.11. (Honest/adversarial tines)
```
definition honest :: "rtree ⇒ nat list ⇒ nat set ⇒ bool"
  where "honest F t H ≡ last (trace_tine F t) ∈ H"

abbreviation "adversarial F t H ≡ ~ honest F t H"
```

It is relatively straightforward to formalise this function using `trace_tine` and `last`, a function that returns the last element of the list.

Definition 3.12. (Height)
```
definition height :: "rtree ⇒ nat"
  where "height F ≡ Max (size ‘ tines F)"
```

Then, `height F` is simply an upper bound of the set of the size of tines in `F`. For `height` to be valid, it is necessary to know that the set of tines of any tree is not an empty set and is finite. This is because these properties are needed in order for an upper bound, `Max`, or a lower bound, `Min`, to make sense.

3.4 Subtrees

There are a number of instances where tine prefixes and suffixes are mentioned in the Ouroboros paper. Sometimes, we need to compare two different tines that share a prefix, so there is supposed to be a way to only reason about their suffixes where they started to fork. We then formalise subtrees to help make intermediate steps look cleaner in the formal proof.

Definition 3.13. (Subtrees)
```
inductive subtreesP :: "rtree ⇒ rtree ⇒ bool"
  where
    Self: "subtreesP F F"
  | Descendant: "⟦s ∈ set Fs; subtreesP s s'⟧
                 ⟹ subtreesP (Tree l Fs) s'"

definition subtrees :: "rtree ⇒ rtree set"
  where "subtrees F ≡ {s. subtreesP F s}"
```

The function `subtreesP` is comparable to `tinesP`. More specifically, the `Nil` and `Sucs` cases of `tinesP` can be compared respectively to `Self` and `Descendent` cases of `subtrees`. The set `subtrees F` collects all subtrees in the tree `F`.

Definition 3.14. (Getting-subtree function)

```
fun get_subtree :: "rtree ⇒ nat list ⇒ rtree"
  where "get_subtree (Tree l Fs) s =
    (case s of [] ⇒ Tree l Fs
             | Cons k s' ⇒ if k < size Fs then get_subtree (Fs!k) s'
                                          else Tree l Fs)"
```

Lemma 3.15.

```
lemma get_subtree_is_subtree: "subtreesP F (get_subtree F t)"
```

Then, the function `get_subtree` is defined in the same way as `trace_tine` is. As we mentioned before, the string `trace_tine F t` represents a label on the path of tine `t` in the tree `F`, but the vertex at the end of that tine is the root of the tree `get_subtree F t`.

To ensure `get_subtree` is working correctly, we provide and prove lemma 3.15.

3.4.1 Relation between subtrees and tines

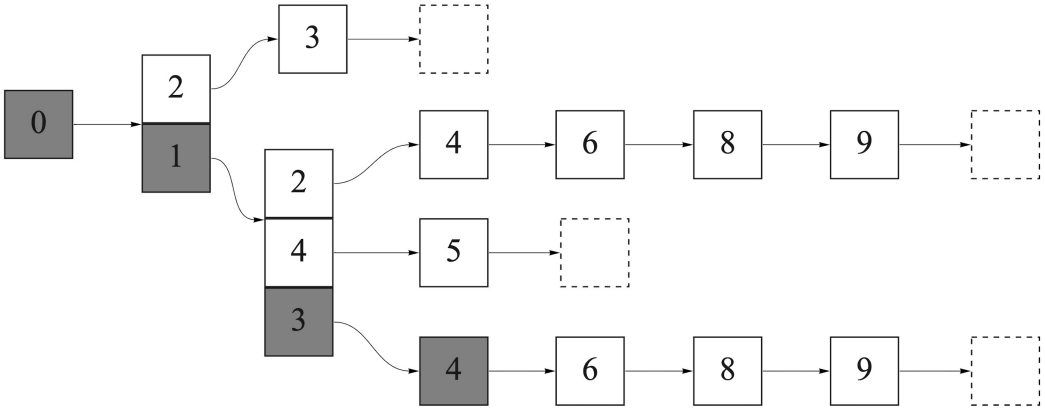
We did mention in the last section that we formalise subtrees to help with the formal proof. In this subsection, we give empirical evidence of our statement by presenting some example lemmas related to both subtrees and tines. They are mainly involved with the functions `trace_tine` and `get_subtree`.

Lemma 3.16.

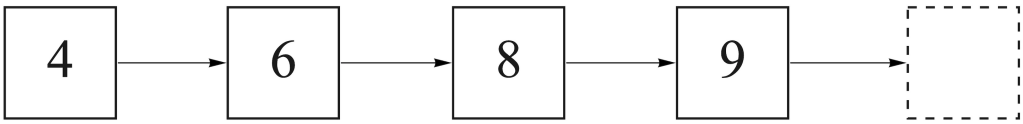
```
lemma get_subtree_tines_append_in_tines :
  assumes "tinesP F t"
  shows "tinesP (get_subtree F (take i t)) (drop i t)"

lemma mod_get_subtree_tines_append_in_tines :
  assumes "tinesP F (t@t')"
  shows "tinesP (get_subtree F t) t'"
```

First of all, we prove that if we split a tine into two parts, the suffix part is a tine of a subtree we get from tracing its root with the prefix part. With lemma 3.16, if we have a pair of edge-sharing tines, we can opt to reason about their edge-disjoint suffixes.



(a)



(b)

Figure 3.5: Let `F` be `rtree` in 3.5a, `trace_tine F [1, 2, 0] = [0, 1, 3, 4]` and `get_subtree F [1, 2, 0] = Tree 4 [Tree 6 [Tree 8 [Tree 9 []]]]` as shown in 3.5a and 3.5b respectively.

Lemma 3.17. **lemma** `get_subtree_append`:
```
  assumes "tinesP F t"
  shows "get_subtree F t
         = get_subtree (get_subtree F (take i t)) (drop i t)"
```

As a result of any tine of a tree being a tine of a corresponding subtree, we can split a tine we use to get a subtree into 2 or more parts and trace them to automatically get an intermediate subtree before the final one. This is useful when we need to identify important vertices along the line and explore different branches from those vertices.

Lemma 3.18.
lemma `last_in_tine_is_label_in_subtree`:
```
  "last (trace_tine F t) = label (get_subtree F t)"
```

Lemma 3.19. **lemma** `trace_tine_append_trace_tine_subtree`:
```
  assumes "tinesP F t"
  shows  "trace_tine F t = trace_tine F (take i t) @
                           (tl
                            (trace_tine
                             (get_subtree F (take i t)) (drop i t)))"
```

Structurally similar to `get_subtree`, appends also preserve `trace_tine`. More specifically, when a tine is split into two parts, they can be traced with their corresponding trees using `trace_tree`, and their results appended equal to using `trace_tine` to trace the original tree with the original tine. However, there is exactly one shared vertex as the last of the first result and the head of the second result are the same vertex. Hence, we put `tl` in the equation in this lemma to get rid of one instance of this vertex. This lemma is by far the most frequently used one to reason about tines.

3.5 Flat forks and the $\sim$ relation

Definition 3.20. (Flat forks and the $\sim$ relation) *For two tines t_1 and t_2 of a fork F, we write $t_1 \sim t_2$ if they share an edge. Note that $\sim$ is an equivalence relation on the set of nontrivial tines; on the other hand, if t_ϵ denotes the "empty" tine consisting solely of the root vertex, then $t_\epsilon \nsim t$ for any tine t. We say that a fork is flat if it has two tines $t_1 \nsim t_2$ of length equal to the height of the fork. A string $w \in \{0,1\}^*$ is said to be forkable if there is a flat fork $F \vdash w$.*

The $\sim$ relation between two tines denotes that those tines share an edge, so we

formalise $\sim$ by the function equiv ($\cong$), which is defined to hold if two tines in a tree have the first edge from the root shared.

Definition 3.21. (The $\sim$ relation and its negation)

```
inductive equiv :: "nat list ⇒ nat list ⇒ bool" (infix "≅" 50)
  where "(t0#_) ≅ (t0#_)"

abbreviation not_equiv :: "nat list ⇒ nat list ⇒ bool" (infix "'/≅" 50)
  where  "not_equiv t1 t2 ≡ ~ (t1 ≅ t2)"
```

We seem to use the negated form /$\cong$ of $\cong$ more than the positive form because edge-disjoint tines are relevant to several important results.

Definition 3.22. (Flat forks)

```
inductive flat :: "rtree ⇒ bool"
  where "⟦t1 /≅ t2;
          size t1 = height F;
          size t2 = height F;
          t1 ∈ tines F;
          t2 ∈ tines F⟧ ⟹ flat F"
```

Definition 3.23. (Forkable strings)

```
inductive forkable :: "bool list ⇒ bool"
  where "⟦F ⊩ w; flat F⟧ ⟹ forkable w"
```

Flat forks and forkable strings are then formalised straightforwardly.

We have already discussed the idea behind flat forks and forkable strings in chapter 3. Flat forks represent situations where there are two completely different suffixes of the chain in the combined views of all honest stakeholders through an execution of the protocol π_{iSPoS} during a certain time period of n slots. The forkable strings terms are used for characteristic strings where there is a possibility for these situations to occur.

3.6 Fork prefixes

One of the most important definitions related to forkable strings is that of fork prefixes. A fork prefix $F \vdash w$ of $F' \vdash w'$, is a consistently-labelled subtree of F', or $F \sqsubseteq F'$, and w is a string prefix of w'. In the Ouroboros paper, there is no formal definition of fork prefixes. Hence, the formalisation in this section is only from an informal description.

Definition 3.24. (Match for lists)
```
inductive match for P
  where match_Nil_Nil: "match P [] []"
  | match_true: "⟦P x y; match P xs ys⟧ ⟹ match P (x#xs) (y#ys)"
  | match_extra: "match P xs ys ⟹ match P xs (y#ys)"
```

We start by defining `match` for `P`, where `P` here is a relation from a list to a list. We can see that the type of `P` depends on the types of its parameters (`P` is polymorphic).

Definition 3.25. (Tree prefixes)
```
inductive prefix :: "rtree ⇒ rtree ⇒ bool"
  where "match prefix Fs Fs' ⟹ prefix (Tree l Fs) (Tree l Fs')"
    monos match_mono
```

The predicate `prefix` is defined by simply adding the condition that two trees have the same-labelled roots and their lists of successors, together, satisfy `match`.

Lemma 3.26.
```
lemma match_Exist:
  assumes "match P (x#xs) ys"
  shows "∀a ∈ set (x#xs). ∃ y ∈ set ys. P a y"
using assms by (induction ys, auto simp:)
```

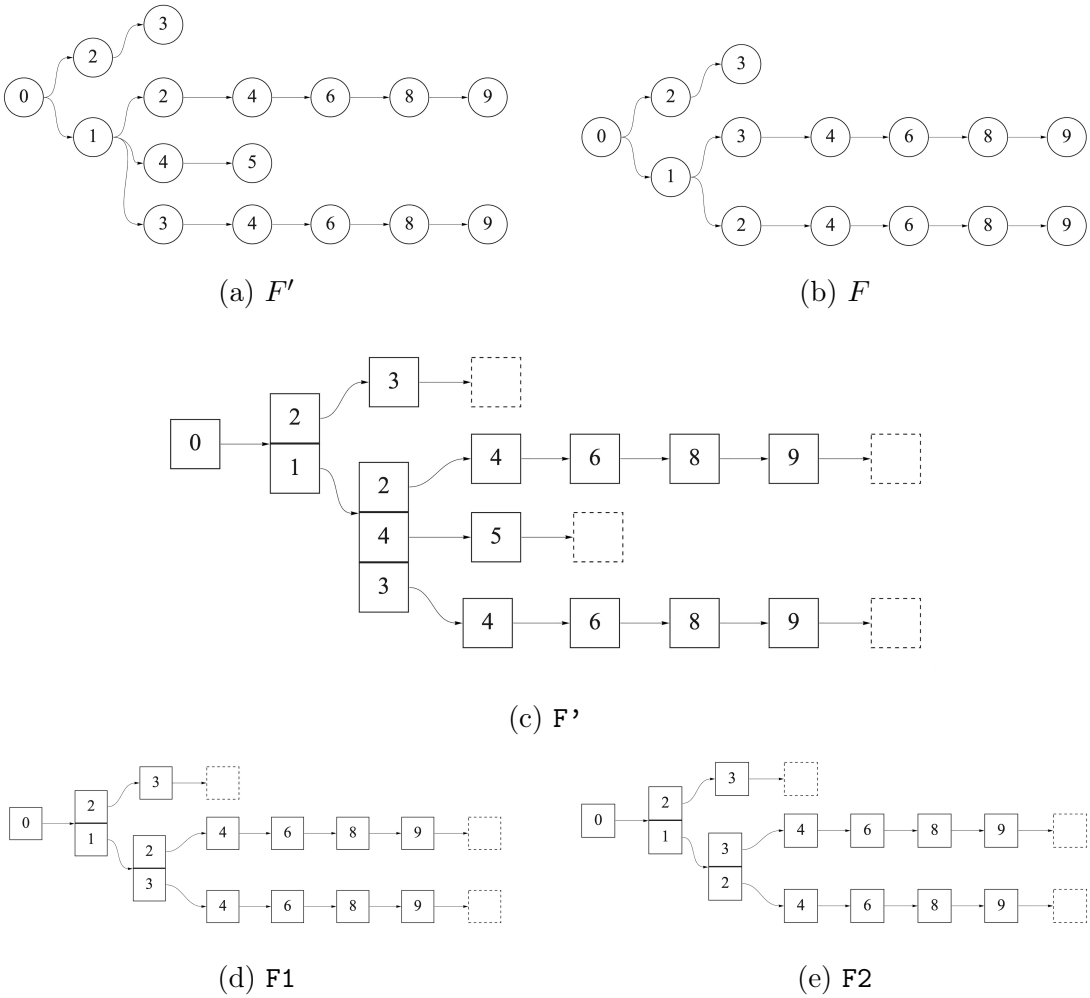
It is useful to mention that we use lemma 3.26 very frequently. When we have `prefix F F'`, and we know there is a branch `f` in the tree `F`, this lemma helps state the existence of a suffix branch in `F'`.

The relation `prefix` is a partial order on `rtree` — we can instantiate `order` with `rtree` by proving that `prefix` is reflexive, antisymetric, and transitive. As we instantiate `rtree` as a partial order through the operator `prefix`, now we can represent `prefix` by $\leq$. We have to note that the function `prefix` is comparable to $\sqsubseteq$ but is indeed not a formalisation of $\sqsubseteq$. This is because the only requirement for $F \sqsubseteq F'$ is for all labels and edges in F to appear in F', whereas for `prefix`, the order of the successors of a tree is taken into account. Therefore, assuming `F` and `F'` are the formalisation of F and F', respectively, if `prefix F F'` holds, then $F \sqsubseteq F'$ holds, but not the other way around.

Next, strings prefixes are defined to complement tree prefixes.

Definition 3.27. (String prefixes)
```
definition string_prefix :: "bool list ⇒ bool list ⇒ bool"
  where "string_prefix w w' ≡ ∃x. w @ x = w'"
```

Fork prefixes can finally be formalised as shown below.



(a) F' (b) F

(c) `F'`

(d) `F1` (e) `F2`

Figure 3.6: While $\sqsubseteq$ does not consider an order of branches, `prefix` does. For example, let F', F be trees represented in 3.6a and 3.6b respectively, and `F', F1, F2` be `rtrees` represented in 3.6c, 3.6d, and 3.6e respectively, it is clear that `F'` is formalisation of F', and both `F1`, `F2` are formalisation of F. However, while $F \sqsubseteq F'$ and `prefix F1 F'`, $\neg$ `prefix F2 F'`.

Definition 3.28. (Fork prefixes)
inductive

```
fork_prefix :: "rtree ⇒ rtree ⇒ bool list ⇒ bool list ⇒ bool"
where "⟦F ⊩ w; F' ⊩ w'; string_prefix w w'; F ≤ F'⟧
       ⟹ fork_prefix F F' w w'"
```

3.6.1 Tine prefixes

Another important issue about prefixes is to define prefixes for tines. We first consider that we cannot define prefixes of a tine without the condition that those tines are from two trees where one is a prefix tree of the other. As a result, we arrive at this definition of `tine_prefix`:

Definition 3.29. (Tine prefixes)
inductive

```
tine_prefix :: "rtree ⇒ rtree ⇒ nat list ⇒ nat list ⇒ bool"
where
  Nil: "⟦F ≤ F'; tinesP F' t⟧ ⟹ tine_prefix F F' [] t"
| Subtree: "⟦k < size (sucs F);
             k' < size (sucs F');
             tine_prefix (sucs F ! k) (sucs F' ! k') t t';
             F ≤ F'⟧⟹ tine_prefix F F' (k#t) (k'#t')"
```

This definition contains a lot of information, especially in the `Subtree` case. In contrast, prefixes of tines are mentioned briefly without detailed descriptions in the Ouroboros paper. Therefore, this definition can be our substantial example reflecting how tedious this project can be, that is we may get exhaustively long formal proofs out of their originally short informal versions.

Moreover, it is worth mentioning that we have this lemma:

Lemma 3.30.

```
lemma prefix_tree_exist :
  assumes "F ≤ F'" "t ∈ tines F"
  shows "∃t' ∈ tines F'. tine_prefix F F' t t'"
```

from the statement in the paper, "If $F \sqsubseteq F'$, each tine of F appears as the prefix of a tine in F'", since it will be useful in future proofs.

3.7 Closed forks

A *closed* fork represents the combinations of chains that are diffused by honest parties only, so in each maximal chain, the most recent block is declared by an honest party. In other words, all leaves in the fork have to be labelled with honest slot numbers. Later on in chapter 4, closed forks will be used to explain how weak characteristic strings are to an adversary's attack.

We start at formalising closed trees and then closed forks.

Definition 3.31. (Closed trees)
```
inductive closed_tree:: "rtree ⇒ bool list ⇒ bool"
  where
    Tree : "⟦∀x ∈ set (Fh#Ftl). closed_tree x w⟧
             ⟹ closed_tree (Tree l (Fh#Ftl)) w"
  | Leaf : "l ∈ honest_indices w ⟹ closed_tree (Tree l []) w"
```

Definition 3.32. (Closed forks)
```
inductive closed_fork
  where "⟦closed_tree F w; F ⊩ w⟧ ⟹ closed_fork F w"
```

Similar to working with `fork_prefix`, we define closed trees and then closed forks using `inductives`.

Lemma 3.33.
```
lemma closed_tree_imply_closed_subtree:
  assumes "subtreesP F F'" "closed_tree F w"
  shows "closed_tree F' w"
```

It is quite obvious that `subtrees` preserves `closed_tree`, but since we do not define `rtree` in a way that we can mention leaves easily except for tracing from the root until we find vertices that have no children, we need lemma 3.33 for it.

Definition 3.34. (Count indices in a tree from a set)
```
definition count_tree_set :: "rtree ⇒ nat set ⇒ nat" where
    "count_tree_set F h ≡ (∑x ∈ h. count (mset_of_tree F) x)"
```

Lemma 3.35.
```
lemma non_exist_closed_tree_count_tree_set_0:
  assumes "count_tree_set F (honest_indices w) = 0"
  shows "¬ (∃F'. (F' ≤ F) ∧ closed_tree F' w)"
```

Function `count_tree_set` is defined mainly for counting all honest vertices in a

tree. It follows that if there are no honest vertices in a tree, that tree does not have a prefix tree that is closed. This result is too obvious to mention in the pen-and-paper proof.

Lemma 3.36.
```
lemma exist_closed_tree_prefix_aux:
  assumes "count_tree_set F' (honest_indices w) > 0"
  shows "∃F. closed_tree F w ∧ F ≤ F' ∧ (∀h. h ∈ (honest_indices w) ⟶
count (mset_of_tree F) h = count (mset_of_tree F') h)"
```

Lemma 3.37.
```
lemma exist_closed_tree_prefix:
  assumes "(∀h. h ∈ (honest_indices w)
            ⟶ count (mset_of_tree F') h = 1)"
  shows "∃F. closed_tree F w ∧ F ≤ F' ∧
          (∀h. h ∈ (honest_indices w) ⟶ count (mset_of_tree F) h = 1)"
```

Lemma 3.38.
```
lemma exist_closed_fork_prefix:
  assumes "F' ⊢ w"
  shows "∃F. closed_fork F w ∧ F ≤ F'"
```

Then we can prove that we have a prefix tree that maintains all honest vertices, and, eventually, it can be applied for forks.

Lemma 3.39.
```
lemma exist_closed_fork: "∃F. closed_fork F w"
proof (induction w rule: rev_induct)
```

Another important result with closed forks is there is always a closed fork for any characteristic string. This can be proved easily by the reverse induction on strings using the rule `rev_induct`.

3.8 Trivial trees, forks

Trivial trees are not defined formally in the Ouroboros paper due to their triviality. A trivial tree is basically a tree with one vertex: its root.

Lemma 3.40.
```
lemma adversarial_strings_contain_trivial_tree_fork:
  assumes "∀x ∈ set w. x"
  shows "Tree 0 [] ⊢ w"
```

Lemma 3.41.
lemma trivial_tree_forks_adversarial_strings:
assumes "Tree 0 [] ⊢w"
shows "∀x ∈ set w. x"

There is only one type of characteristic string that has a trivial fork (a fork that is a trivial tree): ones without honest indices.

Lemma 3.42.
lemma trivial_tree_flat:
assumes "sucs F = []"
shows "flat F"

A trivial tree is always a flat tree as []/≅[] and [] is the longest, and the only, tine in a trivial tree.

Lemma 3.43.
lemma closed_tree_empty_sucs:
assumes "closed_tree (Tree l []) w"
shows "l ∈ honest_indices w"

Lemma 3.44.
lemma closed_fork_trivial :
"Tree 0 [] ⊢ w ⟹ closed_fork (Tree 0 []) w"

However, a trivial tree is closed only if the root is honest. Hence, if the root is labelled by 0, the tree is closed. Therefore, if a trivial tree is a fork, it is a closed fork.

3.9 Chapter summary

We have presented the formalisation of basic elements of the notion of forkable strings in Isabelle/HOL, some of which are complicated. We achieved the goal of formalising flat forks and closed forks, which are the two most significant kinds of forks we intend to discuss in this project. Some definitions that do not exist as definitions but only rather descriptions in the Ouroboros paper were presented here as formal definitions, since they are necessary for us to properly reason about them. Among them, fork prefixes are the most noteworthy concept, as they see usage in several results later on. Finally, in each section, we often provide a few lemmas that seem useful and are related to each definition in the section.

Chapter 4

Formalisation of 'forkable strings are rare'

One important part of this project is to formalise the upper bound of the density of forkable strings in a set of fixed-length strings. This result will lead to the formalisation of how the Ouroboros protocol meets the common prefix property. This result is also the biggest part of the project in terms of how exhaustive and delicate the pen-and-paper proof is. While we state that this is an important result to be formalised, we do have more than one candidate for this upper bound to be proved as discussed in 2.2.2. These candidates are only different in the sense that there are different theorems in probability theory used, but the discrete structure part of their proof is based on the same results. As the discrete structure part of the proof consists of several results which sometimes do not convey any important messages or meanings on their own, we will refer to them as intermediate results.

We will explain how we formalise the density of forkable strings in two sections: section 4.1 discusses and formalises the intermediate results, and section 4.2 discusses which result we pick to formalise for the density of forkable strings and why, as well as its formalisation.

However, we need to stress that the results in section 4.2 contains both finished machine proofs and a plan for when there are specific theorems regarding probability getting formalised in the future.

4.1 Formalisation of intermediate results

The reason we call results in this section "intermediate" is that we only use them for formalising a major lemma in section 4.2. There are many formalisation tasks

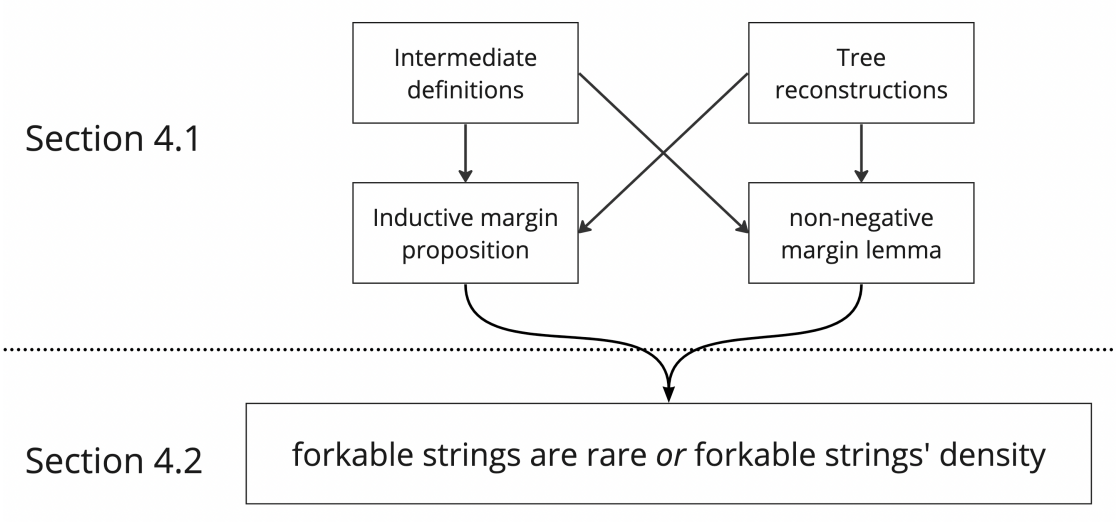


Figure 4.1: An organisation of chapter 4

explained in this section, but we can group them into four parts: 4.1.1, 4.1.2, 4.1.3, and 4.1.4.

The first part is to describe intermediate definitions called gap, reserve, reach, and margin. These definitions are less primitive than tines, depth, and height of a tree that we discussed and formalised earlier in chapter 3. Then, the second part lists options on how we can reconstruct trees (**type** `rtree` to be precise) from the given ones. In the third part, we discuss a proposition regarding the relationship between the forkability of a characteristic string and the margin of its forks. We then formalising the proof of that proposition. Lastly, in the forth part, we explain and formalise how the margin of strings can be defined inductively. As shown in figure 4.1, the first and the second parts are prerequisites for the third and the forth ones.

4.1.1 Intermediate definitions I: gap, reserve, reach, and margin

This section focuses on margin: a value defined on forks. However, before we can formalise margin, we need to formalise gap, reserve, and reach; values defined on tines in a fork. For the convenience of describing definitions and formalisations in this section, we will assume t, t_1, and t_2 are tines in fork F of a string w, and `t`, `t1`, `t2`, `F`, and `w` are their formalisation, respectively.

The *gap* of a tine t in a tree F is the difference between the length of t and the height of F. The following is our formalisation for gap:

Definition 4.1. (Gap)
```
definition gap :: "rtree ⇒ nat list ⇒ nat"
  where "gap F t ≡ height F - size t"
```

The *reserve* of t is the number of adversarial indices appearing in w after the last index labelled in t. The following is our formalisation for reserve:

Definition 4.2. (Reserve)
```
definition reserve :: "rtree ⇒ bool list ⇒ nat list ⇒ nat"
  where
   "reserve F w t ≡ count (mset (drop (last (trace_tine F t)) w)) True"
```

The *reach* of t is the difference between its reserve and its gap. This value can be negative, so `reach` is the only one function out of the three that returns `int`. The following is our formalisation for reach:

Definition 4.3. (Reach)
```
definition reach :: "rtree ⇒ bool list ⇒ nat list ⇒ int"
  where "reach F w t ≡ int (reserve F w t) - int (gap F t)"
```

Regarding the names of these values, one can say that, an adversary, when considering a tine t of a fork $F \vdash w$, can extend t to 'reach' the length of the longest tine in F by filling the 'gap' between them with the adversarial slots it 'reserves' or gains control of.

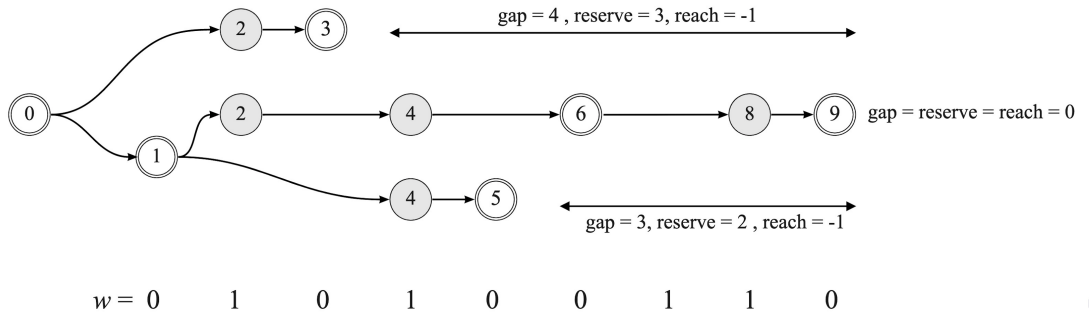


Figure 4.2: Although only three tines are labeled with these values, all of their sub-tines have their own gap, reserve, and reach.

4.1.1.1 Margin

Finally, we can formalise margin, but there is also another value used inseparably to margin named ρ. These two values are developed from *reach*. They are not defined for each tine in a fork but for an entire fork, and they only concern and are applied to closed forks.

Definition 4.4. (Margin) *For a closed fork $F \vdash w$ we define $\rho(F)$ to be the maximum reach taken over all tines in F:*

$$\rho(F) = \max_t \operatorname{reach}(t).$$

The following is our formalisation for ρ:

```
definition rho
  where "rho F w ≡ Max ((reach F w) ‘ (tines F))"
```

Likewise, we define the margin of F, denoted $\mu(F)$, to be the "penultimate" reach taken over edge-disjoint tines of F: specifically,

$$\operatorname{margin}(F) = \mu(F) = \max_{t_1 \nsim t_2} \left(\min \Big(\operatorname{reach}(t_1), \operatorname{reach}(t_2) \Big) \right).$$

The following is our formalisation for margin (μ):

```
inductive marginP
  where "⟦t1 /≅ t2; t1 ∈ tines F; t2 ∈ tines F; F ⊨ w⟧
         ⟹ marginP F w (min (reach F w t1) (reach F w t2))"

definition margin
  where "margin F w ≡ Max {t. marginP F w t}"
```

We formalise margin in a relatively straightforward fashion with a slight adaptation to collect all pairs of edge-disjoint tines using **inductive**.

Unlike gap, reserve, or reach, margin is more complicated and less intuitive than these three, and ρ is defined just to help inductively 'define' margin; we can see this clearer in 4.1.3.

We revisit the issue of function `Max`. The issue of `Max` is that we need to ensure it applies to finite and non-empty sets. Both formalisations for ρ and *margin* use `Max` and apply it with different sets. However both of the sets are subsets of the set of all tines of the concerned fork. Therefore, if we can prove that the set of all tines of any fork is finite and non-empty, `rho` and `margin` will always make sense.

Lemma 4.5. (Tine existence)
lemma `tinesP_exists: "∃t. tinesP F t"`
using `tinesP.Nil` **by** `auto`

Lemma 4.6. (Non-empty set of tines)
lemma `tines_nonempty: "tines F ≠ {}"`
by `(simp add: tinesP_exists tines_def)`

The formal proof that the set of tines is not empty is trivial because the 0-length tine is included in every fork by `tinesP.Nil` so it can be done by `simp` and `auto`.

Lemma 4.7. (Finite set of tines)
lemma `tines_finite: "finite (tines F)"`

The formal proof that the set of all tines is finite needs a structural induction proof on `rtree` and then on `sucs`.

4.1.1.2 Redefining margin

When we consider characteristic strings, each string has many possible forks (infinitely many if there is at least one adversarial slot). Therefore, ρ and margin, μ, can be redefined/overloaded from functions for forks to functions for strings. Originally in the Ouroboros paper, the names of these functions are not changed, but we change their names in our formalisation to be able to use each version of them when suitable. As we mentioned already, these values are only considered for closed forks, so we only redefine ρ and margin as follows:

$$\rho(w) = \max_{\substack{F \vdash w \\ F closed}} \rho(F), \quad \mu(w) = \max_{\substack{F \vdash w \\ F closed}} \mu(F).$$

The formalisation of the redefined versions is straightforward.

Definition 4.8. (Set of closed forks)
definition `closed_forks_set`
where `"closed_forks_set w ≡ {F. closed_fork F w}"`

First, we define sets of closed forks for each string.

Definition 4.9. (Margin for strings)
definition `rho_s`
where `"rho_s w ≡ Max ((λf. rho f w) ‘ (closed_forks_set w))"`

definition `margin_s`
where `"margin_s w ≡ Max ((λf. margin f w) ‘ (closed_forks_set w))"`

We put the suffix "`_s`" in the name of these two functions just to distinguish them from the old versions in the formal proof.

The use of `Max` this time requires the set of closed forks of a string to be non-empty and finite. This is a lot more complicated to formalise than those of the set of tines. Therefore, we came up with a simpler way: formalise the upper bound and the lower bound of the reach for each string.

Lemma 4.10. (Reach upper and lower bounds)
lemma `min_reach:`
assumes `"F ⊢ w"`
shows `"reach F w t ≥ - int (size w)"`

lemma `max_reach:`
`"reach F w t ≤ int (size w)"`

By doing so, we can now establish that the set of possible reach values is finite (since we have their ranges and they can only be integers). Then we can prove that the set of all ρ values and the set of all $margin$ values are also finite because they are subsets of the set of possible reach values.

Lemma 4.11. (Finite set of ρ)
lemma `finite_set_of_rho:`
`"finite ((λf. rho f w) ‘ (closed_forks_set w))"`

Lemma 4.12. (Finite set of margin)
lemma `finite_set_of_margin:`
`"finite ((λf. margin f w) ‘ (closed_forks_set w))"`

Ultimately, we can avoid the need to formalise that the set of closed forks is finite.

To prove that the set of closed forks is non-empty, is trivial and not needed to declare a lemma, since we have already established lemma 3.39 (**lemma** `exist_closed_fork`) in the previous chapter.

4.1.2 Reconstructing trees: I

Reconstructing trees in pen-and-paper proofs is not a difficult issue to be addressed. For example, it is obvious if we extend any tines of a tree, the old one will automatically be a tree prefix of the new one. In formal proofs, however, for this kind of reasoning to be clean and not repetitive, some functions and lemmas must be declared and proved to facilitate heavy usages of particular properties.

There are many ways to reconstruct trees. However, in this section, there are two main methods we will be describing: 1) extending a tine, and 2) cutting a sub-tree. We opt to describe these two due to their helpfulness to our important results in 4.1.3 and 4.1.4.

4.1.2.1 Extending a tine

Extending a tine is not a difficult process to formalise. However, if we wish to extend any tine with a list, that list needs to be transformed into a tree. Therefore, we define `list_to_tree` as follows:

Definition 4.13.

```
fun list_to_tree :: "nat list ⇒ rtree" where
   "list_to_tree [] = Tree 0 []"
 | "list_to_tree [x] = Tree x []"
 | "list_to_tree (k1#(k2#t)) = Tree k1 [list_to_tree (k2#t)]"
```

We use **fun** to define this function because it needs to be defined using recursion. As we intend to change from a list to a tree, indices in the tree need to come from elements in the list only. Therefore, the first case of definition 4.13 is never used as there is no tree with an empty set of indices. This reason also applies to the second and the third case emphasising that we cannot have only one case for non-empty lists.

Definition 4.14. (Tine extension)

```
fun Extend_tine :: "rtree ⇒ nat list ⇒ nat list ⇒ rtree"
  where "Extend_tine (Tree l Fs) [] list
          = Tree l ((list_to_tree list)#Fs)"
      | "Extend_tine (Tree l Fs) (k#t) list
          = Tree l ((take k Fs)
          @ (Extend_tine (Fs ! k) t list) # (drop (Suc k) Fs))"
```

We can see that `Extend_tine` traces the input tine recursively until the tine ends, and then it puts the extended part as the first successor of the end of that tine. This is simply related to how we formalise tree prefixes through `match`: this definition

is consistent with `match_extra: "match P xs ys ⟹ match P xs (y#ys)"`, so it is almost automatic to prove the lemma below.

Lemma 4.15.
```
lemma Extend_tine_implies_prefix:
 assumes "tine ∈ tines F"
 shows "F ≤ Extend_tine F tine list"
```

Most of the time, when we reason with trees, those trees are forks. Thus, it is useful to have some tools in our hands to show how extending a tine affects properties of forks. For example, being a strictly increasing tree is one property of forks. If we extend a strictly increasing tree at one tine ending with x with a strictly increasing list whose head contains a number greater than x, the new tree is also strictly increasing.

Lemma 4.16.
```
lemma Extend_tine_preserves_strictly_inc:
  assumes "strictly_inc F" "strictly_inc_list ls ∧ ls ≠ []"
          "tine ∈ tines F" "label (get_subtree F tine) < hd ls"
  shows "strictly_inc (Extend_tine F tine ls)"
```

Ultimately, extending a tine can build, from a fork, another fork that is associated with the same string. Doing so needs some restrictions, such as, a list being an extending part

- does not contain honest indices since it will change the count of those honest indices
- is strictly increasing, and
- does not contain a number greater than the length of an associated string.

Lemma 4.17. (A fork form by extending another fork)
```
lemma Extend_tine_with_reserve_preserves_forks:
  assumes "forks F w" "tine ∈ tines F"
          "S = {x. x > last (trace_tine F tine) ∧ x ≤ size w
           ∧ x ∉ honest_indices w}"
          "strictly_inc_list ls" "ls≠[]" "∀x ∈ set ls. x ∈ S"
  shows "forks (Extend_tine F tine ls) w"
```

4.1.2.2 Cutting a sub-tree

Another method we present here is how we formalise cutting a sub-tree from a tree, and it is less complicated than the previous one. Although we say that it is

the method for cutting a sub-tree, the initial intention is for us to be able to cut the ending part of any tine which is one kind of sub-trees.

We start on how we cut out an element in a list in the position we aim for cutting.

Definition 4.18.
```
primrec cut_nth:: "nat ⇒ 'a list ⇒ 'a list"
  where   "cut_nth n [] = []"
        | "cut_nth n (x#xs)
           = (case n of 0 ⇒ xs | Suc k ⇒ x # (cut_nth k xs))"
```

Lemma 4.19.
```
lemma cut_nth_take_drop_Suc:
  assumes "i < size l"
  shows "cut_nth i l = (take i l) @ (drop (Suc i) l)"
```

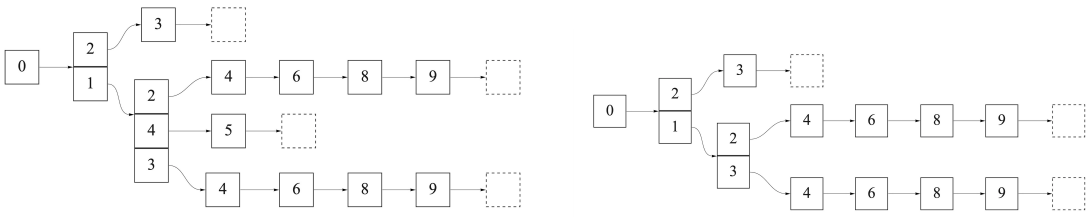
In spite of `cut_nth` being defined inductively using **primrec**, we simplify it in the proof using lemma 4.19 (**lemma** `cut_nth_take_drop_Suc`) more often than the original form.

Definition 4.20. (Cut a subtree)
```
fun Cut_subtree:: "rtree ⇒ nat list ⇒ nat ⇒ rtree"
  where "Cut_subtree (Tree l Fs) [] n = Tree l (cut_nth n Fs)" |
        "Cut_subtree (Tree l Fs) (k#t) n =
         Tree l
              ((take k Fs)@(Cut_subtree (Fs!k) t n)#(drop (Suc k) Fs))"
```

Similar to extending a tine, cutting a sub-tree results in a new tree that is a tree prefix of the first one.

Lemma 4.21.
```
lemma Cut_subtree_hd_prefix:
  assumes "tinesP F t" "t ≠ []"
  shows "Cut_subtree F (butlast t) (last t)  ≤ F"
```

Proving that function `Cut_subtree` preserves properties of forks is almost trivial. This is because we do not add any new vertices to the tree, resulting in the preservation of the depth of each vertex, the upper bound of vertices' labels the tree can have, and so on. However, unlike the case of using `Extend_tine` to extend a fork and preserve the forkness of the resulting tree, `Cut_subtree` is used mostly for changing a fork of one string to a fork of another shorter string. This involves many cases, which will be presented later when we discuss their usage for particular cases.



(a) (b)

Figure 4.3: Let `F` be `rtree` in 4.3a, `Cut_subtree F [1] 1` can be represented by `rtree` in 4.3b.

4.1.3 Non-negative margin closed forks

When we consider forkable strings from definition 3.22, we now only have one way to prove that a string is forkable: that is to find a flat fork of that string. There is, however, a more efficient way than an existential proof to verify if a certain string is a forkable string; it is through the margin of that string. This section explains how the margin of a string is related to forkability. The final goal is to formalise the following proposition:

Proposition 4.22. *A string w is forkable if and only if there is a closed fork $F \vdash w$ for which* $\text{margin}(F) \geq 0$.

While this preposition is easily proven in the Ouroboros paper with a half-page long proof, it takes more than 600 lines of formal proof excluding intermediate results, such as those involving reconstructing trees. We follow the steps in the original proof: each direction of the equivalence is proved by cases; a trivial case and a normal case.

Trivial cases
Trivial cases are when there are no honest parties selected to be a slot leader in n consecutive slots. It is not hard to see that the trivial fork, a fork with one node which is the root as described in section 3.8, is one of the forks of this string, and it is proved that the trivial fork is flat and closed. What is left is the proof that a trivial fork has a non-negative margin. This is not hard to see, since there is only one tine in a trivial fork which is an empty tine. Therefore, the margin of this fork is the reach of an empty tine, which is 0, hence the result.

Forkable string has non-negative-margin closed fork
Then, we focus first on the left-to-right side, which is

Lemma 4.23. (Proposition 4.22: left-to-right)
```
lemma forkable_string_imp_non_neg_margin:
  assumes "forkable w"
  shows "∃F. closed_fork F w ∧ margin F w ≥ 0"
```

Briefly, we assume we have a flat fork `F` of `w` from the definition of `forkable`, and then it is not hard to see that there is a closed prefix `F'` of `F` using this lemma:

Lemma 4.24. (Unique closed fork prefix)
```
lemma exist_closed_fork_prefix:
  assumes "F ⊢ w"
  shows "∃F'. closed_fork F' w ∧ F' ≤ F"
proof (the proof is omitted here)
```

Lastly, we argue that, in `F'`, there are edge-disjoint prefixes of the two longest edge-disjoint tines of `F`, which have non-negative reach values leading to the non-negative margin of `F'`.

The only tedious and tricky part is proving that prefixes of the longest tines have non-negative reach. First, we argue that `F'` has its unique longest tine `t` ending with the last honest index `m`, as it is formalised as this **lemma**:

Lemma 4.25. (Height of a closed fork)
```
lemma closed_fork_depth_max_honest_index_eq_height:
  assumes "closed_fork F w"
  shows "depth F (Max (honest_indices w)) = height F"
```

Then, we can see that the lengths of two prefixes `t1'` and `t2'`, in `F'`, of the longest tines `t1` and `t2`, respectively, in `F`, are less than or equal to the length of `t`. This is proven by cases, and it looks easy in the pen-and-paper proof. The trickiest case is when the two original tines, `t1` and `t2`, both contain honest indices rather than the root: we need to formalise that these two edge-disjoint tines have two edge-disjoint tine prefixes. This trickiness arises because of our formalisation of fork prefixes (`prefix`) cannot track which elements of the prefix list are the prefixes of certain elements of the suffix list. We declare a lemma specifically for this case to make the full proof look more readable.

Lemma 4.26.

```
lemma non_sharing_edged_tines_prefix:
  assumes "fork_prefix F F' w w" "closed_fork F w"
          "t1' ∈ tines F' ∧ t2' ∈ tines F' ∧ t1' /≅ t2'"
          "h1 ∈ set (trace_tine F' t1')" "h2 ∈ set (trace_tine F' t2')"
          "h1 ∈ honest_indices w"
          "h2 ∈ honest_indices w"
  shows "∃ t1 t2. t1 ∈ tines F ∧ t2 ∈ tines F ∧  t1 /≅ t2
         ∧ tine_prefix F F' t1 t1'
         ∧ tine_prefix F F' t2 t2' ∧ last (trace_tine F t1) = h1
         ∧ last (trace_tine F t2) = h2"
```

Therefore, after we formally prove that we have edge-disjoint tines `t1'` and `t2'` in the fork prefix `F'`, it is easy to see that they have `reserve`, the number of adversarial slots after the ending labels of tines, at least their `gap`.

Non-negative-margin closed fork implies forkable string
What is left to prove is the right-to-left side, which is

Lemma 4.27. (Proposition 4.22: right-to-left)

```
lemma non_neg_margin_imp_forkable:
  assumes "closed_fork F w" "margin F w ≥ 0"
  shows "forkable w"
```

This direction is more straightforward to prove and formalise than the other one. We have a closed fork `F` of `w` from the assumption, and we also know that `margin F w ≥ 0`. It follows that there are two edge-disjoint tines `t1` and `t2` that both have `reach` greater than or equal to `0`. From the definition of `reach`, for each of these two tines we know that there are enough adversarial slots reserved to extend it to reach the length of the longest tine in `F`. To do that, we extend tines `t1` and `t2` using function `Extend_tine` with the lists whose elements are from the sets of the adversarial indices left in `w` after the end of tines `t1` and `t2`, respectively. The extended parts of these two tines have to be strictly increasing lists which we know exist. In the pen-and-paper version, proving this is not required, but in the formal proof, we need to have instances of such lists, and the easiest way is to have the following lemma formalised.

Lemma 4.28.

```
lemma exist_strictly_inc_from_set:
  assumes "finite S"
  shows "∃l. (∀x ∈ set l. x ∈ S) ∧ size l = card S
         ∧ strictly_inc_list l"
```

Then, we can prove that the newly extended tree is a fork of the same string `w`. We have covered this with lemma 4.17 (`Extend_tine_with_reserve_preserves_forks`) and as its assumptions are met, the newly extended tree is a fork and it is flat. Finally, we can formalise lemma 4.22:

```
lemma forkable_iff_non_neg_margin:
"forkable w ⟷ (∃F. closed_fork F w ∧ margin F w ≥ 0)"
using forkable_imp_non_neg_margin non_neg_margin_imp_forkable
  by auto
```

4.1.4 Inductive margin on strings

Now that we have described and formalised how margin, or μ, is relevant to forkable strings, another result to be emphasised is the inductive definition of margin on strings. In the following, it describes how margin is inductively defined. Before we present this lemma, we define $\mathbf{m}$ to be a pair of ρ and μ, so that, for all strings w, $\mathbf{m}(w) = (\rho(w), \mu(w))$ with a straightforward formalisation: `m w ≡ (rho_s w, margin_s w)"`.

Lemma 4.29. $\mathbf{m}(\epsilon) = (0, 0)$ *and, for all nonempty strings* $w \in \{0,1\}^*$,

$$\mathbf{m}(w1) = (\rho(w) + 1, \mu(w) + 1),$$

and

$$\mathbf{m}(w0) = \begin{cases} (\rho(w) - 1, 0) & \text{if } \rho(w) > \mu(w) = 0, \\ (0, \mu(w) - 1) & \text{if } \rho(w) = 0, \\ (\rho(w) - 1, \mu(w) - 1) & \text{otherwise.} \end{cases}$$

Furthermore, for every string w*, there is a closed fork* $F_w \vdash w$ *for which* $\mathbf{m}(w) = (\rho(F_w), \mu(F_w))$.

For the readability, from now on we will mention such a closed fork as a representative closed fork, that is, F_w is a representative closed fork for w.

From lemma 4.29, we can see that the inductive definition is not for margin (μ) alone but, instead, for a pair of values ρ and μ; however, ultimately, ρ is never used again to prove other significant results; it is only defined to help inductively reason with margin values for strings. The original proof of this lemma has a length of two and a half pages and is the combination of a structural induction and case analysis with similar techniques used in Proposition 4.22. The formalisation takes around 4800 lines of code, so it is not rational to go into all the details of its formal proof, but instead we select some of the most complicated parts and describe them. As

presented in 4.1.3, the formalisation in this section also follows the steps of the pen-and-paper proof of lemma 4.29 in the Ourboros paper.

Basic step
The basic step is when w is an empty string (ϵ), so from the definition of fork, there is only one fork for this string, which is a trivial fork. Hence ρ and μ of the empty string are ρ and μ of a trivial fork, respectively, and, from 4.1.1, we can see that $\mathbf{m}(\epsilon) = (0, 0)$ and F_ϵ is a trivial fork.

Lemma 4.30. (Lemma 4.29: structural induction - basic step)
lemma `recursive_margin_Nil:`

```
"m [] = (0,0) ∧ (∃Fnil. m [] = (rho Fnil [], margin Fnil []))"
```

Inductive step
The inductive step of this lemma is quite complicated. First, we need to understand that there are two parts in this lemma for us to prove:

- that the value of $m(w)$ is stated correctly for w (by lemma 4.29), and
- that there exists a closed fork $F_w \vdash w$ that $\rho(F_w) = \rho(w)$ and $\mu(F_w) = \mu(w)$

To be precise, we phrase the inductive step as an argument as follows: "assuming that there exists a closed fork $F_w \vdash w$ that $\rho(F_w) = \rho(w)$ and $\mu(F_w) = \mu(w)$, the value of $m(wx)$ is stated correctly by lemma 4.29 for wx, and there exists a closed fork $F_{wx} \vdash wx$ that $\rho(F_{wx}) = \rho(wx)$ and $\mu(F_{wx}) = \mu(wx)$, where $x \in \{0, 1\}$". Therefore, we separate the formalisation of this argument into two main cases: x equals 1: characteristic strings end with an adversarial slot, and x equals 0: characteristic strings end with an honest slot.

Inductive step – strings ending with an adversarial slot
We start with the case where $x = 1$, so we aim to formalise that "assuming that there exists a closed fork $F_w \vdash w$ that $\rho(F_w) = \rho(w)$ and $\mu(F_w) = \mu(w)$, $m(w1) = (\rho(F_w)+1, \mu(F_w)+1)$, and there exists a closed fork $F_{w1} \vdash w1$ that $\rho(F_{w1}) = \rho(w1)$ and $\mu(F_{w1}) = \mu(w1)$". The final result of formalisation in the form of Isabelle lemma is as follows:

Lemma 4.31.
lemma `recursive_margin_1:`

```
  assumes "∃Fw ∈ closed_forks_set w. m w = (rho Fw w, margin Fw w)"
  shows "m (w@[True])= (rho_s w + 1, margin_s w + 1)
         ∧ (∃Fx ∈ closed_forks_set (w@[True]).
            m (w@[True]) = (rho Fx (w@[True]), margin Fx (w@[True])))"
```

The formalisation is quite direct as each closed fork of the string w (formalised

by `w`) is also a closed fork of the string $w1$ (formalised by `w@[True]`) and the other way around. This is because the slot number of the last adversarial slot added to w cannot appear in any closed fork of $w1$, since it will have to be a label for leaves due to the fact that forks are increasing trees. This can be summed-up by this lemma:

Lemma 4.32.
```
lemma closed_fork_adversarial_ending_eq_butlast :
  "closed_fork F (w@[True]) = closed_fork F w"
```

Consequently, for each tine t in a closed fork F of w and $w1$, t has the same gap for both w and $w1$. However, the reserve of t is increased by 1 if we consider F as a closed fork of $w1$ instead of w. Therefore, $reach_w(t) + 1 = reach_{w1}(t)$, where $reach_\square(t)$ is a reach of tine of a fork associated with a string $\square$ (however in a formalised version, `reach` is always obvious for the string it is associated with).

Lemma 4.33.
```
lemma reach_tine_fork_ending_True_plus1:
  assumes "closed_fork F w" "t ∈ tines F"
  shows "reach F w t + 1 = reach F (w@[True]) t"
```

Then, it is not hard to see how to prove lemma 4.31 (**lemma** `recursive_margin_1`) as `w` and `w@[True]` have the same representative closed fork, since the maximum ρ and margin values can still be selected from the same fork.

Inductive step – strings ending with an honest slot
The next case is where $x = 0$. This case is much more delicate than the first one and requires formalising new functions in order to reason properly with the relations between a fork and its prefixes.

For this case, there are three sub-cases:

- $\rho(w) > \mu(w)$ and $\mu(w) = 0$,
- $\rho(w) = 0$, or
- $\rho(w) < 0$

The formalisations of the three are in the same form as in the previous case with adversarial ending. We show below the formalisation of only the first sub-case:

Lemma 4.34.

```
lemma recursive_margin_0_case_1:
  assumes "∃Fx ∈ closed_forks_set w. m w = (rho Fx w, margin Fx w)"
          "rho_s w > margin_s w ∧ margin_s w = 0"
  shows "∃Fx ∈ closed_forks_set (w@[False]).
          m (w@[False]) = (rho Fx (w@[False]), margin Fx (w@[False]))
          ∧ m (w@[False])= (rho_s w - 1, 0)"
```

For the result for the first sub-case and those for the other two sub-cases, each takes around 500 lines of formal proof script excluding intermediate results, such as those involving properties of the functions `Extend_tine` and `Cut_subtree`. For each one of them, the proof has two steps:

1. proving that the lower bounds of both `rho_s` and `margin_s` are the desired values, and
2. proving that the upper bounds of both `rho_s` and `margin_s` are the desired values.

Proving (1) involves extending a representative closed fork `Fw` of `w`. We start from formalising that there is a pair of edge-disjoint tines in `Fw` that one of them, `t1`, has its `reach` equal to `rho` of `Fw` and another, `t2`, has its `reach` equal to `margin` of `Fw`.

Lemma 4.35.

```
lemma exist_two_tines_eq_margin_rho_not_equiv:
  assumes "closed_fork Fw w"
  shows "∃t1 t2. t1 ∈ tines Fw ∧ t2 ∈ tines Fw ∧ t1 /≅ t2
         ∧ reach Fw w t1 = rho Fw w ∧ reach Fw w t2 = margin Fw w"
```

Then we extend this fork to get a new tree that is a closed fork of `Fw0`. Depending on the sub-cases we are working with, we can extend this fork to have a new pair of edge-disjoint tines, `t1'` and `t2'`, that has been extended from `t1` and `t2` that has their `reach` equal to the desired quantities of those of `rho_s` and `margin_s` in lemma 4.29. Therefore, we can find the lower bounds of `rho_s` and `margin_s` as we wanted, because `rho_s` and `margin_s` of `Fw0` are going to be at least the reach of `t1'` and `t2'`, respectively.

Proving (2) is a reverse way of proving (1). More specifically, we assume `F*` as a representative closed fork for `w0`. Then, instead of extending it, we cut the longest tine containing the honest node with the greatest number (the number is always the length of `w0`) until the new tree is a closed fork for `w`. We can always do this as in the lemma presented below:

Lemma 4.36.

```
lemma closed_fork_wF_cut_subtree_closed_fork_w:
  assumes "tinesP F t" "t ≠ []" "closed_fork F (w@[False])"
    "∀x ∈ honest_indices w. count (mset_of_tree (get_subtree F t)) x = 0"
    "count (mset_of_tree (get_subtree F t)) (Suc (size w)) = 1"
    "last (trace_tine F (butlast t)) ∈ honest_indices w
     ∨ size (sucs (get_subtree F (butlast t))) > 1"
  shows "closed_fork (Cut_subtree F (butlast t) (last t)) w"
```

With similar reasoning as (1), we can get the upper bounds of `rho_s` and `margin_s` for `w0` as we wanted. These values match with the desired values in lemma 4.29. The representative closed fork is then the fork `Fw0` we used in the proof of (1).

4.2 Towards formalisation of the density of forkable strings

In this section, the goal is to formalise the upper bound of the forkable strings. We mentioned two different results as theorems 2.11 and 2.12, and we revisit them here as:

Theorem 4.37. *(Theorem 2.11 revisited) Let $\epsilon \in (0,1)$ and let w be a string drawn from $\{0,1\}^n$ by independently assigning each $w_i = 1$ with probability $(1-\epsilon)/2$. Then*

$$\Pr[w \text{ is forkable}] = 2^{-\Omega(\sqrt{n})},$$

Theorem 4.38. *(Theorem 2.12 revisited) Let $\epsilon \in (0,1)$ and let w be a string drawn from $\{0,1\}^n$ by independently assigning each $w_i = 1$ with probability $(1-\epsilon)/2$. Then*

$$\Pr[w \text{ is forkable}] = \exp(-2\epsilon^4(1 - O(\epsilon))n),$$

We have already discussed the different concepts adopted by the two results: Markov chains and supermartingale, respectively, both of which work to describe a stochastic process. The only stochastic process with which we are concerned in this project is Bernoulli process, which is a sequence of independent and identically distributed Bernoulli random variables. More specifically, if we consider an execution ε of the static stake protocol over n slots, the leader selection process results can be presented as a characteristic string $w = w_1 w_2 ... w_n$ of length n, and as the process selects the leader for each slot independently, w_i are independent random variables. If we are only interested in slot leaders' honesty, w_t can be considered a Bernoulli random variable, which is 0 when the leader of a slot sl_t is

honest, or 1 if adversarial. Furthermore, because in the static stake protocol, the stake transferred between these n slots does not affect the weight of the selection, all w_i are identically distributed.

The core ideas of proving theorems 4.37 and 4.38 are largely the same despite using different stochastic models. They use lemma 4.29 to derive the upper bound value of the expectation that margin is non-negative, and then use 4.22 to reason that it is the same as the expectation that strings are forkable.

In this project, we would like to select one of the two results to formalise. Eventually, we selected theorem 4.38. There are two main reasons why we selected the second result. Firstly, one can see that

$$\begin{aligned} 2^{-\Omega(\sqrt{n})} &= exp(-\Omega(\sqrt{n})) \\ &\geq exp(-\Omega(n)) \\ &= exp(-2\epsilon^4(1-O(\epsilon))n), \end{aligned} \tag{4.1}$$

which means theorem 4.38 has a tighter upper bound than that of theorem 4.37. For this reason, when we finish formalising theorem 4.38, we can imply formalisation of theorem 4.37. Secondly, the pen-and-paper proof of theorem 4.37 covers almost three pages and contains mostly proofs by exhaustion, while the pen-and-paper proof of theorem 4.38 is only a page long and concerns only changes of random variables' space and expectation of random variables. This difference of complication and length of the proofs will be emphasised when it comes to formalisation.

As this section presents the first description of the formalisation of probabilistic proofs, we will show it in a different fashion than what we did for the previous section. First, we will explain the proof of theorem 4.38 almost directly porting from Russell et al's work [22] in 4.2.1. Then, we go through the steps in the proof needed to be formalised in 4.2.2 and 4.2.5. For that, we will explain some functions that appear for the first time in this thesis and are specifically used to formalise probability theory; we will discuss mainly how they practically facilitate our proofs and not go into deeper details of measure theory, which they are actually based on. However, in 4.2.6, we will discuss issues we encounter while trying to formalise the density of forkable strings and what we need to compromise and assume without proofs in order to proceed to formalise further important results in the Ouroboros paper. Lastly, we finished the formalisation of theorem 4.38 using the results we assume in 4.2.6.

4.2.1 Forkable strings are rare: original proof

The core of the proof for theorem 4.38 is to find a sequence of random variables that is supermartingale, and then apply it with Azuma-Hoeffding's inequality (stated below) to get an upper bound of forkable strings as we desire. The sketch of the original pen-and-paper proof of theorem 4.38 from Russell et al's paper [22] is described below.

We need to stress that the proof below is entirely not our work. It is almost identically ported from Russell et al's paper in order to see exactly what formalisation tasks need to be done in order to formalise the final result. [22]

Theorem 4.39. (Azuma; Hoeffding. See [33] for discussions) *Let $X_0, ..., X_n$ be a sequence of real-valued random variables so that, for all t, $\mathbb{E}[X_{t+1}|X_0, ..., X_t] \leq X_t$ and $|X_{t+1} - X_t| \leq c$ for some constant c. Then for every $\Lambda \geq 0$,*

$$\Pr[X_n - X_0 \geq \Lambda] \leq \exp\left(-\frac{\Lambda^2}{2nc^2}\right)$$

We need some steps to explain how random variables that fit the assumptions of theorem 4.39 are selected. Firstly, we define random variables $\rho_t = \rho(w_1...w_t)$ and $\mu_t = \mu(w_1...w_t) = margin(w_1...w_t)$, where w_i is a $\{0, 1\}$-random variable representing the honest/adversary slot leader in a slot sl_i and ρ and $margin(\mu)$ are definitions from 4.1.1.

As we know the honest stake ratio has an advantage ϵ against the adversarial stake ratio, $\Pr[w_i = 0] = (1+\epsilon)/2$, and for the sake of convenience later on in the proof, we define the associated $\{\pm 1\}$-valued random variables $W_t = (-1)^{(1+w_t)}$, which makes $\mathbb{E}[W_t] = -\epsilon$.

We then define the ancillary random variables $\overline{\mu}_t = min(0, \mu_t)$. Next, we define the random variables $\Phi_t \in \mathbb{R}$,

$$\Phi_t = \rho_t + \alpha\overline{\mu}_t \tag{4.2}$$

Where ϵ is the advantage of the honest stake ratio against the adversarial stake ratio and $\alpha = (1+\epsilon)/(2\epsilon) \geq 1$.

Let $\Delta_t \in [-(1+\alpha), 1+\alpha]$; $\Delta_t = \Phi_t - \Phi_{t-1}$, $\widetilde{\Phi}_t = \Phi_t + \epsilon t$, and $\widetilde{\Delta}_t = \Delta_t + \epsilon$. It follows that

$$|\widetilde{\Delta}_t| \leq 1 + \alpha + \epsilon \tag{4.3}$$

Then, we can calculate the expectation of $\widetilde{\Phi}_{t+1}$ conditioned on $w_1...w_t$ by consid-

ering possible fixed values $w'_1...w'_t$ for $w_1...w_t$ (from this, we can assume we know the values of $\widetilde{\Phi}_t$ and (ρ, μ) since these random variables only depend on $w_1...w_t$, so we let $\widetilde{\Phi}_t = \widetilde{\Phi}$ and $(\rho_t, \mu_t) = (\rho, \mu)$)

$$\begin{aligned}\mathbb{E}[\widetilde{\Phi}_{t+1}|w_1, ..., w_t] &= \widetilde{\Phi} + \mathbb{E}[\widetilde{\Delta}_{t+1}|w_1, ..., w_t] \\ &= \widetilde{\Phi} + \mathbb{E}[\Delta_{t+1} + \epsilon|w_1, ..., w_t] \\ &= \widetilde{\Phi} + \mathbb{E}[\Delta_{t+1}|w_1, ..., w_t] + \epsilon\end{aligned} \tag{4.4}$$

From here on, we can analyse the four cases that $\mathbb{E}[\Delta_{t+1}|w_1, ..., w_t] \leq -\epsilon$ using lemma 4.29 describing how ρ and $margin\,(\mu)$ can be calculated inductively:

1. when $\rho > 0$ and $\mu < 0$, $\rho_t + 1 = \rho + w_{t+1}$ and $\overline{\mu}_{t+1} = \overline{\mu} + W_{t+1}$, where $\overline{\mu} = min(0, \mu)$; then $\Delta_{t+1} = (1+\alpha)W_{t+1}$ and $E[\Delta_{t+1}|w_1, ..., w_t] = -(1+\alpha)\epsilon \leq -\epsilon$,
2. when $\rho > 0$ and $\mu \geq 0$, $\rho_t + 1 = \rho + w_{t+1}$ but $\overline{\mu}_{t+1} = \overline{\mu}$ so that $\Delta_{t+1} = W_{t+1}$ and $E[\Delta_{t+1}|w_1, ..., w_t] = -\epsilon$,
3. when $\rho = 0$ and $\mu < 0$, $\overline{\mu}_{t+1} = \overline{\mu} + W_{t+1}$ while $\rho_{t+1} = \rho + max(0, W_{t+1}) = \rho + w_{t+1}$; we may compute

$$\mathbb{E}[\Delta_{t+1}|w_1, ..., w_t] = \frac{1-\epsilon}{2} - \epsilon\alpha = \frac{1-\epsilon}{2} - \epsilon\left(\frac{1}{\epsilon} \cdot \frac{1+\epsilon}{2}\right) = -\epsilon$$

   ,and
4. when $\rho = \mu = 0$ exactly, one of the two random variables ρ_{t+1} and μ_{t+1} differs from zero: if $W_{t+1} = 1$ then $(\rho_{t+1}, \overline{\mu}_{t+1}) = (1, 0)$; likewise, if $W_{t+1} = -1$ then $(\rho_{t+1}, \overline{\mu}_{t+1}) = (0, -1)$. It follows that

$$\mathbb{E}[\Delta_{t+1}|w_1, ..., w_t] = \frac{1-\epsilon}{2} - \left(\frac{1+\epsilon}{2}\right)\alpha \leq \epsilon$$

   .

We can then conclude from 4.4 that

$$\mathbb{E}[\widetilde{\Phi}_{t+1}|w_1, ..., w_t] \leq \widetilde{\Phi} + (-\epsilon) + \epsilon = \widetilde{\Phi} \tag{4.5}$$

Therefore,

$$\mathbb{E}[\widetilde{\Phi}_{t+1}|\widetilde{\Phi}_1, ..., \widetilde{\Phi}_t] = \mathbb{E}[\widetilde{\Phi}_{t+1}|w_1, ..., w_t] \leq \widetilde{\Phi}_t, \tag{4.6}$$

Hence, the random variables $\widetilde{\Phi}_t$ will form the desired sequence of random variables for Azuma-Hoeffding's inequality. Next, from Azuma-Hoeffding's inequality we

derive:

$$\begin{aligned}
\Pr[w \text{ is forkable}] &= \Pr[\overline{\mu}_n = 0] \\
&\leq \Pr[\Phi_n \geq 0] \\
&= \Pr[\widetilde{\Phi}_n \geq \epsilon n] \\
&\leq \exp\left(-\frac{\epsilon^2 n^2}{2n(1+\alpha+\epsilon)^2}\right) \\
&\leq \exp\left(-\frac{2\epsilon^4}{1+35\epsilon} \cdot n\right)
\end{aligned} \tag{4.7}$$

Analysis for necessary tasks – to finish up to the final result – theorem 4.38) – there are many formalisation tasks to be done following the order they appear in the original pen-and-paper proof:

1. Random variables. There are several ways to formalise random variables. However, the way we selected here is using type `'a pmf` that formalises probability mass functions because they are enough for discrete random variables as all random variabled concerned in this project are discrete.

2. Independent and identically distributed (i.i.d.) random variables. This type of random variables are defined from finite number of identical random variables. This can be done by using theory `pi_pmf` along with already-formalised variables from task 1.

3. Conditional expectations. Conditional expectations are formalised in Isabelle/HOL, and it is for both continuous and discrete random variables. However, it is involved measure theory such as proving measurability of random variables (in our case, the type `'a pmf`) which is a complication we would like to avoid. The different path we pick is to we mechanise conditional expectations specifically for the type `'a pmf`. In long run, it is more sustainable method to go through, but the elaborative library containing theorems to simplify calculations regarding conditional expectations is still needed. For this reason, we think it is reasonable to leave this for future work.

4. Azuma-Hoeffding's inequality. There is no formalisation of this theorem in Isabelle/HOL. This theorem has pre-conditions described in form of conditional expectations which needs task 3 to be finished first. Therefore, we also leave this for future work.

4.2.2 Formalising relevant discrete random variables

Discrete random variables and their distributions can be described together by probability mass functions. From the previous section, we can see that all random variables in the proof are based on random variables w_i, where $i \in \{1, ..., n\}$, which have the Bernoulli distribution with $p = (1-\epsilon)/2$, where $\epsilon \in (0, 1]$. The probability mass function of the Bernoulli distribution on Boolean values has already been formalised in Isabelle/HOL in theory `Probability_Mass_Function` as follows:

Definition 4.40. (Bernoulli distribution)
lift_definition `bernoulli_pmf :: "real ⇒ bool pmf" is`
`"λp b. ((λp. if b then p else 1 - p) ∘ min 1 ∘ max 0) p"`

In Isabelle/HOL, the probability mass function is formalised by a type `'a pmf`. `bernoulli_pmf p :: bool pmf` represents the probability mass function that maps `True` to `min 1 (max 0 p)` and `False` to `1-(min 1 (max 0 p))`, or to be written in lemmas as follows:

Lemma 4.41.
lemma `pmf_bernoulli_True: "0 ≤ p ⟹ p ≤ 1`
`⟹ pmf (bernoulli_pmf p) True = p"`

Lemma 4.42.
lemma `pmf_bernoulli_False: "0 ≤ p ⟹ p ≤ 1`
`⟹ pmf (bernoulli_pmf p) False = 1 - p"`

Therefore, `bernoulli_pmf p` can only represent the Bernoulli distribution with parameter p when `p` is between `0` and `1` inclusively. It can be noticed that although we stated that `bernoulli_pmf p` represents a "function", we actually need to use `pmf` in the front of `bernoulli_pmf` to use it as a function. This is because the type `'a pmf` also guarantees that its instances operate on valid measure spaces, which cannot be guaranteed by only providing a function alone (in the Bernoulli case, the concerned function is the function "λp. if b then p else 1 - p) ∘ min 1 ∘ max 0) p" in the definition of `bernoulli_pmf`).

We can use `set_pmf` to get the set of the measure space of our interest. This set simply collects all values that result in positive numbers when applying with `pmf`.

Lemma 4.43.
lemma `set_pmf_bernoulli:`
`"0 < p ⟹ p < 1 ⟹ set_pmf (bernoulli_pmf p) = UNIV"`

In this case, `UNIV` is `{True, False}` (a set of all instance of type `bool`). In some cases, `set_pmf` does not return `UNIV`; for example, `set_pmf bernoulli_pmf 1` would be equal to `{True}` as `pmf (bernoulli_pmf 1) False = 0`, so `False` is not in `set_pmf`.

Another important function to mention is `measure`. When applied to `M :: 'a pmf`, this function returns a measure for the measure space that `M` takes values on. We can see, for example, how `measure` works on `bernoulli_pmf p` through these four lemmas:

Example 4.44.

lemma "0 ≤ p ⟹ p ≤ 1 ⟹ measure (bernoulli_pmf p) {} = 0"

lemma "0 ≤ p ⟹ p ≤ 1 ⟹ measure (bernoulli_pmf p) {True} = p"

lemma "0 ≤ p ⟹ p ≤ 1 ⟹ measure (bernoulli_pmf p) {False} = 1 - p"

lemma "0 ≤ p ⟹ p ≤ 1 ⟹ measure (bernoulli_pmf p) {True,False} = 1"

(Note that we created these four lemmas just to use them as examples here; they have no use for any other purposes.)

We can start formalising each random variable w_i which has $Pr[w_i = 1] = (1 - \epsilon)/2$; we start by defining ϵ and α in terms of ϵ to be used through the whole formalisation:

context
  fixes ε::real **assumes** ep_gt_0 : "0 < ε" **and** ep_le_1: "ε ≤ 1"
  fixes α::real **assumes** α: "α = (1 + ε)/(2*ε)"

Then, `bernoulli_pmf ((1-ε)/2))` can be used to represent w_i, except that the set of a measure space is still `{True, False}`. We can change its measure space from `{True, False}` to `{1,0}` by the function `map_pmf` as follows:

Example 4.45.

definition w_n_pmf :: "nat pmf" **where**
 "w_n_pmf = map_pmf (λb. (if b then 1 else 0)) (bernoulli_pmf ((1-ε)/2))"

We will not explain in detail how `map_pmf` is defined as it depends on theory `Giry_Monad`, which is beyond the scope of this project, but, briefly, it maps a set of a measure space to a new set and changes other elements in the measure space (a σ-algebra and a measure) accordingly.

4.2.3 Formalising independent and identically distributed Bernoulli random variables

Although we can formalise Bernoulli random variables, it is not straightforward to formalise n independent and identically distributed (i.i.d.) Bernoulli random variables, since there is no function that can automatically do so in the Isabelle/HOL library yet. Therefore, we discuss two potential options of formalising it:

1. Induction — we can use `map_pmf` inductively to formalise a fixed-length list of i.i.d. Bernoulli random variables:

   Definition 4.46.
   ```
   fun wn :: "nat ⇒ (bool list) pmf" where
      "wn 0 = return_pmf []"
    | "wn (Suc n) = map_pmf (λ(x,y). x # y)
                            (pair_pmf (bernoulli_pmf ((1-ε)/2)) (wn n))"
   ```

   The function `pair_pmf` works as its name suggests: it pairs two random variables and their distributions together independently and calculate a new measure space from both of its argument as shown in these lemmas from theory `Probability_Mass_Function`:

   Lemma 4.47.
   ```
   lemma pmf_pair: "pmf (pair_pmf M N) (a, b) = pmf M a * pmf N b"
   ```

   Lemma 4.48.
   ```
   lemma set_pair_pmf: "set_pmf (pair_pmf A B)
                               = set_pmf A × set_pmf B"
   ```

   The function `wn` has a type of `(bool list) pmf`, so to access the i-th Bernoulli random variable, one needs to use `map_pmf (λl. nth l i) (wn n)`; here, `i` can range from `0` to `n-1`.

2. Index function — Eberl has developed a theory devoted solely to building an indexed collection of probability mass functions in which each of them can differ from others by its index (as part of a verification of the Skip Lists data structure [39]). The relevant operator we use is `Pi_pmf`:

   ```
   definition Pi_pmf :: "'a set ⇒ 'b ⇒ ('a ⇒ 'b pmf) ⇒ ('a ⇒ 'b) pmf"
   where
     "Pi_pmf A dflt p =
        embed_pmf (λf. if (∀x. x ∉ A ⟶ f x = dflt)
                       then ∏x∈A. pmf (p x) (f x) else 0)"
   ```

The operator `Pi_pmf` has three arguments as follows:

- `A :: 'a set`: an index set,
- `dflt :: 'b`: a default value that out-of-index probability mass functions take, and
- `p :: ('a ⇒ 'b pmf)`: a function that accepts an index value and returns a probability mass function according to the index.

Therefore, in order to create a collection of i.i.d. random variables from `Pi_pmf`, the third argument, `p`, will just ignore its input (index) and return the same probability mass function. We can see how we define `w_i_pmf` below:

Definition 4.49.
definition `w_i_pmf :: "nat ⇒ (nat ⇒ bool) pmf"`
where `"w_i_pmf n = Pi_pmf {..<n} False (λ_. bernoulli_pmf ((1-ε)/2))"`

For the sake of simplicity, we assigned the index set to be `{..<n}` in the case of n random variables. Next, we can use `map_pmf` to define `w_pmf`:

Definition 4.50.
definition `w_pmf :: "nat ⇒ (bool list) pmf"`
where `"w_pmf n = map_pmf (λf. map f [0..<n]) (w_i_pmf n)"`

Having considered both options of formalising a sequence of i.i.d. random variables, we have decided to use the second option: theory `Pi_pmf`. This is because it comes with lemmas to reason with measure theory proofs for each random variable in the sequence separately or altogether. In the former option, although it is less of a black box and is more applicable to our work, it is a rather laborious process to further declare and prove those lemmas without making our formalisation repetitive.

4.2.4 Formalising sequences of ρ_t, μ_t, and Φ_t

From the previous subsection, we can formalise other random variables which are defined on top of w with the help of `map_pmf`. Our aim below is to find functions to use with `map_pmf` to apply with `w_pmf n` from definition 4.50 to derive the formalisation for ρ_t, μ_t, Φ_t, and ultimately $\widetilde{\Phi}_t$ from the proof in section 4.2.1.

For the first two sequences, `rho_s` and `margin_s` from section 4.1 can be used almost directly. However, the way these two functions are defined is tied to the structure of `forks`. Even though we have already proved that they can be calculated inductively by lemma 4.29, defining a new one using **fun** offers an advantage to use induction proofs directly:

Definition 4.51.

```
fun rev_m :: "bool list ⇒ int × int"
 where "rev_m [] = (0,0)"
 |"rev_m (True#w) = (fst (rev_m w) + 1, snd (rev_m w) + 1)"
 |"rev_m (False#w) =
  (if fst (rev_m w) > snd (rev_m w) ∧ snd (rev_m w) = 0
  then (fst (rev_m w) - 1, 0)
  else
    (if fst (rev_m w) = 0
    then (0, snd (rev_m w) - 1)
    else (fst (rev_m w) - 1, snd (rev_m w) - 1)))"
```

It is obvious that `rev_m` is defined following the result of lemma 4.29; a small but important change is applied, however, as `rev_m` calculates reversely to what the original definition of **m** does. This is because in order to prove lemmas involving `rev_m` by induction on its input, defining `rev_m` inductively on `bool list` structure is a more intuitive approach. To ensure that `rev_m` works correctly as an alternative formalisation for **m**, we have proved this lemma:

Lemma 4.52.

```
lemma rev_m_m_rev:"rev_m l = m (rev l)"
```

The four following intermediate functions constructed from `rev_m` are to be used with `map_pmf` and `w_pmf` to formalise sequences of these random variables Φ_t, $\widetilde{\Phi}_t$, Δ_t, and $\widetilde{\Delta}_t$, respectively:

Definition 4.53.

```
definition Φ_n :: "nat ⇒ bool list ⇒ real" where
  "Φ_n n l = (λp. real_of_int (fst p)
              + α * (real_of_int (min 0 (snd p))))
                (rev_m (drop (size l - n) l))"
```

Definition 4.54.

```
definition Φ_n' :: "nat ⇒ bool list ⇒ real" where
  "Φ_n' n l = (λp. real_of_int (fst p)
               + α * (real_of_int (min 0 (snd p))))
                 (rev_m (drop (size l - n) l)) + n * ε"
```

Definition 4.55.

```
definition Δ_n :: "nat ⇒ bool list ⇒ real" where
  "Δ_n n l = Φ_n n l - Φ_n (n-1) l"
```

Definition 4.56.
definition Δ_n' :: "nat ⇒ bool list ⇒ real" **where**
"Δ_n' n l = Φ_n n l - Φ_n (n-1) l + ε"

To formalise random variables, only the second and the forth functions are of our interest, since other random variables are only for intermediate purposes. Therefore, the formalisation of the sequence of $\widetilde{\Phi}_t$, and the sequence of $\widetilde{\Delta}_t$ can be done as follows:

Definition 4.57.
definition Φ_n_pmf' **where**
"Φ_n_pmf' n = map_pmf (λx. Φ_n' n x) (w_pmf n)"

Definition 4.58. definition Δ_n_pmf' **where**
"Δ_n_pmf' n = map_pmf (λx. Δ_n' n x) (w_pmf n)"

4.2.5 Expectations and conditional expectations

One of the assumptions of the Azuma-Hoeffding's inequality that needed to be satisfied in order to prove theorem 4.38 is that the sequence of $\widetilde{\Phi}_t$ is supermartingale, so conditional expectations need to be mechanised. Theory `Conditional_Expectation` in Isabelle/HOL-Probability formalises conditional expectations, but its use is not quite suitable for the type `pmf`, as it requires measurability proofs which we wish to avoid in this project. Therefore, it is more appropriate in our case to use resources available in theory `Probability_Mass_Function` to formalise conditional expectations.

The conditional expectation of X given $Y = y$ is,

$$\begin{aligned}\mathbb{E}[X|Y=y] &= \sum_{x\in\Omega_X} x\Pr(X=x|Y=y)\\ &= \sum_{x\in\Omega_X} x\frac{\Pr(X=x,Y=y)}{\Pr(Y=y)}\\ &= \begin{cases}\mathbb{E}(X\mathbb{1}_{Y=y}(Y))/\Pr(Y=y) & ;\Pr(Y=y)\neq 0\\ 0 & ;\Pr(Y=y)=0\end{cases}\end{aligned} \tag{4.8}$$

A formalisation of conditional expectations can be done as follows[1]:

[1]This definition is originally a suggestion by Andreas Lochbihler in the discussion through the Isabelle mailing list under the topic "(Super-)Martingale and conditional expectation"

Definition 4.59. (Conditional Expectation)
definition `cond_exp :: "'a pmf ⇒ ('a ⇒ real) ⇒ ('a ⇒ 'b) ⇒ 'b ⇒ real"`
where
`"cond_exp p X Y y = measure_pmf.expectation (cond_pmf p (Y -‘ {y})) X"`

The function `cond_exp` has four arguments as follows:

- `p :: 'a pmf` represents a joint distribution for all random variables involved (i.e. X and Y),
- `X :: 'a ⇒ real` represents a formalisation of the real-valued random variable X by mapping from the set of measure space in the joint distribution `p` to real numbers,
- `Y :: 'a ⇒ 'b` represents a formalisation of the random variable Y and the set of measure space in the joint distribution `p` to `'b`, where `'b` is a type that represents what the set Y takes values from, and
- `y :: 'b` represents a value y that we would like to find for an expectation of X, given that $Y = y$.

The functions we have not introduced before in the definition of `cond_exp` are `measure_pmf.expectation` and `cond_pmf` (defined in theory `Probability_Mass_Function`). Similar to when we introduced `map_pmf`, we omit the details on how they are formalised. Instead we present two lemmas from theory `Probability_Mass_Function` which are significant for our context:

Lemma 4.60.
lemma `integral_measure_pmf:`
fixes `f :: "'a ⇒ 'b::{banach, second_countable_topology}"`
assumes `A: "finite A"`
shows `"(⋀a. a ∈ set_pmf M ⟹ f a ≠ 0 ⟹ a ∈ A)`
`⟹ (measure_pmf.expectation M f) = (∑a∈A. pmf M a *R f a)"`

Lemma 4.60 basically explains that the function `measure_pmf.expectation` formalises $\mathbb{E}$, but it needs to work on two arguments to ensure that it calculates the expectation of a real-valued random variable: a distribution (of a type `'a pmf`) and a real-valued random variable (of type `'a ⇒ real`).

Lemma 4.61. **lemma** `pmf_cond:`
fixes `p :: "'a pmf"` **and** `s :: "'a set"`
assumes `not_empty: "set_pmf p ∩ s ≠ {}"`
shows `"pmf cond_pmf x = (if x ∈ s then pmf p x / measure p s else 0)"`

We need to clarify that, in theory `Probability_Mass_Function`, lemma 4.61 is not shown exactly as presented here, but the fixed variables (`p` and `s`) and the assumption (`not_empty`) are provided for the whole section related to `cond_pmf`: this lemma is one of many results that use these variables in their statements. When this lemma is considered by the cases with lemma 4.60, shows the formalisation by splitting the cases in the last line of equation 4.8.

4.2.6 Issues regarding conditional expectations

Although we have established how conditional expectations can be formalised for discrete random variables, we lack an elaborative library to simplify expressions with conditional expectations. The absence of such a library causes some tasks in our project to be too exhaustive to finish. This can compare to the advantages we gained from having theory `Pi_pmf` in Isabelle. Without theory `Pi_pmf`, formalising any products of a collection of probability mass functions would have been significantly more laborious.

Due to lack of such library, we are listing some of our necessary results we assert and assume and assume without proofs in order to finish formalising theorems 4.38. Isabelle command **sorry** is used to indicate an unfinished proof a current goal as true.

4.2.6.1 Azuma-Hoeffding's inequality

Formalising Azuma-Hoeffding's inequality as a side project seems unrealistic given that it is in the different scope from our work. This is because we do not have enough materials to work from preconditions as they are described using conditional expectations. We have decided to leave this theorem for future formalisation work or to be done by a person or a group specialising in probability theory.

Our formalised version of Azuma-Hoeffding's inequality that works for type `'a pmf` we assume is as follows:

Theorem 4.62.
theorem `Azuma_Hoeffding_Inequality:`
assumes
`"⋀i x. x ∈ set_pmf P ∧ i < n ⟹`
`cond_exp P (X (i+1))`
`(λw. map (λj. X j w) [0..<i])`
`((λw. map (λj. X j w) [0..<i]) x) ≤ X i x"`
`"⋀i x. x ∈ set_pmf P ∧ i < n ⟹ abs (X (i + 1) x - X i x) ≤ c "`
shows `"⋀Λ. 𝒫(x in P. X n x - X 0 x ≥ Λ) = exp (- (Λ * Λ) / (2 *`
`(n::real) * c))"`
sorry

We previously use `cond_exp` provided in definition 4.59 to state this theorem. The assumption part of this theorem contains two statements:

1. the sequence of random variables `X :: nat ⇒ 'a ⇒ real` mapping from a joint distribution `P :: 'a pmf` to `real` is supermartingale. `X i :: 'a⇒ real` represents each random variable with `P :: 'a pmf`.
2. the absolute value of the difference of each adjacent random variable `X (i + 1)` and `X i` under the same event (`x`) has a boundary.

4.2.6.2 Fulfilling Azuma-Hoeffding's inequality assumptions for our results

We need to formalise equations 4.3 and 4.6:

1. the difference between an adjacent pair of random variables $\widetilde{\Phi}_{i+1}$ and $\widetilde{\Phi}_i$ is defined by a random variable $\widetilde{\Delta}_{i+1}$. We can formalise an upper and a lower bound of $\widetilde{\Delta}_{i+1}$, which is formalised in 4.2.4, by the following lemmas:

   Lemma 4.63. **lemma** `set_pmf_Δ_n_pmf':`
   assumes `"a ∈ set_pmf (Δ_n_pmf' n)"`
   shows `"a ≤ 1 + α + ε ∨ a ≥ - (1 + α) + ε"`

   lemma `abs_set_pmf_Δ_n_pmf':`
   assumes `"a ∈ set_pmf (Δ_n_pmf' n)"`
   shows `"abs a ≤ 1 + α + ε"`

2. The following two lemmas mechanise equation 4.6:

Lemma 4.64.

```
lemma Φ'_supermartingale_1:
  assumes "w ∈ set_pmf (w_pmf n)" "t<n"
  shows "(cond_exp (w_pmf n)
                   (Φ_n' (t + 1))
                   (λw. map (λx. Φ_n' (nat x) w) [0..<t])
                   (map (λx. Φ_n' (nat x) w) [0..<t])) =
         (cond_exp (w_pmf n)
                   (Φ_n' (t + 1))
                   (λw. map (λx. w ! nat x) [0..<t])
                   (map (λx. w ! nat x) [0..<t]))"
sorry
```

Lemma 4.65.

```
lemma Φ'_supermartingale_2:
  assumes "w ∈ set_pmf (w_pmf n)" "t < n"
  shows "cond_exp (w_pmf n)
                  (Φ_n' (t + 1))
                  (λw. map (λx. (rev w) ! (nat x) ) [0..<t])
                  (map (λx. (rev w) ! (nat x)) [0..<t])
         ≤ Φ_n' t  w"
sorry
```

Lemma 4.64 can help prove 4.65 following the order of sub-equations appearing in equation 4.6. The difficulties we are facing are on reasoning for getting expectation values when using lemma 4.60 and lemma 4.61 to unfold expression with `cond_exp`.

4.2.7 Formalisation of the density of forkable strings

To formalise theorem 4.38, we can simply substitute all variables in theorem 4.39 with those in theorems 4.64 and 4.65 as the preconditions of Azuma-Hoeffding's inequality are satisfied, so we get this formalised probability:

Lemma 4.66.

```
lemma Azuma_Hoeffding_Inequality_applied_rev:
  fixes n::nat
  shows  "⋀Λ. 𝒫(x in w_pmf n. Φ_n' n (rev x) - Φ_n' 0 (rev x) ≥ Λ)
          = exp (- (Λ * Λ) / (2 * (n::real) * (1 + α + ε)))"
```

Then, we can use the result above with Λ `= n *` ε to formalise the series of results following inequality 4.7:

Lemma 4.67.
lemma Φ_ge_0_cal:
fixes n::nat
shows "$\mathcal{P}$(w in w_pmf n. Φ_n n (rev w) ≥ 0)
= exp (- ((real n) * ε * (real n) * ε)
/ (2 * (n::real) * (1 + α + ε)))"

Lemma 4.68.
lemma forkable_prob_prep:
"$\mathcal{P}$(w in w_pmf n. rev_μ_bar (rev w) = 0)
≤ exp (- ((real n) * ε * (real n) * ε)
/ (2 * (n::real) * (1 + α + ε)))"

Lemma 4.69.
lemma forkable_eq_rev_μ:
"$\mathcal{P}$(w in w_pmf n. forkable w) = $\mathcal{P}$(w in w_pmf n. rev_μ_bar (rev w) = 0)"

Lemma 4.70. Formalisation of inequality 4.7
lemma forkable_prob_lem:
"$\mathcal{P}$(w in w_pmf n. forkable w)
≤ exp (- ((real n) * ε * (real n) * ε)
/ (2 * (n::real) * (1 + α + ε)))"

We can see that this does not involve the Landau symbol Ω presented in inequality 4.7. We omit the use of Landau symbol in our formalisation as we will use this result to continue our formal proof furthermore, and Landau symbols will only hide necessary details. Instead, we define c0 as a local constant, so the formalisation looks more concise and cleaner:

context
fixes c0::real **assumes** "c0 = (ε ^ 2) / (2 * (1 + α + ε))"

This simplified result is formalised below:

Lemma 4.71. Density of forkable strings: simplified
lemma forkable_prob_lem_simplified:
fixes n
assumes "n >0"
shows "$\mathcal{P}$(w in w_pmf n. forkable w) ≤ exp (- (real n) * c0)"

4.2.7.1 Generalising forkable string's density

As `w_pmf L` is a string of `L` i.i.d. random variables, any `n`-length sub-strings have the same distribution which is `w_pmf n`. We first formalise this property for the prefix and suffix sub-strings:

Lemma 4.72. Probability on sub-string prefixes
```
lemma function_prob_eq_prefix:
  fixes a::nat and L::nat and f
  assumes "L ≥ 0" "a > 0" "a ≤ L"
  shows "𝒫(x in w_pmf L.  f (take a x)) = 𝒫(x in w_pmf a.  f x)"
```

Lemma 4.73. Probability on sub-string suffixes
```
lemma function_prob_eq_suffix:
  fixes a::nat and L::nat and f
  assumes "L ≥ 0" "a ≤ L"
  shows "𝒫(x in w_pmf L.  f (drop a x)) = 𝒫(x in w_pmf (L - a).  f x)"
```

The technique used to formalise these two lemmas are as follows: 1) unpack definition of probability, 2) formalise that for the part of the string that does not concern in lemma does affect the probability, and 3) simplify the result to probability form again[2]. Then we can combine them and substitute a generic function `f` to `forkable` to have the complete generalised lemma:

Lemma 4.74. Probability on sub-strings
```
lemma forkable_prob_eq_trim:
  fixes a::nat and  b::nat and L::nat
  assumes "L ≥ 0" " a < b" "b ≤ L"
  shows "𝒫(x in w_pmf L.  forkable (drop a (take b x)))
         = 𝒫(x in w_pmf (b-a).  forkable x)"
```

Therefore, we can generalise lemma 4.71 to the following:

Lemma 4.75. Density of forkable sub-strings
```
lemma forkable_prob:
  assumes "L ≥ 0" "a < b" "b ≤ L"
  shows "𝒫(x in w_pmf L.  forkable (drop a (take b x)))
         ≤ exp (- c0*(b-a))"
```

This is the final version that we will use onwards in other formal proofs requiring

[2]This technique could be considered to be useful in future work for formalising our lemma related to conditional expectations.

forkable strings probability.

4.3 Chapter summary

We have presented two main formalisation tasks important to formalise the density of forkable strings: the intermediate results concerning discrete constructions, and the probabilistic part. All the intermediate results are to be used to formalise the probabilistic result. The formalisations in this chapter were given along with brief descriptions summarised from the original definitions and proofs.

To begin with intermediate results, margin was formalised (along with ρ); this requires the formalisation of other quantities namely gap, reserve, and reach. While margin and ρ are bound with a fork, other values are bound with a tine. However, margin and ρ can be redefined to be functions of characteristic strings instead of functions of forks. Next, we redirected slightly to construct useful functions to reconstruct trees. In our complete formal proof script, we do have numerous functions in this category, some of them were chosen to be displayed based on their necessity for formalising margin a function inductively defined on characteristic strings. The main functions we presented are `Extend_tine` and `Cut_subtree`, and they were presented with lemmas facilitating their usages in the formal proof.

As we have formalised margin, we proceeded to formalise two important results. First, we presented how we formalised the connection between margin and forkability by lemma 4.22. Then, we have given a brief formalisation of how margin can be constructed inductively through lemma 4.29. This lemma is a good example of how complicated our formal proof can get since the original proof is two and half pages long, but the formal proof comprises a few thousand lines of code.

Continuing from intermediate results, we described the attempt to formalise the probabilistic part of the density of forkable strings result. We started with a brief analysis of and a comparison between the two different results: lemma 4.37 and lemma 4.38, from both probabilistic and formalisation perspectives, and we selected to formalise the latter. We then almost ported the pen-and-paper proof from Russell et al's "Forkable Strings are Rare" paper [22] with a slight difference to better fit the context of our thesis.

The main formalisation tasks for probabilistic part are as follows. First, we employed the type `'a pmf` from theory `Probability_Mass_Function` to describe discrete random variables: one function for both random variables and their discrete probability distributions. Second, we discussed two techniques for formalising independent and identically distributed (i.i.d.) random variables: 1) an induction with `map_pmf`, and 2) the index building operator `Pi_pmf` from the theory with the same

name [39], and revealed that we picked the latter due to the built-in facilitation that comes with the mentioned theory. Third, although there is a mechanisation of conditional expectations in Isabelle/HOL-Probability, we suggested a version specifically for the type `'a pmf`. This is to avoid challenges with measurability that comes with the already-mechanised one since it can also apply to continuous random variables, while the type `'a pmf` does not have to involve measurability proofs. However, with careful consideration, we conclude that there are not enough materials for formalising our results related to conditional expectations – lemmas 4.64 and 4.65 – effectively in Isabelle. In the meantime, it is reasonable for us to assume them and proceed to formalise more significant results. A similar issue applies to Azuma-Hoeffding's inequality as it has not been formalised in Isabelle yet, so we also assume it without proof. Lastly, we describe how we finalise our formalisation of lemma 4.38 and also present the generalised version of it to be used further in this project.

Chapter 5

Formalisation of common prefix for Ouroboros

Common prefix is one of the three elemental properties required for the blockchain protocols to satisfy persistence and liveness — the properties of the robust transaction ledgers. In this chapter, we aim to formalise that Ouroboros satisfies common prefix property, but before outlining our sequence in achieving this goal we need to recall common prefix and how to address it in terms of forkable strings notation.

Here, we recall common prefix:

Common Prefix (CP); with parameters $k \in \mathbb{N}$. The chains $\mathcal{C}_1$, $\mathcal{C}_2$ adopted by two honest parties at the onset of the slots $sl_1 < sl_2$ are such that $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$, where $\mathcal{C}^{\lceil k}$ denotes the chain obtained by removing the last k blocks from $\mathcal{C}$, and $\preceq$ denotes the prefix relation.

Like closed forks representing potential combined views of chains diffused by honest parties, the forkable strings notion can be used to precisely describe common prefix (CP) properties. To do so, we need to discuss what it means for chains to be adopted by honest parties. Chains that are adopted by honest parties are outputs of $\mathsf{maxvalid}(\mathcal{C}, \mathbb{C})$ in the chain extension step of the execution of the protocol π for each honest party. As a result, consider a chain adopted by an honest participant, this chain has the length "no less than any chain previously diffused by an honest player" (see p. 19 of the Ouroboros paper [14]) because if it is shorter than any chain previously diffused by an honest player, it is not the longest valid chain and cannot be adopted by an honest participant. Viable tines are defined to represent these chains:

Definition 5.1. (Viability) *Let $F \vdash w$ be a fork for a string $w \in \{0,1\}^*$. We say that t is viable if, for all honest indices $h \leq \ell(t)$, we have*

$$d(h) \leq length(t).$$

```
inductive viable :: "rtree ⇒ bool list ⇒ nat list ⇒ bool" where
  "⟦F ⊢ w ; t ∈ tines F;
    ⋀h. (h ∈ honest_indices w ∧ h ≤ last (trace_tine F t))
        ⟹ depth F h ≤ length t⟧
   ⟹ viable F w t"
```

(Recall that $\ell(t)$ is the label of the terminal vertex of t)

Then, the abstract version of the common prefix adopting the forkable strings notion is described as follows:

Definition 5.2. ***Common Prefix (CP); with parameters*** *$k \in \mathbb{N}$. The string possesses k-cp if, for every fork $F \vdash w$ and every pair of viable tines t_1 and t_2 of F for which $\ell(t_1) \leq \ell(t_2)$, the tine $t_1^{\lceil k}$ is a prefix of t_2, where $t^{\lceil k}$ denotes the tine obtained by removing the last k edges from t.*
(Equivalently, $length(t_1) - length(t_1 \cap t_2) \leq k$, where $t_1 \cap t_2$ denotes the common prefix of the two tines.)

The goal in this chapter is to formalise theorem 5.3 as described below.

Theorem 5.3. *Let $k, L \in \mathbb{N}$ and $\epsilon \in (0, 1/2)$. Let $w = w_1 ... w_L$ of length L, where each w_i is independently distributed in $\{0,1\}$ so that $\mathbb{E}[w_i] = 1/2 - \epsilon$. Then*

$$Pr[w \text{ does not possess } k\text{-}cp] = \epsilon_{cp}(k; L, \epsilon) = (L/c_0) \cdot \exp(-c_0(k-1)).$$

where $c_0 = \epsilon^3/(1+\epsilon) * (2\epsilon + 1)$.

This result is comparable to the k-cp violation upper bound theorem in the ouroboros papar. However, we do not use landau symbol Ω to describe our result meaning that there is no hidden information in this theorem.

We organise this chapter to formalise theorem 5.3 as follows. We start by exploring another intermediate definition called divergence in section 5.1; different angles between the pen-and-paper definition and the formalisation of divergence are also discussed in the same fashion as in 4.1.1. Next, section 5.2 contains continuous contents from 4.1.2 as we address more ways of how one can reconstruct the structure of trees and how we formalise them. These new approaches of reconstructing trees will be used in section 5.3 where we focus on the connection between forka-

bility of characteristic strings and divergence through lemmas and theorems and also their formalisations. The formal proof of theorem 5.3 is then presented in section 5.4. Finally, the chapter summary is provided in section 5.5.

5.1 Intermediate definitions II: divergence

In this section, we capture a quantity called divergence, its formalisation, and relevant lemmas. Divergence is a way to reduce from characteristic strings satisfying k-cp to their forkability, which were well-developed and formalised in previous chapters.

Firstly, we provide the definition of divergence

Definition 5.4. *Let F be a fork for a string $w \in \{0,1\}^*$. For two viable tines t and t' of F, we define the notation t/t' by the rule*

$$t/t' = length(t) - length(t \cap t'),$$

where $t \cap t'$ denotes the common prefix of t and t'. Then, we define the divergence of two viable tines t_1 and t_2 to be the quantity

$$div(t_1, t_2) = \begin{cases} t_1/t_2 & if\ \ell(t_1) < \ell(t_2) \\ t_2/t_1 & if\ \ell(t_2) < \ell(t_1) \\ \max(t_1/t_2, t_2/t_1) & if\ \ell(t_1) = \ell(t_2) \end{cases}$$

We overload this notation by defining divergence for F as the maximum over all pairs of viable tines:

$$div(F) = \max_{\substack{t_1,t_2\ viable \\ tines\ of\ F}} div(t_1, t_2).$$

Finally, we define the divergence of w to be the maximum such divergence over all possible forks for w:

$$div(w) = \max_{F \vdash w} div(F).$$

Observe that if $div(t_1, t_2) \leq k$ and, say, $\ell(t_1) \leq \ell(t_2)$, the tine $t^{\lceil k}$ is a prefix of t_2.

Then, we provide its formalisation for the first version of divergence, $div(t_1, t_2)$, where t_1, t_2 are *tines*.

Definition 5.5. (Divergence for Tines)

```
definition divs :: "rtree ⇒  nat list ⇒ nat list ⇒ nat"
  where "divs F t1 t2 ≡ (if lb F t1 ≤ lb F t2 then size t1
         - size (longest_common_prefix_tines t1 t2) else 0)"

definition div_tines :: "rtree ⇒ nat list ⇒ nat list ⇒ nat"
  where
    "div_tines F t1 t2 ≡ max (divs F t1 t2) (divs F t2 t1)"
```

Despite the slight difference between the formalisation and the original definition of divergence, `div_tines` turns out to be an accurate formalisation of function *div* for two tines. The intermediate step where we need to define / is not in the formalisation of divergence, since the version of the Ouroboros paper from which we formalised did not use /. It was updated later, nevertheless. However, this is not a problem as / only appears once in one pen-and-paper proof in a way that we can use `divs` as its formalisation.

One important point to be noted is that, for the sake of simplicity, `divs_tines` does not check if the input tines are viable or not. Using `viable` in the definition of `div_tines` will complicate the process, since we need the function to return non-natural numbers when it is undefined for those non-viable input tines. We could do that by adding characteristic strings as another parameter and changing the output type of the function from `nat` to `nat option`, so the new function type would be: `div_tines :: "rtree ⇒ bool list ⇒ nat list ⇒ nat list ⇒ nat option"`, where `div_tines F w t1 t2 = Some n` when `viable F w t1` and `viable F w t2`, or `div_tines F w t1 t2 = None` when `~viable F w t1` or `~viable F w t2`. However, in our formalisation we never use non-viable tines as inputs for `div_tines`, so the method above is unnecessary.

Definition 5.6. (Divergence for Fork)

```
definition div_f :: "rtree ⇒ bool list ⇒ nat" where
  "div_f F w ≡ Max (⋃t1 ∈ tines F. (⋃t2 ∈ tines F.
                   {x. viable F w t1 ∧ viable F w t2
                    ∧ x = div_tines F t1 t2}))"
```

The are two redefined versions of the *div* function. The first one is to apply it to forks, which are formalised by `div_f`, and this version uses `viable`. The set `(⋃t1 ∈ tines F. (⋃t2 ∈ tines F.{x. viable F w t1 ∧ viable F w t2 ∧ x = div_tines F t1 t2}))` collects the divergence of two viable tines in `F`. There is an instance of `Max` again, so proving that this set is non-empty and finite is required.

Definition 5.7. (Divergence for String)
definition `div_s :: "bool list ⇒ nat" where`
`"div_s w ≡ Max ((λx. div_f x w) ‘ (forks_set w))"`

The second redefining of *div* is from applying to forks to applying to (characteristic) strings. The instance of `Max` this time needs a similar adaptation as done in 4.1.1 describing formalisation of margin.

5.1.1 Link between divergence and common prefix

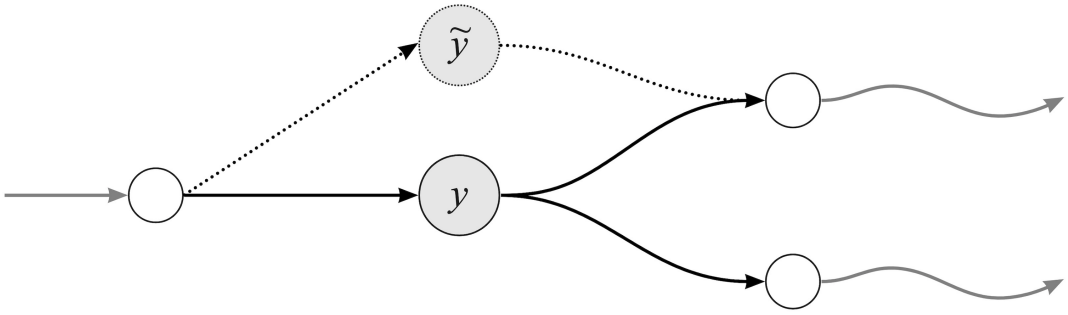
The property k-cp from definition 5.3 can then be simplified in the form of divergence. More specifically, the violation of k-cp can be described through divergence that "the characteristic string w satisfies k-cp if and only if $div(w) \leq k$" [14]. Formalising this result is not complicated as proving it requires mostly notational conventions between first-order logic formulas.

5.2 Reconstructing trees: II

Subsequent to 4.1.2, we resume discussing further specific ways of reconstructing trees which mostly are easy and do not require detailed explanations in pen-and-paper proofs, but are very delicate in formal proofs. This time, the focus is on other necessary methods of three reconstructions to prove theorem 5.3. The processes we are presenting in this section are either built from the ideas in 4.1.2 or completely new.

5.2.1 Subtree redirection

Suppose a fork $F \vdash w$, and $w_a = 1$, there can be any number (can be 0) of vertices in F that are labeled by a. Therefore, if vertex y is one of those vertices and has x as its predecessor, we can add a new vertex $\widetilde{y}$ labeled by a to be another successor of x, and name a new tree F'. It is not difficult to see that this new tree F' is still a fork of w. Then we can move any subtree that has a root as one of the successors of y to extend the vertex $\widetilde{y}$ instead.



From what we have described, we need to clarify steps in formalising the tine redirection. For each step, we sometimes present formalisation if it may require new functions to be defined. The notations used from the following paragraph can be referred to in these steps:

1. Cut a subtree of a tree - we already defined `Cut_subtree` in chapter 4.
2. Extend a tree with a tree - We extended a tree with a tine before with the function `Extend_tine`. Now, we are building a more intuitive function, as we already have the tree we would like to extend with another tree from the previous step using function `Extend_tree` as defined below:

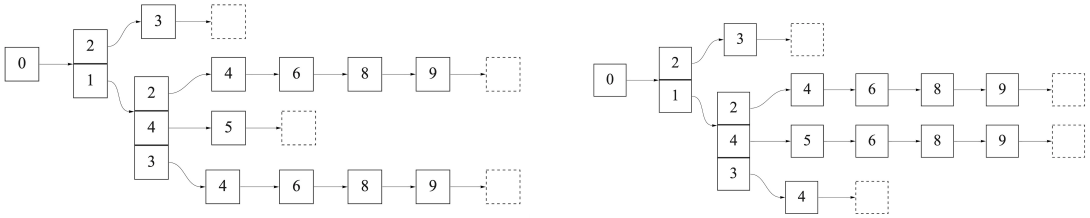
   Definition 5.8.

```
fun Extend_tree :: "rtree ⇒ nat list ⇒ rtree ⇒ rtree"
  where
    "Extend_tree (Tree l Fs) [] rt = Tree l (rt#Fs)"
  | "Extend_tree (Tree l Fs) (k#t) rt
     = Tree l ((take k Fs) @
               (Extend_tree (Fs ! k) t rt) # (drop (Suc k) Fs))"
```

We can then redirect a subtree as we want in the example in the first paragraph of this subsection by one line of code:

```
"F'' = Extend_tree (Cut_subtree F t k)
                   (take ((size t) - 1) t)
                   (Tree (lb F t) [get_subtree F (t @ [k])])"
```

Above, where `F` is a formalisation of F, `t` is a formalisation of a tine that ends with the vertex y, `k` indicates that x is a k-th successor of y, and `lb` is a formalisation of the function l.



(a) (b)

Figure 5.1: Let `F` be `rtree` in 5.1a, `rtree` in 5.1b can be built by `Extend_tree (Cut_subtree F [1, 2, 0] 0) [1, 1, 0] (get_subtree F [1, 2, 0, 0])`, which can be represented by `rtree` in 5.1b.

5.2.2 Fork of string suffix built by subtrees

Let $w = w_1 w_2 ... w_n$ and $w' = w_{a+1} w_{a+2} ... w_n$, in this subsection, we would like to present a specific process of building a fork $F' \vdash w'$ from a fork $F \vdash w$. First, we need to note that the process we are presenting works only when F has a certain behaviour. This behaviour is that there exists an honest vertex y in F which has $d(y) = d$ and is labeled with a, where other vertices deeper than y have greater labels.

This process contains two steps:

1. Build a tree from all relevant vertices - we start by collecting all subtrees of F that have vertices of depth $d + 1$ as their roots. Then, we put them in a list of successors of a root y' which is labeled with a. We name a new tree Ft. We can see that Ft has these following properties:

   - Ft is strictly increasing,
   - for each honest index h of w, if $h \geq a$, there is exactly one vertex with label h in Ft,
   - for two vertices i and j in Ft' that $\ell(i), \ell(j)$ are honest indices of w, if $\ell(i) < \ell(j)$ then $d(i) < d(j)$

   Functions `tines_for_suffix` and `Suffix_pinched` below are defined to formalise constructing the aforementioned Ft.

   Definition 5.9.

```
definition tines_for_suffix:: "rtree ⇒ nat ⇒ (nat list) set"
  where "tines_for_suffix F n = {t. tinesP F t ∧ size t = n}"
```

The function `tines_for_suffix` collects all tines with a certain length from a tree.

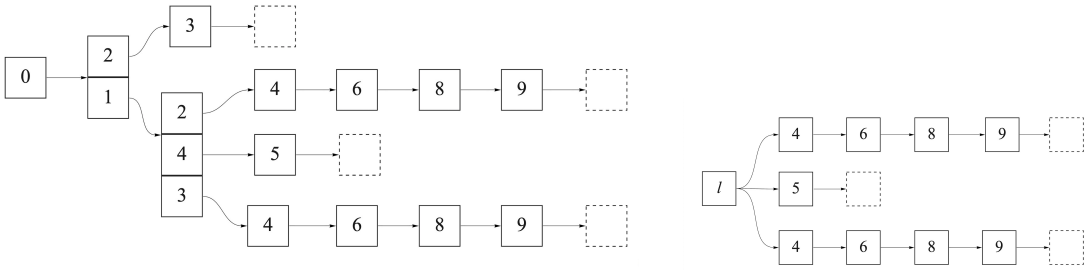
Definition 5.10.

```
definition Suffix_pinched:: "rtree ⇒ nat ⇒ nat ⇒ rtree"
  where "Suffix_pinched F n l =
Tree l (map (get_subtree F) (list_of (tines_for_suffix F n)))"
```

The function `Suffix_pinched` uses `tines_for_suffix` to get all tines with a certain length and put them in a list, and then map `get_subtree` on this list to build a list of successors of a tree.

2. Shift labels of all vertices - we can see that even though we have built a tree Ft, it is not a fork of w'. To adjust Ft to be a fork of w', we need to change labels in Ft from indices associated with w to indices associated with w'. As $w' = w_{a+1}w_{a+2}...w_n$, we redefine it to $w' = w'_1w'_2...w'_{n-a}$, so $w_{a+i} = w'_i$. Therefore, a label $a+j$ associated with w is the same as a label j associated with w'. Consider subtracting all labels in Ft by a, naming and name this tree Ft', this tree has these following properties:

   - root of Ft' is labeled by 0
   - Ft' is strictly increasing,
   - for each honest index h of w', there is exactly one vertex with label h in Ft' because $w'_h = w_{a+h}$ meaning that $a+h$ is an honest of w and there is exactly one vertex with label $a+h$ in Ft,
   - for two vertices i and j in Ft' that $\ell(i), \ell(j)$ are honest indices of w',



(a) (b)

Figure 5.2: Let `F` be `rtree` in 5.2a, `rtree` in 5.2b can be built by `Suffix_pinched F 2 l`.

if $\ell(i) < \ell(j)$ then $d(i) < d(j)$ (by the same reason as the previous property)

Thus, Ft' is a fork of w'. A function that subtracts all labels in a tree with a certain number is what we need, so we define `reduce_tree` as follows,

Definition 5.11.

```
fun reduce_tree:: "rtree ⇒ nat ⇒ rtree"
  where "reduce_tree (Tree l Fs) n = Tree (l - n)
                                     (map (λx. reduce_tree x n) Fs)"
```

After we get all the necessary functions formalised, we can formalise building F' from F in the first paragraph of this subsection:

```
define F'
where "F' = reduce_tree (Suffix_pinched F (Suc (size t)) a) a"
```

Above, where `F` is a formalisation of F, `t` is a formalisation of a tine that ends with the vertex y, and `a` is a formalisation of value a.

5.3 Linking forkability to common prefix

This section aims to explain the result below.

Theorem 5.12. *Let $w \in \{0,1\}^*$. Then there is a forkable substring $\check{w}$ of w with $|\check{w}| \geq div(w)$.*

```
theorem exist_forkable_substring_length_ge_div:
  assumes "length w = l"
  shows "∃w' i j. forkable w' ∧ i ≤ j ∧ j ≤ l
                  ∧ (drop i (take j w)) = w' ∧ length w' ≥ div_s w"
```

This theorem is basically the link between forkability of strings and divergence, and we have already known from section 5.1 that divergence is related to violation of k-cp property. The pen-and-paper proof of this theorem takes two and a half pages, so it is not unexpected that the formal proof in Isabelle/HOL covers about 7000 lines, which is the longest formal proof we have so far. Similar to how we explained the formalisation of theorem 4.29, we will just explain the core ideas and tricky parts to finish the formalisation.

The proof of theorem 5.12 is a proof of existence. Following the original pen-and-paper version, the proof can be separated into two parts:

1. Designate a potentially forkable substring w' where $length(w') \geq div(w)$ — selecting w' can be done by selecting a correct pair of tines t_1, t_2 in a fork $F \vdash w$, where $div(t_1, t_2) = div(F) = div(w)$. However, there are some conditions on how to select the correct pair of tines, but they are irrelevant here, as the formalisation of this part is not significant and quite independent from other formal proofs.

2. Prove that w' is forkable — this part uses two methods of tree reconstruction presented in the previous chapter. The subtree redirection from 5.2.1 is used to help construct the proof by the contradiction that we cannot add a vertex and redirect a tree to it if the vertex is labelled by honest indices. Then, the process of building a fork of a string suffix by subtrees from 5.2.2 can be used to construct a fork of F'' of a string w'' from subtrees of F (from the previous step), where w' is a substring of w'' and their heads start at the same position. Finally, we can search for a flat fork $F' \vdash w'$ which is a prefix of F''.

5.4 Formalisation of k-cp violation probability

We restate the theorem with proof that is adapted from the Ouroboros paper[1] as we have now explained and formalised lemmas necessary to make sense of and complete this proof.

Theorem 5.3. *Let $k, L \in \mathbb{N}$ and $\epsilon \in (0, 1/2)$. Let $w = w_1 ... w_L$ of length L, where each w_i is independently distributed in $\{0, 1\}$ so that $\mathbb{E}[w_i] = 1/2 - \epsilon$. Then*

$$Pr[w \text{ does not possess } k-cp] = \epsilon_{cp}(k; L, \epsilon) = (L/c_0) \cdot \exp(-c_0(k-1)).$$

where $c_0 = \epsilon^3/(1+\epsilon) * (2\epsilon + 1)$.

Proof. Recall that w violates $k - cp$ precisely when there is a fork $F \vdash w$ for which $div(F) \geq k$. Thus we wish to show that the probability that $div(w) \geq k$ is no more than $\exp(-c_0(k-1) + \ln(L/c_0))$. It follows from theorem 5.12 that if $div(w) \geq k$,

[1] The technique used for the proof here is similar to the Ouroboros paper. However, we use a better result from "forkable strings are rare" paper.[22] Therefore, the calculation of integrals here results in a different result to that of the Ouroboros paper.

there is a forkable substring $\check{w}$ of length at least k. Thus

$$\begin{aligned} Pr[w\ does\ not\ possess\ k{-}cp] &\leq Pr[\exists \alpha, \beta \in \{1, ..., L\}\ so\ that\ \alpha + k - 1 \leq \beta \\ &\qquad w_\alpha ... w_\beta\ is\ forkable] \\ &\leq \sum_{1 \leq \alpha \leq L} \underbrace{\sum_{\alpha+k-1 \leq \beta \leq L} Pr[w_\alpha ... w_\beta\ is\ forkable]}_{*} \end{aligned} \tag{5.1}$$

According to lemma 4.75, the probability that a string of length t drawn from this distribution is forkable is no more than $\exp(-c_0 t)$; here, c_0 is used instead of the formalised version `c0`, but it represents the same value; $c_0 = \epsilon^3/(1+\epsilon) * (2\epsilon + 1)$.

Note that for any $\alpha \geq 1$,

$$\sum_{t=\alpha+k-1}^{L} e^{-c_0 t} \leq \int_{k-1}^{\infty} e^{-c_0 t} dt = (1/c_0) e^{-c_0(k-1)} \tag{5.2}$$

and it follows that the sum $(*)$ above is less than or equal to $\exp(-c_0(k-1))$. Thus

$$Pr[w\ does\ not\ possess\ k{-}cp] \leq L(1/c_0) e^{(-c_0(k-1))} \leq e^{(\ln(L/c_0) - c_0(k-1))} \tag{5.3}$$

as desired. □

Note on the difference of our result to the result in the ouroboros paper – in the Ouroboros paper, they state that:

> "the probability that a string of length t drawn from this distribution is forkable is no more than $\exp(c\sqrt{t})$ for a positive constant c."

using lemma 2.11. Then, they proceed to use this statement to calculate an upper bound of $(*)$ in inequality 5.1. However, we see this as an inaccurate statement since the only proof we get from lemma 2.11 is that there the upper bound of forkable strings' density is $\exp(-\Omega(\sqrt{t}))$ in which we can assume only the following:

> There exist a natural number t_0 and a positive constant c in which the probability that a string of length $t \geq t_0$ drawn from this distribution is forkable is no more than $\exp(c\sqrt{t})$.

We avoid this inaccuracy by having the final result of our formalisation of lemma 2.12: lemma 4.71, not be in the landau symbol form like in the original pen-and-

paper version.

We finished the formalisation of inequality 5.1 as follows:

```
lemma k_cp_violation_upper bound_1:
  assumes "k ≤ L"
  shows "𝒫(x in w_pmf L. ¬ k_cp k x)
         ≤ 𝒫(x in w_pmf L. ∃a b. a ∈ {1..L} ∧ b ∈ {0..L}
             ∧ a + k - 1 ≤ b ∧ forkable (drop (a-1) (take b x)))"

lemma  k_cp_violation_upper bound_2:
  assumes "k ≤ L"
  shows "𝒫(x in w_pmf L. ∃a b. a ∈ {1..L} ∧ b ∈ {0..L}
          ∧ a + k - 1 ≤ b ∧ forkable (drop (a-1) (take b x)))
         ≤ (∑a ∈ {1..L}.(∑ b ∈ {(a+k-1)..L}.
             𝒫(x in w_pmf L. forkable (drop (a-1) (take b x)))))"
```

The lemmas represent the first and second sub-inequalities in inequality 5.1, respectively. The first one is straightforward to formalise as the predicate on the left implies the predicate on the right.

On the other hand, the second one requires the use of the inclusion-exclusion principle. Hence, the formal proof becomes quite tedious. We separate this into two parts of inequality as follows:

```
lemma  k_cp_violation_upper bound_2_1:
  fixes a::nat and k::nat and L::nat
  assumes "a ∈ {1..L}" "k ≤ L" "k>1"
  shows "𝒫(x in w_pmf L.
         ∃b. b ∈ {1..L} ∧ a + k - 1 ≤ b
         ∧ forkable (drop (a-1) (take b x)))
         ≤ (∑ b ∈ {(a+k-1)..L}.
            𝒫(x in w_pmf L. forkable (drop (a-1) (take b x))))"
```

and

```
lemma  k_cp_violation_upper bound_2_2:
  assumes "k ≤ L" "k>1"
  shows "𝒫(x in w_pmf L.
          ∃a b. a ∈ {1..L} ∧ b ∈ {1..L} ∧ a + k - 1 ≤ b
          ∧ forkable (drop (a-1) (take b x)))
          ≤ (∑a ∈ {1..L}. 𝒫(x in w_pmf L.
             ∃b. b ∈ {1..L} ∧ a + k - 1 ≤ b
             ∧ forkable (drop (a-1) (take b x))))"
```

We use a similar technique for both parts by unpacking the probability to smaller parts then sum them up. The first part is done by fixing `a` (α in pen-and-paper version) to one value.

While the left-hand side of inequality 5.2 can be proved by induction, we generalise it to all integral for non-negative function first:

Lemma 5.13.

```
lemma sum_le_integral:
  fixes f :: "real ⇒ real"
  assumes "a ≤ b" "n≠0" and ant: "antimono_on {a..b} f"
  shows "(∑i=1..n. f(a + (real i / real n) * (b-a))) * (b-a) / n
         ≤ integral {a..b} f"
```

Then, we apply the desired value which is our exponential function from lemma 4.75:

Lemma 5.14.

```
lemma sum_le_integral_applied:
  fixes k::nat and l::nat
  assumes "k-1>0" "c >0"
  shows "(∑t ∈ {k..k+l}. exp (- c* (real_of_nat t)))
         ≤ (integral {(k-1::real)..(k+l::real)} (λt . exp (- c* t)))"
```

Then last step is to extend the integral side to improper integral over an unbounded interval on the right side. In Isabelle/HOL, we use `at_top`, to describe taking a limit to infinity.

Lemma 5.15.

```
lemma inf_integral_le_interval_integral:
  fixes h ::"real ⇒ real"
  assumes int: "⋀y::real. h integrable_on {a..y}"
    and lim: "((λy. integral {a..y} h)  l) at_top"
    and nonneg: "⋀y. y ≥ a ⟹ h y ≥ 0"
   and "a ≤ b"
 shows "integral {a..b} h ≤ l"
```

Formalising an integral result is a simple task in Isabelle: we can simply prove that one function is a derivative of another function using `has_vector_derivative`.and then we can prove the reverse of that using this corollary[2]:

corollary `fundamental_theorem_of_calculus_strong:`

[2]Available in theory `Henstock_Kurzweil_integration`.

```
  fixes f:: "real ⇒ 'a:: banach"
  assumes "finite S"
  and "a ≤ b"
  and vec: "⋀x. x ∈ {a..b} - S
           ⟹ (f has_vector_derivative f'(x)) (at x)"
  and "continuous_on {a..b} f"
shows "(f' has_integral (f b - f a)) {a..b}"
```

Therefore, we can formalise an improper integral for our specific exponential function:

Lemma 5.16.

```
lemma integral_for_specific_exp_tmp:
  fixes c::real
  assumes "k-1>0" "c>0"
  shows "((λy. integral {k-1..y} (λt . exp (- c* t)))  (1/c)
         * (exp (-c * (k-1)))) at_top"
```

We can apply this lemma to get the final result of inequality 5.2:

Lemma 5.17.

```
lemma sum_le_integral_at_top:
  fixes k::nat and l::nat and c::real
  assumes  "k-1>0" "c > 0"
  shows "(∑t =k..k + l. exp (- c* (real_of_nat t)))
         ≤ (1/c) * (exp (-c * (k-1)))"
```

The formalisation of inequality 5.3 is simply done by using that 5.1 and 5.2, and applying `c0` for all `c` using lemma below:

lemma[3] `sum_le_card_Max: "finite A ⟹ sum f A ≤ card A * Max (f ‘ A)"`

Then, we get our final result in this project:

Lemma 5.18. Formalisation of k-cp violation probability

```
lemma k_cp_violation_upper bound:
  fixes L::nat and k::nat
  assumes "k-1>0" "k≤L" "L > 0"
  shows "𝒫(x in w_pmf L. ¬ k_cp k x)
         ≤ exp (ln (real L * (1/c0))  - c0 * (k-1))"
```

[3]This lemma is available in theory `Lattices_Big`.

5.5 Chapter summary

Common prefix in the context of the forkable strings notion was explained, for which a definition of viability was required and given. This is followed by the provision of the theorem representing the statement that the protocol satisfies common prefix. We then set out a framework to formalise this theorem.

Firstly, divergence, another intermediate definition linked directly to the common prefix property, was given and formalised. Divergence can be used to explain pairs of tines, forks, and then characteristic strings. `Max` is used in the same fashion as we formalised *margin*. Secondly, we continued on how we formally reconstruct trees: redirecting subtrees and pinching tree suffixes. New functions introduced for this purpose are `Suffix_pinched` and `reduce_tree`. Thirdly, another significant but intermediate result on how divergence of the characteristic strings is linked to their forkability was formalised. The structure of the formalisation was only provided in brief because the main complication of the formalisation was on reconstructing trees that we explained already. Finally, with the result of chapter 4, we finished our final result formalising the upper bound of common prefix property violation.

Chapter 6

Related work and evaluation

We have discussed the growing body of research in blockchain protocols in chapter 1 covering mainly proof-of-stake and proof-of-work based protocols. In this chapter, we will focus more on lines of work related to our project, especially the ones that apply formal methods to blockchain protocols.

Firstly, in section 6.1, we present several pieces of closely related work. They range from pen-and-paper proofs of security and correctness of various other blockchain protocols to formalisation of those proofs or the protocols directly. Among them, there are both PoW-based and PoS-based blockchain protocols. We also compare our work to them in different perspectives, such as, tools used, the difference of each formalised protocol and protocols' assumptions, and, most importantly, how strong their formalised security statements are compared to ours. Then, in section 6.2 the Ouroboros family formalisation is presented and discussed; we will address what has been done and how plausible for us to co-develop the full formal proofs in the near future. Lastly, remotely related work will be examined in section 6.3.

6.1 Blockchains' security analyses and formalisations

The introduction of Bitcoin by Nakamoto in 2008 [2] has revolutionised the field of Byzentine Agreement [40] from using a majority of votes from all parties, which is referred to as a quorum-based protocol, to letting any individual party participate in a lottery to have a right to decide on a consensus for all parties in order to maintain security: persistence and liveness. Blockchain protocols that follow this method — using the lottery system to reach a consensus — are called Nakamoto-style blockchains (or NSB). However, blockchain protocols have been developed

widely in both styles — NSB and quorum-based blockchain protocols — and the security of such protocols has been rigorously analysed over the years.

6.1.1 Security of Nakamoto-style blockchains

A Nakamoto-style blockchain was analysed for the first time in Gary et al's work [29] for the case of PoW protocols, referred to as the bitcoin backbone protocol (BBP). This work introduced three elementary properties mentioned already in chapter 2: common prefix, chain quality, and chain growth, which taken together can prove that a blockchain protocol satisfies persistence and liveness. There have been numerous extensions of this work in different perspectives: Kamp et al. generalised how protocols select a chain where there are many candidates by applying a weight function for each chain before selecting the best chain, instead of just selecting the longest chain as Bitcoin does. They also prove an additional guarantee called an optimistic responsiveness for cases where the weight functions are exponentially growing [41]. In the work by Pass et al., a fully asynchronous network was assumed and applied, which was more realistic than the synchronous network assumption in the original work [42]. Simplification of the original analysis was achieved by Ren [43]. Having the security of NSB protocols analysed also helps improve the performance of protocols and proving their security in the same framework; Prism, for example, is a PoW NSB protocol that is designed for a better transaction rate than bitcoin while maintaining similar security guarantees [44].

NSB has been employed as the core protocol algorithm for numerous PoW-based and PoS-based protocols, but we will only focus on PoS protocols with a rigorous proof of security. Ouroboros classic [14] is the first PoS NSB to have a rigorous analysis for security guarantees assuming a synchronous network, non-adaptive adversary, and honest majority stake. Ouroboros Praos, Genesis, and Chronos have then been developed from the classic version in sucession; Praos [23] aimed at weakening the assumption of a network from synchronous to semi-asynchronous and strengthening its security for adaptive adversarial attacks, Genesis [24] introduced a PoS NSB protocol that has fully dynamic availability where parties can bootstrap from genesis blocks, and Chronos [25] introduced a novel synchronisation mechanism to get rid of the need for the global clock, assuming that local clock speeds are the same. Snow White [16] is also one of the early PoS NSB protocols that have been rigorously analysed. It is a subsequent work by Pass et al's "the sleepy model of consensus" [45] which aims to address problems with parties disconnecting from and rejoining consensus protocols by removing the PoW mechanism and replacing it with the one-vote-per-party mechanism (comparable to the scenario where each party has exactly one coin in their stake) assuming an honest majority.

Although NSB protocols could use the method of exploiting these three elementary properties to guarantee their security, there are some differences in applying it between PoW-based and PoS-based protocols. Thomsen and Spitters stated that proving that a protocol satisfies common prefix is greatly different between PoW-based and PoS-based protocols [46]. This is because winning the lottery in PoS-based protocols allows an adversarial slot leader—the winner of the lottery in a certain slot— to extend many chains with different blocks in whatever fashion they like and send them to different parties. The only exception is in each chain there can only be at most one block from each slot. By contrast, in PoW-based protocols, the winner of the lottery only declares one block for that win. This makes it more difficult for PoS-based protocols to sustain common prefix. For example, the work done by Gaži et al. [47] calculates a bound for PoW NSB to satisfy consistency through the similar notion of forkable (characteristic) strings as used in Ouroboros classic; forks in this work (called PoW Δ-forks) do not allow multiple nodes from one adversarial vertex. Moreover, there is a study by Blum et al. [48] contributing to optimising a lower bound for the length of the slot for PoS blockchain protocols to compete with PoW blockchain protocols.

Formalisation of NSB protocols — there have been formalisation attempts at NSB protocols and their security. The first attempt is Toychain [49] by Pîrlea et al. in the Coq proof assistant [50]. This work focuses on formal guarantees for general NSB protocols, especially PoW ones. They used state transition systems to define a relation on global states of the protocol and prove the properties in terms of where global states can reach. One property proved in this work is, considering a semi-synchronous network, if all messages sent were delivered, all parties would reach a consensus on the current best chain. Thomsen and Spitters criticised that it is "not enough to argue about how the tree of blocks evolves" for the real-world applications because the case where there are no messages in transit might never happen [46]. It does not formalise security properties of the protocol; only functional correctness is formalised. There is also an extension work from Toychain called Kaizen [51] that includes an analysis focusing on NSB protocol's actual performant implementation, but it does not improve statements in the original Toychain work [49].

Gopinathan et al's Probchain [52] has attempted to formalise Gary et al's BBP [29] security analysis. This work is not complete. It uses some techniques from Toychain [49] like relations on global states and the use of Oracle in Coq to represent non-determinism. The main theorems they aim to formalise are about chain growth and common prefix. In this unfinished work, they assert, instead of formally verifying, that the probability that the protocol's global state will execute typically (Typical Executions is their terminology) is $1 - \exp(-\Omega(k))$. When the

protocol executes typically, the common prefix holds, assuming k is big enough.

PoS NSB's security was formalised for the first time by Thomsen and Spitters [46] and also in the Coq proof assistant that demonstrates both safety (referred to as persistence in our work) and liveness. Thomsen and Spitters also claimed that their work was the first to achieve the machine checked proof of a consensus algorithm in general for these two properties. Therefore, they provide a relatively thorough comparison between their work and other studies done in Coq for consensus protocols. This work formalises the three elementary properties but adapts them to be associated with the PoS lottery mechanism instead of the PoW one. The protocol they considered to represent PoS NSB in their work is similar to a static stake protocol of Ouroboros Praos [23], which highly resembles the protocol we use in our work, except for the leader selection process in which each slot can have multiple slot leaders. Nevertheless, this work employs the same approach as used in Toychain [49], proving properties on reachable global states and assuming a stronger network setting, which is synchronous.

Evaluation with closely related work — while many formalisation works discussed in this paragraph involve NSB protocols' security properties that are similar to our project, all of them focus on different levels of formalisation from ours and hence use different techniques. Particularly, we only work on the formalisation of the combinatorial analysis of the common prefix property and the static stake protocol of Ouroboros classic, under the assumption that the implementation (more specifically the leader selection process) works correctly, while other projects try to define semantics for the protocols' executions and prove properties through the use of state transition systems. Both Gopinathan et al's Probchain [52] and Thomsen and Spitters' PoS NSB formalisation [46] focus on three elementary properties (the common prefix and the chain growth properties in Probchain and all three in the PoS NSB formalisation). Although we cover only the common prefix property, our formalisation of the common prefix property is based on more complicated pen-and-paper proofs than the other work which is the combinatorial analysis of the characteristic strings. This gains us some advantages as follows. Under the assumption of honest majority of stake, our upper bound of the probability for the common prefix property for the PoS NSB $(1 - \exp(\ln\ L - \Omega(\sqrt{k}))$, where L is the number of slots the protocol has run) is almost as good as that of the PoW NSB protocol in Probchain $(1 - \exp(-\Omega(k)))$ while working on PoS which is more complicated for proving this property than PoW due to the adversarial parties' ability to declare multiple blocks when picked as slot leaders. Comparing the PoS NSB protocols, Thomsen et al. formally prove that common prefix from a much stronger assumption than ours: that at least 2/3 of the stake is honest.

6.1.2 Other blockchain protocols

Rigorous analyses have also been performed for blockchains that use traditional quorum-based protocols. The difference between the quorum-based protocols and NSBs is that the quorum-based ones decide each block together, while NSBs pick one party (or a few parties) to declare a block individually. In the context of blockchain protocols, quorum-based protocols let parties vote and have an agreement on a new block to be put in a shared ledger. They usually require stronger honest assumptions than NSBs do, but they require only a partial synchrony of a network to be secure with an unknown upper bound of message delivery time [53]. Apart from Ouroboros BFT (discussed above in section 6.2), which now has its formalisation of functional correctness but not for security guarantees, we will discuss some quorum-based PoS blockchain protocols that have been formalised.

Algorand [15] is a PoS blockchain that claims to process a block in a very fast manner (10 minutes per block) with rigorous analysis for its security in both permissioned or permissionless environments. For the permissionless environment, it requires that more than 2/3 of stake is honest to be secure. Its safety was formalised in Coq in the work by Alturki et al. [54], but not its liveness. The work was done assuming a partially synchronous execution of the protocol.

Ethereum has a project, Casper, attempting to replace its current PoW consensus protocol for its finality layer with PoS protocols. A few protocols were proposed with rigorous analysis on security. Casper CBC [55] is one of the proposals for this project and it was originally formalised in Isabelle/HOL, but only for safety [56]. Another version of Casper, Casper FFG [57], was formalised in Coq [58] including its safety and plausible liveness, which is a weaker version of liveness not covering deadlock cases. Getting extended from this result, the revised version of Casper FFG, Gasper [59], was also formalised in Coq for the same properties; this protocol is for Ethereum 2.0.

6.2 Ouroboros protocols family formalisation

We described the Ouroboros protocol family in chapter 2. There is ongoing work on formalising two of the protocols in this family in Isabelle/HOL [60]. What has been done is the formalisation of Ouroboros Praos [23] and Ouroboros BFT [26] by Diaz et al.[1] The work is a formalisation of the implementations of the protocols using labeled transition systems. They have formally proved the functional correctness of the implementation. More precisely, they have proved the equivalence of a broadcasting network—the communication mechanism assumed by both

[1]This project is also supported by IOHK and developed by IOHK formal methods team.

Ouroboros papers [23, 26]— and forwarding packets in a peer-to-peer network— the communication mechanism used by real implementations. However, there are no formal proofs regarding the security of these two protocols in their project. By-products of the project include two process calculi called ♮-calculus and ⊳-calculus, the former having been already published by Jeltsch [61], one of the project contributors. We have a connection with this team and have discussed a potential collaboration in the near future. Since the two versions of Ouroboros they formalised are based on Ouroboros classic, their security guarantees are rooted to those of Ouroboros classic with few challenges required for us to adapt some of our work to be compatible with theirs; one of them is mentioned below.

While the Ouroboros family uses forkable (characteristic) strings to prove their security guarantees related to the common prefix property, the difference in network settings indicates the different versions of the notions used. More specifically, in Ouroboros Praos, the network setting assumes only a partially asynchronous operation, in contrast to Ouroboros classic where synchronisation is one of the assumptions. Therefore, the forkable strings notion used in this version of the protocols is expanded to accommodate the delays of message delivery, i.e. the definition of forks is expanded to Δ-forks where all the requirements are the same except the strictly increasing honest indices, where the delay time (Δ) is considered [23]. Another challenge is to formally prove that it is realistic that forks represent possible shared views under a known result of the leader selection process in the implementation of these protocols in Diaz et al's work [60].

6.3 Other related work

There are some remotely related studies on the topic of blockchain protocols' formalisation that we have not mentioned or discussed. Since there are various topics with tangential details close to our work, we group them into this section without any suggestion that they are connected.

6.3.1 Smart contracts and their formalisation

A smart contract is one of the most significant and widely-used applications of blockchain protocols. It marks the second generation of blockchain protocols [62] (the first one being cryptocurrencies, and the third one involving the public sector, e.g. Corda [63]). A smart contract is a computer program that is executed automatically when agreements or contracts are met; this is to reduce trusted third-parties while remaining secure when executing the program. They were introduced in 1990s by Szabo [6]. Szabo later proposed that smart contracts can be

implemented by cryptographic hash chains [64].

Ethereum is a well-known platform for providing smart contracts. They run on its decentralised virtual machine called the Ethereum Virtual Machine. This machine's instruction set is Turing-complete [65]. Hirai defined EVM in Lem, a language that can be compiled by a few interactive theorem provers, and then used Isabelle/HOL to test and formally prove some of its safety properties [66]. This work was extended by Amani et al. also in Isabelle/HOL at the level of EVM bytecode [67]. There is also a separate project named KEVM presenting a complete formal semantics of EVM's bytecode language as a foundation for further formal analyses; its correctness and performance were evaluated using the official Ethereum test suite [68]. The work by Osterland et al. [69] attempts to formalise Solidity code, which is in a higher level than KEVM (bytecode language), and then analyses it using SPIN (a model-checking tool).

6.3.2 Other formal methods

Other formal methods have been applied to blockchain protocols for their security aspects. DiGiacomo-Castillo et al. [70] used UPPAAL-SMC, a statistical model checker, to examine Garay et al's BBP [29] properties by varying protocol parameters; it has been shown that there is a trade-off between the failure rate of chain quality and that of the common prefix using different tie-breaking rules from selecting the best chain.

6.3.3 E-cash

Before the cryptocurrency era, the untraceable e-cash systems were proposed by Chaum in 1993. [3] The protocols in this work were formalised by Dreier et al. [71] in the applied π calculus [72], and the results are amendable to be verified by ProVerif (an automatic tool for verifying cryptographic protocols) [73]. This work covers security properties that are different from what we mentioned for consensus protocols: the identified unforgery and privacy properties.

Chapter 7

Conclusion

We have described through various chapters how we have attempted to formalise security guarantees of a PoS-based protocol. We spent two chapters (2 and 3) particularly addressing the background knowledge of our work. We began by explaining what is required to understand that protocol, which is the static stake protocol of Ouroboros classic, its assumptions, and its properties we aimed to formalise. Persistence and liveness were explained in detail, while we clarified that Ouroboros is proved to satisfy them through elementary properties: common prefix, chain quality, and chain growth. The notion of forkable strings was presented in the continued background chapter as it is the main part of the combinatorial analysis of the leader selection process of the protocol.

We proceeded to show our formalisation work in Isabelle/HOL with the final goal of formalising that Ouroboros satisfies common prefix with overwhelming probability. This can be separated into three formalisation tasks: chapter (3: fundamental elements of the notion of forkable strings, chapter 4: "forkable strings are rare", and chapter 5): common prefix. The second part is unfinished due to time constraints and inadequate resources in Isabelle/HOL probability theory. Although we did not show full formal proofs of any lemmas or theorems, we added discussions about the ideas behind them and how they were different from the pen-and-paper proofs of the same results. Finally, we reviewed recent work on the formalisation of blockchain (and other pieces of remotely related work) with comparisons to our work as an evaluation approach in chapter 6.

Below, the contributions of the thesis, our final results, and our plan for future work are presented in three sections, respectively.

7.1 Contributions of this thesis

We have contributed an almost complete formalisation that Ouroboros satisfies common prefix. Common prefix is one of the three elementary properties proposed by Gary et al. [29] for the Nakamoto-style blockchain protocols to satisfy persistence and liveness. Our formalisation captures a detailed combinatorial analysis of characteristic strings in the Ouroboros paper [14]. Characteristic strings represent simplified results of the leader selection process (the lottery mechanism in Ouroboros) that involve the honesty of the selected leaders only. The said combinatorial analysis is also referred to as the notion of forkable strings.

We can separate the contributions from this project into two parts: discrete structure and probability theory. The former is complete, while the latter is ongoing.

Discrete structure — we have finished the formalisation of all results regarding the discrete structure required for proving that the common prefix property holds for the protocol. All of definitions and lemmas in the Ouroboros paper relevant to the notion of forkable strings are formalised. These include:

- fundamental definitions — for example, `height`, `depth`, and `strictly_inc`, and fundamental properties related to them,
- intermediate definition — for example, `gap`, `reserve`, `reach`, `margin` and `divs`, and
- intermediate theorems — for example lemma 4.22, lemma 4.29, and lemma 5.12.

Apart from the main results that are clearly stated in the Ouroboros paper, there were a few issues regarding mainly the discrepancies between the pen-and-paper proofs and the formalised ones. Hence, we needed to assert more definitions and prove lemmas related to them to deal with these discrepancies. We introduced a number of ways to reconstruct `rtree`, such as `get_subtree`, `Cut_subtree`, and `Extend_tree` and also provided their properties in the form of lemmas. In the pen-and-paper proofs, constructing new trees or getting some parts of trees (e.g. tines, subtrees or prefixes) from existing ones can be done easily and is obvious to the readers if those new trees or those parts satisfy certain properties or not, but it is not trivial for the theorem provers to solve. Moreover, one significant difference between `rtree` and rose trees in the pen-and-paper proofs is that there is the ordering of branches of tree in the former but no ordering in the latter. This, while helping us trace tines in a tree in the formalisation easily, created a few problems with formally proving the results regarding prefixes of trees. A number of lemmas need to be introduced and proved in our formalisation due to this issue;

the best example is lemma 4.26 (**lemma** `non_sharing_edged_tines_prefix`) where positioning the prefixes of two tines is difficult.

In conclusion, we have built a very detailed and thorough theory around a newly-introduced **datatype** `rtree` along with **datatype** `bool list` (presenting characteristic strings) to formalise the notion of forkable strings. This theory can be used in the future to analyse other PoS NSB blockchains with similar assumptions on the leader selection process, the network settings, and the stake distributions because the way we constructed our theory is also feasible for a minor adaptation for a notion of forkable strings with a slightly different set of rules. Also, as a by-product, one can use our theory to reason out rose trees with different usages from blockchain protocols.

Probability theory — although we did not finish this part, we have formalised a great amount of relevant random variables, equations and inequalities used towards proving the bound in Russell et al's "Forkable Strings are Rare" [22].

There are a couple of reasons why we did not finish formalising some of the probabilistic proofs. One main reason is the required theorem, Azuma-Hoeffding's inequality (see theorem 4.39), has not yet been formalised in Isabelle/HOL. Secondly, we use the theory of probability mass functions to formalise random variables and their distributions, but the problem is we do not have the definition of conditional expectations to use with type `'a pmf` formalised prior to our project. This leads to our own formalisation of conditional expectations for type `'a pmf`. However, the absence of supported lemmas for it renders the results that rely on using conditional expectation, such as lemmas 4.64 and 4.65, slightly too tedious to be finished under the time constraint we had.

7.2 Final results

We were not the first to attempt to formalise the PoS security blockchain protocols, given the work by Thomsen and Spitters [46]. They make a much stronger assumption—of a 2/3 honest majority instead of just 1/2—allowing simpler techniques. Our formalisation comprises almost 26,000 lines of Isabelle/HOL code. The work is stored at https://bitbucket.org/wkawin/ouroboros with the theory files named `Ouro` and `Ouro2`; we only separate our code into two theories due to its overall large size. The formalisation works for the latest version of Isabelle – Isabelle2023.

7.3 Future work

- Short-term — finishing the ongoing tasks is one of our reasonable short-term goals. However, we might need to reroute our ways of formalising random variables and super-martingale as the process became drastically tedious and we could not finish it in time. (In the mean time, we could expect Azuma-Hoeffding to be formalised in Isabelle/HOL.) After that, we consider formally proving chain quality and chain growth properties for Ouroboros. Finally, while our formal proofs of different results are still sometimes repetitive, more simplifications and optimisations can be made.
- Long-term — it is mentioned in chapter 6 that there is a formalisation of the implementation of protocols in the Ouroboros family in Isabelle/HOL by the IOHK formal methods team. The work only provides a formal proof of the functional correctness of the protocols. There is a strong possibility of continuing the work with this team to link the result of the leader selection process to characteristic strings and use our formalisation of the notion of forkable strings to formally prove their security guarantees. These protocols, however, differ from Ouroboros classic, and some versions might require a different notion of forkable strings, which could also be done by altering our results.

Bibliography

[1] Stuart Haber and W Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[3] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[4] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Whitepaper: https://translatewhitepaper.com/wp-content/uploads/2021/04/EthereumOrijinal-ETH-English.pdf*, 3(37), 2014.

[5] David Schwartz, Noah Youngs, Arthur Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper: https://cryptoguide.ch/cryptocurrency/ripple/whitepaper.pdf*, 5(8):151, 2014.

[6] Nick Szabo. Formalizing and securing relationships on public networks. *First monday*, 1997.

[7] Sarah Underwood. Blockchain beyond bitcoin. *Communications of the ACM*, 59(11):15–17, 2016.

[8] Mark Gimein. Virtual bitcoin mining is a real-world environmental disaster. *Bloomberg Business*, 2013.

[9] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *International Conference on Security and Cryptography for Networks*, pages 538–557. Springer, 2014.

[10] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Annual Cryptology Conference*, pages 585–605. Springer, 2015.

[11] Sunoo Park, Krzysztof Pietrzak, Joël Alwen, Georg Fuchsbauer, and Peter Gazi. Spacecoin: A cryptocurrency based on proofs of space. Technical report, 2015.

[12] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract] y. *ACM SIGMETRICS Performance Evaluation Review*, 42(3):34–37, 2014.

[13] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *International Conference on Financial Cryptography and Data Security*, pages 142–157. Springer, 2016.

[14] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[15] Silvio Micali. Algorand: the efficient and democratic ledger. *arXiv preprint arXiv:1607.01341*, 2016.

[16] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.

[17] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[18] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[19] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper: https://www.chainwhy.top/upload/default/20180619/126a057fef926dc286accb372da46955.pdf, August*, 19, 2012.

[20] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6: https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf, fall*, 2014.

[21] Andrew Poelstra. Distributed consensus from proof of stake is impossible, 2014.

[22] Alexander Russell, Cristopher Moore, Aggelos Kiayias, and Saad Quader. Forkable strings are rare. *IACR Cryptology ePrint Archive*, 2017:241, 2017.

[23] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 66–98. Springer, 2018.

[24] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 913–930, 2018.

[25] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros chronos: Permissionless clock synchronization via proof-of-stake. *IACR Cryptol. ePrint Arch.*, 2019:838, 2019.

[26] Aggelos Kiayias and Alexander Russell. Ouroboros-bft: A simple byzantine fault tolerant consensus protocol. *IACR Cryptol. ePrint Arch.*, 2018:1049, 2018.

[27] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 157–174. IEEE, 2019.

[28] Christian Badertscher and Aggelos Kiayias. Tutorial: The ouroboros protocol family. 2019. https://aft.acm.org/wp-content/uploads/2019/10/Ouroboros_AFT19_Tutorial.pdf.

[29] Juan A Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, pages 281–310, 2015.

[30] Ran Canetti. Universally composable signature, certification, and authentication. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 219–233. IEEE, 2004.

[31] Charles Charles Miller Grinstead and James Laurie Snell. Introduction to probability. 1997.

[32] Noga Alon and Joel H Spencer. The probabilistic method. 2008.

[33] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[34] Lawrence C Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[35] Michael JC Gordon and Tom F Melham. Introduction to hol a theorem proving environment for higher order logic. 1993.

[36] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[37] Lawrence C Paulson. The inductive approach to verifying cryptographic protocols. *Journal of computer security*, 6(1-2):85–128, 1998.

[38] Tobias Nipkow and David Von Oheimb. Java light is type-safe—definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–170, 1998.

[39] Max W. Haslbeck and Manuel Eberl. Skip lists. *Archive of Formal Proofs*, January 2020. https://isa-afp.org/entries/Skip_Lists.html, Formal proof development.

[40] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[41] Simon Holmgaard Kamp, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, Søren Eller Thomsen, and Daniel Tschudi. Weight-based nakamoto-style blockchains. In Patrick Longa and Carla Ràfols, editors, *Progress in Cryptology – LATINCRYPT 2021*, pages 299–319, Cham, 2021. Springer International Publishing.

[42] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

[43] Ling Ren. Analysis of nakamoto consensus. *IACR Cryptol. ePrint Arch.*, 2019:943, 2019.

[44] Jing Li and Dongning Guo. On analysis of the bitcoin and prism backbone protocols in synchronous networks. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 17–24. IEEE, 2019.

[45] Rafael Pass and Elaine Shi. The sleepy model of consensus. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 380–409. Springer, 2017.

[46] Søren Eller Thomsen and Bas Spitters. Formalizing nakamoto-style proof of stake. *arXiv preprint arXiv:2007.12105*, 2020.

[47] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Tight consistency bounds for bitcoin. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 819–838, 2020.

[48] Erica Blum, Aggelos Kiayias, Cristopher Moore, Saad Quader, and Alexander Russell. The combinatorics of the longest-chain rule: Linear consistency for proof-of-stake blockchains. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1135–1154. SIAM, 2020.

[49] George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 78–90, 2018.

[50] The coq proof assistant. https://coq.inria.fr/.

[51] Faria Kalim, Karl Palmskog, Jayasi Mehar, Adithya Murali, Indranil Gupta, and Phalgun Madhusudan. Kaizen: Building a performant blockchain system verified for consensus and integrity. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 96–104. IEEE, 2019.

[52] Kiran Gopinathan and Ilya Sergey. Towards mechanising probabilistic properties of a blockchain. CoqPL, 2019.

[53] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[54] Musab A Alturki, Jing Chen, Victor Luchangco, Brandon Moore, Karl Palmskog, Lucas Peña, and Grigore Roşu. Towards a verified model of the algorand consensus protocol in coq. In *International Symposium on Formal Methods*, pages 362–367. Springer, 2019.

[55] Vlad Zamfir. Casper the friendly ghost: A correct by construction blockchain consensus protocol. *Whitepaper: https://github.com/ethereum/research/blob/master/papers/caspertfg/caspertfg.pdf*, 2017.

[56] Ryuya Nakamura, Takayuki Jimba, and Dominik Harz. Refinement and verification of cbc casper. In *2019 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 26–38. IEEE, 2019.

[57] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[58] Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Roşu. Verification of casper in the coq proof assistant. Technical report, 2018. http://hdl.handle.net/2142/102075.

[59] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper. *arXiv preprint arXiv:2003.03052*, 2020.

[60] Javier D´ıaz and Wolfgang Jeltsch. Towards a formalization of the ouroboros protocol family.

[61] Wolfgang Jeltsch. A process calculus for formally verifying blockchain consensus protocols. In *Declarative Programming and Knowledge Management*, pages 24–39. Springer, 2019.

[62] Kaiwen Zhang and Hans-Arno Jacobsen. Towards dependable, scalable, and pervasive distributed ledgers with blockchains. In *ICDCS*, pages 1337–1346, 2018.

[63] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, Whitepaper: https://blockchainlab.com/pdf/corda-introductory-whitepaper-final.pdf, August*, 1:15, 2016.

[64] Nick Szabo. Secure property titles with owner authority. *Online at http://szabo. best. vwh. net/securetitle. html*, 1998.

[65] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.

[66] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*, pages 520–535. Springer, 2017.

[67] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77, 2018.

[68] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[69] Thomas Osterland and Thomas Rose. Model checking smart contracts for ethereum. *Pervasive and Mobile Computing*, 63:101129, 2020.

[70] Max DiGiacomo-Castillo, Yiyun Liang, Advay Pal, and John C Mitchell. Model checking bitcoin and other proof-of-work consensus protocols. In *2020*

IEEE International Conference on Blockchain (Blockchain), pages 351–358. IEEE, 2020.

[71] Jannik Dreier, Ali Kassem, and Pascal Lafourcade. Automated verification of e-cash protocols. In *International Conference on E-Business and Telecommunications*, pages 223–244. Springer, 2015.

[72] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *Acm Sigplan Notices*, 36(3):104–115, 2001.

[73] Bruno Blanchet et al. An efficient cryptographic protocol verifier based on prolog rules. In *csfw*, volume 1, pages 82–96. Citeseer, 2001.