MASTER IN COMPUTER SCIENCE

Proof-of-stake blockchain: Ouroboros

Analysis of epoch duration and transaction confirmation time

Master Thesis

Arbër Kuçi

Faculty of Science, University of Bern

March 2021

Prof. Christian Cachin

Dr. Giorgia Azzurra Marson

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Proof-of-stake based blockchains are an *eco-friendly* alternative of *proof-of-work* based blockchains. Ouroboros protocol is the first provably secure *proof-of-stake* based blockchain protocol. Although the research paper introducing Ouroboros comes with rigorous security guarantees, its security analysis is asymptotic and therefore insufficient for selecting concrete parameters that make the protocol secure. The focus of our work is to analyze the protocol and make claims about its parameters.

For blockchain protocols that follow the so-called *Nakamoto style consensus* with probabilistic finality, including Ouroboros, the transaction confirmation time should be set such that the probability of reverting a transaction after it is confirmed by the system is negligible. The transaction confirmation time is analyzed in this thesis.

In the Ouroboros protocol, randomness must be produced periodically to be able to *randomly select a committee of parties* that governs the system for a period of time called *epoch*. This randomness is generated by the *randomly elected committee of parties* through a secure coin tossing protocol. This randomness can not be controlled by an adversary as long as the majority of parties playing the coin tossing protocol are correct. The duration of an *epoch* should be such that the majority of parties playing the coin tossing protocol are correct with overwhelming probability. The epoch duration is analyzed in this thesis.

Contents

1 Introduction

The celebrated Bitcoin system [18] which realizes an electronic payment system has inspired a lot of new research. The Bitcoin system solves a problem that before has been thought to be very challenging, namely, it realizes a distributed protocol that is run by parties that do not trust each other, where the participation is granted to everyone who wishes to join the protocol and as some results (e.g. [11]) have shown, under certain assumptions like *correct majority of computational power* and *synchronous communication* the protocol is secure.

In Bitcoin, parties can contribute to the network only when they can prove to the network that they have solved the so called *proof-of-work*, i.e., that they have invested an expected amount of computational resources. The amount of computational resources depends on the participation on the network; higher participation means more computational resources are expected to be able to contribute in the system. As the participation has grown, the estimated number of hashes that the Bitcoin network evaluates every second at the time of this writing is 170 quintillion [5].

Although Bitcoin is indeed an ingenious solution, the *proof-of-work* is very energy demanding and at the time of this writing the network consumes energy as much as Argentina according to [2].

This issue became obvious as the network grew and was discussed extensively in the Bitcoin community and the idea of replacing *proof-of-work* with *proof-of-stake* was proposed.

The idea of a *proof-of-stake* blockchain is to select parties randomly with probability proportional to the stake that they own. In this way, instead of running the system based on the computational resources, the system is run based on the stake that the parties own, i.e., the higher the stake that the party owns, the bigger the role that the party has in the governance of the system. Realizing such a system where parties are randomly selected is challenging in a permissionless setting.

In *proof-of-work* based blockchains, the *proof-of-work* can be tuned in such a way that the probability of coming up with a *solution* of the *proof-of-work* without investing the expected amount of computational resources is negligible. This is not the case with *proof-of-stake* based blockchains like *Ouroboros* which makes it very hard to design such protocols that are secure.

Another major aspect of the protocols that follow the so-called *Nakamoto-style* consensus is the confirmation time. In protocols that follow this style of consensus there is no immediate agreement on the order of blocks; however, parties converge towards agreement as the time passes and typically it is acceptable that parties do not agree only on some of the most recent blocks (e.g. most recent k blocks). This kind of protocols guarantees with overwhelming probability that correct parties agree on the chain only up to the last k blocks and the last k blocks are not immutable and might change over time. Considering this, once a *transaction* is part of a chain, it is necessary to wait for k blocks before claiming that the transaction is confirmed from the system. The value of k should be picked such that once the system confirms a transaction, the probability of reverting the transaction should be negligible.

The confirmation time e.g. for Bitcoin is 6 blocks. The choice of 6 is not necessarily derived from a close analysis of the protocol and as e.g. [16] shows, an *adversary* controlling only *25%* of the computational resources can create forks of length 6 with probability *5%*.

Ouroboros [15] was one of the first provably secure *proof-of-stake* blockchain. While the results in [15] are outstanding, implementing Ouroboros is not straightforward. One of the reasons is that the asymptotic security analysis done in [15] does not allow for estimation of parameters of the protocol. The goal of this thesis is to estimate relevant parameters that are necessary for implementing the protocol. One of the parameters that is analyzed in this thesis is *transaction confirmation time*.

The Ouroboros protocol is governed by a *randomly selected committee of parties*. This *randomly selected committee of parties* runs the system only for a period of time called *epoch* and then the *committee* is refreshed based on some randomness produced by playing a secure coin tossing protocol. This randomness can not be controlled by an adversary as long as the *committee* playing the coin tossing protocol has a correct majority. The duration of an *epoch* should be picked such that the probability of a correct majority playing the secure coin tossing protocol happens with overwhelming probability. The other parameter that is estimated in this thesis is *epoch duration*.

There are two main contributions in this thesis:

1. We estimate the duration of an epoch. The Ouroboros protocol is run by a *randomly selected committee* of parties. This committee is responsible for running the protocol for a period of time and is periodically refreshed. The time that a *committee* is responsible for running the protocol is called an *epoch*. An *epoch* could last something like one day, one week or one month. However, it is desirable to be able to set the duration of an epoch to an appropriate duration based on some formal reasoning. We estimate the duration of an epoch by taking into account the adversarial stake.
2. We estimate the transaction confirmation time. The transaction confirmation time is an important aspect of systems that follow the so called *Nakamoto consensus*. Here we estimate the transaction confirmation time assuming a *covert adversary* and taking into account the adversarial stake.

1.1 Organization

Chapter 2 of the thesis presents relevant background that is necessary to be able to follow the thesis.
Chapter 3 describes the Ouroboros protocol as well as some other results from [15] relevant for the theory of blockchains.
Chapter 4 is about estimating the duration of an epoch of the Ouroboros protocol.
Chapter 5 estimates the transaction confirmation time assuming a covert adversary and assuming a

particular attack.

Chapter 6 is a discussion about the progress of this thesis as well as some possible future work that extends the work of this thesis.

2
Preliminaries and Notation

2.1 Blockchain Notations

A *blockchain* (or simply a *chain*) consists of a sequence of *blocks* where each *block* contains a set of operations (transactions in the context of cryptocurrencies). Each *block* of the *blockchain* has a pointer to the previous block.

The i^{th} block of a blockchain, denoted as B_i is a tuple consisting of four elements $B_i = (h_i, \bar{x}, sl, \sigma)$ where $h_i = H(B_{i-1}) \in \{0,1\}^{\lambda}$ and H is a cryptographic hash function. The element $\bar{x} \in \{0,1\}^{*}$ is a set of operations and σ is the signature signed by the party who produced the block B_i. sl can be considered as the time when the block is produced; more precisely the slot sl is the round when the block is produced. A round here is called a slot as in [15].

The first block B_0 of the blockchain is a special block called the *genesis block* and depending on the blockchain protocol, it can be different from the rest of the blocks of the blockchain. The length of a chain C is the number of its blocks and is denoted as $|C| = n$ if the chain C has n blocks.

By pruning the last k blocks from the chain C where $k \in \mathbb{N}$ a new chain that is identical to C except for the last k blocks is obtained and is denoted as $C^{\lceil k}$. If $k \geqslant |C|$ then $C^{\lceil k} = \varepsilon$ where ε is the empty chain.

If two chains C_1 and C_2 of the same length are identical except for the last k blocks then it denoted as $C_1^{\lceil k} \preceq C_2$ and $C_2^{\lceil k} \preceq C_1$ where $\preceq$ is the prefix relation between two chains. The last block of the chain C is denoted as *head(C)* .

2.2 Timing Assumptions

An important aspect of distributed systems is arguing about the behavior of parties concerning the passage of time. Knowing whether it is possible to make assumptions or not on how long does it take for a party to perform a local computation step or to send a point to point message is very important.

There are systems where it is assumed that the parties do not have access to local clocks and no assumptions are made on how long does it take for a party to either perform a local computation step or to send a point to point message or both. Systems that operate under these assumptions are called *asynchronous systems*.

If assumptions are made on how long does it take for a party to perform a local computation step or to send a point to point message then these systems are called *synchronous systems*. If the system is modeled in the *synchronous model*, the following assumptions are made:

1. *Synchronous computation*. There is a known upper bound on how long does it take for a party to perform a local computation step.
2. *Synchronous communication*. There is a known upper bound on how long does it take for a party to send a point to point message.
3. *Synchronous physical clocks*. Every party has access to a clock and there is a known upper bound on how much can a clock deviate from a global real-time clock.

For more details on how the time is modeled in distributed systems, see, e.g. [7].

2.3 Cryptographic Primitives

2.3.1 Cryptographic hash functions

Hash functions [19]. A cryptographic hash function is a function that maps the set of arbitrary long bit strings to the set of bit strings of fixed length λ, i.e., H: $\{0,1\}^* \mapsto \{0,1\}^\lambda$, and satisfies the following security properties:

- *Collision resistance.* It should be computationally hard to find $x \in \{0,1\}^*$ and $y \in \{0,1\}^*$ such that $x \neq y$ and $H(x) = H(y)$.
- *Second-preimage resistance.* Given a value $x \in \{0,1\}^*$ it is computationally hard to find another value $y \in \{0,1\}^*$ such that $y \neq x$ and $H(x) = H(y)$.

2.3.2 Digital signatures

Digital signatures [13]. A digital signature scheme is a tuple of three algorithms (*KeyGen*, *Sign*, *Ver*) defined as follows:

1. The key generation algorithm *KeyGen* is a randomized algorithm that outputs a key pair (pk, sk); the *public key* pk and the *secret key* sk.
2. The signing algorithm *Sign* is a randomized algorithm that takes as input a secret key sk and a message m from the message space and outputs a signature σ.
3. The verification algorithm *Ver* is a deterministic algorithm that takes as input a public key pk, a message m and a signature σ. It outputs a single bit b, where $b = 1$ means that signature σ is a valid signature for the given message m and $b = 0$ means the signature σ is not valid for message m.

A digital signature scheme, with probability 1 should satisfy the following:

- For every message m from the message space, for every key pair (pk, sk) output by *KeyGen* algorithm, $Ver(pk,\ m,\ Sign(sk,\ m)) = 1$.

When modeling the security of a digital signature scheme, the adversary $\mathcal{A}$ should have the possibility to ask for signatures on any message m he chooses and, of course, can verify the signatures on its own.

A digital signature scheme is secure against *existential forgery attacks* if it is computationally hard for an adversary $\mathcal{A}$ to come up with a signature σ and a message m such that the verification algorithm verifies the signature σ for this particular message m and $\mathcal{A}$ has never asked for the signature on the message m.

2.4 Asymptotic analysis of functions

Besides arguing if an algorithm is correct, one important issue to consider is the performance of the algorithm. One straight forward way of evaluating the performance of an algorithm is to try to take into consideration every operation of the algorithm by evaluating the cost of each operation of the algorithm. This approach is correct but it is very tedious.

For convenience reasons, not every operation is taken into consideration. Instead, it is common to find how the algorithm performs depending on the input of the algorithm; namely, figure out how many "*steps*" does the algorithm perform depending on the input. In general, when analyzing algorithms, an upper bound, a lower bound or a function that is a lower and an upper bound is needed to be able to evaluate the performance of an algorithm. In the next sections, the lower bound Ω, the upper bound O as well as the lower and upper bound Θ notation is introduced based on [8].

The set of functions Ω

The Ω notation is a lower bound for a set of functions. Let $g, f : \mathbb{N} \to R_{\geqslant 0}$ be two functions from natural numbers to non-negative real numbers.

The set $\Omega(g(n))$ is the set of all functions f, such that such that $\exists\ c \in R_{>0}$ and $n_0 \in \mathbb{N}$ such that $cg(n) \leqslant f(n)$ for all $n \geqslant n_0$.

In other words, the set $\Omega(g(n))$ is the set of all functions f, such that function f grows at least as fast a constant multiple of the function $g(n)$. If a function $f \in \Omega(g(n))$ it is common to denote is as $f = \Omega(g(n))$.

Example. Let $f(n) = 2^n$ and $g(n) = log_2(n)$. Then $f \in \Omega(g(n))$ because for a constant $c = 1$ and $n_0 = 1$ the function $2^n \geqslant log_2(n)$ for all $n \geqslant n_0$.

The set of functions O

Contrary to Ω notation, the O notation is an upper bound for a set of functions. Let g, f be two functions defined as above.

The set $O(g(n))$ is the set of all functions f, such that such that $\exists\ c \in R_{>0}$ and $n_0 \in \mathbb{N}$ such that $cg(n) \geqslant f(n)$ for all $n \geqslant n_0$.

In other words, the set $O(g(n))$ is the set of all functions f, such that function f grows no faster than a constant multiple of the function $g(n)$.

Example. Let $g(n) = 2^n$ and $f(n) = log_2(n)$. Then $f \in O(g(n))$ because for a constant $c = 1$ and $n_0 = 1$ the function $2^n \geqslant log_2(n)$ for all $n \geqslant n_0$.

The set of functions Θ

The Θ notation is a lower bound and an upper bound for a set of functions. Let g, f be two functions defined as above. A function $f = \Theta(g(n)) \iff f = O(g(n)) \wedge f = \Omega(g(n))$.

Example. Function $f(n) = n(n-1)/2 = \Theta(n^2)$. This is because $f(n) \in \Omega(n^2)$ since $0.1n^2 \leqslant n(n-1)/2$ for all $n \geqslant 2$ and $f(n) \in O(n^2)$ since $n^2 \geqslant n(n-1)/2$ for $n \geqslant 1$.

2.5 Probabilistic preliminaries

2.5.1 Discrete random variables, expectation and the Chernoff bound

When analyzing randomized experiments, determining the set of possible outcomes ω is the first step. E.g. when tossing a coin, the set of possible outcomes is $\omega = \{head, tail\}$. The experiment of coin tossing could be posed like this: if *head* comes it is a win, if *tail* comes it a loss and it is convenient to write 1 whenever *head* comes and 0 whenever *tail* comes. Such a function that maps the set of possible outcomes to the real numbers is called a random variable.

Definition 1 (Random variables [17]). A random variable *X* on a sample space ω is a real-valued function on ω; that is, *X*: $\omega \mapsto \mathbb{R}$. A discrete random variable is a random variable that takes only a finite or countably infinite number of values.

A random variable that takes value 1 if the experiment succeeds and 0 otherwise is called a *Bernoulli* random variable [17].

One characteristic of a random variable is its expectation. The expected value of a random variable is a weighted average of the values it can take, where each value is weighted by the probability that the random variable takes that value.

Definition 2 (Expected value [17]). The expected value of a random variable *X* is denoted by $E[X]$ and computed as follows:

$$E[X] = \sum_i iPr[X = i]$$

where the summation is done over all values that *X* can take.

Example. Consider the experiment of coin tossing and let *X* be the random variable representing coin tossing as described above, then $E[X] = 1 \cdot Pr[X = 1] + 0 \cdot Pr[X = 0] = 1/2$.

Theorem 1 ([17]). *For a finite number of discrete random variables $X_1, ...X_n$ with finite expectations,*

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

Theorem 2 (Chernoff bound [17]). *Let $X_1, ..., X_n$ be independent Bernoulli random variables such that $Pr[X_i = 1] = p$ for $p \in [0,1]$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. The following bound holds: $Pr[X \geqslant (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$ for any $\delta > 0$.*

Note: The Chernoff bound above holds not only for random variables that are identically distributed, but for the purpose of this thesis, this is enough.

3 Ouroboros

In this chapter, the Ouroboros protocol is described based on [15] and [9]. Besides describing the Ouroboros protocol, some other results from [15] such as the fork abstraction that seem relevant for the theory of blockchain protocols are described.

3.1 Introduction

The Ouroboros protocol runs in slots and a sequence of slots makes an epoch. Before the start of an epoch, a committee of parties that is responsible for producing blocks during an epoch is elected. The election of the committee must be random and for this election to be randomized some source of randomness is necessary before the start of every epoch. The source of randomness for electing the first committee that produces the blocks of the first epoch comes from the genesis block. The randomness for a particular epoch after the first one comes from the coin tossing protocol played by the committee of the previous epoch. The coin tossing protocol guarantees randomness that can not be controlled by an adversary $\mathcal{A}$ as long as the majority of the committee members participating in the protocol are correct. That is why it is very important to make sure that the majority of the elected committee is correct in every epoch.

The idea of proof-of-stake (PoS) based blockchains is completely different on how new blocks are produced. While in Bitcoin, a new block is produced as soon as a party manages to solve a cryptographic hard problem, in PoS blockchains, the responsibility for producing a block at a certain point in time is done randomly based on the stake that the party owns. More precisely, as soon as the the randomness is made available either for the first epoch from the genesis block or for subsequent epochs from the coin tossing protocol, it can be verified immediately who is responsible for producing blocks at any point in time during the whole epoch. The probability of a party being elected to produce a block is proportional to the stake that the party owns, otherwise this could be exploited by $\mathcal{A}$ e.g. to get more corrupted parties to produce blocks.

3.2 Model

Timing assumptions. The time is split into discrete units called *slots* and at every slot, the parties that maintain the blockchain produce one block if the elected party for the respective slot is correct. The model in which the protocol operates is *synchronous* and it is assumed that parties have access to clocks which should be *roughly* synchronized. Clocks are allowed to deviate but this deviation should be significantly less than the duration of one slot. In every slot there is one party that is elected to produce one block called the *leader* of the slot; the *slot leader* is publicly known and has the exclusive right to produce one block in the respective slot. A number of slots is grouped into epochs. During an epoch, the parties, except for producing blocks they run a secure coin tossing protocol to produce randomness which is then used to elect the slot leaders of the next epoch and because of this, the length of an epoch should be such that it accommodates the execution of the coin tossing protocol. Given that the slot leaders are known in advance for a whole epoch, this approach of having the slot leaders known publicly for a complete epoch in advance gives the adversary an advantage to interfere, in particular the eclipse attack becomes relevant if the length of en epoch is very long.

Note [3]: The length of one slot in blockchains like Cardano that are based on one of the Ouroboros protocols is 20 seconds and an epoch is 21600 slots long, which is about 5 days.

Correct majority. The correct majority in the Ouroboros protocol is assumed to be the set of correct players that collectively own more than 50% of the total stake in the system during the whole execution. The correct majority is assumed to follow the protocol strictly during the whole execution and does not deviate from the protocol for any reason.

Adversary $\mathcal{A}$. In this model, it is assumed that the adversary $\mathcal{A}$ is in full control of the network. Because of this $\mathcal{A}$ can see the content of the messages exchanged between parties and can reorder messages. $\mathcal{A}$ can also delay messages with the exception that the synchronous model assumption holds. The adversary can also corrupt correct parties and the set of corrupted parties can collude in any imaginable way with the goal of breaking the system. The corruption is done under the following assumptions:

- From the moment that the adversary attempts to corrupt a correct party, it takes some time until the party falls under control of the adversary. This means that corruption does not happen instantly. This delay is called *corruption delay* and is expressed in terms of slots.
- Throughout the execution of the protocol, the adversary can corrupt parties under the assumption of the correct majority.

3.3 Properties of the protocol

Blockchain is one possible way to realize a distributed ledger. In a distributed ledger, it is desirable to be able to confirm transactions fast enough and once confirmed to become immutable. These characteristics of a robust transaction ledger are captured with the following two properties:

- **Persistence with parameter** $k \in \mathbb{N}$. Once a correct party of the system claims that a transaction *tx* is *stable*, the remaining nodes, if queried will report the transaction in the same position in the ledger and will agree on the entire prefix of the ledger prior to *tx*. A transaction is declared *stable* if and only if it is k blocks deep in the ledger.
- **Liveness, with parameter** $u \in \mathbb{N}$. Once a certain transaction is available in the network, after the passage of time that corresponds to u slots, all correct parties, if queried will report the transaction as *stable*.

The *persistence* property of a distributed ledger can be derived from the following property of a blockchain, given that the distributed ledger is realized from a blockchain:

- **Common prefix (CP), with parameter** $k \in \mathbb{N}$. Assume C_1 and C_2 are two chains adopted by two correct parties. Assume *head(C_1)* is produced at slot sl_1 and *head(C_2)* is produced at slot sl_2 such that $sl_1 \leqslant sl_2$ then $C_1^{\lceil k} \preceq C_2$.

The *liveness* property of a distributed ledger can be derived from the following properties of a blockchain, given that the distributed ledger is realized from a blockchain:

- **Honest Chain Growth (HCG) with parameters** $\tau_{hcg} \in (0,1]$ and $s_{hcg} \in \mathbb{N}$. Assume *C* is a chain adopted by a correct party where *head(C)* is produced at the slot sl_2. Assume sl_1 is a slot prior to sl_2 where *C* contains a block generated by a correct party. If $sl_1 + s_{hcg} \leqslant sl_2$ then the number of blocks appearing in the chain *C* after the slot sl_1 is at least $\tau_{hcg}s_{hcg}$.
- **Existential Chain Quality (∃CQ) with parameter** $s_{\exists cq} \in \mathbb{N}$. Assume *C* is a chain adopted by a correct party and *sl* is the slot where *head(C)* is produced. At any portion of $s_{\exists cq}$ consecutive slots up to the slot *sl*, the chain *C* contains at least one block generated by a correct party.

The *liveness* property of a distributed ledger can be derived by the following blockchain property as well, called chain growth (CG) property which itself can be derived by combining ∃CQ and HCG properties:

- **Chain Growth (CG) property with parameters** $\tau_{cg} = \frac{\tau_{hcg}s_{hcg}}{2s_{\exists cq} + s_{hcg}}$ and $s_{cg} = 2s_{\exists cq} + s_{hcg}$. Assume *C* is a chain adopted by a correct party and *sl* is the slot where *head(C)* is produced. At any portion of s_{cg} consecutive slots up to the slot *sl*, the chain *C* contains at least $\tau_{cg}\ s_{cg}$ blocks.

3.4 Ouroboros protocol

In this section, the Ouroboros protocol is described. The protocol is split in three main parts, the **Ouroboros protocol (Protocol 1)**, the **Slot leader algorithm (Algorithm 1)** and the **Chain validation algorithm (Algorithm 2)**.

Ouroboros protocol. The **Ouroboros** protocol describes what a party does in a slot, how a party advances to the next slot and how a party advances from one epoch to the next.

The Ouroboros protocol runs in slots where each slot lasts Δ units of time and after a certain number of R slots, a new epoch is started.

During the first epoch, the parties that are the leaders of at least one slot of the first epoch, i.e., the elected committee to produce the blocks of the first epoch, invoke the coin tossing protocol to get randomness for the second epoch (line 11 of **Protocol 1**). The coin tossing protocol must finish and output the randomness to all correct parties before the current running epoch is finished. The output of the coin tossing protocol is stored in a hash table ρ indexed by epochs (line 5 of **Protocol 1**).

For the first epoch of the protocol, the randomness and the stake distribution is present in the *genesis block* B_0. Based on the initial randomness and stake distribution from the *genesis block* B_0, parties can figure out who can produce a block at a certain slot of the first epoch by invoking the **Slot Leader** algorithm.

At every slot, each party checks if they are the leader of the slot. If a party is the leader of the slot, then the party selects some transactions from the set of transactions (transaction set is declared at line 2 of **Protocol 1**), constructs the block associated with the respective slot and broadcasts the block. As soon as a party sees a transaction, the party stores the transaction in its set of transactions and broadcasts it to the other parties.

The protocol is parameterized with the predicate **ValidChain**(C) which is true for a chain C if the chain does not contain transactions that spend more than once the same coin.

Once the protocol is at the last slot of an epoch, the protocol moves to the next epoch, calculates the stake distribution S based on the local chain C (excluding the last k blocks). The calculation of the stake distribution (line 26 of **Protocol 1**) must be done in a deterministic manner. Once the stake distribution is calculated then the slot leaders are determined by invoking the **Slot Leader** algorithm. Once the leaders of the epoch that is about to start are known, the coin tossing protocol is invoked to get the randomness for the following epoch and then the next slot starts after Δ units of time.

Protocol 1 Ouroborus protocol as run by party P_i. Parameterized by slot duration Δ, *genesis block* B_0, length of an epoch R, hash function *H*, the predicate **ValidChain**(*C*) and the key pair (pk_i, sk_i)

```
1:  $C \leftarrow \varepsilon$
2:  $transactions \leftarrow \emptyset$
3:  $slot \leftarrow 0$
4:  $epoch \leftarrow 1$
5:  $\rho[epoch] \leftarrow B_0$                      ▷ Randomness for the initial epoch.
6:  $S \leftarrow B_0$                               ▷ The initial stake distribution.
7:  for i = 1 to R do                                ▷ Determine the leaders of the initial epoch.
8:      $leaders[i] \leftarrow$ SlotLeader($S$, $\rho[epoch]$, i)
9:  end for
10: if $P_i$ is the leader of any of the slots of the first epoch then
11:     Invoke CoinTossing protocol for the epoch ($epoch$+1)
12: end if
13: while True do
14:     $slot \leftarrow slot + 1$
15:     if $leaders[slot] = pk_i$ then
16:         Pick $trxs \in transactions$ s.t. ValidChain($C$) holds when appending a new block with $trxs$ in $C$.
17:         if $C = \varepsilon$ then
18:             $h \leftarrow H(B_0)$
19:         else
20:             $h \leftarrow H(head(C))$
21:         end if
22:         $transactions \leftarrow transactions \setminus trxs$
23:         $\sigma \leftarrow Sign(trxs \,||\, slot \,||\, h, sk_i)$
24:         $B_{slot} \leftarrow (trxs, slot, h, \sigma)$
25:         $C \leftarrow C + B_{slot}$                ▷ Add block $B_{slot}$ at the end of the chain $C$.
26:         Broadcast($C$)
27:     end if
28:     if $slot$ mod R = 0 then
29:         $epoch \leftarrow epoch + 1$
30:         $S \leftarrow$ Calculate stake distribution based on the local chain $C$.    ▷ Deterministic operation.
31:         for i = (($epoch$ - 1) * R + 1) to ($epoch$ * R) do
32:             $leaders[i] \leftarrow$ SlotLeader($S$, $\rho[epoch]$, i)    ▷ Determine the leaders of the next epoch.
33:         end for
34:         if $P_i$ is the leader of any of the slots of the epoch $epoch$ then
35:             Invoke CoinTossing protocol for the epoch ($epoch$+1)
36:         end if
37:     end if
38:     Wait $\Delta$ units of time
39: end while
```

Slot leader algorithm. The *Slot leader algorithm* determines the leaders of slots. Determining the leader of a slot is done based on the randomness ρ, the stake distribution $S = \{(pk_1,s_1),...,(pk_n,s_n)\}$ where pk_i is the public key of user U_i and s_i is the stake that U_i owns and based on the slot.

The *Slot leader* algorithm maps the set $\{1,2,...,t\}$ to the set $\{pk_1,...,pk_N\}$ based on stake distribution, where *t* is the number of coins (the smallest unit of currency) in the system and the set $\{pk_1,...,pk_N\}$ is the set of stakeholders. After this mapping is constructed, one of the elements $c_i \in \{1,...,t\}$ is randomly selected and the image of c_i by this construction is returned, i.e., one stakeholder is randomly selected with probability proportional to the stake that the user owns. More coins that a user U_i has, more elements of $\{1,2,...,t\}$ are mapped to its public key pk_i and the probability that U_i is selected as the leader of a certain slot is proportional to the stake that U_i owns.

The function *sort* should be a deterministic function to make sure that all parties construct the same mapping.

Algorithm 1 Slot Leader algorithm parameterized by the deterministic function *sort*, by the number of coins (smallest unit of currency) in the system *t*, by stake distribution *S*, slot *sl* and randomness ρ.

```
1: function SLOTLEADER($S$, $\rho$, $sl$)
2:     $index \leftarrow 1$
3:     $\mathbf{F} \leftarrow \emptyset$
4:     for all $(pk_i, s_i) \in sort(S)$ do          ▷ Sort is a deterministic function.
5:         $s \leftarrow$ Convert $s_i$ to the smallest unit of the currency.
6:         $s \leftarrow s$ + index - 1
7:         repeat
8:             $\mathbf{F}(index) \leftarrow pk_i$
9:             $index \leftarrow index + 1$
10:        until $s > index$
11:    end for
12:    $h \leftarrow (H(\rho || sl) \bmod t) + 1$
13:    return $\mathbf{F}(h)$
14: end function
```

Chain Validation algorithm. The **Chain validation** algorithm shows how a party validates blockchains broadcast by other parties and how the party decides whether to keep its local chain or to accept the chain from the other party.

The following two assumptions are made in the **Chain validation** algorithm:

- **Chain Validation** algorithm can access and update the chain *C* (line 1 **Protocol 1**) that the party maintains locally.
- Parties have access to a function that returns the current slot called *CurrentSlot()*.
- **Chain Validation** algorithm has access to leaders of the slots; leaders of the slots are stored in the variable *leaders* in **Protocol 1**.

The algorithm is parameterized by the predicate **ViableChain**(C') which is defined as:

- **ValidChain**(C') $\wedge\ C' \neq \varepsilon \wedge |C| < |\ C'\ | \wedge sl \leqslant$ *CurrentSlot()* where *sl* is the slot where *head(C')* is produced and *C* is the local chain maintained by the party.

If the predicate **ViableChain** is true for the chain C' then the algorithm checks if the structure of the chain is correct. If the structure of the chain is correct then the following are true:

- The sequence of slots is strictly monotonic increasing namely, $sl_i < sl_{i+1}$ for all slots where the chain C' has blocks.
- At every block B_i, the value h_i should be the hash value of the previous block in the chain.
- Every block B_i produced at a slot sl_i is produced by the leader of the slot sl_i.

Whenever a chain C' is delivered to a party, if the predicate **ViableChain**(C') is true and the structure of the chain is correct, the party will adopt the chain C'.

Algorithm 2 Chain validation parameterized by *genesis block* B_0, the chain *C* that the party maintains locally and the predicate **ViableChain**(C')

```
function CHAINVALIDATION($C'$)
    isViable ← ViableChain($C'$)
    if isViable then
        repeat
            $B_{i+1}$ ← head($C'$)
            $C' \leftarrow C'^{\lceil 1}$
            if $|C'| = \varepsilon$ then
                $B_i \leftarrow B_0$
            else
                $B_i$ ← head($C'$)
            end if
            $(trxs_{i+1}, sl_{i+1}, h_{i+1}, \sigma_{i+1}) \leftarrow B_{i+1}$
            $(trxs_i, sl_i, h_i, \sigma_i) \leftarrow B_i$
            isViable ← $sl_{i+1} > sl_i \wedge h_{i+1} = H(B_i) \wedge$ Ver(leaders[$sl_{i+1}$], $trxs_{i+1}||sl_{i+1}||h_{i+1}$, $\sigma_{i+1}$)
        until $|C'| \neq \varepsilon \wedge$ isViable $\neq$ False
    end if
    if isViable then
        Update local chain C to $C'$.
    end if
end function
```

3.5 Fork abstraction and forkable strings

In the case when no corrupted nodes are elected as the leaders of any slot, then only correct parties are responsible for running the system; correct parties follow strictly the protocol and in this case no adversary $\mathcal{A}$ poses a risk to the system.

However, as the *Slot leader* algorithm determines the slot leaders randomly with the probability of a party being selected to lead a slot proportional to the stake that the party has, then the adversary $\mathcal{A}$ can corrupt some parties, buy some stake (assuming less than the majority of stake) and if the system runs for a long time, the corrupted parties will eventually be the slot leaders of some slot. Once corrupted nodes are slot leaders, they have a window of opportunity for interfering with the system and break the guarantees of the protocol.

To be able to argue about the chance of corrupted parties interfering with the system, it is necessary to introduce some convenient notation from [15] that describes a protocol execution and models the number of slots led by corrupted parties and the number of slots led by correct parties. One way of doing this is with *characteristic strings*. The characteristic string is a bit string, where a 0 bit corresponds to a slot where the leader of a slot is a correct party and a 1 bit corresponds to a slot where the leader is a corrupted party. More formally:

Definition 3 (Characteristic string)**.** Assume *E* is an execution of **Protocol 1** that runs for *m* slots. Then the *characteristic string* $w \in \{0, 1\}^m$ of *E* is defined as:

$$w_i = \begin{cases} 1 & \text{if the leader of the slot } i \text{ is a corrupted party.} \\ 0 & \text{if the leader of the slot } i \text{ is a correct party.} \end{cases}$$

for i ∈ {1, 2, 3, ..., *m*}. For a certain bit of the characteristic string if w_i = 0 then we say that *i* is a correct index otherwise we say that *i* is a corrupted index of w.

For an execution of the **Protocol 1** that runs for *m* slots, the characteristic string $w = 0^m$ would not give any chance to the adversary $\mathcal{A}$ to interfere with the chain, whereas for a characteristic string of $w = 1^m$ then the adversary $\mathcal{A}$ would have have the chance to present completely different chains to different correct parties and thus easily break properties of the system.

For a given characteristic string w of an execution of the **Protocol 1**, to be able to argue about what kind of chains can the adversary $\mathcal{A}$ build and present to the correct parties and what kind of chains can parties adopt it is necessary to introduce the notion of the fork from [15].

Definition 4 (Fork)**.** Assume $w \in \{0, 1\}^m$ is a characteristic string of an execution of **Protocol 1** and H = {i | w_i = 0} is the set of correct indices. A fork for the string w is a directed, rooted tree *F* = (V, E) with a labeling function *l* : V ↦ {0, 1,..., m} with the following properties:

- P_1 Each edge of *F* is directed away from the root.
- P_2 The root r ∈ V is given the label *l*(r) = 0.
- P_3 The labels of vertices along any path in the tree are strictly increasing.
- P_4 Each *i* ∈ H is the label of exactly one unique vertex in *F*.
- P_5 The function **d** : H ↦ {1, 2,..., m} is defined such that **d**(*i*) is the depth of the unique vertex *v* in *F* for which *l*(*v*) = i. The function **d** is strictly increasing.

F ⊢ w is the notation used to indicate that the *F* is a fork for the string w.

3.5.1 Examples of protocol executions and forks

Example 1. Assume that **Protocol 1** runs for 9 slots and the following characteristic string w = 000111000 corresponds to this execution. One possible fork *F* ⊢ w is the one in Figure 3.1.

The leaders of the first three slots are correct parties, they follow the protocol, they produce blocks in each slot and broadcast these blocks. Every correct party, before the end of the third slot will receive the chain C_2 containing nodes from the slots 1, 2 and 3. The leaders of the next three slots, namely the leaders of the slots 4, 5 and 6 are corrupted parties. They can deviate from the protocol in any imaginable way. One possible strategy that they can follow is to produce two chains in parallel, one chain committing to the block from the third slot produced by a correct party and one chain committing only to the *genesis block*. Assume that the slot leader P_i of the seventh slot joins the system only in the beginning of the seventh slot. Since the adversary $\mathcal{A}$ is in full control of the network, $\mathcal{A}$ broadcasts first the chain C_1 to the party P_i, party P_i adopts the chain C_1, extends C_1 with a new block and broadcasts it. In the view of correct parties, the chain C_1 including the node from the seventh slot is correct and the longest chain that they have seen so far. In the next slots, eight and nine correct parties responsible for producing the blocks simply extend the chain C_1.

This particular execution of the protocol that gives such a characteristic string where it is possible to come up with two different chains of the same length and they share only the genesis block is a situation that can cause issues. The reason for this is because in the tenth slot, $\mathcal{A}$ can present the chain C_1 to some parties and chain C_2 to some other parties; both chains are correct and so two groups of correct parties would accept two different chains. This would be very dangerous if $\mathcal{A}$ manages to run the system for a long time with such two parallel chains.

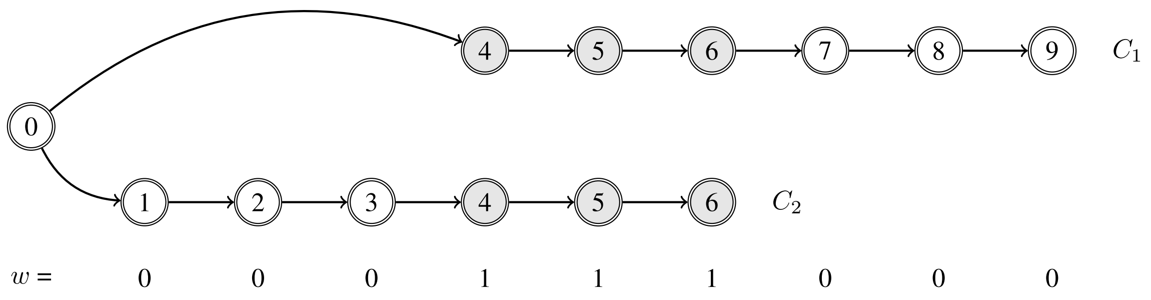

Figure 3.1: An example of fork $F \vdash w$ = 000111000

Example 2. In this example another execution of **Protocol 1** is shown. Assume the characteristic string of this execution is w = 00100100 and one possible for $F \vdash w$ of w is shown in Figure 3.2.

The first two slots are led by correct parties and they follow the protocol as expected. However, the third slot is led by a corrupted party and this party decides to build not on the longest chain C_3, but on the chain that does not contain vertex 2. In this way, transactions that are part of the vertex 2 are discarded because of malicious actions of the corrupted party. In the same way, the transactions that are part of the block 5 are discarded because of the malicious actions of the corrupted party that is the leader of the sixth slot.

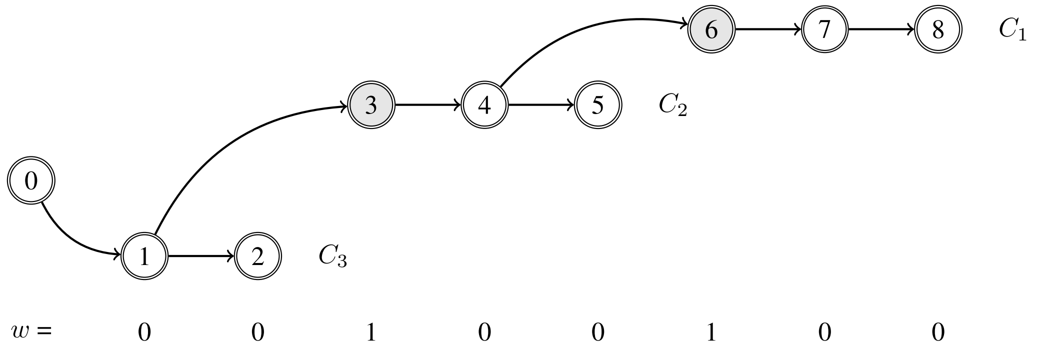

Figure 3.2: An example of fork $F \vdash w$ = 00100100

In Figure 3.1 the fork F has two chains, C_1 and C_2. In Figure 3.2 the fork F has three chains, C_1, C_2 and

C_3. For a chain C the length of the chain is denoted as *length(C)*, equal to the number of edges on the chain starting from the root vertex. The *height* of a fork is the length of the longest tine.

The $\sim$ relation. For two chains C_1 and C_2 of a fork *F* that share an edge the following notation is used: $C_1 \sim C_2$. If they share no edge then $C_1 \nsim C_2$.

Chains C_1 and C_2 in Figure 3.1 do not share an edge, i.e., $C_1 \nsim C_2$, whereas all possible pairwise combinations of chains in Figure 3.2 share an edge, i.e., $C_1 \sim C_2$, $C_1 \sim C_3$ and $C_2 \sim C_3$.

Flat forks. A fork is called *flat fork* if it has two chains C_1 and C_2 such that $C_1 \nsim C_2$ and their length equal to the height of the fork.

The fork *F* in Figure 3.1 is a flat fork and the fork *F* in Figure 3.2 is not a flat fork.

Forkable strings. A characteristic string w is called forkable string if there exists a flat fork $F \vdash w$.

Characteristic string w = 000111000 in the Figure 3.1 is a forkable string.

Hamming weight. The Hamming weight of a characteristic string w, denoted as *weight*(w) is the number of ones in the characteristic string w.

3.5.2 Characteristic strings with low hamming weight

In this section we show that characteristic strings that contain correct indices of at least more than $2/3$ of the total bit strings are not forkable. This proof is essentially the same as the proof of the Proposition 3.7 from [14].

Theorem 3. *Let* $w \in \{0,1\}^m$. *If* weight$(w) < m/3$ *then characteristic string* w *is not forkable.*

Proof. Assume that w is a forkable string. We show that the *weight*$(w) \geqslant m/3$. By proving this contrapositive, the statement of the theorem is proved.

Assuming w is forkable, then there exists a *flat fork* $F \vdash w$. Since *F* is a *flat fork*, then there exist at least two chains C_1 and C_2 of *F* such that $C_1 \nsim C_2$ of the same length, i.e., *length(C_1)* = *length(C_2)* and, the length of these two chains is the same as the height of the fork *F*.

Assume *t* = *weight*(w) then the characteristic string w has $m - t$ correct indices and *t* corrupted ones.

The corrupted indices of w can contribute with either zero or one vertex in C_1. The same applies to C_2.

Given that there are *t* corrupted indices, the maximum number of nodes from corrupted indices in both chains C_1 and C_2 combined would be $2t$ (if all corrupted indices contribute with exactly one vertex in chain C_1 and one vertex in C_2). The minimum number of nodes from corrupted indices in both chains C_1 and C_2 would be zero if every corrupted index contributes with no vertex at all.

From the property P_4 of **Definition 4**, each correct index corresponds to exactly one vertex in one of the chains of the fork. Given that there are in total $m - t$ correct indices, the number of nodes from correct indices is exactly $m - t$.

It follows that the *maximum number* of nodes would be $2t$ (from the corrupted indices) and $m - t$ (from correct indices), in total $2t + m - t = t + m$.

From the property P_5 of **Definition 4**, the depth of the nodes corresponding to correct indices is strictly increasing. Given that there are in total $m - t$ correct indices, the depth of the vertex corresponding to the last correct index is at least $m - t$. This means that the length of the chain that contains the node that corresponds to the last correct index is at least $m - t$, meaning that the height of the fork *F* is at least $m - t$.

Assuming that the height of F is at least $m - t$ and given that F is forkable then $m - t = \textit{length}(C_1) = \textit{length}(C_2)$; from $\textit{length}(C_1) = m - t$, C_1 has at least $m - t$ nodes; with the same logic C_2 has at least $m - t$ nodes; it follows that from correct indices both chains C_1 and C_2 combined have at least $m - t + m - t = 2(m - t)$ nodes.

Assuming that every corrupted index contributes with zero nodes in both chains C_1 and C_2 then the *minimum number* of nodes in both chains C_1 and C_2 combined is $2(m - t)$.

The *maximum number* of nodes should be at least as big as the *minimum number* of nodes in chains C_1 and C_2, i.e., $m + t \geqslant 2(m - t) \implies t = \textit{weight}(\mathrm{w}) \geqslant m/3$. □

3.5.3 Characteristic strings with high hamming weight

In this section we show that *characteristic strings* that have correct indices less than or equal to the half are always forkable.

Theorem 4. *Let $w \in \{0,1\}^m$. If the weight(w) $\geqslant m/2$ then characteristic string w is forkable.*

Proof. Let $\epsilon \in [0, 1/2]$. Assume that *weight*$(w) = \lceil (1/2 + \epsilon)m \rceil$, then the number of zeros in w is $\lfloor (1/2 - \epsilon)m \rfloor$.

From property P_4 of **Definition 4**, a chain C_1 that commits to the *root node* node can be constructed only with nodes corresponding to the correct indices with *length(C_1)* = $\lfloor (1/2 - \epsilon)m \rfloor$.

By selecting $\lfloor (1/2-\epsilon)m \rfloor$ indices out of $\lceil (1/2+\epsilon)m \rceil$ corrupted indices, yet another chain C_2 that commits to the *root node* with *length(C_2)* = $\lfloor (1/2 - \epsilon)m \rfloor$ can be constructed containing nodes corresponding only to corrupted indices.

The constructed fork F has two chains C_1 and C_2 that do not share any edge ($C_1 \nsim C_2$) with *length(C_1)* = *length(C_2)* which is the height of the fork F. This shows that there exists a flat fork $F \vdash w$, hence w is forkable. □

3.5.4 The density of forkable strings

In the previous sections we showed that for a given characteristic string $w \in \{0,1\}^n$, if the *weight(w)* $\geqslant n/2$ then w is always forkable and if the *weight(w)* $< n/3$ then w can not be forked.

While these two results give some insight on when an adversary might fork a chain, the assumptions to show these two results do not correspond to an execution of the **Protocol 1**. The reason for this is that we are assuming a fixed *hamming weight of the characteristic string*. More appropriate would be to ask what is the probability that a randomly chosen *characteristic string* $w \in \{0,1\}^m$ is forkable?

In [15] it is shown that for a characteristic string $w \in \{0,1\}^n$, if each w_i is assigned independently 1 with probability ϵ and 0 with probability $1 - \epsilon$ for some $\epsilon \in (0, 1/2)$ then w is forkable with probability that drops exponentially in n, i.e., probability that w is forkable is $2^{-\Omega(\sqrt{n})}$. In [20] this upper bound is improved to $2^{-\Omega(n)}$.

4
Epoch duration

In this chapter we estimate the duration of an epoch. The duration of an epoch should be picked such that the *randomly elected committee of parties* which carries out the secure coin tossing protocol for producing *randomness* has a correct majority with overwhelming probability.

4.1 Introduction

During an epoch, the most crucial events that happen in the Ouroboros protocol are the following ones:

- Blocks are produced from the committee of slot leaders of the epoch.
- The same committee of slot leaders utilizing a coin tossing protocol produces randomness used for refreshing the committee of slot leaders in the subsequent epoch.

In the previous section, it is shown that once the *hamming weight* of a *characteristic string* of length m is more than or equal to $m/2$, then this is certainly a forkable string. The *characteristic string* for an epoch is known as soon as the randomness for that epoch is available and the randomness for every epoch is available before the start of the epoch, meaning that the *characteristic string* for a whole epoch is known before the start of the epoch. This means that the adversary $\mathcal{A}$ knows before the start of an epoch if a fork can be constructed using the slots of the upcoming epoch. Ideally, the characteristic string of every epoch is not forkable. That is why it is important to argue about the probability of a corrupted majority of slot leaders of an epoch because a corrupted majority is essentially a forkable *characteristic string* of that epoch.

As already stated, the coin tossing protocol guarantees randomness that can not be controlled by adversary $\mathcal{A}$ as long as the majority of the participating parties in the coin tossing protocol are correct. Once the majority of the *randomly selected committee* of an epoch are corrupted parties, they can control the output of the coin tossing protocol. The result of this is that $\mathcal{A}$ has a chance to create a majority in the next epoch as well, and this may continue repeatedly. This is another important reason why we should argue about the probability of a corrupted majority of slot leaders of an epoch.

The duration of an epoch is estimated by arguing about the probability of a corrupted majority of slot leaders of an epoch. It is necessary to know how many slots should an epoch be such that the probability of a corrupted majority of slot leaders of the epoch is minimized. This estimation should also take into consideration the stake that the adversary $\mathcal{A}$ has. When minimizing this probability, i.e. when finding an upper bound on this probability, this upper bound should depend not only on the length of the epoch but also on the stake that $\mathcal{A}$ has. The stake that $\mathcal{A}$ has should also be considered because the probability of a party being elected for leading a slot is proportional to the stake that the party owns and this means that the probability of being slot leader of any of the parties corrupted by $\mathcal{A}$ is proportional to the stake that $\mathcal{A}$ has.

4.2 Probabilistic estimation of the duration of an epoch

The length of some epoch $e_j, j = 1, 2, ...$ will be denoted by R, where R is the number of slots of an epoch.

The idea is to keep track of whenever $\mathcal{A}$ is selected to lead a slot and denote this event with 1 and whenever a correct party is selected to lead a slot and denote this event with 0. More formally, the following random variable is introduced:

$$X_i = \begin{cases} 1 & \text{if the leader of the slot } i \text{ is a corrupted party.} \\ 0 & \text{if the leader of the slot } i \text{ is a correct party.} \end{cases}$$

If $X_i = 1$, it means that the leader of the slot i of some epoch e_j is a corrupted party and if $X_i = 0$, it means that the leader of the slot i of some epoch e_j is a correct party.

As already argued, the probability of a party begin selected to lead a slot is proportional to the stake that the party owns; it means that the probability of any of the corrupted parties of being selected to lead a slot is proportional to the stake that these parties collectively own. From the assumptions of the correct majority from the **Section 3.2**, $\mathcal{A}$ has no more than 50% of the total stake in the system. From this, $Pr[X_i = 1] = \epsilon$ for some $\epsilon \in (0, 1/2)$ and $Pr[X_i = 0] = 1 - \epsilon$. The expected value of X_i, $E[X_i] = 1 \cdot Pr[X_i = 1] + 0 \cdot Pr[X_i = 0] = \epsilon$.

We want to argue about the probability that the majority of slot leaders of an epoch of length R is corrupted, i.e., $R/2$ or more slot leaders are corrupted. For this, the following random variable is introduced:

$$X = \sum_{i=1}^{R} X_i$$

The expected value of X is the following:

$$E[X] = E\left[\sum_{i=1}^{R} X_i\right] \overset{*}{=} \sum_{i=1}^{R} E[X_i] = \sum_{i=1}^{R} \epsilon = R\epsilon$$

where $\overset{*}{=}$ holds by **Theorem 1**. We want to argue about the probability that $\mathcal{A}$ manages to take over a majority of slots of an epoch and find an upper bound of this probability that is a function of R and ϵ. This allows us to derive concrete values of the length of an epoch depending on the adversarial stake.

Theorem 5 (Safe epoch duration). *Let $R \in \mathbb{N}^+$ be the duration of some epoch $e_j, j = 1, 2, ...$ in slots. Let $\epsilon \in (0, 1/2)$ be the stake owned by an adversary $\mathcal{A}$. Let $\gamma = Pr[\mathcal{A}$ leads a majority of slots of epoch $e_j]$. Then,*

$$\gamma < \left[\frac{\exp\left(\frac{1-2\epsilon}{2\epsilon}\right)}{\left(1 + \frac{1-2\epsilon}{2\epsilon}\right)^{1+\frac{1-2\epsilon}{2\epsilon}}} \right]^{R\epsilon}$$

Proof.

$$\begin{aligned}
\gamma = Pr\left[X \geqslant \frac{R}{2}\right] = Pr\left[X \geqslant \frac{R\epsilon}{2\epsilon}\right] &= Pr\left[X \geqslant \frac{E[X]}{2\epsilon}\right] \\
&= Pr\left[X \geqslant \frac{2\epsilon + 1 - 2\epsilon}{2\epsilon} \cdot E[X]\right] \\
&= Pr\left[X \geqslant \left(1 + \frac{1-2\epsilon}{2\epsilon}\right) \cdot E[X]\right] \\
&\overset{*}{<} \left[\frac{\exp\left(\frac{1-2\epsilon}{2\epsilon}\right)}{\left(1 + \frac{1-2\epsilon}{2\epsilon}\right)^{1+\frac{1-2\epsilon}{2\epsilon}}} \right]^{R\epsilon}
\end{aligned}$$

where $\overset{*}{<}$ holds by **Theorem 2** because $\frac{1-2\epsilon}{2\epsilon} > 0$ for $\epsilon \in (0, 1/2)$.

□

4.3 Concrete duration of an epoch in slots

In this section, based on the upper bound derived in **Theorem 5**, we derive concrete values for epoch duration R that gives us an upper bound on the probability that $\mathcal{A}$ succeeds on leading a majority of slots of an epoch.

In Table 4.1, the (safe) duration of an epoch is shown depending on the stake that $\mathcal{A}$ owns and assuming that we want the probability of corrupted majority of slot leaders to be less than 10^{-40}.

Adversarial stake	Epoch duration in slots
10%	228
30%	1663
45%	34364
49%	908733
49.9%	$9.19808 \cdot 10^7$

Table 4.1: The safe duration of an epoch as a function of different adversarial stakes and assuming that we want a corrupted majority of an epoch with probability less than 10^{-40}.

As shown in Table 4.1, as soon as $\mathcal{A}$ possesses a considerable amount of stake in the system (e.g. 45% or 49%), according to this analysis, to be on the safe side, one should set the duration of an epoch to a very long period of time. Assuming that the length of a slot is 20 seconds, and assuming that $\mathcal{A}$ has 49% stake in the system then 908733 slots are roughly 30 weeks. Setting the duration of an epoch to 30 weeks, essentially makes the system predictable for 30 weeks and makes the system very vulnerable.

However, one should also consider that assuming that $\mathcal{A}$ owns 45% or 49% of the stake in the system is a very conservative assumption; that is because at the time of this writing the market capitalization of Cardano (which is the implementation of one of the Ouroboros protocols) is about 37 billion dollars according to [4] and $\mathcal{A}$ owning 49% means that $\mathcal{A}$ has managed to corrupt parties that collectively own stake worth more than 18 billion dollars.

A more realistic assumption would be to assume that $\mathcal{A}$ manages to corrupt parties that collectively own e.g., 20% of the stake in the system - according to [4] it means that $\mathcal{A}$ has stake worth about 7.4 billion dollars. If we want the upper bound of probability of a corrupted majority of slot leaders of an epoch to be still less than 10^{-40}, then the duration of an epoch should be at least 583 slots. Assuming that a slot is 20 seconds long, then an epoch lasts about 3 hours and 15 minutes.

In Figure 4.1, we show the upper bound (from **Theorem 5**) of the probability of a corrupted majority for an epoch as a function of the epoch duration and for different adversarial stakes. As expected, for an adversary that has e.g. 10% of the stake, the upper bound probability of a corrupted majority for an epoch goes down very quickly as the number of slots increases, however, for a rich $\mathcal{A}$ with 49% of the stake, even after setting the duration of an epoch to 500 slots, the upper bound probability is still close to 1.

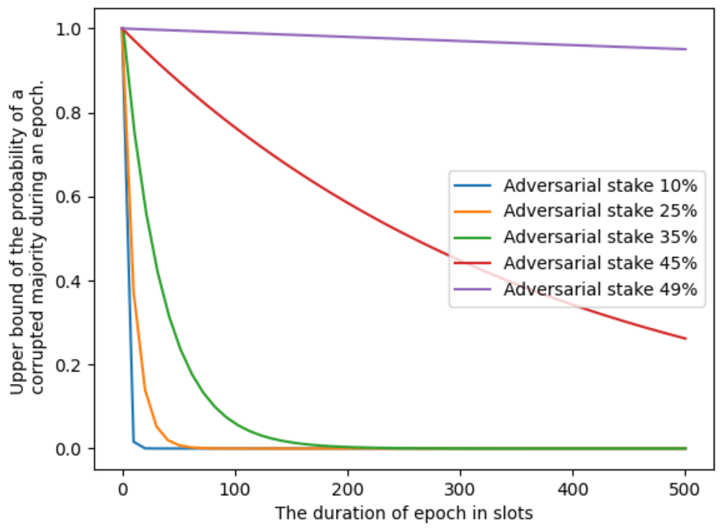

Figure 4.1: The upper bound of the probability of a corrupted majority for an epoch as a function of the duration of an epoch and $\mathcal{A}$ stake.

4.4 Discussion

One possible issue worth mentioning here is that $\mathcal{A}$ could buy stake and scatter the stake to as many users as possible. This might lead to a situation where for an epoch we have a correct majority of slot leaders but looking only on the number of parties of the committee of the epoch then the majority is corrupted. However, as already explained in [15] this issue might be easily avoided by simply adding an extra condition on the leader election process, i.e. just by considering those parties who have at least 1% of the stake as eligible candidates for being part of the committee.

5 Transaction confirmation time assuming a covert adversary

In this chapter, using a similar approach as in the previous chapter we estimate the transaction confirmation time. We do this by assuming that parties consider a transaction as confirmed only when the transaction stays in the chain adopted by correct parties for a certain number of slots. In the Ouroboros protocol, contrary to other *proof-of-work* based blockchains (where the number of blocks is critical for transaction confirmation time), because of the way that blocks are produced, i.e., it is publicly known for a whole epoch who has the right to produce blocks at what point in time, it is more convenient to consider transaction confirmation time in slots instead of blocks. The estimation of transaction confirmation time is done by assuming a particular adversary $\mathcal{A}$ called *covert* adversary and not a *general* adversary. A covert adversary is an adversary that attempts to cheat but does so in such a way to avoid being caught. To make clear the distinction between a *covert* adversary and a *general* adversary, we present examples of attacks from both kinds of adversaries in the next sections. We start with an attack from a *general* adversary in the next section and the subsequent section is about *covert* adversary.

5.1 General adversary

In this section, we show a particular attack from a general adversary. The attack is a fork that $\mathcal{A}$ manages to create during the course of a few slots. We show that the way that $\mathcal{A}$ performs this attack results in a situation where correct parties can certainly detect the malicious activity.

Assume that the **Protocol 1** runs for six slots, the resulting *characteristic string* is w = 001100, and the resulting fork is the one in Figure 5.1. It can be seen that adversarial parties run two slots, i.e., slot 3 and 4. In this execution $\mathcal{A}$ has the opportunity to broadcast two different chains C_1 and C_2 at the end of the sixth slot to different parties, i.e. divide the set of correct parties in two groups, one that adopts the chain C_1 and one that adopts the chain C_2 and does so. By behaving in such a way, adversary $\mathcal{A}$ can be detected by correct parties by only observing the chains that correct parties adopt, i.e., it can be seen that $\mathcal{A}$ has constructed on purpose two blocks in the slot 3 and 4 and consequently two different chains are adopted by different parties in the seventh slot. This is a clear and obvious deviation from the protocol specification.

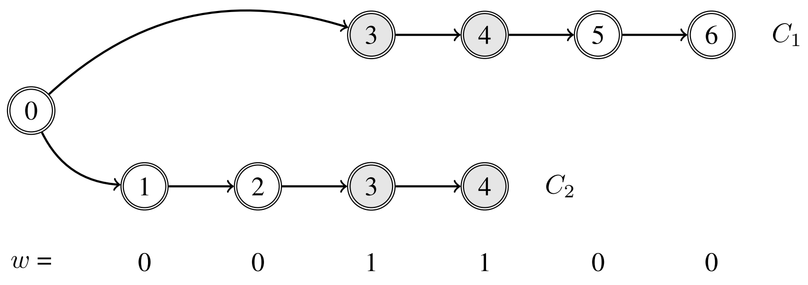

Figure 5.1: An example of fork $F \vdash w = 001100$

A covert adversary tries to avoid this situation, i.e., a covert $\mathcal{A}$ still tries to cheat the system but tries to do so by behaving in such a way that in the eyes of correct parties, it is not obvious that $\mathcal{A}$ has deviated the protocol.

In the following section, we present an example of how a covert adversary can manipulate the system.

5.2 Covert adversary

In this section we show an example of an attack from a *covert* adversary. The reason why this particular adversary is considered is because a covert adversary $\mathcal{A}$ is believed to be an adversary that models adversarial behavior in real life settings such as commercial, political and social settings. Covert adversaries typically deviate arbitrarily from the protocol in an attempt to cheat, but do it with the intention of not getting caught. In many real-world situations, parties are willing to cheat, but only if they are not caught doing so. This is believed to be the case in many fields (financial, political, business, diplomatic) where the correct behavior can not be assumed, yet parties can not afford the loss of reputation if they are caught cheating [6].

Now we show a particular scenario that could happen from malicious activity of a covert adversary $\mathcal{A}$. The covert adversary $\mathcal{A}$ in this scenario manages to revert a transaction that is confirmed by the network. The idea here is to issue a transaction, e.g., a payment from an adversarial account holder to a victim recipient, have the transaction confirmed by the system and then revert the transaction by publishing a chain built in secret by $\mathcal{A}$.

Example of reverting a confirmed transaction from a covert adversary. In this example, it is assumed that the confirmation time for a transaction is eight slots, i.e., after a transaction is included in some block in the chain then parties will consider a transaction as confirmed as long as the transaction is in the chain for eight slots.

The following strategy followed by a covert $\mathcal{A}$ leads to reverting an already confirmed transaction by the system:

1. $\mathcal{A}$ broadcasts a transaction **trx** for sending some money to Bob.
2. Transaction **trx** is eventually included in some block of the chain e.g. in the block B_{50} of the slot 50.

3. Bob and $\mathcal{A}$ wait for 8 slots after the slot 50 because the confirmation time of a transaction is 8 slots and (presumably) the transaction can not be reverted after 8 slots.
4. $\mathcal{A}$ is selected to lead 5 out of 8 slots, e.g., $\mathcal{A}$ leads slots e.g. 52, 54, 55, 57, 58.
5. $\mathcal{A}$ forks the chain to the block of the slot 49 and builds a chain in secret, i.e., $\mathcal{A}$ does not include blocks from correct parties, extends the chain from the block of slot 49 and builds its own chain with the blocks from the slots 52, 54, 55, 57, 58 and initially does not broadcast the chain.
6. At the beginning of the slot 59, Bob observes that the transaction **trx** is still in the chain and thus releases the goods and sends them to $\mathcal{A}$. As soon as $\mathcal{A}$ gets the goods, $\mathcal{A}$ releases its own secrete chain which does not include the transaction **trx**.
7. At the slot 59 every party switches to the secret chain of $\mathcal{A}$ because it is longer than the one where **trx** is included.
8. After the slot 59, since the chain where **trx** is included is not adopted by parties, $\mathcal{A}$ can later use the same amount of coins that he used in transaction **trx**. The issue that this situation causes is that $\mathcal{A}$ has already obtained the goods from Bob, yet Bob does not have the money, although the system confirmed the transaction for a short period of time.

This example of reverting a confirmed transaction from a covert adversary is captured in Figure 5.2.

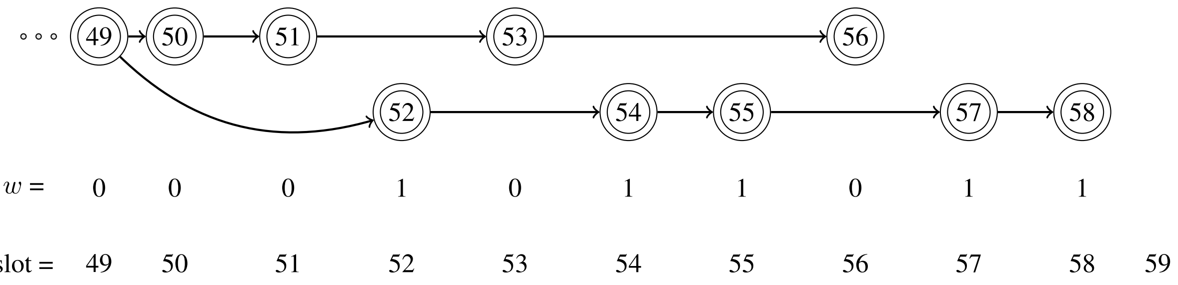

Figure 5.2: Reverting a confirmed transaction by the system.

Note how in this example $\mathcal{A}$ manages to intentionally revert a transaction that spent his money and there is no way to know who is the adversary because for someone who would not know which parties are adversarial, the behavior of both parties, i.e. correct and corrupted parties seem as expected. This kind of behavior describes a covert adversary $\mathcal{A}$; $\mathcal{A}$ manages to cheat but does it in such a way that $\mathcal{A}$ can not be held responsible (e.g. $\mathcal{A}$ can just blame the network conditions).

5.3 Probabilistic estimation of transaction confirmation time

The confirmation time for a transaction **trx** included in some block B_i of slot $i \in \mathbb{N}$ will be denoted by $k \in \mathbb{N}$, i.e., the transaction **trx** should be considered as confirmed by the system only if it is part of the chain until the slot $i + k$.

Just like in the previous section, to estimate transaction confirmation time, we need to keep track of whenever $\mathcal{A}$ leads a slot in the range $[i + 1, ..., i + k]$ and denote it with a 1, and whenever a correct party leads a slot in the same range we denote it with 0. For this purpose, the following random variable is introduced:

$$X_j = \begin{cases} 1 & \text{if the leader of the slot } j \text{ is a corrupted party.} \\ 0 & \text{if the leader of the slot } j \text{ is a correct party.} \end{cases}$$

for $j \in [i+1, ..., i+k]$.

If $X_j = 1$ for some $j \in [i+1, ..., i+k]$ it means that $\mathcal{A}$ leads this slot, whereas if $X_j = 0$ it means that a correct party leads the slot.

The probability of an adversarial party being selected to lead a slot is proportional to the stake that the party owns; the probability of any of the adversarial parties being selected to lead a slot is proportional to the stake that $\mathcal{A}$ owns. From the correct majority of the **Section 3.2** it is assumed that $\mathcal{A}$ has less than 50% of the stake at any point in time in any execution of the protocol, hence $Pr[X_j = 1] = \epsilon$ for $\epsilon \in (0, 1/2)$ and $Pr[X_j = 0] = 1 - \epsilon$. The expected value of X_j is $E[X_j] = 1 \cdot Pr[X_j = 1] + 0 \cdot Pr[X_j = 0] = \epsilon$.

We need to make sure that the range $[i+1, ..., i+k]$ that we specify for some k is such that the majority of blocks, i.e. strictly more than 50% of the blocks in that range are produced by correct parties with high probability. This can be done by simply arguing about the complementary event, i.e., argue about the probability that strictly more than 50% of blocks in this range of slots are produced by adversarial parties, and minimize this probability. For this, we introduce the following random variable:

$$X = \sum_{j=i+1}^{i+k} X_j$$

The expected value of X is the following:

$$E[X] = E\left[\sum_{j=i+1}^{i+k} X_j\right] \overset{*}{=} \sum_{i=i+1}^{i+k} E[X_j] = \sum_{j=i+1}^{i+k} \epsilon = k\epsilon$$

where $\overset{*}{=}$ holds by **Theorem 1**. We want to argue about the probability that $\mathcal{A}$ manages to take over a majority of slots on the range $[i+1, ..., i+k]$ and find an upper bound of this probability that is a function of k and ϵ. This allows us to derive concrete values of transaction confirmation time k depending on the adversarial stake ϵ. It follows that:

$$\begin{aligned} Pr\left[X \geqslant \frac{k}{2} + 1\right] &= Pr\left[X \geqslant \frac{k+2}{2}\right] = Pr\left[X \geqslant \frac{k+2}{2} \cdot \frac{k\epsilon}{k\epsilon}\right] \\ &= Pr\left[X \geqslant \frac{2k\epsilon - 2k\epsilon + k + 2}{2k\epsilon} \cdot E[X]\right] \\ &= Pr\left[X \geqslant \left(1 + \frac{k - 2k\epsilon + 2}{2k\epsilon}\right) \cdot E[X]\right] \\ &\overset{*}{<} \left[\frac{\exp\left(\frac{k - 2k\epsilon + 2}{2k\epsilon}\right)}{\left(1 + \frac{k - 2k\epsilon + 2}{2k\epsilon}\right)^{1 + \frac{k - 2k\epsilon + 2}{2k\epsilon}}}\right]^{k\epsilon} \end{aligned}$$

where $\overset{*}{<}$ holds from **Theorem 2** because $\dfrac{k - 2k\epsilon + 2}{2k\epsilon} > 0$ for $k \in \mathbb{N}$ and $\epsilon \in (0, 1/2)$.

5.4 Transaction confirmation time in slots

In Table 5.1, based on the upper bound in the previous section, the transaction confirmation time is shown assuming different adversarial stakes and assuming we want the probability of a successful double spending attack to be less than 10^{-40}.

Adversarial stake	Transaction confirmation time in slots
10%	224
30%	1653
45%	34325
49%	908534
49.9%	$9.9179 \cdot 10^7$

Table 5.1: Transaction confirmation time in slots as a function of different adversarial stakes and assuming that we want the probability of success for the double spending attack from a covert adversary to be less than 10^{-40}.

As it can be seen in Table 5.1, according to our analysis, assuming a rich $\mathcal{A}$ with 49% stake and if we want to say with high confidence e.g. with probability 10^{-40} that the transaction can not be reverted from a covert adversary then the transaction should be in the chain for 908534 slots; assuming a slot is 20 seconds, then one should wait for more than 30 weeks to say with such high confidence that transaction can not be reverted by a covert adversary.

Note that, as already argued in the **Section 4.3**, assuming that $\mathcal{A}$ manages to corrupt parties that collectively own 49% of the stake is a very conservative assumption because this means that these parties collectively own stake worth more than 18 billion dollars.

A more realistic assumption would be to assume that $\mathcal{A}$ has e.g. 20% of the stake, i.e., parties corrupted by $\mathcal{A}$ collectively own stake worth 7.4 billion dollars. If we want to say that the probability of reverting a transaction (from an attack from **Section 5.2**) is less than 10^{-40} then, the transaction confirmation time should be set to 577 slots; assuming each slot is 20 seconds, it follows that confirmation time is a bit more than 3 hours.

By relaxing the upper bound probability of reverting a transaction to 10^{-20} and assuming $\mathcal{A}$ has 20% of the stake, transaction confirmation should be set to 286 slots; assuming slots last 20 seconds, it takes about 1.5 hours to confirm transactions. Assuming the same adversarial stake but lowering the upper bound probability of reverting a transaction to 10^{-5}, transaction confirmation time should be set to 67 slot; assuming slots last 20 seconds, it follows that transaction confirmation time is a bit more than 22 minutes.

In Figure 5.3, we show the upper bound from the previous section on the probability that a covert adversary reverts a transaction after the confirmation time passes as a function of the confirmation time in slots and different adversarial stakes. As it can be seen from Figure 5.3 a covert adversary with stake e.g. 10% has a very small chance of reverting a transaction after the confirmation time, whereas if we assume a very rich covert adversary e.g. with stake 49% even if we wait 500 slots to confirm the transaction, the upper bound on the probability of reverting a transaction after the confirmation is still close to 1.

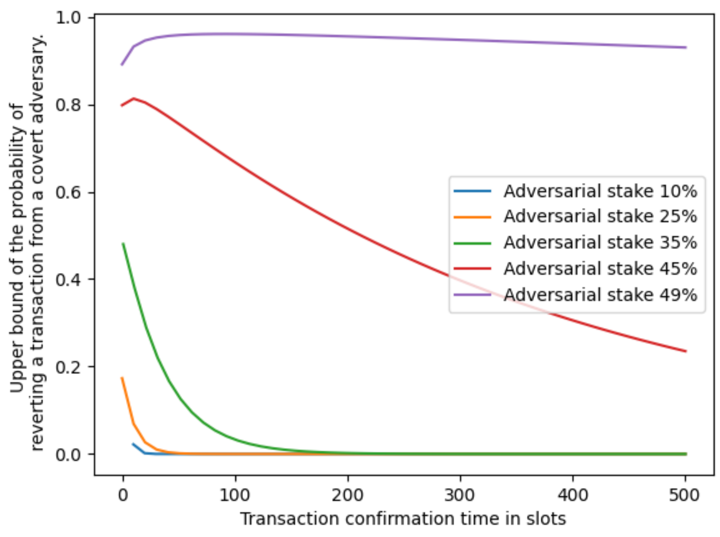

Figure 5.3: The upper bound of the probability of reverting a transaction from a covert adversary.

5.5 Discussion

In this section we estimated the transaction confirmation time for a particular adversary that does not want to look like it is attacking the system on purpose, i.e., a covert adversary. This estimation is done by also taking into consideration the adversarial stakes.

In [15] the Ouroboros protocol is proved secure under the assumption that $\mathcal{A}$ has less than 50% of the stake in the system at any point in time.

There are two things worth discussing and pointing out:

- The adversary considered in this section, while it is indeed a realistic adversary, it is not the most general/powerful one and when security guarantees are shown, ideally, a general adversary is considered. Given this, intuitively speaking, under a general adversary the probability of a successful attack considered in this section, i.e., reverting a confirmed transaction would be even higher.
- When arguing about security, ideally, we assume an adversary that is as powerful as it gets, in this case as rich as it can get e.g. with stake 49% on the system. As shown in Table 5.1, the transaction confirmation time for such an adversary is very very long. In practice, we would want to have blockchain systems that confirm transactions much faster.

6 Conclusion

6.1 Discussion

The initial idea for this work has been to analyze also other provably secure proof-of-stake based blockchains such as Algorand [12]. More precisely the idea has been to extract common parameters of these protocols and to compare them based on these parameters.

After studying and analyzing closely Algorand, we found out that the design of these protocols were different. E.g. Algorand designs a new Byzantine Agreement protocol for realizing a blockchain protocol, and e.g. the concept of transaction confirmation time does not make much sense in Algorand because the parties know if there was agreement or not on a block right after a certain number of rounds of voting whereas in Ouroboros (and other protocols that are based on the so called *Nakamoto style consensus*) the confirmation time for a transaction is something that should be manually set.

Initially, we thought that we would be able to extract these main parameters straight from the papers such as [15]. However, in this paper, the work is focused around showing bounds of the probability that the protocol satisfies certain security properties, and these bounds are asymptotic bounds which hide certain parameters; because of this, we spent quite some time analyzing the paper and trying to extract the parameters from security proofs and in the end concluded that it is very challenging to extract concrete values for some of the parameters of the protocol from the security proofs.

After concluding that it is very challenging to extract concrete values for relevant parameters from security proofs of the papers we turned towards implementations of Ouroboros protocol, namely Cardano. After spending some time searching through documentation and through code, it was not clear which version of the Ouroboros protocol was actually implemented and very challenging to even find if there is actually an implementation of the first Ouroboros protocol in Cardano.

Finally, we took a step back and started by simply splitting the protocol into small pieces and trying to see what are some of the relevant parameters. Because the Ouroboros protocol follows the *Nakamoto-style* consensus it is necessary to argue about when is a transaction considered confirmed; our analysis on this issue is shown in **Chapter 5**. The other relevant parameter of the protocol was the duration of an epoch. Analysis on the duration of an epoch is shown in **Chapter 4**.

6.2 Future work

In this work, we use the same approach as [15] of assuming that there are either correct parties that follow strictly the protocol or parties controlled by adversary. While this certainly simplifies the formal part of reasoning, this assumption is not very practical for protocols intended to be *permissionless*. For future work it would be interesting to extend this assumption by also considering parties that are correct but anyway fail to follow the protocol and analyze the protocol under this extra assumption.

In **Chapter 5** we analyzed the confirmation time of a transaction under the assumption of a covert adversary. Future work might include doing the same analysis for a general adversary.

Bibliography

[1] 'Hamming weight'(2020) *Wikipedia*. Available at `https://en.wikipedia.org/wiki/Hamming_weight` (Accessed: 2020-08-01).

[2] Bitcoin consumes more electricity than argentina. `https://www.bbc.com/news/technology-56012952`. Accessed at 2021-02-28.

[3] Cardano blockchain explorer. `https://explorer.cardano.org`. Accessed at 2020-07-15.

[4] Cryptocurrency prices by market cap. `https://www.coingecko.com`. Accessed at 2021-02-28.

[5] What is bitcoin's current hash rate? `https://www.coindesk.com/what-does-hashrate-mean`. Accessed at 2021-02-28.

[6] Aumann, Y. and Lindell, Y. (2007). Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography Conference*, pages 137–156. Springer.

[7] Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

[9] David, B. Iohk research, proof-of-stake protocol. `https://www.youtube.com/watch?v=hMgxZOsTlQc`. Accessed at 2020-07-15.

[10] Gaži, P. Iohk research, ouroboros. `https://www.youtube.com/watch?v=PoNaw-Mtxgo`. Accessed at 2020-07-15.

[11] Garay, J., Kiayias, A., and Leonardos, N. (2015). The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer.

[12] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. (2017). Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68.

[13] Katz, J. and Lindell, Y. (2020). *Introduction to modern cryptography*. CRC press.

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

[15] Kiayias, A., Russell, A., David, B., and Oliynykov, R. (2017). Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer.

[16] Kiffer, L., Rajaraman, R., and Shelat, A. (2018). A better method to analyze blockchain consistency. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 729–744.

[17] Mitzenmacher, M. and Upfal, E. (2017). *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.

[18] Nakamoto, S. (2008). Bitcoin.

[19] Rosulek, M. (2020). *The Joy of Cryptography*.

[20] Russell, A., Moore, C., Kiayias, A., and Quader, S. (2017). Forkable strings are rare. *IACR Cryptol. ePrint Arch.*, 2017:241.