Thesis Dissertation

CONSERVATIVE STUBBORN MINING: EXTENDING, MODELING AND VERIFYING SELFISH MINING STRATEGIES ON BITCOIN

Andreas Tsouloupas

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

January 2021

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Conservative Stubborn Mining: Extending, Modeling and Verifying Selfish Mining Strategies on Bitcoin

Andreas Tsouloupas

Supervisor

Dr. Anna Philippou

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

January 2021

Acknowledgments

I would like to thank my thesis supervisor Dr. Anna Philippou, Associate Professor at the Department of Computer Science of the University of Cyprus, for the valuable guidance, encouragement, and support during this project. Her contribution and feedback were critical to complete and write this dissertation.

Abstract

Bitcoin is the first and most popular digital currency for online payments, realized as a decentralized peer-to-peer electronic cash system. Bitcoin maintains an ordered ledger of all transactions, and the longest chain of the blockchain is selected as the valid ledger by the participants.

Selfish mining is a well-known attack where a selfish miner, under certain conditions, can gain a disproportionate share of reward by deviating from honest behavior. A selfish miner exploits the conflicting resolution rule of Bitcoin in order to increase its revenue. The standard Bitcoin protocol requires nodes to quickly distribute newly created blocks; however, malicious nodes can gain higher payoffs by withholding blocks they create and selectively defer their publication. Fortunately, selfish mining becomes profitable only when the malicious nodes possess a relatively large proportion of the entire network's computational power and/or are sufficiently well connected to the rest of the network.

In this thesis, we extend the mining strategy space to include "conservative stubborn" variations. These variations allow selfish miners to earn more than the pre-existing selfish mining strategies for specific parameter space regions. Consequently, we show that the basic selfish mining and stubborn mining attacks are not globally optimal.

Furthermore, we provide a formalization of selfish mining without propagation delays on Bitcoin as an UPPAAL model. UPPAAL is a tool for modeling, validating, and verifying real-time systems modeled as networks of timed automata, extended with data types. In this work, the UPPAAL STRATEGO extension is used, which enables statistical model checking and allows the exploration of strategy space defined by the automata. The model on UPPAAL STRATEGO is used to show the dominant strategies of selfish mining, among every strategy in the extended strategy space of selfish mining, for every region of the parameter space, estimate the dominant strategies' revenue, and verify our conservative stubborn variations' Risk Safety property. Finally, we estimate the probability of a selfish miner gaining more than its fair share in terms of revenue. Bitcoin is not incentive-compatible for a vast portion of the parameter space if we consider a proportion of the honest miners receiving first the selfish block instead of depending on the complexity of propagation delays, and no countermeasures are in place.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Contents

1.1 Overview

Since its inception, Bitcoin [23], a decentralized cryptocurrency, captured the interest of the entire world. Due to its popularity, a large number of research work have been conducted to study its various features, *e.g.,* consensus protocol and incentive schemes. The groundbreaking, distributed nature of Bitcoin eliminates the need for a trusted third party, such as banks, to process payments. Instead, a publicly known ledger handles transactions for which its participants consent. Miners are responsible for maintaining and extending the ordered ledger of transactions by pursuing the standard mining procedure. For this purpose, a consensus protocol was designed to solve ambiguities and form an agreement. In order to ensure that miners will follow the standard Bitcoin protocol, incentives in the form of rewards are in place.

The incentives in Bitcoin have a crucial role since they prevent miners from deviating from the prespecified consensus protocol and, primarily, to keep the Bitcoin "alive". Incentives are rewards in the form of bitcoins, which is the currency of the Bitcoin system. The block creator includes a batch of transactions, to be confirmed, into a block whose

creation requires generating the solution to computationally expensive proof-of-work puzzles. Once a new block is created, rewards are awarded to its creator. Therefore, honest participation is incentivized, and as a consequence, the security of Bitcoin is reinforced.

In a disruptive paper [18], its authors highlighted a defect in the incentive scheme in Bitcoin. A single miner or a pool of miners who possesses enough computational power and/or is extremely well connected to the rest of the network can increase its expected rewards by deviating from the block publication rule. This requires that most of the participants follow the predetermined Bitcoin protocol. The block publication rule of the Bitcoin protocol requires nodes to publish any block they mine immediately; however, in [18], they have shown that if miners selectively withhold blocks they may increase their revenue. For the rest of this work, we refer to this first selfish mining strategy, called Selfish-Mine by its inventors, as the basic selfish mining.

When every participant of the Bitcoin network strictly follows the consensus protocol, we expect that miners are rewarded according to the computational power they control. More specifically, we expect miners to reap the same fraction of rewards as the computational power they possess. Nevertheless, selfish mining allows an attacker to increase its revenue at the expense of other miners. This is accomplished by exploiting the conflicting resolution rule of the Bitcoin protocol. The longest chain is what individual nodes accept as the valid ledger of the blockchain. Its blocks are the only blocks considered valid, and hence they are the only blocks that receive rewards. This rule allows every node in the network to agree on what the blockchain looks like and therefore consent on the same transaction history. To exploit this vulnerability, an attacker deliberately creates a fork, which is a side chain, in order to force honest miners to periodically abandon the previously longest chain. According to the Bitcoin protocol, miners work on the longest chain; therefore, miners select to extend the longest fork in the presence of forks. However, the selfish miner keeps hidden the hopefully (for them) longest fork; hence, the unaware miners work on a shorter fork. Consequently, honest miners waste time and money mining on top of a shorter chain that will never become the longest.

Unfortunately, selfish mining attacks can potentially cause catastrophic consequences to the Bitcoin network. The rate of growth of a successful selfish mining attacker is steady. This is a result of the increased revenue compared to honest participants of the network. Consequently, other miners may join the pool of the selfish miner to enjoy the increased

profit, or the miner can multiply its computational power by buying more equipment. Moreover, the revenue from selfish mining is proportional to the computational power in possession of the attacker; therefore, the attack will become more effective over time. Eventually, the attacker can collect every reward of the network by performing the well-known 51% attack, which requires more than half of the entire network's computational power.

The above-mentioned reasons were the impetus for further study of selfish mining in order to encourage interest in future research and more particular, in application and development of formal techniques for the analysis of blockchains. Undoubtedly, the detection, alleviation, and security against such attacks must be immediate and effective.

1.2 Purpose

Our goal in this thesis is to considerably understand the selfish mining strategy space and, therefore, try to extend it by introducing more effective strategies. In [24], an earlier paper regarding selfish mining expands the mining strategy space to include a novel strategy family, called "stubborn". These new stubborn mining strategies offer the miner more revenue than basic selfish mining, which is the first formally described selfish mining strategy. Thus, basic selfish mining is not optimal.

Furthermore, this work focuses on exploring formal techniques towards the analysis of selfish mining on Bitcoin. We aim to investigate the UPPAAL [4] model checker and the capabilities offered by its various extensions, such as UPPAAL STRATEGO [15]. UPPAAL STRATEGO allows an effective strategy space search, which fit our purpose perfectly.

1.3 Contributions

This thesis makes the following contributions:

- We present the conservative stubborn mining variants, which further expand the strategy space of selfish mining. More specifically, in this work, we discuss two variants, the Safe-Lead, and Safe-Equal-Fork, which are based on stubborn mining. What distinguishes them is the alternation between basic selfish mining and stub-

born mining to achieve better revenue results. We also provide pseudocode, which allows any combination of selfish mining variations to form hybrid strategies.

- We implement and evaluate the entire space of selfish mining strategies on UPPAAL model checker. UPPAAL STRATEGO is used to identify the best strategy for every parametrization of the parameter space. Furthermore, the Risk Safety properties of basic selfish mining and the novel conservative stubborn mining are verified. Risk Safety property allows bounded risk at any time of the selfish mining.

- Stubborn mining is not optimal for a large parameter space. Our conservative stubborn mining outperforms pre-existing strategies in some regions of the parameter space.

1.4 Outline

The rest of the thesis is organized in chapters as follows. In Chapter 2, we provide the background information required for understanding the subsequent chapters. More specifically, there is an introduction to the well-known Bitcoin cryptosystem and the model checking tool UPPAAL, alongside its UPPAAL STRATEGO extension. Chapter 3 provides an overview of the existing selfish mining strategies and our newly introduced variations modeled as state machines. Moreover, an algorithm is given which is capable of constructing hybrid selfish mining strategies by combining variations. In Chapter 4, we explain in detail the automata of our implementation on UPPAAL STRATEGO and the testing code, which secure the correctness of our model. In Chapter 5, we evaluate our new conservative stubborn strategies and verify the Risk Safety property satisfaction. In Chapter 6, we refer to related work concerning selfish mining and previous work modeling Bitcoin on UPPAAL. Finally, in Chapter 7, we summarize our work and propose possible future work extending our model with parameters or countermeasures not studied in the thesis at hand.

Chapter 2

Background

Contents

2.1 Overview

This chapter provides background information for the rest of the thesis. We briefly discuss how Bitcoin realizes a distributed ledger of transactions by providing essential information about transactions, blocks, and blockchain. Then, we shift the attention to the unwanted but inevitable blockchain forks that are causing problems to the Bitcoin network. Without blockchain forks, there will be no need to study selfish mining, which is the main topic of this work. Finally, we provide an overview of UPPAAL [4] and more specifically, of the UPPAAL STRATEGO [15] extension, the model checking tool used to conduct this work.

2.2 Bitcoin

Bitcoin is the first decentralized global currency system introduced by Satoshi Nakamoto [23] in 2008. Unlike traditional currencies, such as Euro and USD, which are maintained by centralized authorities (banks), Bitcoin is a currency maintained by volunteer participants worldwide, called miners. Each participant (miners and non-miners) of Bitcoin's network keeps a replica of the transaction's ledger that sometimes may slightly differ from others. This ledger tracks the balance of all accounts in the system. To verify new transactions into the distributed ledger, Bitcoin miners stick to a consensus protocol, an agreement of the rules they should follow during the mining procedure. In this work, we abstract the complexity of transactions' and blocks' structures since they are out of this study's scope—more details about how Bitcoin works can be found in [1, 8, 22].

2.2.1 Transactions

In general, a transaction transfers bitcoins (the currency) from one or more source accounts to one or more destination accounts. This becomes possible due to the capability to enter more than one input and output in each transaction. Each input is linked to an output of a previous ledger's transaction, which has not yet been spent. Such transactions are called unspent transactions. It goes without saying that in order for an unspent transaction to be linked to an input of a transaction, its ownership[1] must be proved. On the other hand, each output can refer to a different account and, hence, be claimed later as input to another transaction. Moreover, the amount of bitcoin transferred to other accounts should never exceed the amount of bitcoin accumulated by inputs. Any bitcoins leftover from the transaction's inputs, that is, they are not transferred to an account, remain as transaction fees to the miner. The miner who will include the transaction in a block of the ledger will be the one who will obtain the transaction fees of the transactions included in the block. Therefore, this adds an incentive to miners who waste power during the mining procedure. In addition to the transaction fees, the miner of a newly mined block is incentivized by receiving an additional predefined reward. This reward is in the form of a transaction, namely coinbase transaction, which allows the miner to include a transaction

[1] Bitcoin uses ECDSA to secure the authenticity of transactions and to create accounts from public-private key pairs

into the block without inputs (this is the only exception to the rule) but with outputs that can spend the predefined reward.

2.2.2 Blocks

The purpose of blocks is to confirm the validity of transactions. A block contains a set of transactions that will be validated. The creator of a block, the so-called miner, can decide whether a transaction will be included in the block and transactions' order in that block.

When creating a block, the mining procedure is followed, which requires solving a computationally difficult puzzle. This puzzle is called proof-of-work. Bitcoin uses the Hashcash [9] proof-of-work system, which was originally used to limit email spam and denial-of-service attacks. Each block can be serialized, and its digest can be found after applying a cryptographic hash function. To solve the puzzle, a miner needs to find a valid nonce that is part of the block; therefore, its digest will change after applying a new nonce. As cryptographic hash functions are one-way, it is difficult to find a nonce that produces a correct solution, but it is straightforward for others to verify it. A valid nonce must evaluate the block's digest to a value less than or equal to the target determined by the consensus protocol (can also be seen as leading zeros). Therefore, since finding an acceptable digest is probabilistic, the higher the computing power of the network a miner possesses, the greater the chances that this miner will be the next block's owner by finding a valid nonce. Moreover, the target changes every 2016 blocks so that a new block will require approximately 10 minutes to be mined. Finally, a miner will publish its block immediately after creating it so that it can be disseminated to every node in the network as soon as possible to avoid forks.

2.2.3 Blockchain

So far, we have seen how transactions are validated and ordered into a block. However, this is not enough to create a distributed ledger of transactions, and therefore there is a need to define the order of transactions between different blocks. The blockchain serves this purpose, which is essentially a directed tree of blocks.

In the blockchain, each block is linked to another block, which is called its parent. A block defines its parent by including its digest within itself. The pointer to the parent of

a block implies that a block can not be altered by modifying its transactions or metadata later. If that is the case, its digest will completely change; therefore, its children will become orphans and might be invalided according to the target value. The genesis block is the root of the blockchain, and it is hardcoded. The height of a block is calculated based on the distance from the genesis block. For example, the genesis block has 0 height.

Assuredly, there can be many branches into the blockchain; however, only the longest branch is considered valid, that is, the branch with the greater height. To be more precise, the longest chain is the chain with the most work, which is almost always the same as just comparing the height, unless the fork spans a re-target. Therefore, the transactions in the blocks of the longest chain are the only transactions that are considered valid. The block at the end of the longest branch is called the head of the blockchain. Miners extend their local blockchain on top of that block.

2.2.4 Blockchain Forks

We already observed that blockchains can have multiple branches simultaneously; however, according to the consensus protocol, miners mine on top of the longest branch. These branches are called blockchain forks, and they are the reasons for many possible attacks to the Bitcoin system.

A blockchain fork can be caused either from deliberately mining on top of a side chain which is not the longest or from information propagation delays of the network, which can cause concurrent block mining from distinct miners. In our work, information propagation delays studied in [17] are considered negligible compared to the time needed to mine a new block (forks caused by concurrent mining are omitted). Moreover, Blocks that are not in the longest fork are called stale blocks.

In the presence of blockchain forks, miners can decide to mine on top of different forks. As a consequence, the longest fork can change at any time if a side fork surpasses the current longest fork in terms of height. Therefore, the Bitcoin network never commits a transaction or a block definitively because the longest fork can change. Hence, the blocks of the longest fork will change, and as a result, the valid transactions and their order will also change.

As discussed later, in Chapter 3, Blockchain forks are exploited in selfish mining to increase miners' revenue without following the consensus protocol. Therefore, they can

deceive honest miners by creating deliberate forks.

2.3 UPPAAL Model Checker

UPPAAL is a tool for modeling, validating, and verifying real-time systems modeled as networks of timed automata [7], which are composed into a system, extended with data types.

Each UPPAAL automaton consists of locations, edges, and local declarations of functions and variables. A location represents a state of the automaton, and it can be either initial (only one per automaton), urgent, which freezes the time or committed, which also freezes the time; however, the next transition must be taken from a committed location. The last restriction does not apply to urgent locations; thus, this distinguishes them. Furthermore, a location has an invariant, which determines how long an automaton can stay there or after how long it can leave from there. In addition, edges allow the transition from one location to another and, therefore, the transition of the system from one state to another. An edge consists of updates, guards, synchronizations, and selections (not used in this work). Firstly, updates are expressions that are executed during the transition, and they can be used to update local variables of the automaton or global variables of the system. Moreover, guards are expressions that state the conditions under which the transition is executable (enabled). Finally, synchronizations contain the channel on which the transition must be synchronized with another automaton or every other automaton in the case of a broadcast channel. The transitions of automata are executed simultaneously when a synchronization occurs.

UPPAAL uses TCTL [5, 6] temporal logic language to verify its properties. An extended tutorial is provided in [10]. In general, there are five properties, namely possibly (satisfied), invariantly (always satisfied), potentially always, eventually (there is always a chance to be satisfied), and leads to (response). In this thesis, we use solely the invariantly property which is denoted by:

```
A[] Expression
```
(2.1)

It basically means that the expression is satisfied permanently without a moment of un-

satisfaction. Overall, there are many extensions of UPPAAL, but for the purposes of this work, we focus on UPPAAL STRATEGO.

2.3.1 UPPAAL STRATEGO

This extension of UPPAAL integrates UPPAAL SMC [11, 16], which enables statistical model checking. UPPAAL SMC essentially replaces the non-deterministic choices between multiple enabled transitions by probabilistic choices and the non-deterministic choices of time delays by probability distributions. The probability distributions are either uniform distributions in cases with time-bounded delays or exponential distributions in cases of unbounded delays. If an exponential distribution specifies the time delay, then instead of invariant, the distribution rate λ is specified at a location. These changes permit probability estimation, hypothesis testing, and probability comparison. In this work, we used the capability of probability estimation queries, which is denoted by:

```
Pr[bound;#sim](<> or [] Expression)
```
(2.2)

This query estimates the satisfaction probability of a path expression, where the diamond means in the future while the square means always. The model exploration is bounded by an expression that can set a limit based on a clock value, model time (used in this thesis) or the number of steps. Furthermore, UPPAAL SMC allows value estimation, which estimates the expected mean value of the minimum or maximum evaluation of an expression by running a given number of simulations, which is denoted by:

```
E[bound;#sim](min: or max: Expression)
```
(2.3)

The number of simulations is specified after the bound, which was explained earlier. Moreover, note that for the probability estimation query, it is not required to set the number of simulations because UPPAAL handles it.

Furthermore, UPPAAL STRATEGO integrates UPPAAL TIGA [12] and statistical learning methods proposed in [14]. As a result, UPPAAL STRATEGO allows the exploration of strategy space defined by the automata, and queries offered by UPPAAL SMC can be examined under specific strategies that maximize or minimize a property and/or guarantee

an objective. The query of maximizing an objective is denoted by:

```
strategy S = maxE (Expression) [bound]
```
(2.4)

where S is an identifier which can be applied to the queries offered by UPPAAL SMC by adding at the end of the queries the words "under S" (bound implies the time limit as explained earlier). For this purpose, UPPAAL STRATEGO allows the definition of controlled and uncontrolled edges. The controlled edges are depicted as solid lines and they are controlled by the controller; thus, the controller is able to remember a path or transition selection to maximize or minimize and/or guaranty an objective.

2.3.2 Test Cases

The test cases tool can be used to examine the correctness of UPPAAL models. It generates traces from the implemented model. The produced traces are then translated into test cases, which are output files (one per test case), based on test code entered into the model on edges and locations. Test code at locations can be entered either on entering or exiting the location. Therefore, the correctness of the implementation can be checked by entering the appropriate test code in edges and locations, which will produce a file for each test case that can be used later for testing the implementation. It is up to the developer how test cases will be used. In Figure 2.1, we present the test case procedure for our implementation which is described later in Section 4.5. An extended tutorial on the test cases tool is located in the demos folder of the current (4.1) development release of the academic version of UPPAAL.

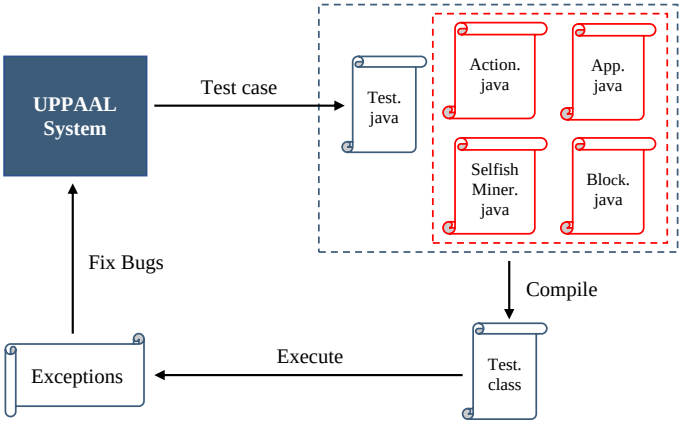


Figure 2.1: Test cases procedure.

Chapter 3

Selfish Mining Strategies

Contents

3.1 Overview

This chapter provides an overview of the existing selfish mining strategies alongside our newly introduced variations. We begin with the notations used throughout this thesis and how we are modeling the strategies as state machine diagrams. After that, we depict the basic selfish mining strategy proposed in [18]. In the subsequent section, we describe

Lead, Equal-Fork, and Trail stubborn selfish mining variations proposed later in [24]. The stubborn selfish mining strategies contain the notion of stubbornness, which was our inspiration to introduce two new selfish mining strategies, the Safe-Lead and Safe-Equal-Fork. We will discuss them in more detail later in this chapter, but for now, note that they are strongly related and similar to Lead and Equal-Fork respectively. The chapter ends by providing an efficient way to construct hybrid selfish mining strategies. A hybrid selfish mining strategy combines more than one of the variations reported above in order to increase its stubbornness and profit. We also provide an outline of the pseudocode summarizing the actions performed by any feasible hybrid strategy.

Honest miners strictly follow the consensus protocol described in Section 2.2. On the other hand, selfish miners deviate from the consensus protocol regarding the publication of blocks. They deliberately withhold blocks that they have mined in order to reveal them later in such a way that will be more profitable for them. A selfish miner has the dilemma of when and how many blocks to reveal. These decisions define the strategy which the miner follows. The functionality of each one of the strategies used in this thesis will be described in this chapter.

3.2 Modeling of the Mining Procedure

This section provides a list of the notations used for the rest of this thesis. Furthermore, for each variation of the selfish mining attack, a state machine will be provided.

The mining process consists of a set of entities that interact with each other. There are two types of entities, those that are selfish and those that are honest. Their interaction depends on the strategy followed by the attacker, also known as selfish miner. In Table 3.1, we give a list of notations with the notations used in this work, accompanied by a short description, where affected by the connectivity means that they receive first the selfish block when a fork occurs.

3.2.1 Defining the Mining Environment

The environment of the mining procedure consists of a set of entities that interact with each other. Each one of the entities holds a proportion of the total computing power of the entire blockchain network. In total, there are two different entities, of which one is divided

Entities of Environment (Section 3.2)	
A	Alice – Selfish miners coalition
B	Bob – The set of all honest miners
GB	Good Bob – The subset of Bob which is not affected by its connectivity with Alice
BB	Bad Bob – The subset of Bob which is affected by its connectivity with Alice
Network's Mining Power Distribution Parameters (Section 3.2)	
α	The fraction of the total computing power of the network held by Alice
β	The fraction of the total computing power of the network held by Bob
γ	The fraction of computing power held by Bob which belongs to Bad Bob
Stubborn Mining Variations (Section 3.4)	
S	Basic Selfish mining (Section 3.3) – The first formally described selfish mining strategy
L	Lead stubborn – Stubborn variation which insists on lead
F	Equal-Fork stubborn – Stubborn variation which quickly gains lead
T_i	Trail stubborn - Stubborn variation where Bob pulls ahead of Alice in terms of lead, where i is the trail stubbornness degree
Conservative Stubborn Mining Variations (Section 3.5)	
L_S	Safe-Lead conservative stubborn – Conservative stubborn variation which insists conservatively on lead
F_S	Safe-Equal-Fork conservative stubborn – Conservative stubborn variation which gains lead quickly in a conservative fashion

Table 3.1: Notation table.

into two sub-entities. More specifically, we have the following entities and sub-entities:

- Alice (A) – This entity represents the coalition of selfish miners. Selfish miners are the miners in the network who do not follow the predetermined consensus protocol rules and deviate in order to have a personal profit. More specifically, they follow their own rules of publishing a newly created block. Their mining behavior is described as one of the selfish mining strategies in Sections 3.3, 3.4, 3.5, and 3.6. The fraction of computing power they possess is denoted by the letter α, where $0 \leq \alpha \leq 1$.

- Bob (B) – This entity represents the entire set of honest miners. Honest miners are

the miners in the network who strictly follow the rules of the consensus protocol. As a consequence, they have no selfish intentions. The fraction of computing power they possess is denoted by the letter β, where $0 \leq \beta \leq 1$ and $\alpha + \beta = 1$.

- Good Bob (*GB*) - Bob's first sub-entity consisting of miners who inherit Bob's honest behavior. The only difference is that Alice is not well connected with him. As a result, when two forks (Alice's and Bob's) of equal length co-occurred, he will mine on top of Bob's block. The fraction of computing power corresponding to this sub-entity is $1 - \gamma$ of Bob's computing power, that is, $(1 - \gamma)\beta$ of the network's total computing power, where $0 \leq \gamma \leq 1$.

- Bad Bob (*BB*) - Bob's second sub-entity consisting of miners who again inherit Bob's honest behavior. The only difference is that Alice is well connected with him. As a result, when two forks (Alice's and Bob's) of equal length co-occurred, he will unintentionally mine on top of Alice's fork. The fraction of computing power corresponding to this sub-entity is γ of Bob's computing power, that is, $\gamma\beta$ of the network's total computing power, where $0 \leq \gamma \leq 1$.

The entities of the environment also have a local blockchain. The blockchain is maintained as described in 2.2. Any reference to the public and private blockchains are made to highlight the fork on top of which the entities are mining. Alice strictly works on her private blockchain, but at the same time, she is inspecting the blocks mined on the public blockchain. On the other hand, Bob mainly works on the public blockchain, with Bad Bob sometimes accidentally working on the private blockchain. Furthermore, Bob may not have a complete picture of the entire private blockchain since Alice may not have revealed all of her blocks yet.

3.2.2 Strategies as State Machines

The strategies in this work have been modeled as state machines. More specifically, for each strategy, we provide Alice's state machine, which depicts a complete picture of Alice's actions upon an event. A state machine consists of:

- A set of states, the representation of which will be explained below.

- A set of transitions from one state to another. Moreover, each transition contains the following information as a label. The probability, given that Alice is currently located in the state from which the transition exits, for the transition to be taken. As a result Alice performs an action during the transition to the new state. This is denoted with the label $\frac{propability}{action}$ placed on the transition. A transition can also be denoted by the tuple $(\sigma_1, pr, act, \sigma_2)$, where σ_1 is the current state, σ_2 is the state to which the transition leads, pr is the probability for the transition to be taken and act is the action being performed during the transition.

As mentioned above, Alice has to decide which action to perform each time a probabilistic event occurs. These decisions taken by Alice define the strategy which she follows. Overall, there are four different actions to choose from. "No action" (n), Alice remains idle (no blocks of the private chain are published). "All" (a), Alice will publish all of her unpublished blocks. "Match" (m), Alice will publish as many blocks as needed to match the length of the public blockchain. "Restart" (r), Alice abandons her effort and continues the mining process starting from the head of the public blockchain, that is, she loses all of her mined blocks that existed solely in her private blockchain. In Table 3.2, there is a summary of the possible actions that can be performed by Alice every time an event occurs.

Action	*Description*
n	No action – Alice will not publish blocks (idle)
a	All – Alice will publish all unpublished blocks of her private chain
m	Match – Alice will publish as many blocks as required to match public chain
r	Restart – Alice will abandon her effort

Table 3.2: Alice's possible actions.

Regarding the names of the states, they have been used in such a way as to give us enough information about the lead that Alice has at her disposal as well as the state of the environment (but not for the risk she runs). The *lead* can take any value from the set of integers ($lead = \cdots, -1, 0, 1, 2, 3, \cdots$). In contrast, the *risk* can take any value from the set of positive integers ($risk = 0, 1, 2, \cdots$). When we refer to the lead of Alice compared to Bob, we mean the difference in the length of private and public blockchains,

$lead = len(pubchain) - len(prchain)$. On the other hand, the risk is the number of blocks that are included in the private chain, whether published or not, which are not included in the public chain. (Note that when we refer to public and private chains, we mean the longest chains, that is, the chains with most blocks). For instance, if Alice currently has lead and risk equal to five, then if Good Bob mines the next block, Alice's lead reduces to four. However, the risk remains the same because the block she revealed when Goob Bob mined the block is not included in the public chain.

As we have seen, there is a difference between risk and lead; the next step is to explain how and what exactly a state represents. Firstly, a state is given an integer number, either negative or positive. This number indicates the lead of Alice compared to Bob at that state. Then the state is classified into one of the state types. In total, there are three types of states, without an apostrophe, with a single apostrophe ($'$), and with a double apostrophe ($''$). The apostrophes follow the number given to that state, namely the lead of Alice. All states, which are of the first type without apostrophe, denote a state in which (from Alice's point of view) there are no forks, that is, Bob has not mined a block yet since the moment Alice started her new cycle of selfish behavior. A new cycle of selfish behavior is started every time Alice performs the actions "all" or "restart". With a single apostrophe, we denote the states in which there is a fork (from Alice's point of view) and Bad Bob mines on top of Alice's fork, while with a double apostrophe, Bad Bob mines on top of Bob's fork. For instance, with $1'$ we will denote the state in which Alice has a lead of one block and Bad Bob mines on top of Alice's fork, whereas with $0''$ we denote the state where Alice has no lead and Bad Bob mines on top of Bob's fork. It is important to note that the initial state for every state machine is state 0. Finally, we want to mention a special case in our state machines, where we give information about the risk that Alice runs and is denoted by $0'_s$. In this state, we know that Alice has no lead, there is a fork (Bad Bob mines on top of Alice's fork), and Alice runs a risk equal to one.

3.3 Basic Selfish Mining

The first strategy that we will study was initially proposed in [18]. As we mentioned earlier, the inspiration of this work was to take advantage of block withholding by Alice and her excellent network connectivity with Bad Bob. As a result, the basic selfish mining was

born. The strategies in Sections 3.4 and 3.5 are based on this strategy. (The differences in the new strategies concern Alice's chosen actions upon the probabilistic events)

It is essential to mention that there is a difference when Bad Bob mines a block from when Good Bob does. The difference is that Bad Bob will mine on top of Alice's fork when there is a fork, and the public fork is of equal length with Alice's fork. Into our state machines, this is represented by the states with a single apostrophe. As a result, all previous Alice's blocks in the current private chain will be confirmed and successfully inserted into the public chain making Good Bob continue mining on top of Bad Bob's block when Bad Bob first mines a block. The difference is not particularly evident in the basic selfish mining strategy, but we will see later that in the new strategies, different actions may be taken when Bad Bob mines a block compared to when Good Bob does.

In brief, the fundamental function of the basic selfish mining strategy is as follows. Alice will reveal a block, if any, to match the public chain every time Bob mines a block. When Alice matches the length of the public chain, she divides Bob's computing power because Good Bob will mine on top of Bob's fork and Bad Bob on top of Alice's fork. Moreover, Alice will never risk losing her mined blocks. In such a situation, Alice will publish all of her unpublished blocks to ensure her revenue. As a result, the selfish mining has the effect to waste the mining power of Bob who is mining blocks on a fork that is eventually abandoned because it is no longer the longest fork.

In order to describe a strategy in more detail, we need to clarify Alice's actions being performed for every reachable state of the state machine for this specific strategy. The following rules apply for the basic selfish mining:

- When Alice is in states ≥ 0 or $\geq 2'$ and Alice mines a block, then Alice remains idle.
- When Alice is in state 2 or $2'$ and Bob mines a block then Alice reveals all of her blocks.
- When Alice is in state $0'$ and Alice mines a block, then Alice reveals all of her blocks.
- When Alice is in states $1, \geq 3$ or $\geq 3'$ and Bob mines a block, then Alice reveals a block and matches the public chain.

- Alice is not negative lead tolerant (states $\leq -1''$ not reachable), thus she always restarts when the lead is lost.

This strategy is depicted as a state machine in Figure 3.1, where we can see the states and their corresponding transitions that describe Alice's behavior in each situation. The state machine presented in our work is not the same as presented in [18], as we wanted to represent additional information with slightly different modeling, *i.e.,* the presence of public and private forks. Thus, we translated it into our modeling scheme. Nevertheless, it is equivalent in terms of the behavior of Alice.

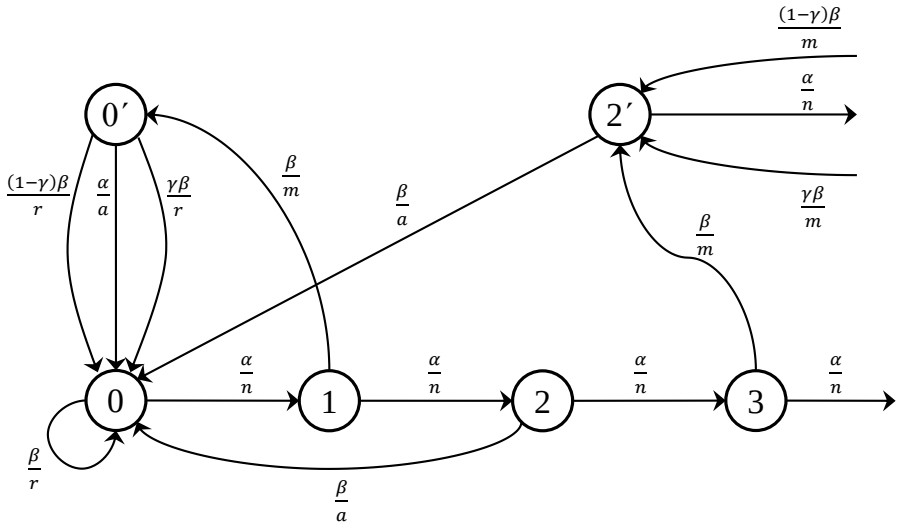


Figure 3.1: State machine of basic selfish mining strategy.

Now that we know the exact functionality of the strategy, we can conclude that this strategy has a property. We define the Risk Safety property with maximum risk of i, denoted with RS_i, where $i > 0$. If this property is satisfied by a strategy then the strategy always has risk below i. A strategy that satisfies RS_j, where $j > 0$, also satisfies RS_i, for $i \geq j$.

The basic selfish mining satisfies the RS_1 property. This implies that Alice never risks more than one block since the only case in which she risks a block is when she is in state $0'$. This state is only reachable if Alice first mines a block during a new cycle of selfish mining behavior, and Bob mines the next block. Therefore Alice cannot run a risk of more than one block for the strategy currently discussed. As we will see later, this strategy is the most conservative since the risks taken are minimal.

3.4 Stubborn Selfish Mining

At the end of the previous section, we briefly discussed the Risk safety property of the selfish mining strategy. The risk taken by Alice in the stubborn strategies [24] can become arbitrarily large; hence, we can say that they satisfy RS_∞, but for the rest of this thesis we say that strategies which satisfy RS_∞ do not satisfy the Risk Safety property. More specifically, in the paper mentioned above, the authors perceived that new strategies could be created by increasing the stubbornness of basic selfish mining by introducing small variations in Alice's decision rules. A strategy is considered more stubborn when it performs less frequently the actions "all" and "restart"; that is, Alice prefers to insist on the effort she is currently in, rather than starting a new cycle of selfish mining behavior. In total, they designed three variations, which they named Lead, Equal-Fork, and Trail stubborn. The state machine that illustrates how these strategies work is shown in Figure 3.2, which includes all the variations color separated. The state machine we present is not the same as presented in [24] since our modeling is slightly different, and our basic strategy is not the same; therefore, we translated it to meet our modifications. Their basic strategy was a version of the basic selfish mining that omitted Alice's matching rule when Bob mines a block. The matching rule was only part of the Lead stubborn mining, which will be described in short. We made this differentiation (in Equal-Fork and Trail stubborn variations) because there was no reason not to include the matching rule, as it does not insert extra insistence from Alice on its inclusion. Instead, with the matching rule Bob's computing power is divided into Good Bob's and Bad Bob's. The rest of this thesis works with these slightly different stubborn variations.

3.4.1 Lead

Lead stubborn is the strategy in which Alice is more stubborn in terms of maintaining her lead. Besides, this is evident from the name given to the strategy.

In short, the main idea of this strategy is that Alice will never perform the action "all" when she has any positive lead. Thus, she increases her stubbornness regarding her lead, trying to exploit it maximally. However, this has, as a consequence, a more dangerous amount of risk. No one can guarantee that this strategy will have bounded risk as applies in basic selfish mining, which satisfies the RS_1 property.

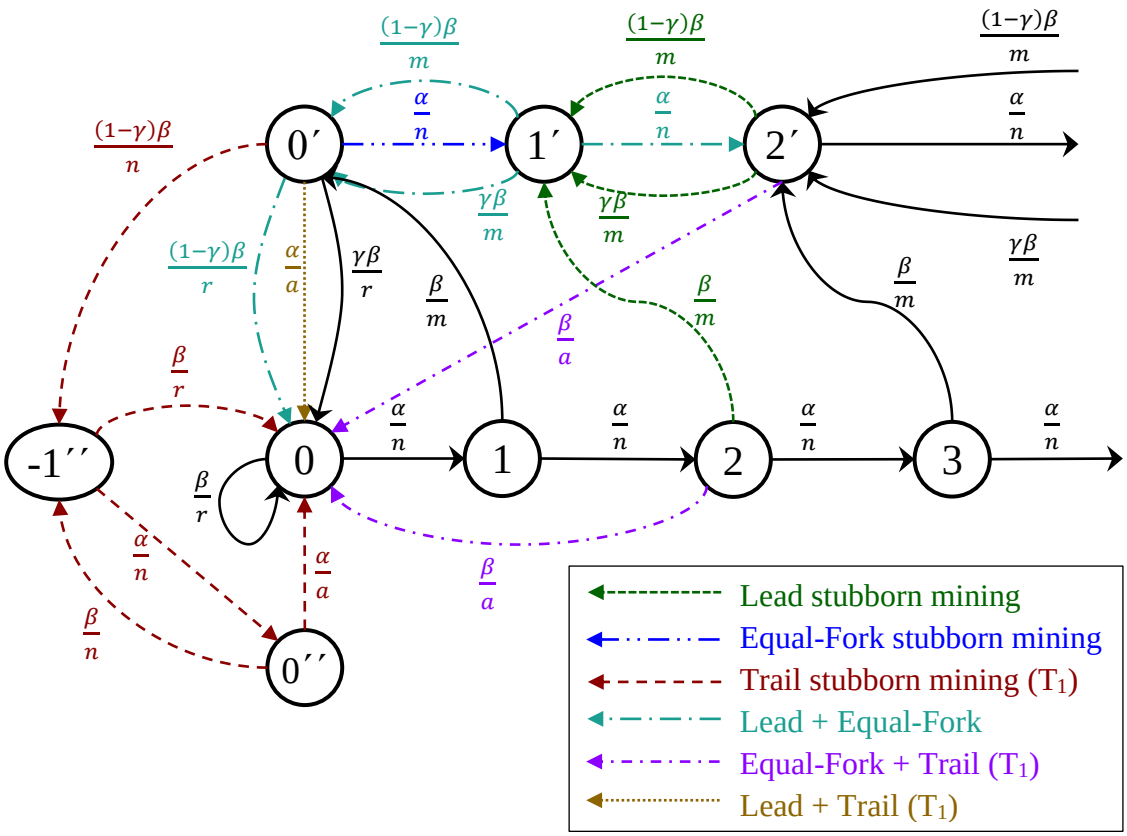


Figure 3.2: State machine of Lead, Equal-Fork and Trail (T_1) mining variations. (1) Green + Cyan + Gold + **Black** indicates the state machine of Lead stubborn strategy. (2) Blue + Cyan + Purple + **Black** indicates the state machine of Equal-Fork stubborn strategy. (3) Bordeaux + Purple + Gold + **Black** indicates the state machine of Trail (T_1) stubborn strategy.

As previously done, to describe the strategy, we need to clarify Alice's decisions in each of the states in which she may find herself. In the previous section, we described the basic strategy in detail, so from now on, we will only mention the differences towards it, which for Lead stubborn are as follows:

- When Alice is in 2 or 2′ states and Bob mines a block, then Alice instead of revealing all of her blocks, she reveals a block to match the public chain.
- When Alice is in state 1′ and Alice mines a block then Alice takes no action.
- When Alice is in state 1′ and Bob mines a block, then Alice reveals a block to match the public chain.

For this strategy, the state 1′ is reachable as opposed to basic selfish mining. This is the result of Alice's higher stubbornness level.

3.4.2 Equal-Fork

Equal-Fork stubborn is the strategy in which Alice is more stubborn when public and private forks are of equal length; Alice has already revealed all of her blocks and has no other block to reveal.

If we recall the basic selfish mining functionality, we can observe that Alice did not take risky decisions when she was in a situation with equal forks (state $0'$). If she mined the next block, she would have immediately revealed it to secure all of her previously mined blocks. This is not the case for this stubborn strategy. If Alice is in such a situation and mines the next block, she will keep it secret instead of revealing it. The incentive behind this variation is that Alice wants to gain a lead immediately and therefore pull ahead of Bob. Consequently, this slight change eliminates the Risk Safety property since an infinite loop between states with a lead of zero and one without Bad Bob mining a block intermediately is possible, which can arbitrarily increase risk.

As before, we will only refer to the differences between this strategy and the basic selfish mining, which are the following:

- When Alice is in $0'$ or $1'$ states and Alice mines a block, then Alice performs no action.
- When Alice is in state $1'$ and Bob mines a block, then Alice reveals a block to match the public chain.

We notice that the $1'$ state is also reachable in this strategy in contrast to basic selfish mining.

3.4.3 Trail

So far, the strategies analyzed were referring to one and only one strategy. Trail stubborn defines a set of strategies with a distinction that characterizes them all. Alice remains stubborn even if the public chain is pulled ahead of her private chain; that is, the lead she poses is negative. Undoubtedly, these strategies are more stubborn than the basic selfish mining, as they not only allow Alice to lose the lead from her competitors but also she loses the advantage of Bad Bob (who unintentionally mines on top of her fork). This happens because when Bob is ahead of Alice and mines a block, Alice will not be able

to match his public chain fork as she lacks blocks. Therefore, Bad Bob will not mine anymore in favor of Alice. The different Trail stubborn strategies are distinguished by the negative lead that Alice allows before giving up. The parameter i, where $i \geq 1$, is the identifier for these strategies, and it is placed as a subscript after the word Trail. In general, Alice gives up when the lead is equal to $-(i+1)$. The main incentive behind these strategies is that Alice hopes that with her computational power, she will be able to overtake the public chain to secure blocks that with negative lead would have been otherwise lost. We want to mention that the only Trail stubborn strategy that was studied was T_1 because the results in [24] showed that higher profit could not be achieved for greater i. Additionally note that this strategy does not satisfy the Risk Safety property since the negative lead allows arbitrarily large paths between double apostrophe type states.

Regarding the differences between these strategies and the basic selfish mining, they are as follows:

- When Alice is in state $0'$ and Good Bob mines a block, then Alice performs "no action" (this allows negative lead).
- When Alice is in states $-i'' < s \leq 0''$ and Bob mines a block, then Alice performs "no action".
- When Alice is in state $-i''$ and Bob mines a block, then Alice abandons her effort and performs "restart".
- When Alice is in states $-i'' \leq s \leq -1''$ and Alice mines a block, then Alice performs "no action".
- When Alice is in state $0''$ and Alice mines a block, then Alice reveals all of her blocks.

Note that Alice still abandons her effort when she is in the $0'$ state and Bad Bob mines a block. The reason that led to this differentiation is that when Bad Bob mines a block, he confirms all of Alice's previous blocks. Therefore, Alice does not lose any blocks and there is not any reason to take additional risks. Alice will also lose the advantage of Bad Bob (when behind), which will make her effort much more difficult. Differentiating this behavior can lead to worse results.

3.5 Conservative Stubborn Selfish Mining

In this section, we present two new conservative variations inspired by the stubborn strategies of Section 3.4. The new strategies are designed because the corresponding stubborn strategies do not satisfy the Risk Safety property. Note that there are no changes for Trail stubborn strategies that can turn them into conservative stubborn, as we have named the new strategies. This unsatisfaction aroused our interest, and by making the appropriate changes to the Lead and Equal-Fork stubborn, we achieved the satisfaction of the Risk Safety property. The only difference is that Alice's maximum risk is two as opposed to one in basic selfish mining; thus, RS_2 property is satisfied. The satisfaction of this property will be studied in more detail in Section 5.5. For the time being, we can perceive it intuitively through the descriptions that will follow.

For better understanding the required strategy changes, in Figure 3.3, we illustrate six blockchain figures covering all the edge cases that need to be discussed. First, let us look at Figures 3.3b, d and f, which relate to the changes that had to be made to Lead stubborn to produce Safe-Lead conservative stubborn strategy. In Figure 3.3b, we see that Alice is in state $1'$ with a lead of one block, while previously she was in state $2'$ with a lead of two blocks where a new block emerged from Bad Bob. This resulted in securing all of Alice's previous blocks, as Bad Bob mined on top of Alice's fork. So, Alice will risk only her last two blocks in case of abandonment of her effort. On the other hand, in Figure 3.3d, the only difference is that when Alice was in state $2'$, the resulting block was from Good Bob. However, that does not bound Alice's risk since Good Bob mined on top of the public chain. So, Alice's risk is greater than two. Thus, Alice may decide to give up her effort, leading to a loss of an arbitrarily large number of blocks. Therefore, this made us think that by differentiating Alice's action depending on whether Bad Bob or Good Bob mined the last block will help. In Figure 3.3f, we see the last case of transiting to state $1'$, but this time from state 2, where there are not two forks yet. The risk is always two regardless of whether Bad Bob or Good Bob mined the last block, so a differentiation is not needed.

Figures 3.3a, c and e show the changes that needed to be made to the Equal-Fork stubborn to produce the Safe-Equal-Fork conservative stubborn. In Figure 3.3a, we see that Alice is in state $0'$, while previously she was in state $1'$, where a new block from Bad Bob emerged. This resulted in securing all of Alice's previous blocks, as Bad Bob

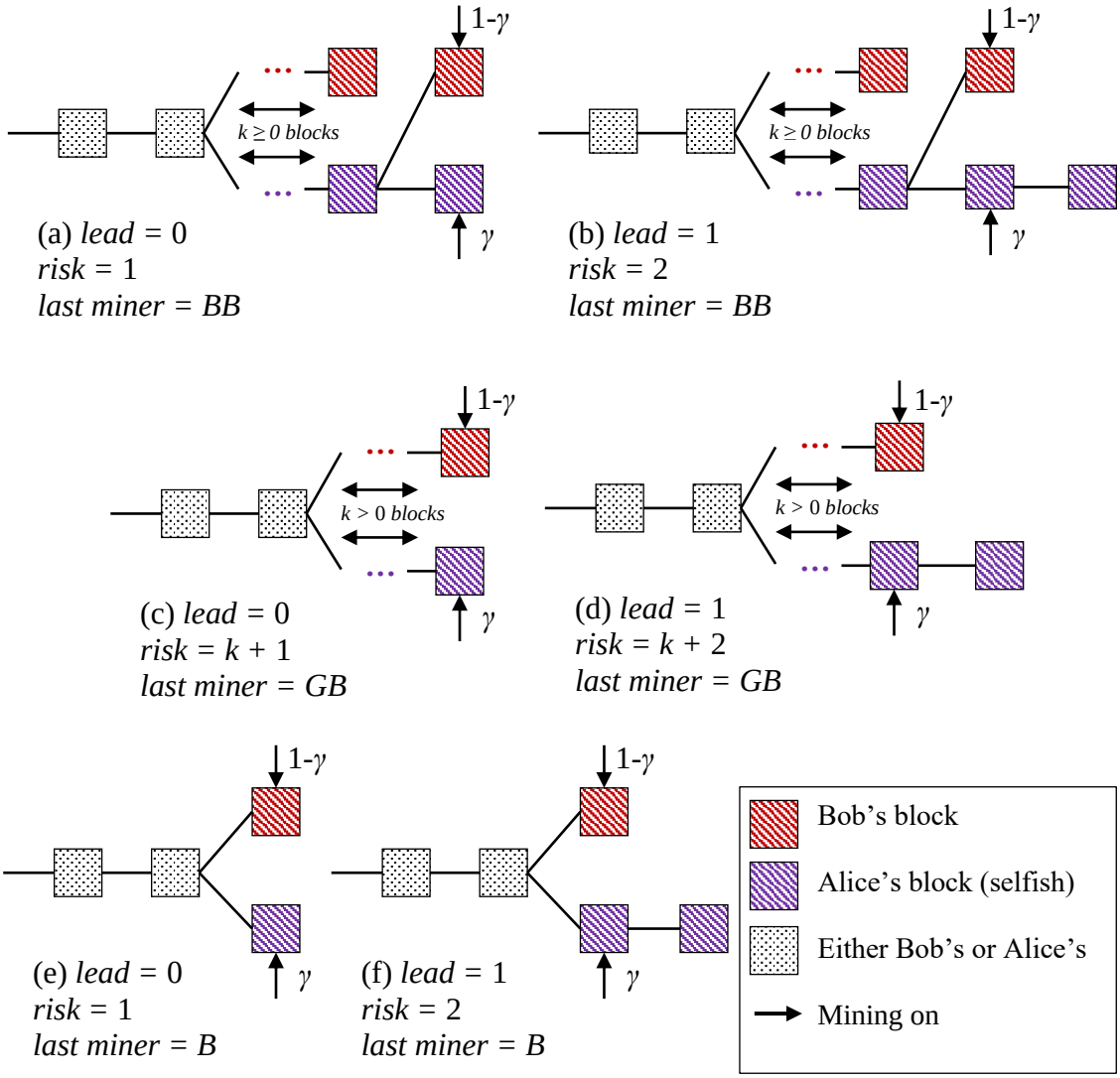


Figure 3.3: Representation of blockchain's states that motivated us to introduce the Safe-Lead and Safe-Equal-Fork conservative stubborn mining variations. Each state shows the lead of the selfish miner (lead), the number of selfish blocks that are not yet included in the public chain (risk) and the miner of last mined block. The arrows indicate the fraction of Bob's power which is mining on each block.

continued mining on top of Alice's fork. So, Alice will risk only her last block in case of abandonment of her effort. On the other hand, in Figure 3.3c, the only difference is that when Alice was in state $1'$, the resulting block was from Good Bob. However, that does not limit Alice's risk since Good Bob mined on top of the public chain. So, Alice's risk is greater than one. Thus, Alice may decide to give up her effort, leading to a loss of an arbitrarily large number of blocks. So, as before, we thought that a differentiation in Alice's actions depending on whether Bad Bob or Good Bob mined the last block would

help. In Figure 3.3e, we see the last case of transiting to state $0'$, but this time from state 1 where there are not two forks yet. The risk is always one regardless of whether Bad Bob or Good Bob mined the last block, so a differentiation is not needed.

With all the above observations, an introduction was made to the logic behind the modifications needed to transform stubborn strategies into conservative ones. Besides, the strategies were called conservative stubborn as they use "all" and "restart" actions less frequently than the basic selfish mining, but more often than stubborn strategies. In Figure 3.4, the state machine that illustrates how these strategies work is depicted; the two strategies are color separated.

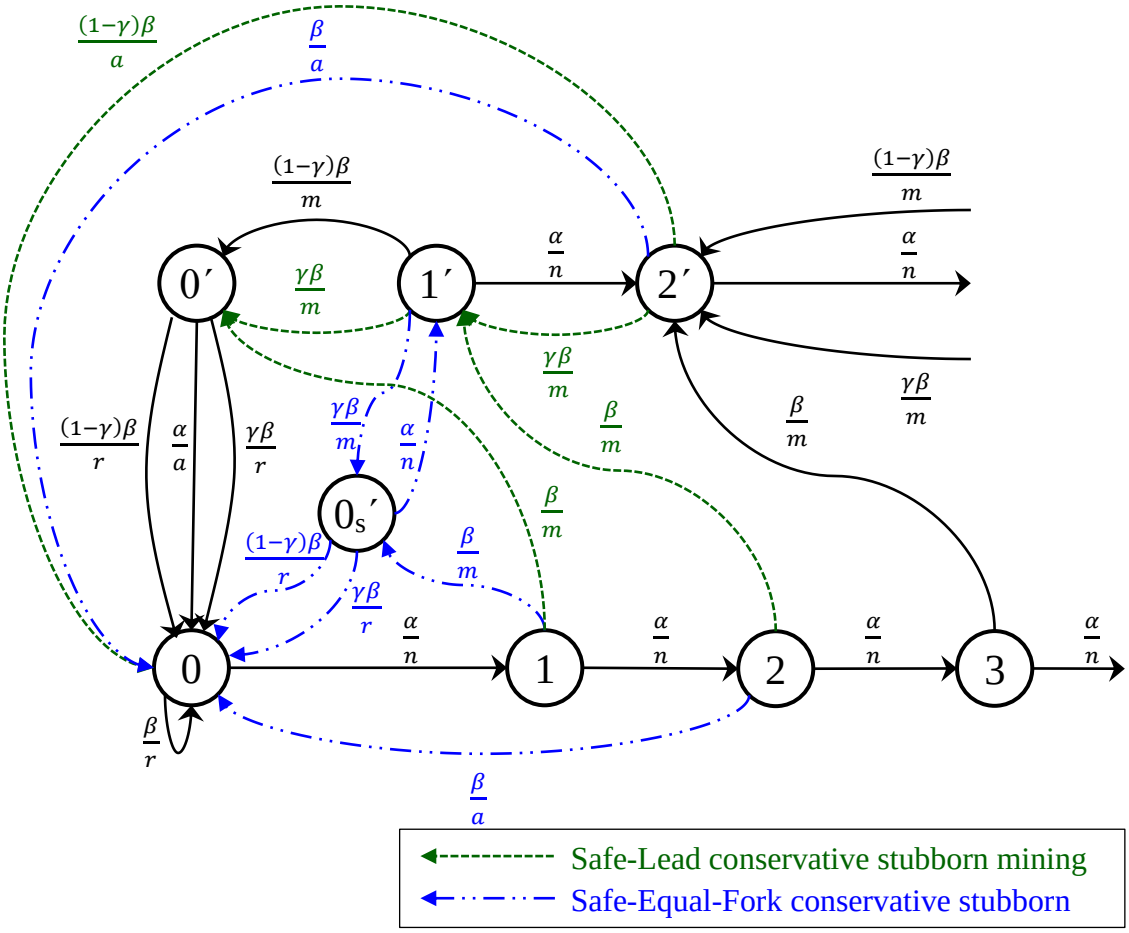


Figure 3.4: State machine of Safe-Lead and Safe-Equal-Fork conservative stubborn mining variations. (1) Green + **Black** indicates the state machine of Safe-Lead conservative stubborn strategy. (2) Blue + **Black** indicates the state machine of Safe-Equal-Fork conservative stubborn strategy.

3.5.1 Safe-Lead

Safe-Lead conservative stubborn is the strategy designed to satisfy RS_2 property when Alice wants to risk at most two blocks and to insist more on the lead she holds.

This strategy is almost identical to Lead stubborn. The only difference is that the action performed by Alice is differentiated when she has a lead of two blocks, and the next block is mined either from Bad Bob or Good Bob. Due to its conservative nature, when the next block is mined from Good Bob, Alice will behave the same as in basic selfish mining, while when the next block is mined from Bad Bob, Alice will behave like in the Lead stubborn mining. This, as we saw in Figure 3.3, ensures a maximum risk equal to two because Good Bob is mining solely on the public chain whereas Bad Bob is mining on Alice's fork; thus, he is acting like a safety barrier. This safety barrier allows Alice to be more stubborn when Bad Bob mines the next block. The reader can run any example of execution on the state machine, which we provide in Figure 3.4, to confirm that risk is always bounded to two blocks.

In more detail, the differences between this strategy and Lead stubborn are as follows (this strategy inherits all behavioral features of Lead stubborn):

- When Alice is in state $2'$ and Bad Bob mines a block, then Alice reveals a block and matches the public chain.
- When Alice is in state $2'$ and Good Bob mines a block, then Alice reveals all of her blocks.

Note that when Alice is in state 2, the entity that will mine the next block does not influence her decision; she will always match the public chain as in Lead stubborn.

3.5.2 Safe-Equal-Fork

Safe-Equal-Fork conservative stubborn is the strategy designed to satisfy RS_2 property when Alice wants to risk at most two blocks and insist on the presence of two equal forks.

As before, this strategy is almost identical to the corresponding stubborn strategy. The only difference is that in Safe-Equal-Fork, Alice sometimes behaves like in the basic selfish mining. This depends on the risk she runs when she is in a state where the lead is equal to zero. When Alice has a lead of zero, and the risk is less than or equal to one,

Alice follows the Equal-Fork stubborn strategy rules. This is especially the case when Alice has one block lead and Bad Bob mined the next block. Therefore, Bad Bob secures all of Alice's previous blocks, leaving only the last one in danger. In all other cases, specifically, when the risk is greater than one (and the lead is zero), Alice behaves as in the basic selfish mining. Hence, Alice's future actions depend on whether Good Bob or Bad Bob is the latest block's miner. In the latter case, Good Bob is always the miner of the latest mined block. We ensure that Alice will never risk more than two blocks by following this strategy with these changes.

In more detail, the differences between this strategy and Equal-Fork stubborn are as follows (this strategy inherits all behavioral features of Equal-Fork):

- When Alice is in state $1'$ and Good Bob mines a block, then Alice reveals a block to match the public chain. (Transition to state $0'$, which will affect future actions according to basic selfish mining)
- When Alice is in state $1'$ and Bad Bob mines a block, then Alice reveals a block to match the public chain. (Transition to state $0'_s$, which will affect future actions according to Equal-Fork stubborn)
- When Alice is in state $0'$ and Alice mines a block, then Alice reveals all of her blocks.
- When Alice is in state $0'_s$ and Bob mines a block, then Alice abandons her effort to restart a new cycle of selfish behavior.
- When Alice is in state $0'_s$ and Alice mines a block, then Alice performs "no action".

This strategy is the only one in which the state $0'_s$ is reachable. This is a consequence of the differentiation in Alice's decisions when there is a risk of one block, *i.e.,* Alice will risk in the presence of equal forks only when she runs a risk of one block since she wants to avoid the infinite loop between states with a lead of zero and one without Bad Bob mining intermediately a block, which can arbitrarily increase risk. Therefore, a dedicated state was necessary to distinguish such situations. As mentioned before, this state is the only one in our state machines that provides information about Alice's risk, which is one block.

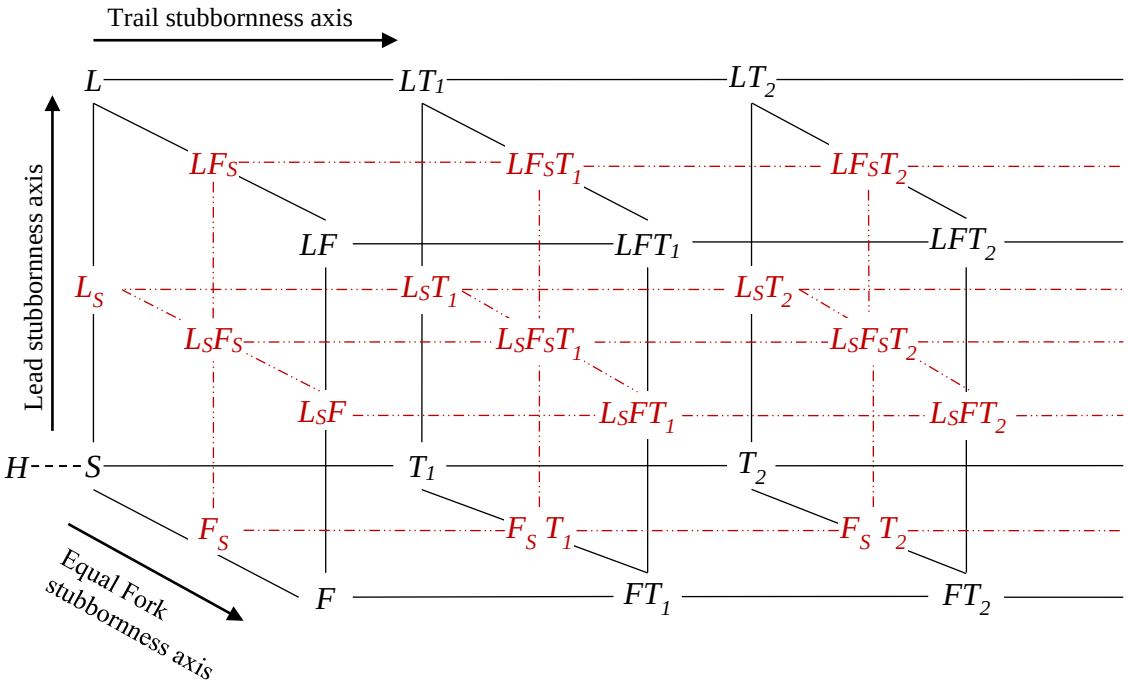


Figure 3.5: Strategies 3D space. A strategy is coordinated in the 3D space based on its stubbornness on the Lead, Equal Fork and Trail categories. Moving to the direction of the arrows means that the strategy is becoming more stubborn on that category. With dark red (dashed) color we mark our new conservative stubborn mining strategies in the 3D space.

3.6 Hybrid Strategies

The strategies described in the previous sections are not mutually exclusive, so they can be combined to form hybrid strategies that are more stubborn. The only combinations that can not be formed are those of Lead and Equal-Fork stubborn with their corresponding conservative stubborn strategies.

In total, there are three measures of stubbornness that characterize each hybrid selfish mining strategy. Therefore, all strategies can be presented in a three-dimensional space, as shown in Figure 3.5. The three measures are Lead, Trail and Equal-Fork stubbornness: (1) Safe-Lead and Lead strategies can increase the lead stubbornness of a hybrid strategy, (2) Safe-Equal-Fork and Equal-Fork strategies can increase the Equal-Fork stubbornness of a hybrid strategy, and (3) Trail strategies can increase the Trail stubbornness of a hybrid strategy. There is an infinite number of Trail strategies; thus, this dimension spans to infinity. In addition, note that Safe-Lead and Safe-Equal-Fork strategies do not always contribute the same amount of stubbornness to the corresponding measure. The degree

of contribution depends on the computational power of Bad Bob (γ). As γ approaches 0, then the behavior of both strategies approaches the basic selfish mining. On the other hand, as γ approaches 1, both strategies' behavior approaches the corresponding stubborn strategy's. Consequently, in the former case, they will not contribute to the stubbornness in the respective measure, while in the latter case, they will tend to contribute the same amount of stubbornness as the stubborn strategies do. The basic selfish mining is the strategy with zero Lead, Equal-Fork, and Trail stubbornness, so it is coordinated on the origin of the three-dimensional space.

The combination of strategies is not straight-forward because there are collisions between the decisions of which action Alice should perform on a specific occasion. For example, we have a collision when we want to form the Safe-Lead-Safe-Equal-Fork strategy (L_SF_S), and Alice has a lead of two blocks without the presence of a public fork yet (state 2 of state machine). When Bob mines the next block, the Safe-Lead strategy will ask Alice to perform the action "match", while the Safe-Equal-Fork will ask Alice to perform the action "all". The question is how we will resolve these conflicts.

To address these ambiguities, we will define some transitions for each strategy that we will call special transitions. These transitions characterize each strategy's differentiation from basic selfish mining and have no collisions between each other. In the existence of a collision, the transition that will be preferred is the transition of the strategy which is special, according to Table 3.3 which presents the special transitions for each of the strategies. In this way, all conflicts will be resolved, and the combination of strategies will be feasible since there are no conflicts between special transitions. The notation used to describe special transitions is given in Section 3.2. Concerning the collision of L_SF_S discussed earlier, we managed to resolve it since the special transition $(2, \beta, m, 1')$ of Safe-Lead strategy will be preferred. It is important to note that every hybrid strategy which contains at least one of the conservative stubborn variations is considered in the category of conservative stubborn strategies. Also, hybrid strategies are given the name of all combined variations, *e.g.,* combination of all stubborn variations will result to LFT_1 selfish mining hybrid strategy.

Algorithm 1 presents the pseudocode for the formation of hybrid strategies. This pseudocode was used to implement UPPAAL's automata and test cases described in Chapter 4. It shows how Alice's behavior changes depending on the simultaneously activated vari-

Strategy	***Special Transitions***
Lead (L)	$(2', (1-\gamma)\beta, m, 1'), (2', \gamma\beta, m, 1'), (2, \beta, m, 1')$
Equal-Fork (F)	$(0', \alpha, n, 1')$
Trail (T_i)	$(0', (1-\gamma)\beta, n, -1'')$
Safe-Lead (L_s)	$(2', (1-\gamma)\beta, a, 0), (2', \gamma\beta, m, 1'), (2, \beta, m, 1')$
Safe-Equal-Fork (F_s)	$(1, \beta, m, 0_s'), (1', \gamma\beta, m, 0_s')$

Table 3.3: Special transitions for each strategy.

ations. The activated variations depend on the $lead$, $slead$, $fork$, $sfork$, $trail$, and T_{len} input parameters. The last parameter determines the trail stubbornness, which we have seen in the discussion of Trail strategies, and is assigned with a positive integer value. The corresponding $lead$, $fork$, and $trail$ parameters must be set to true to activate the Lead, Equal-Fork, and Trail variations. To activate Safe-Lead conservative stubborn mining, set both $lead$ and $slead$ parameters to true; the same applies for Safe-Equal-Fork (set both $fork$ and $sfork$ to true). Alice is behaving as in basic selfish mining if all variations are disabled.

The pseudocode determines the decisions and necessary changes to all of Alice's variables and data structures for three possible events. The events that may occur are the initialization of miners *(line 1)*, the mining of blocks by miners belonging to Alice's coalition *(line 9)*, and the mining of blocks by miners who do not belong to Alice's coalition *(line 21)*. Regarding the miners' initialization *(lines 1-7)*, there is a simple update of the local blockchain with the publicly known blocks, and the selfish miner will start mining on top of the private chain.

As we have seen in the initialization, there is no differentiation caused by activated strategies. However, strategies will influence the decisions taken in the other two events. We will then try to uncomplicate the pseudocode with references to it by providing a more simplistic description so that the reader can easily understand it, but without referring to details concerning the updates of variables.

For each of the two remaining events *(line 9 and line 21)*, it must be made clear which action Alice decides to perform in any state of the state machine she can be found. Starting from the event of block mining by the miners of the coalition of Alice *(lines 9-19)*, the following apply:

1. *(lines 13-14)* Alice is in state $0'$ or $0'_s$. If Equal-Fork is activated then she does not perform any action *(SM1 lines 2-3)*. If Safe-Equal-Fork is activated then she does not perform any action only if she is in safe zero state *(SM1 lines 2-3)*. In all other occasions she reveals all of her unpublished blocks *(SM1 lines 4-6)*.

2. *(lines 15-18)* Alice is in state $0''$ and she reveals all of her unpublished blocks.

3. She does not perform any action in the other states (indirectly stated).

Continuing to the event of block mining by miners who do not belong to Alice's coalition *(lines 21-40)*, the following apply:

1. *(lines 24-25)* Alice is in states s, $-T''_{len} < s \leq 0''$ and she does not perform an action.

2. *(lines 26-29)* Alice is in states 0 or $-T''_{len}$ and she abandons her effort.

3. *(lines 30-32)* Alice is in state $0'$. If Trail is activated and the block belongs to Good Bob then she does not perform an action *(SM2 lines 2-3)*. In all other occasions she abandons her effort *(SM2 lines 4-6)*.

4. *(lines 33-35)* Alice is in state 1 or $1'$ and she reveals her last unpublished block. If Safe-Equal-Fork is activated and the block belongs to Bad Bob or she was in state 1 then she transits to state $0'_s$ *(SM3 lines 2-3)*.

5. *(lines 36-37)* Alice is in state 2 or $2'$. If Lead is activated then she reveals one of the unpublished blocks *(SM4 lines 2-3)*. If Safe-Lead is activated and the block belongs to Bad Bob or she was in state 2 then she reveals one of the unpublished blocks *(SM4 lines 2-3)*. In all other occasions she reveals all of her unpublished blocks *(SM4 lines 4-6)*.

6. *(lines 38-39)* Alice is in a state with lead greater than 2 and she reveals one of her unpublished blocks.

Regarding the variables *privateBranchLen, isBehind, and safeZero*, they are used to distinguish the three types of states. The *privateBranchLen* variable helps to distinguish the single apostrophe states from the states without an apostrophe. The *isBehind* variable helps to distinguish the states with a double apostrophe, and the *safeZero* variable helps

us to distinguish the state $0'_s$. Furthermore, marked in red are the pseudocode changes to support L_S and F_S conservative stubborn variations alongside the rest of the strategies.

Algorithm 1 Hybrid Strategies

Input: *lead, slead, fork, sfork, trail,* T_{len}

on Init
    public chain ← publicly known blocks
    private chain ← publicly known blocks
    privateBranchLen ← 0
    isBehind ← false
    safeZero ← false
    Mine at the head of the private chain

on My Miners found a block
    Δ_{prev} ← length(private chain) - length(public chain)
    append new block to private chain
    privateBranchLen ← *privateBranchLen* + 1
    if $\Delta_{prev} = 0$ *and privateBranchLen* ≥ 2 *and not isBehind* **then** ▷ $State = 0', 0'_s$
        SM1
    else if $\Delta_{prev} = 0$ *and isBehind* **then** ▷ $State = 0''$
        publish all of the private chain
        privateBranchLen ← *0*
        isBehind ← *false*
    Mine at the new head of the private chain

on Others found a block
    Δ_{prev} ← length(private chain) - length(public chain)
    append new block to public chain
    if $\Delta_{prev} > -T_{len}$ *and isBehind* **then** ▷ $-T''_{len} < State \leq 0''$
        do nothing
    else if ($\Delta_{prev} = 0$ *and privateBranchLen = 0) or* $\Delta_{prev} = -T_{len}$ **then** ▷ $0, -T''_{len}$
        private chain ← public chain
        privateBranchLen ← *0*
        isBehind ← *false*
    else if $\Delta_{prev} = 0$ *and privateBranchLen* ≥ 1 *and not isBehind* **then** ▷ $State = 0'$
        SM2
        safeZero ← *false*
    else if $\Delta_{prev} = 1$ **then** ▷ $State = 1, 1'$
        SM3
        publish last block of the private chain
    else if $\Delta_{prev} = 2$ **then** ▷ $State = 2, 2'$
        SM4
    else if $\Delta_{prev} > 2$ **then** ▷ $State > 2, 2'$
        publish first unpublished block in private chain
    Mine at the head of the private chain

SM1, SM2, SM3 and SM4

```
1: SM1
2:     if (fork and not sfork) or safeZero then
3:         safeZero ← false
4:     else
5:         publish all of the private chain
6:         privateBranchLen ← 0
```

```
1: SM2
2:     if trail and not controlled() then
3:         isBehind ← true
4:     else
5:         private chain ← public chain
6:         privateBranchLen ← 0
```

```
1: SM3
2:     if fork and sfork and (controlled() or privateBranchLen = 1) then
3:         safeZero ← true
```

```
1: SM4
2:     if lead and (not slead or controlled() or privateBranchLen = 2) then
3:         publish first unpublished block in private chain
4:     else
5:         publish all of the private chain
6:         privateBranchLen ← 0
```

controlled(): It is true if and only if the miner who mined the last honest block is *BB* (well connected with the selfish miners), thus the previous block belongs to Alice.

Chapter 4

Implementation

Contents

4.1 Overview

We have implemented the bitcoin's network as UPPAAL's automata [4]. Overall our implementation in UPPAAL STRATEGO [15] consists of three automata that simulate the interaction between our network's entities, namely selfish miners (alongside their strategy) and honest miners. Four automata are composed into a system representing Alice and her strategy, Good Bob, and Bad Bob. Our implementation abstracts the complexity of the Bitcoin protocol and block-transaction structures. It only deals with the dissemination of blocks and the time elapsed between mining two distinct blocks (transactions are completely omitted since they are not affected by the selfish mining strategy, and they would have added unnecessary complexity to the system). In this chapter, firstly, we describe the implementation of the selfish miner (Alice) and its strategy. Then, we provide the implementation of the honest miner (Bob), and at the end of the chapter, there is a short description of how we maintained the correctness of our automata' functionality.

Before proceeding to the first section of this chapter, there is a short introduction to the

exponential distribution. As we discussed in Section 2.3, timed automata [7] are used to model the behavior and test real-time systems' properties. In our distributed bitcoin network system, we wanted somehow to model the elapsed time between mining two distinct blocks. Moreover, this time must variate according to the proportion of computational power the miner poses. For this purpose, the exponential distribution is the appropriate means to represent the time elapsed between events (UPPAAL STRATEGO also supports it). More specifically, the exponential distribution is used to predict the amount of waiting time until the next event, which in our case is a new block. The exponential distribution has a single parameter, which is called rate (λ), and it specifies the event rate, that is, the number of events per time unit (the time unit can be interpreted as any time interval).

4.2 The Selfish Miner

The automaton of selfish miners, called SMiners, which is depicted in Figure 4.1, represents the coalition of selfish miners under Alice's supervision. This automaton consists of seven locations, of which location Mine is the core of the mining procedure while the rest exist for decision-making purposes and they are committed. Furthermore, it maintains a local blockchain where blocks reside. Our automaton has the Finish location which is the game ending location. (In our experiments we "played" a game of 1000 blocks in order to find the best strategy regarding revenue at state Finish)

The automaton begins from location Start. This means that the automaton is not initialized yet; hence initializations must be made through the first transition's updates, such as blockchain initialization with the genesis block (initialize()). The transition to location Mine is initially deactivated until strategies are set from the strategy automaton, depicted in Figure 4.2. This is handled by guard stratsReady(), which checks whether strategies are set.

The mining procedure occurs at location Mine. The exponential distribution is used at mining locations to model the time elapsed between two blocks being mined. Therefore, it is assumed that the probability of leaving the location is distributed according to the exponential probability. In other words, approximately $\frac{rate\_a}{1000}$ (rate_a:1000) blocks are mined every time unit. The time unit is interpreted as one minute; thus, the $rate\_a$, which is one of the automaton's parameters, will determine the mining rate. In this work, we

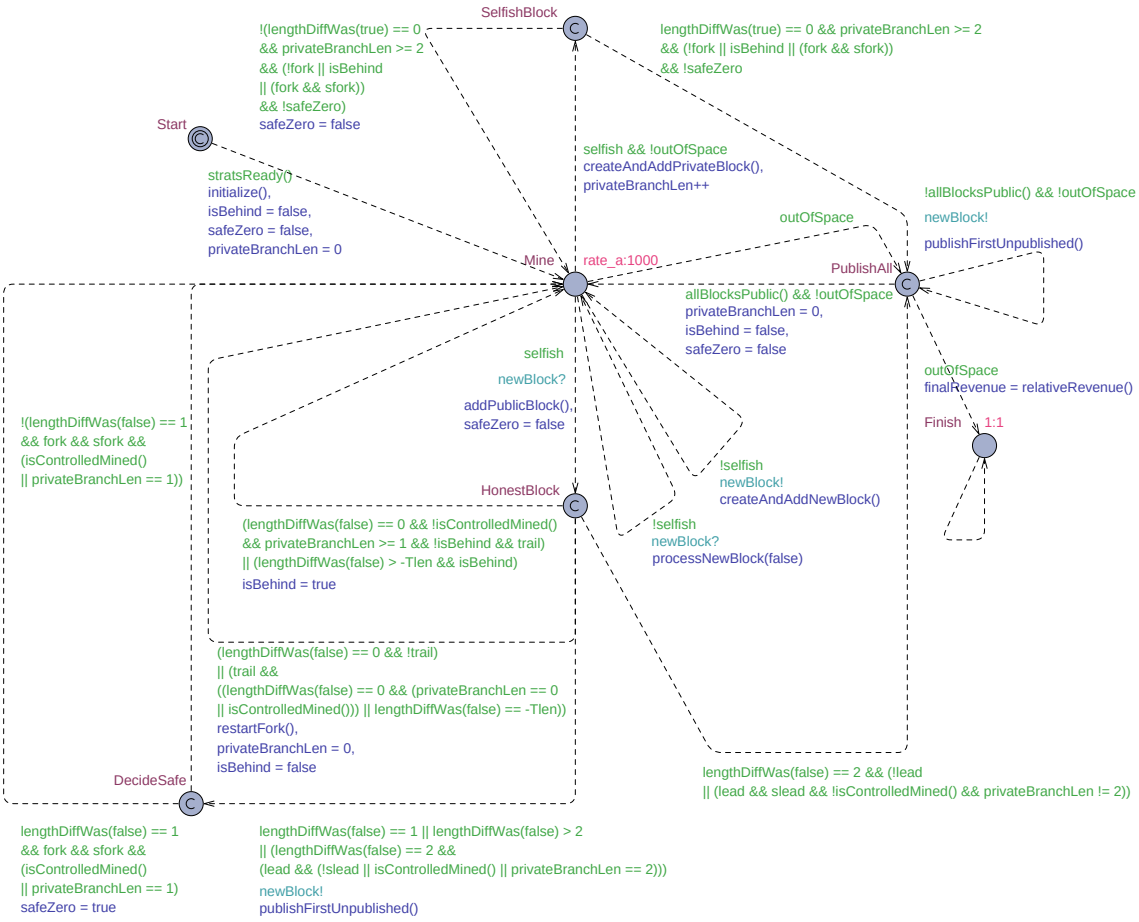


Figure 4.1: Selfish miners automaton UPPAAL STRATEGO.

used *rate_a* to specify the proportion of the network's total mining power, which this coalition of selfish miners poses. For example, setting *rate_a* equal to 50 implies that selfish miners own 50% of the total computing power of the entire network and this miner will mine approximately 1 block every 20 minutes. The denominator 1000 is justified because Bitcoin adjusts mining difficulty so that a block will be mined approximately every 10 minutes (10 minutes multiplied by 100). The denominator can be modified if someone wants to interpret time differently, *i.e.,* time is interpreted as seconds or to allow more precise proportions (this was useful during our experiments). The same applies in Section 4.4. In general, this rate can change according to the needs of model's users.

Regarding the publication of blocks, miners can publish and receive blocks through the broadcast channel newBlock being at location Mine. Recall that in Algorithm 1, we distinguished two types of events, on Others and My Miners. This distinction also applies to the selfish miners automaton. When selfish miners automaton is at location Mine, and

honest miners automata mine a block, then they will broadcast it through the newBlock channel, and eventually, selfish miners will receive it. This signal will result in selfish miners automaton transitioning from location Mine to location HonestBlock, where it will have to decide its next action. Before that, it will add the new block to its local blockchain with the update function addPublicBlock(). At location HonestBlock, there exist four outgoing transitions. These transitions represent the four possible actions that Alice can perform, which are "all", "restart", "match", and "no action". The outgoing transition to the left side of the location represents the action "no action". The outgoing transition to the right bottom represents the action "all", while the rest represent the actions "restart" and "match". The latter is represented by the outgoing transition, which leads to location DecideSafe. The automaton will publish the first unpublished block through newBlock! broadcast channel with the help of publishFirstUnpublished() update function. The location DecideSafe decides whether selfish miners are in a safe zero state, as described in Section 3.5. On the other hand, the transition, which represents the action "restart", will force selfish miners automaton to mine on top of honest miners' fork (restartFork()).

When selfish miners automaton is at location Mine and mines a block as described earlier with the exponential distribution, then the transition to the location SelfishBlock will be taken. However, the automaton will not immediately publish a block while taking this transition. First, a new block will be created and added to the local blockchain; think of that as a withheld block from selfish miners. Then, at location SelfishBlock, a decision will determine the next action. Overall, there are two possible actions at this location, which are "no action" and "all". The outgoing transition to the left side of the location represents the action "no action", whilst the outgoing transition to the right the action "all". At location PublishAll, the automaton will publish all unpublished blocks through newBlock! broadcast channel one by one until allBlocksPublic() is satisfied.

Throughout the automaton description, we omitted the explanation of the transitions that form self-loops onto location Mine. These transitions correspond to the functionality of honest mining described later in Section 4.4. Also, note that all selfish miners automaton's transitions are not controlled (dotted) since they are not the required transitions to be remembered by the controller when finding the best strategy (the execution traces differ every time, although the same strategy is followed). Furthermore, we have not paid much

attention to the transitions' guards since they are complicated and not easy to understand. The correctness of the model is shown later in Section 4.5.

Undoubtedly, the strategy adopted from the automaton of selfish miners plays a pivotal role in the decision-making mechanism at locations HonestBlock, and SelfishBlock. Strategies can significantly affect guards' evaluation of their outgoing transitions. Strategy selection will be the next topic that we will address to complete the description of selfish miners realization as UPPAAL automata.

4.3 The Strategy

Selfish mining strategies is the main topic of this work. So far, we have seen the SMiners automaton but not how selfish miners' strategy is chosen. For this purpose, a dedicated automaton was implemented, shown in Figure 4.2. This automaton consists of six locations (all committed) that make feasible strategy selection. The Strategy automaton must be linked with the corresponding SMiners automaton by providing the SMiners automaton's identity as a parameter (**id**). Therefore, the strategy will be available to the appropriate SMiners automaton depicted in Figure 4.1. The strategy is adopted from SMiners automaton when update function initialize() is executed, which also initializes the strategy alongside other initializations.

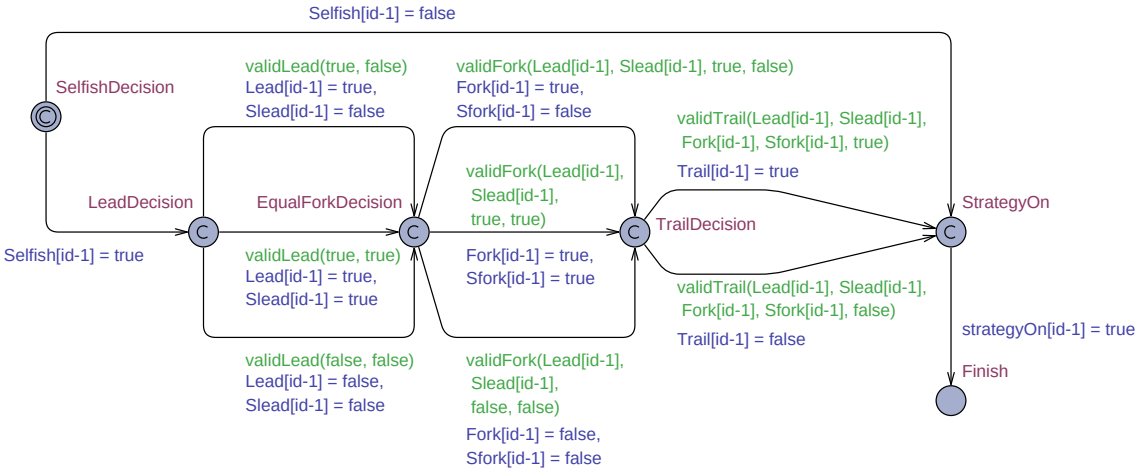


Figure 4.2: Strategy initialization automaton UPPAAL STRATEGO.

The Strategy automaton begins from location SelfishDecision. This location is responsible for deciding whether selfish miners will be selfish or not. If selfish miners are

chosen not to be selfish (Selfish[id-1]=false), then the automaton will skip selfish mining variations and will directly transit to location StrategyOn. On the other hand, if selfish miners are chosen to be selfish, then the automaton will transit to location LeadDecision. At this location, the automaton has three options. A selfish miner can follow the Lead variation (Lead[id-1]=true, Slead[id-1]=false), the Safe-Lead variation (Lead[id-1]=true, Slead[id-1]=true), or none of them (Lead[id-1]=false, Slead[id-1]=false). The automaton will select one of these options, and it will transit to location EqualForkDecision. At this location, the automaton has three options: Equal-Fork, Safe-Equal-Fork, or none of them, but this time the changes are made to the variables **Fork[id-1]** and **Sfork[id-1]** in the same fashion as at LeadDecision location. After selecting the Equal-Fork variation, the automaton transits to location TrailDecision. At this location, there are only two options since there is not a conservative stubborn variation. Therefore, the automaton has to choose whether the Strategy will enable Trail variation (Trail[id-1]=true) or not (Trail[id-1]=false). This is the last decision the automaton has to make, which will subsequently lead to location StrategyOn. This location has only one transition, which signals the completion of strategy selection by setting strategyOn[id-1]=true and transiting to location Finish. Then, the SMiners automaton will be able to take the transition from location Start to location Mine since the Strategy is defined, and its guard is satisfied.

Overall, there are 19 strategies, both basic selfish mining and honest mining included. Strategy automaton offers the capability of disabling strategies. This capability is necessary because some experiments must be done to a restricted number of and specific strategies. For this purpose, the automaton has an array parameter of length 17 (**disable[17]**) (honest strategy and basic selfish mining cannot be turned off). When instantiating the strategy template in system declarations, some strategies can be disabled by providing the appropriate array. To learn more about the appropriate array format, refer to Listing A.5, listed in Appendix A. The guards validLead(...) (on transitions from LeadDecision to EqualForkDecision), validFork(...) (on transitions from EqualForkDecision to TrailDecision), and validTrail(...) (on transitions from TrailDecision to StrategyOn) are handling recursively the request of disabling strategies. More specifically, they turn off paths of the Strategy automaton that define disabled strategies. Furthermore, note that all strategy automaton transitions are controlled (solid) since they are the transitions to be remembered by the controller when finding the best Strategy. They contain all the necessary

information about the adopted strategy.

In addition to the UPPAAL STRATEGO implementation, we provide an UPPAAL SMC implementation. There are very few differences between them. The latter enables the manual assignment of Strategy by providing it through parameters to the automaton template instantiation. Therefore, the Strategy automaton is no longer necessary. The UPPAAL SMC automata can be seen in Figures B.1 and B.2, listed in Appendix B. Moreover, UPPAAL SMC is the implementation used to test our model, but the correctness also generalizes to the UPPAAL STRATEGO since the changes do not affect the SMiner automaton except for its automated strategy selection.

4.4 The Honest Miner

The last entity of our bitcoin network, which was modeled, is honest miners. The automaton of honest miners is called HMiners (Figure 4.3), and it can either represent Good Bob or Bad Bob. Since both entities exist in our network, there are two instantiations of this template, one representing Good Bob and one representing Bad Bob. This purpose serves the parameter **controlled** given to the automaton during instantiation. If parameter **controlled** is set to true, then the automaton's behavior is based on Bad Bob's behavior; otherwise, the automaton behaves such as Good Bob.

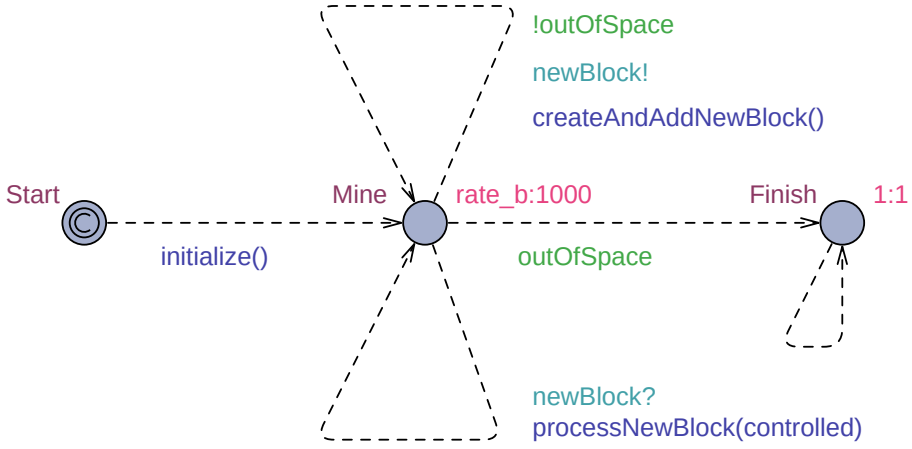


Figure 4.3: Honest miners automaton UPPAAL STRATEGO.

The implementation of HMiners automaton is straightforward because honest miners are only receiving and publishing blocks without any sophisticated strategy. The automaton begins at location Start, and it transits to location Mine. Before this transition, it

performs some initializations (initialize()), such as initializing blockchain with the genesis block.

As mentioned earlier, the exponential distribution is used at mining locations. Therefore, approximately $\frac{rate\_b}{1000}$ (rate_b:1000) blocks will be mined every time unit. The time unit is interpreted as one minute; thus, the *rate_b*, which is one of the automaton's parameters, will determine the mining rate. For example, setting *rate_b* equal to 50 implies that honest miners of this automaton hold 50% of the total computing power of the entire network and this miner will mine approximately 1 block every 20 minutes.

When this automaton is at location Mine, it can either receive a block (newBlock?) or publish a newly mined block (newBlock!) based on exponential distribution. The received block will be added to the local blockchain (processNewBlock(controlled)) of this automaton by taking the transition below location Mine. If the automaton mines a new block, it will choose the transition above the location Mine. The function createAndAddNewBlock() creates the new block, and this block is disseminated through channel newBlock. Also, note that all honest miners automaton's transitions are not controlled (dotted) for the same reason as in SMiners automaton. Lastly, the location Finish is the game ending location as discussed earlier in SMiners automaton.

4.5 Testing the Model

In Section 4.2, we perceived the complexity of guards on the transitions of our SMiners automaton, but at that moment, we deemed that they are correctly defined. In this work, we implemented the testing code as Java classes to test that our automata function correctly according to Algorithm 1. Recall from Section 2.3 that UPPAAL offers the capability of creating a file for a specific trace. More specifically, code can be placed in system declaration to determine the prefix and suffix of the produced file, on transitions and during exiting/entering Locations in the order presented in the trace. The prefix is used to create a class called Test, which contains the main and the suffix to close this class's brackets. Moreover, this class extends the App class, which will be explained in detail in short.

In general, there are four classes, namely, Action, Block which implements the block's structure used in UPPAAL, SelfishMiner which implements the pseudocode of Algorithm

1, and App. The last is responsible for tracking UPPAAL's automata actions and maintaining information provided by automata, such as newly created blocks and SMiners local variables. These classes are presented in Listings C.1, C.2, C.3, and C.4 respectively, listed in Appendix C.

Class App is the most important since it maintains the information provided by the automata. It can be thought of as the core of our testing implementation since it synthesizes every class and provides functions to be placed on automata edges and locations. In general, it handles the code presented on transitions and the code when automata are entering or exiting a Location. This class provides a set of functions that can be placed on automata. We can arrange this set into two sub sets. First are the functions which provide information of the state of automata to the Java implementation and second, are the functions that indicate which action was performed. Here, we reference lines of code for each of the functions provided by class App, listed in Listing C.4. Therefore, the functions which provide information about the state and blocks of automata are the following (this code is placed on SMiners automaton):

1. *(lines 14-17)* Initialize the strategy followed by selfish miners. On the transition from Location Start to Mine.

2. *(lines 19-54)* Check if the local variables of SMiners automaton, which are maintained for decision making, are as expected. On entering Location Mine.

3. *(lines 56-71)* Add and Check if the new block mined from SMiners automaton is as expected. On entering Location SelfishBlock.

4. *(lines 73-78)* Add to blockchain of selfish miners the new block mined from HMiners automaton. On entering Location HonestBlock.

Furthermore, the functions that indicate which action was performed are the following (this code is placed on SMiners automaton):

1. *(lines 80-92)* Check if action "all" was the expected action.

2. *(lines 94-106)* Check if action "no action" was the expected action.

3. *(lines 108-120)* Check if action "restart" was the expected action.

4. *(lines 122-134)* Check if action "match" was the expected action.

These functions are placed on transitions corresponding to the action in question according to Section 4.2.

In conclusion, with the test cases, we can test our implementation by producing a wide range of traces for each selfish miners' strategy and, therefore, their corresponding main function of the Test class. Furthermore, we can detect errors with the exceptions prompted during test cases, and the necessary adjustments can be made. The test cases' procedure is depicted in Figure 2.1. Test cases prove our final implementation's correctness and are an essential part of our work.

Chapter 5

Evaluation

Contents

5.1 Overview

This chapter is dedicated to the evaluation of our new conservative stubborn strategies with UPPAAL STRATEGO. The evaluation required to examine the efficiency of strategy space strategies, stubborn and basic selfish mining included, for different values of α and γ in parameter space. Note that we consider every strategy in the strategy space except those that enable the Trail variation with trail stubbornness greater than 2. More specifically, we consider 18 selfish mining strategies in our experiments. The values of α and γ range from 0.01 to 0.50 and 0 to 1 respectively, both with a step of 0.01. Therefore, there are 5050 different parametrizations to consider when Uppaal verifies the queries in place. Values of α greater than 0.50 have no interest because an attacker with access to more than 50% of the network's computational power can launch the 51% attack, which is by far a stronger attack (the attacker can reap all the rewards of the network). Furthermore, not all γ values are likely because well connectivity with almost the entire network seems impossible. However, for completeness reasons, we included every value of γ, although

their possibility of occurrence is negligible. Moreover, for each execution of the model, a limit of 1000 blocks to be mined from miners is set; hence, a game of 1000 rounds is "played".

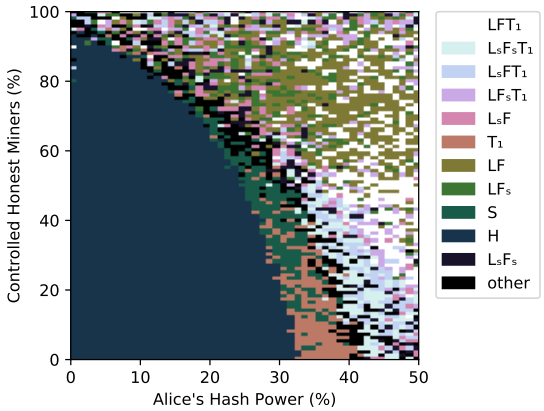
We provide and explain the queries in the corresponding forthcoming sections. First, we discuss the results of dominant strategies for each combination of α and γ. Then, we present the relative revenue of selfish miners compared to the expected honest revenue and stubborn strategies' revenue. After that, we discuss about the fairness of bitcoin's blockchain when miners adopt such strategies. Finally, we present the queries verifying the RS_2 property of our newly introduced conservative stubborn variations and the hybrid strategy combining them.

5.2 Dominant Strategies

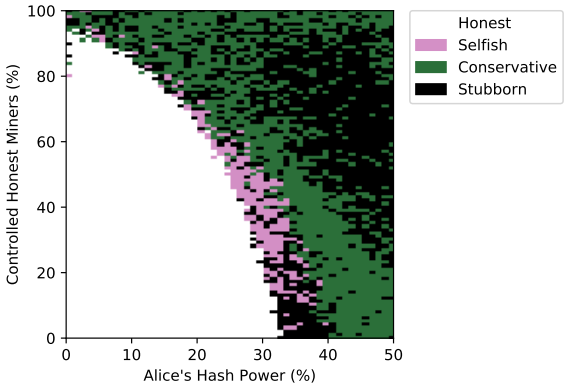
To identify the dominant strategies, we used the queries 5.1 and 5.2. In short, query 5.1 identifies the strategy which maximizes the final revenue of Alice. The bound of model time simulation is set to 20000 because we know that it is sufficient to reach the game ending location (the same applies to any feature query). This query must be the first one to run since the rest of the queries are verified under its strategy. Recall that only the Strategy automaton has controlled transitions that the controller of UPPAAL STRATEGO controls. In order to verify a query under the revenue maximization strategy, the identifier MaxRevenue should be used at the end of the query. However, in order to see which strategy is preferred from UPPAAL STRATEGO, we need to run simulations with query 5.2 under the revenue maximization strategy to track changes in strategy variables. Thus, the preferred strategy will be made known to us by observing which is the most popular strategy selected by the controller over 100 simulations.

```
strategy MaxRevenue =  maxE (Alice.finalRevenue) [<=20000]
                       : <> Alice.Finish
```
(5.1)

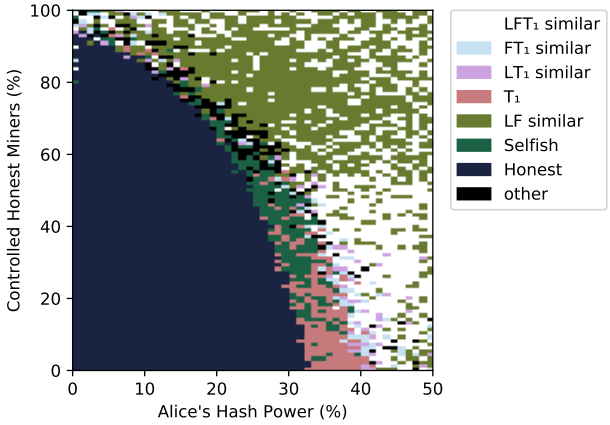
```
simulate 100 [<=50]{Alice.lead, 0.1+Alice.slead, 2+Alice.fork,
                    2.1+Alice.sfork, 4+Alice.trail,
                    6+Alice.selfish} under MaxRevenue
```
(5.2)



(a) Best strategy per region.



(b) Conservative stubborn strategies vs stubborn strategies.



(c) Best family of strategies per region.

Figure 5.1: Best strategies for different values of α (Alice) and γ (controlled honest miners).

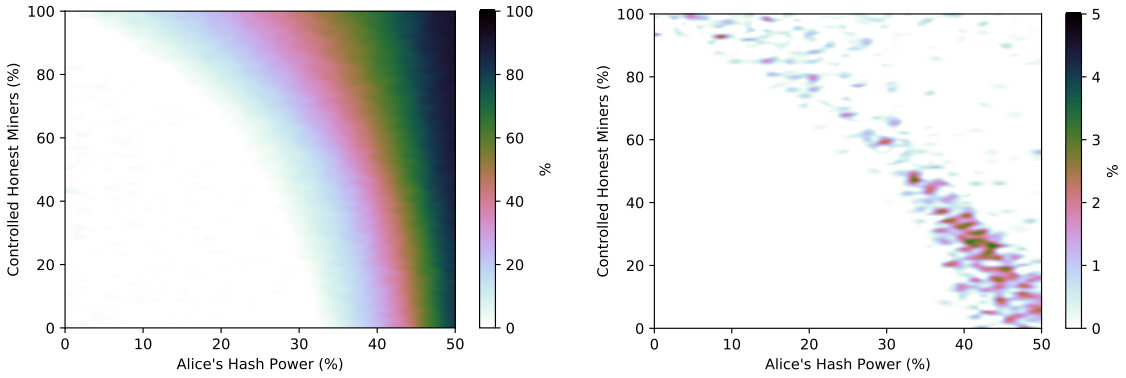
In Figure 5.1, we show the results of our experiments for the dominant strategies. In Figure 5.1a, we present the best selfish mining strategies for different values of α and γ. We observe that hybrid strategies form clusters in the parameter space. Clearly, no single hybrid strategy performs better in the entire space of parameters, but different hybrid strategies are preferred in localized regions.

In Figure 5.1b, we demonstrate a comparison between conservative stubborn and stubborn strategies. Apparently, the new hybrid strategies formed with at least one conservative stubborn variation, in many cases displace the pre-existing hybrid strategies. Hence, stubborn mining is not optimal for a large fraction of the parameter space.

Finally, in Figure 5.1c, we present the best family of strategies for each parametrization. A family of strategies consists of the hybrid strategies, which contain either the stub-

born or the respective conservative stubborn variations. We observe that the LFT_1 family outperforms the FT_1, which was the dominant strategy when solely stubborn strategies were applied, for high α values and low γ values. Recall that hybrid strategies are named after the combination of the names of their enabled variations (Table 3.1). More specifically, the strategies of LFT_1 family that outperform FT_1 are the conservative stubborn members of the family ($L_SF_ST_1$, L_SFT_1, and LF_ST_1), as marked in Figure 5.1.

5.3 Revenue and Comparison with Stubborn Strategies



(a) Relative revenue of best strategy compared to expected honest gain.

(b) Relative revenue of best strategy compared to stubborn variations.

Figure 5.2: Comparisons of relative revenue of Alice's best strategy for different values of α (Alice) and γ (controlled honest miners).

To evaluate a strategy, we consider Alice's relative revenue when performing a strategy, which is the percentage increase compared to the revenue of another strategy. The following equation gives the relative revenue of Alice when performing strategy X compared to strategy Y:

$$relative\_revenue(X,Y) = \frac{revenue_X - revenue_Y}{revenue_Y} \times 100 \tag{5.3}$$

where $revenue_X$ is the fraction of blocks earned by Alice under strategy X. In this section, we compare Alice's relative revenue for honest mining and stubborn mining (the comparison is made among every strategy both honest and stubborn included).

To calculate Alice's final revenue, we used the estimation query of 5.4, which is executed under the revenue maximization strategy. This allowed estimating the fraction of

blocks earned by Alice under the best strategy for each parametrization. The estimation query run 100 simulations for a duration of 20000 time units of the model which is required to reach the final location.

```
E[<=20000;100] (max: Alice.finalRevenue) under MaxRevenue
```
(5.4)

In Figure 5.2, we present Alice's relative revenue in comparison with honest mining and stubborn mining. The relative revenue results compared to honest mining, depicted in Figure 5.2a, are similar to [24]. In general, we observe that selfish mining is more profitable than honest mining in a wide range of parameters. Hence, a miner is incentivized to deviate from the consensus protocol to follow a selfish mining strategy. As parameter α increases, the greater is the percentage increase in the revenue of Alice. The same applies to parameter γ. When parameter α exceeds 0.5, then a 51% attack will be feasible. This will permit Alice to absorb the entire revenue of the blockchain by invalidating every other block.

However, does conservative stubborn mining increase significantly the selfish miner's revenue? The answers are given in Figure 5.2b, where we show Alice's relative revenue compared to stubborn mining. To calculate the relative revenue, we used the best stubborn mining strategy for the respective parameters. Our new conservative stubborn mining strategies offer up to 5% percentage increase to Alice's revenue compared to the best strategy of stubborn mining. This is mainly observed to high α values (0.4 to 0.5) and low γ values (0 to 0.4) where the LFT_1 family outperforms the FT_1 family of hybrid strategies. This signifies that conservative stubborn mining has better results when both γ and α are not small either big at the same time, *i.e.,* the regions with better revenue shown in Figure 5.2b. Therefore, this implies that less stubborn strategies, *i.e.,* conservative stubborn, were necessary for some parameter space regions to handle more careful decision making.

5.4 Fairness of the Blockchain

A blockchain needs to be incentive-compatible in terms of its honest policy in any circumstance. Therefore, it should not entitle anyone to consider deviating from the consensus protocol for personal profit.

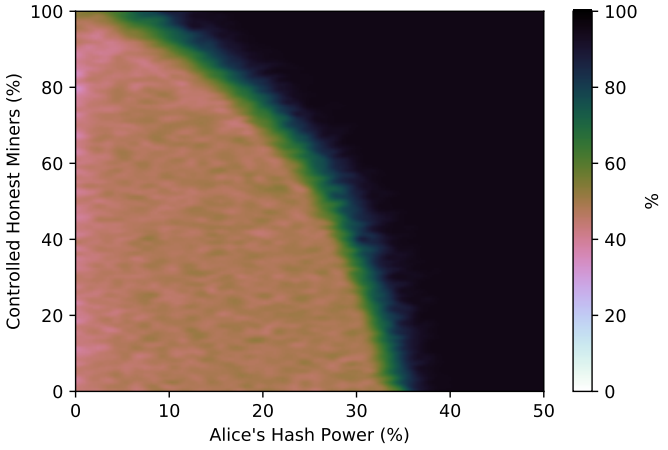


Figure 5.3: Probability of earning more than fair share for different values of α (Alice) and γ (controlled honest miners).

This section observes a selfish miner's probability of earning more than its fair share by following selfish strategies. In a fair protocol, one would expect miners with an α fraction of the computational power to harvest an α fraction of the rewards. The fairness probability can be verified with query 5.5, which calculates the probability for the final revenue to exceed the fair share fraction of α under the best selfish mining strategy.

```
Pr[<=20000](<> Alice.finalRevenue > a) under MaxRevenue      (5.5)
```

In Figure 5.3, we present our results for the probability of exceeding the fair share in terms of revenue. For the values of the parameters in parameter space where the honest policy is preferred, we see that the selfish miner's probability exceeding the fair share is 50%. This happens because inevitably, a miner's profit is very close to its fair share since the honest strategy is followed. Therefore, the miner's rewards will be slightly less or greater than the expected average for different simulations. In the region of parameter space, very close to the region where honest policy is the most profitable, we see that the probability increases around 70%. While we move a little further away from the region of honest policy, we see that the probability increases rapidly to 100%. Overall, regions in parameter space that are incentive-compatible with selfish mining strategies are larger than regions that are incentive-compatible with honest policy. Therefore, this can cause miners to merge in order to reach the desired threshold in terms of computing power, which will allow them to apply a selfish strategy to increase their revenue.

5.5 The Risk Safety Property

In Chapter 3, we discussed about the Risk Safety property that basic selfish mining and conservative stubborn variations impose. Recall that Risk Safety property specifies the maximum risk to which Alice (selfish miner) is susceptible, at any circumstance.

In order to verify these properties of basic selfish mining and conservative variations, we designed two property queries shown in 5.6 and 5.7 respectively. From the state machines of Chapter 3, it is known that Alice loses the race when she performs the action "restart". This appears only when Alice is in a state with lead equal to zero or equal to the negative of the trail stubbornness value. The above clarifications will help in understanding the queries.

```
A[](( 
        (!Alice.lead && !Alice.fork && !Alice.trail
          && Alice.Mine && Alice.lengthDiffIs() == 0
    )
    imply (Alice.risk() <= 1)
   )
```
(5.6)

Query 5.6 verifies the RS_1 property of the basic selfish mining. On the other hand, query 5.7 verifies the RS_2 property of the conservative stubborn variations. Moreover, it verifies that the combination of conservative variants, *i.e.,* L_SF_S conservative stubborn strategy, also satisfies the property above. Finally, after verifying both queries, they are satisfied in our model; thus, our claims about the Risk Safety property are correct.

```
A[](((
        (Alice.lead && Alice.slead && !Alice.trail &&
          (!Alice.fork || Alice.sfork))
        || (Alice.fork && Alice.sfork &&
          (!Alice.lead || Alice.slead) && !Alice.trail)
      )
      && Alice.Mine && (Alice.lengthDiffIs() == 0
        || Alice.lengthDiffIs() == -trail_len)
    )
    imply (Alice.risk() <= 2)
   )
```
(5.7)

Chapter 6

Related Work

Contents

6.1 Overview

The first known discussion about gaining higher payoffs by withholding new blocks and selectively postponing their publication, *i.e.,* selfish mining, took place in one of Bitcoin's forums [3]. Since then, researchers have studied this topic extensively from many aspects, and there are many proposed countermeasures against it. Besides, bitcoin has attracted colossal interest since its inception in 2008. As a consequence, the studies focused not only on selfish mining but also on other possible attacks. Some of them used UPPAAL to conduct their research. Next, we review some of the above-addressed topics.

6.2 Selfish Mining and Countermeasures

Selfish mining has shown tremendous research interest since the first proposed strategy in [18]. In parallel with the work in [24], the work in [27] was conducted. The latter studied optimal selfish mining strategies for any parametrization of α and γ, according to MDP (Markov Decision Processes). They also stated that under propagation delays, the profit threshold, that is, the minimum value of parameter α required for profitable selfish

mining, is 0. Moreover, they explained significant weaknesses of the uniform tie-breaking countermeasure proposed in [18].

Additional work in selfish mining appears in [20]. In that work, a study of selfish mining [18] is pursued when propagation delays are taken into account. A significant observation is that the value of parameter γ can be non-zero only if there is variability in the propagation delay of different nodes. Furthermore, they demonstrated that it is possible to detect block withholding behavior similar to selfish mining by monitoring the production rate of stale blocks.

Many approaches to mitigate selfish mining were also proposed. In [21], a countermeasure called Freshness Preferred was designed. This countermeasure tries to decrease the profitability of selfish mining by using unforgeable timestamps. More specifically, it indicates that if a miner receives two blocks within w seconds and both blocks belong to forks of equal height, then the miner accepts the block with the most recent valid timestamp rather than the one that arrived first. Another approach to alleviate selfish mining, which suggests the introduction of expected transaction confirmation height and block publishing height, appears in [26].

Bitcoin is not incentive compatible since miners deviate from honest mining to gain more than their fair share of the rewards. In [25], a fair blockchain, called FruitChain, is proposed, which is proved to be approximately fair. Therefore, this disincentivizes selfish mining. To achieve fairness, except for blocks, it introduces fruits (mined in parallel with blocks) which hang from blocks of the blockchain. Also, it obliterates the need for mining pools, which are somehow destroying the distributed nature of Bitcoin, by decreasing the variance of mining rewards. Another alternative to prevent selfish mining is discussed in [28].

6.3 UPPAAL

UPPAAL was previously used for modeling and verifying Bitcoin properties. More specifically, the statistical model checker was used to study the probability of successfully deploying specific attacks.

The first study in [13] focused on the known attack of double-spending. This paper presented an abstract model of the Bitcoin protocol with only the essential characteristics

of the protocol implemented in UPPAAL SMC. The aim was to investigate the probability of successful double-spending in an environment consisting of honest and dishonest participants. As we know, Bitcoin is used to process transactions, *e.g.,* buying various goods and services. When the payer wants to proceed with a transaction, he will create a transaction with a transfer to the seller's account. This transaction will be visible to the seller once it is included in a block of the blockchain's longest chain. Although the seller sees the transaction in the longest chain, he still waits until the longest chain extends a few more blocks. According to the classic bitcoin client, this is known as the confirmation depth, which is set to 6 blocks. Sellers who accept bitcoins as payment can and should set their threshold to how many blocks are required until transactions are considered confirmed. Delaying the acceptance of a transaction ensures, to some extent, that a side fork will not replace the longest chain, and hence the transaction will not be revoked. A successful double-spending attacker will force bitcoin nodes to believe that a ledger without his previous payment is the correct one, even though the attacker received the good or service. To achieve this, the malicious miner builds a secret side chain in order to reverse payments.

Finally, the second and last study that we will address is presented in [19]. Bitcoin has a block limit of 1 megabyte, limiting the number of confirmations of transactions per second. That led to the development of many alternatives, such as BTU, which allows flexible block size. BTU suggested using a type of majority attack to force other Bitcoin miners to adopt it [2]. That work modeled the attack mentioned above in UPPAAL SMC and analyzed the time it would take for such an attack to succeed and the success probability according to many strategies, depending on the attackers' computing power and the confirmation depth.

Chapter 7

Conclusion

Contents

7.1 Overview

In this work, we introduced a new family of selfish mining inspired by stubborn mining [24]. The new conservative stubborn mining strategies alternate their behavior between basic selfish mining and stubborn mining depending on the blockchain state. This allows conservative stubborn mining to outperform the existing strategies in a wide range of parametrizations of α (selfish miners' fraction of network's computation power) and γ (fraction of honest miners which are well connected to selfish miners). Overall, it manages to improve efficiency compared to stubborn mining up to 5% in terms of revenue. Our study focused on selfish mining in the absence of propagation delays and other coalitions of selfish miners.

In Chapter 3, we began by presenting how we modeled selfish mining strategies as state machines, and we described the basic selfish mining [18] and stubborn mining [24]. Next, we introduced the Safe-Lead and Safe-Equal-Fork conservative stubborn mining strategies and a straight-forward way to combine every variation. In Chapter 4, we described our implementation on UPPAAL STRATEGO which allowed us later, in Chapter 5, to evaluate the newly introduced variations. Our work's results reinforce the concerns about selfish mining, even though, to the extent of our knowledge, a selfish mining attack

on bitcoin was not deployed successfully yet. Therefore, any attempt to carry out this attack must be detected and tackled effectively. Besides, by doing an overview of the related work, it is clear that there have been studied many countermeasures [21, 26] and variations [25, 28] in order to mitigate selfish mining.

7.2 Future Work

As future work, various parameters that were not included in this thesis and their impact on the efficiency of existing strategies can be studied. More specifically, the presence of many coalitions of selfish miners has not yet been studied. In the presence of multiple coalitions of selfish miners, selfish mining may be disincentivized as it may no longer be profitable. Moreover, in order to retain increased profitability, malicious coalitions may decide that it is more beneficial to merge their power. Therefore, our model can be extended to include and study the effects of multiple distinct selfish miners.

Another parameter that we did not consider in our model is the propagation delay of information in the network, *i.e.,* block dissemination time. Thus, our model can be extended to take into consideration propagation delays instead of the parameter γ. For this purpose, a study on the current propagation delay of bitcoin will be required to describe it as probability distributions.

Selfish mining, as studied in [24], can be combined with eclipse attacks. During an eclipse attack, the attacker tries to compromise every incoming and outgoing connection of a peer. Therefore, the victim is isolated from the rest of the network and unable to view the ledger's current state. As a result, the attacker can filter the victim's view of the blockchain in his favor. An extension of our model, which will study the combination of eclipse attacks with the new extended strategy space of selfish mining, will be interesting.

Furthermore, someone may have a more in-depth look at a dynamic Trail stubborn strategy which may outperform our extended strategy space for some values of the parameter space. In contrast with the traditional family of Trail stubborn strategies, this strategy will dynamically change its trail stubbornness to form a more efficient Trail stubborn variation. For instance, a simple rule will change trail stubbornness to x_1 if risk is below y, otherwise to x_2. Recall that trail stubbornness is the number of blocks allowed to fall behind instead of adopting the public chain and, therefore, restarting a new cycle

of selfish mining.

Finally, some of the defenses presented in Chapter 6, such as FruitChains [25] (a fair blockchain), Freshness Preferred [21], and Zeroblock [28] can be modeled and verified in UPPAAL.

Bibliography

[1] Bitcoin - Open source P2P money. https://bitcoin.org/en/. (Accessed on 12/23/2020).

[2] Bitcoin unlimited miners may be preparing a 51% attack on bitcoin. https://bitcoinmagazine.com/articles/bitcoin-unlimited-miners-may-be-preparing-51-attack-bitcoin. (Accessed on 12/24/2020).

[3] Mining cartel attack. https://bitcointalk.org/index.php?topic=2227.msg30083#msg30083. (Accessed on 11/22/2020).

[4] UPPAAL. https://uppaal.org/. (Accessed on 12/23/2020).

[5] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science, LICS 1990*, pages 414–425. IEEE, 1990.

[6] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for probabilistic real-time systems (extended abstract). In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming, ICALP 1991*, volume 510 of *Lecture Notes in Computer Science*, pages 115–126. Springer, 1991.

[7] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[8] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 2014.

[9] A. Back. Hashcash - a denial of service counter-measure, 2002.

[10] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *Proceedings of the International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[11] P. E. Bulychev, A. David, K. G. Larsen, M. Mikucionis, D. B. Poulsen, A. Legay, and Z. Wang. UPPAAL-SMC: statistical model checking for priced timed automata. In *Proceedings of the 10th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2012*, volume 85 of *EPTCS*, pages 1–16, 2012.

[12] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proceedings of the 16th International Conference on Concurrency Theory, CONCUR 2005*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.

[13] K. Chaudhary, A. Fehnker, J. van de Pol, and M. Stoelinga. Modeling and verification of the bitcoin protocol. In *Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS 2015*, volume 196 of *EPTCS*, pages 46–60, 2015.

[14] A. David, P. G. Jensen, K. G. Larsen, A. Legay, D. Lime, M. G. Sørensen, and J. H. Taankvist. On time with minimal expected cost! In *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis, ATVA 2014*, volume 8837 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2014.

[15] A. David, P. G. Jensen, K. G. Larsen, M. Mikucionis, and J. H. Taankvist. Uppaal stratego. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer, 2015.

[16] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.*, 17(4):397–415, 2015.

[17] C. Decker and R. Wattenhofer. Information propagation in the bitcoin network. In *Proceedings of the 13th IEEE International Conference on Peer-to-Peer Computing, P2P 2013*, pages 1–10. IEEE, 2013.

[18] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security, FC 2014*, volume 8437 of *Lecture Notes in Computer Science*, pages 436–454. Springer, 2014.

[19] A. Fehnker and K. Chaudhary. Twenty percent and a few days - optimising a bitcoin majority attack. In *Proceedings of the 10th International Symposium on NASA Formal Methods, NFM 2018*, volume 10811 of *Lecture Notes in Computer Science*, pages 157–163. Springer, 2018.

[20] J. Göbel, H. P. Keeler, A. E. Krzesinski, and P. G. Taylor. Bitcoin blockchain dynamics: The selfish-mine strategy in the presence of propagation delay. *Perform. Evaluation*, 104:23–41, 2016.

[21] E. Heilman. One weird trick to stop selfish miners: Fresh bitcoins, A solution for the honest miner (poster abstract). In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security, FC 2014*, volume 8438 of *Lecture Notes in Computer Science*, pages 161–162. Springer, 2014.

[22] A. Judmayer, N. Stifter, K. Krombholz, and E. R. Weippl. *Blocks and Chains: Introduction to Bitcoin, Cryptocurrencies, and Their Consensus Mechanisms*. Morgan & Claypool Publishers, 2017.

[23] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[24] K. Nayak, S. Kumar, A. Miller, and E. Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy, EuroS&P 2016*, pages 305–320. IEEE, 2016.

[25] R. Pass and E. Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 315–324. ACM, 2017.

[26] M. Saad, L. Njilla, C. A. Kamhoua, and A. Mohaisen. Countering selfish mining in blockchains. In *Proceedings of the International Conference on Computing, Networking and Communications, ICNC 2019*, pages 360–364. IEEE, 2019.

[27] A. Sapirshtein, Y. Sompolinsky, and A. Zohar. Optimal selfish mining strategies in bitcoin. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security, FC 2016*, volume 9603 of *Lecture Notes in Computer Science*, pages 515–532. Springer, 2016.

[28] S. Solat and M. Potop-Butucaru. Brief announcement: Zeroblock: Timestamp-free prevention of block-withholding attack in bitcoin. In *Proceedings of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2017*, volume 10616 of *Lecture Notes in Computer Science*, pages 356–360. Springer, 2017.

Appendix A

In this appendix, we quote the code of UPPAAL STRATEGO's implementation. More specifically, we provide in Listing A.1, the global declarations of the model, in Listing A.2, the local declarations of selfish miners automaton, in Listing A.3, the local declarations of strategy automaton, in Listing A.4, the local declarations of honest miners automaton, and in Listing A.5, the system declarations of the model.

```
1  // Global declarations.
2
3  /* Necessary for test cases. Do not change.*/
4  int __reach__ = 0;
5  int __single__ = 0;
6  /* Necessary for test cases*/
7
8  // Change Block limit to allow more blocks to be mined.
9  const int BLOCKLIM = 1000;
10
11 // Do not change anything bellow!
12 const int HLIM = 2;
13 const int SLIM = 1;
14
15
16 typedef int[-1, BLOCKLIM] BlockID;
17 typedef int[0, HLIM] HMinerID;
18 typedef int[0,SLIM] SMinerID;
19
20
21 typedef struct{
22 BlockID blockID; // Block id
23 BlockID prevID;  // Previous block id
24 HMinerID hMiner; // Honest miner id (if it is mined by honest)
25 SMinerID sMiner; // Selfish miner id (if it is mined by selfish)
26 int[0,BLOCKLIM] length; // Distance from genesis block.
27 } Block;
28
29
30 // It replaces the hash value of the blocks with a unique number which is
31 // incremented every time a miner mines a new block
32 int [0,BLOCKLIM] blockHash = 1;
33
34 // This channel will be used to publish blocks.
35 broadcast chan newBlock;
36
37 // It is used when we are broadcasting blocks for temporary storage.
38 meta Block tempBlock;
39
40 // It is used to indicate the completion of the simulation. All automatons will
41 // transit to the Finish state.
```

```
42 bool outOfSpace = false;
43
44 // These variables should not become true (Not part of the state).
45 meta bool nextNotFound = false;
46 meta bool prevNotFound = false;
47 meta bool wrongLength = false;
48
49 // Strategies declared from controller
50 bool Lead[SLIM], Slead[SLIM], Fork[SLIM], Sfork[SLIM], Trail[SLIM];
51 bool Selfish[SLIM], strategyOn[SLIM];
```

Listing A.1: Global declarations UPPAAL STRATEGO.

```
1  /** Selfish miner's local declarations.
2   *
3   *  This automaton implements the operation of a selfish miner with various
4   *  options for strategies. Strategy will be chosen arbitrarily.
5   *
6   *  @param id Selfish miners coalition id.
7   *  @param Tlen Trail stubborness number.
8   *  @param rate_a Selfish miner coalition's hash rate of the entire network.
9   */
10
11 int privateBranchLen;
12 bool isBehind;
13 // indicates whether we are on safe zero state (risk=1)
14 bool safeZero = false;
15
16 // This array will maintain all mined blocks.
17 Block chain[BLOCKLIM+1];
18
19 // Keep track of the head of the 2 forks.
20 int publicHead;
21 int privateHead;
22
23 // Index of last public block from the selfish blocks that this miner mined.
24 int indexLastPublic;
25
26 double finalRevenue = 0.0;
27
28 // These variables will be initialized arbitrarily after strategy transitions.
29 bool lead, slead, fork, sfork, trail, selfish;
30
31
32 Block b;
33 // Necessary for test cases
34 int[1, SLIM] identity = id;
35 int tlen = Tlen;
36 // Necessary for test cases
37
38 /*****************************************************************************/
39 /** Check if strategies are initiliazed.
40  *
41  *  @return It returns true if so, otherwise false.
42  */
43 bool stratsReady(){
```

```
44      int i;
45      for (i = 0; i < SLIM; i++){
46          if (!strategyOn[i]){
47              return false;
48          }
49      }
50
51      return true;
52  }
53
54  /** Find the first free slot in chain array.
55   *
56   *  @return It returns the index of the first free slot in the chain array.
57   *          It returns -1 if there is no free slot.
58   */
59  int getChainFreeIndex(){
60      int i, index = -1;
61      for (i = 0;i < BLOCKLIM + 1; i++){
62          // First free slot.
63          if(chain[i].blockID == 0 && chain[i].prevID == 0 ){
64             index = i;
65             i = BLOCKLIM;
66          }
67      }
68      return index;
69  }
70
71  /** Find the index of the given block in the chain.
72   *
73   *  @param id Block's id to search for its index.
74   *  @return It returns the index of the given block in the chain.
75   *          It returns -1 if there is no such block.
76   */
77  int getBlockInChainIndex(BlockID bid){
78      int i, index = -1;
79
80      for (i = 0; i < BLOCKLIM + 1; i++){
81          if (chain[i].blockID == bid){
82              index = i;
83              i = BLOCKLIM;
84          }
85      }
86      return index;
87  }
88
89  /** Find and return the risk of the selfish miner.
90   *  Risk is the number of blocks of the selfish miner that are not yet included
91   *  in the public chain.
92   *
93   *  @return It returns the risk.
94   */
95  int risk(){
96      int index, index2;
97
98      index = publicHead;
99
100     // find last common block of public and private chains.
```

```
101     while (chain[index].blockID != 0){
102
103         index2 = privateHead;
104         while (chain[index2].blockID != 0){
105             if (chain[index2].blockID == chain[index].blockID){
106                 return chain[privateHead].length - chain[index2].length;
107             }
108
109             index2 = getBlockInChainIndex(chain[index2].prevID);
110         }
111
112         index = getBlockInChainIndex(chain[index].prevID);
113     }
114
115     return chain[privateHead].length;
116 }
117
118 /** Check whether the previous block of the head of the public chain was mined
119  *  from this selfish miners coalition.
120  *
121  *  @return It returns true if previous block of the head of the public chain
122  *          was mined from this selfish miner coalition.
123  */
124 bool isControlledMined(){
125     int index;
126     index = getBlockInChainIndex(chain[publicHead].prevID);
127
128     return (chain[index].sMiner == id) ? true : false;
129 }
130
131 /** Find the block which is after a specific block in the blockchain of
132  *  the selfish miners.
133  *
134  *  @param bid Block's id to search for its next block.
135  *  @return It returns the index of the next block.
136  *          It returns -1 if there is no such block.
137  */
138 int getNextBlockIndex(BlockID bid){
139     int i, index = -1;
140
141     for (i = 0; i < BLOCKLIM + 1; i++){
142         // Must be mined from this selfish miner.
143         // There might be a lot of blocks with the same prevID.
144         // In other words a block might have more than one children.
145         if (chain[i].prevID == bid && chain[i].sMiner == id){
146             index = i;
147             i = BLOCKLIM;
148         }
149     }
150     return index;
151 }
152
153 /** Find the relative renenue of selfish miners.
154  *
155  *  Relative revenue = #selfish blocks / (#selfish blocks + #others blocks)
156  *
157  *  @return It returns the relative revenue of selfish miners at the time it was
```

```
158  *          called, multiplied by 10000 and truncated.
159  */
160 double relativeRevenue(){
161     double selfish, others, revenueDouble;
162     int index;
163     int revenueInt;
164     selfish = 0;
165     others = 0;
166 
167     index = publicHead;
168     if (chain[publicHead].length <= chain[privateHead].length){
169         index = privateHead;
170     }
171     // initialize blockchain with genesis block
172     /*chain[0].blockID = 0;
173     chain[0].prevID = -1;
174     chain[0].huMiner = 0;
175     chain[0].hcMiner = 0;
176     chain[0].sMiner = 0;
177     chain[0].length = 0;*/
178     while (chain[index].blockID != 0){
179         if (chain[index].sMiner == id){
180             selfish = selfish + 1;
181         }
182         else{
183             others = others + 1;
184         }
185 
186         index = getBlockInChainIndex(chain[index].prevID);
187     }
188 
189     if (selfish + others == 0){
190         return 0;
191     }
192 
193     // by passing double precision problems with verification
194     revenueDouble = selfish/(selfish + others);
195     //revenueInt = fint(revenueDouble*10000);
196     return revenueDouble;
197     //return revenueInt;
198 }
199 
200 /** Find the number of selfish miners' blocks.
201  *
202  *  @return It returns the number of selfish miners' blocks.
203  */
204 int selfishBlocksInLongestChain(){
205     int index, selfish;
206 
207     index = publicHead;
208     // if chain is full and there is a tie in length of private and public
209     // blockchains then give advantage to public blockchain.
210     if (chain[publicHead].length < chain[privateHead].length){
211         index = privateHead;
212     }
213 
214     while (chain[index].blockID != 0){
```

```
215         if (chain[index].sMiner == id){
216             selfish = selfish + 1;
217         }
218 
219         index = getBlockInChainIndex(chain[index].prevID);
220     }
221     return selfish;
222 }
223 
224 /** Find the number of blocks mined from others.
225  *
226  *  @return It returns the number of blocks mined from others.
227  */
228 int otherBlocksInLongestChain(){
229     int index, selfish, other;
230 
231     index = publicHead;
232     if (chain[publicHead].length <= chain[privateHead].length){
233         index = privateHead;
234     }
235     other = chain[index].length;
236 
237     while (chain[index].blockID != 0){
238         if (chain[index].sMiner == id){
239             selfish = selfish + 1;
240         }
241 
242         index = getBlockInChainIndex(chain[index].prevID);
243     }
244     return other-selfish;
245 }
246 /*****************************************************************************/
247 
248 
249 /** Initialize blockchain with the genesis block.
250  *
251  *  This function is also doing some initializations on local variables.
252  */
253 void initialize(){
254     // Initialize blockchain with genesis block.
255     chain[0].blockID = 0;
256     chain[0].prevID = -1;
257     chain[0].hMiner = 0;
258     chain[0].sMiner = 0;
259     chain[0].length = 0;
260 
261     // Private and public chains start from the genesis block.
262     publicHead = 0;
263     privateHead = 0;
264 
265     indexLastPublic = 0;
266 
267     // initialize strategy which was decided from Strategy automaton.
268     // (the one belongs to this Miner)
269     lead = Lead[id-1];
270     slead = Slead[id-1];
271     fork = Fork[id-1];
```

```
272     sfork = Sfork[id-1];
273     trail = Trail[id-1];
274     selfish = Selfish[id-1];
275 }
276 
277 /** Find the length difference of the 2 forks (public and private).
278  *
279  *  @param selfish Indicates if the previous found block was selfish.
280  *  @return It returns the length difference of the 2 forks before the last
281  *          mined block.
282  */
283 int lengthDiffWas(bool selfish){
284     int publicLenWas;
285     int privateLenWas;
286 
287     if (selfish){
288         publicLenWas = chain[publicHead].length;
289         privateLenWas = chain[privateHead].length - 1;
290     }
291     else{
292         publicLenWas = chain[publicHead].length - 1;
293         privateLenWas = chain[privateHead].length;
294     }
295 
296     return privateLenWas - publicLenWas;
297 }
298 
299 /** Find the length difference of the 2 forks (public and private).
300  *
301  *  @return It returns the current length difference of the 2 forks
302  */
303 int lengthDiffIs(){
304     return chain[privateHead].length - chain[publicHead].length;
305 }
306 
307 /** Add a new block to the chain.
308 */
309 void createAndAddPrivateBlock(){
310     int index;
311     //Block b;
312 
313     index = getChainFreeIndex();
314     if (index != -1){
315         b.blockID = blockHash;
316         b.prevID = chain[privateHead].blockID;
317         b.hMiner = 0;
318         b.sMiner = id;
319 
320         b.length = chain[privateHead].length + 1;
321 
322         chain[index] = b;
323         if (blockHash == BLOCKLIM){
324             outOfSpace = true;
325         }
326         else{
327             blockHash++;
328         }
```

```
329
330         privateHead = index;
331     }
332     else{
333         outOfSpace = true;
334     }
335 }
336
337 /** Check whether all blocks which the selfish miners mined have been published
338  *
339  *  @return It returns true if all private blocks have been published otherwise
340  *          false.
341  */
342 bool allBlocksPublic(){
343     return (indexLastPublic == privateHead) ? true : false;
344 }
345
346 /** This function will publish the first unpublished private block.
347  *
348  *  Automatons are responsible to copy from broadcast tempBlock the broadcasted
349  *  block.
350  */
351 void publishFirstUnpublished(){
352     int nextIndex, currID;
353
354     if (!allBlocksPublic()){
355         currID = chain[indexLastPublic].blockID;
356
357         nextIndex = getNextBlockIndex(currID);
358
359         if (nextIndex != -1){
360             tempBlock = chain[nextIndex];
361
362             indexLastPublic = nextIndex;
363
364             if (chain[publicHead].length < chain[privateHead].length &&
365                     allBlocksPublic()){
366                 publicHead = privateHead;
367             }
368         }
369         else{
370             nextNotFound = true;
371         }
372     }
373 }
374
375 /** This function is responsible to add a block that was broadcasted from
376  *  newBlock broadcast channel.
377  */
378 void addPublicBlock(){
379     int index, prevIndex, headLen;
380     //Block b;
381
382     b = tempBlock;
383     prevIndex = getBlockInChainIndex(b.prevID);
384
385     // Previous block does not exist, something is wrong.
```

```
386     if (prevIndex == -1){
387         prevNotFound = true;
388         return;
389     }
390 
391     if (chain[prevIndex].length + 1 != b.length){
392         b.length = chain[prevIndex].length + 1;
393         wrongLength = true;
394     }
395 
396     index = getChainFreeIndex();
397     if (index != -1){
398         chain[index] = b;
399 
400         // Check if the new block forms a longer chain.
401         headLen = chain[publicHead].length;
402         if (headLen < b.length){
403             publicHead = index;
404         }
405     }
406     else{
407         outOfSpace = true;
408     }
409 }
410 
411 /** Restart private fork.
412  *
413  *  This function will point the head of the private fork to the head of the
414  *  public fork.
415  *
416 */
417 void restartFork(){
418     privateHead = publicHead;
419     indexLastPublic = publicHead;
420 }
421 
422 
423 
424 /** This function is responsible to add a block that was broadcasted from
425  *  newBlock broadcast channel to the chain array.
426  *
427  *  It can give priority to the latest block in a case of a length tie. This is
428  *  used to differentiate honest controlled miners from honest uncontrolled.
429  *
430  *  @param latestPriority In case of chain length tie latest block wins.
431  */
432 void processNewBlock(bool latestPriority){
433     int index, prevIndex, headLen;
434     //Block b;
435 
436     b = tempBlock;
437     prevIndex = getBlockInChainIndex(b.prevID);
438 
439     // Previous block does not exist, something is wrong.
440     if (prevIndex == -1){
441         prevNotFound = true;
442         return;
```

```
443     }
444
445     if (chain[prevIndex].length + 1 != b.length){
446         b.length = chain[prevIndex].length + 1;
447         wrongLength = true;
448     }
449
450     index = getChainFreeIndex();
451     if (index != -1){
452         chain[index] = b;
453
454         // Check if the new block forms a longer chain.
455         headLen = chain[publicHead].length;
456         if ((headLen == b.length && latestPriority) || headLen < b.length){
457             publicHead = index;
458         }
459     }
460     else{
461         outOfSpace = true;
462     }
463 }
464
465 /** Add a new block to the chain and then broadcast.
466 */
467 void createAndAddNewBlock(){
468     int index;
469     //Block b;
470
471     index = getChainFreeIndex();
472     if (index != -1){
473         b.blockID = blockHash;
474         b.prevID = chain[publicHead].blockID;
475         b.hMiner = 0;
476         b.sMiner = id;
477
478         b.length = chain[publicHead].length + 1;
479
480
481         chain[index] = b;
482         if (blockHash == BLOCKLIM){
483             outOfSpace = true;
484         }
485         else{
486             blockHash++;
487         }
488
489         publicHead = index;
490
491         // Broadcast the newly mined block.
492         tempBlock = b;
493     }
494     else{
495         outOfSpace = true;
496     }
497 }
```

Listing A.2: Selfish miners automaton local declarations UPPAAL STRATEGO.

```
/** Strategies local declaration.
*
*  This automaton implements the strategy selection operation.
*
*  @param id Selfish miners coalition id.
*  @param disable Indicates the strategies that are not permitted to be
*                 enabled. Look at the table below to find out the strategy
*                 assigned per index of this array.
*/
const int[0,2] strategies[17][5] = {
                        {1, 1, 1, 1 , 1}, // LSFST
                        {1, 1, 1, 0 , 1}, // LSFT
                        {1, 1, 1, 1 , 0}, // LSFS
                        {1, 1, 1, 0 , 0}, // LSF
                        {1, 1, 0, 2 , 1}, // LST
                        {1, 1, 0, 2 , 0}, // LS
                        {1, 0, 1, 1 , 1}, // LFST
                        {1, 0, 1, 0 , 1}, // LFT
                        {1, 0, 1, 1 , 0}, // LFS
                        {1, 0, 1, 0 , 0}, // LF
                        {1, 0, 0, 2 , 1}, // LT
                        {1, 0, 0, 2 , 0}, // L
                        {0, 2, 1, 1 , 1}, // FST
                        {0, 2, 1, 0 , 1}, // FT
                        {0, 2, 1, 1 , 0}, // FS
                        {0, 2, 1, 0 , 0}, // F
                        {0, 2, 0, 2 , 1}  // T
                        };
/*****************************************************************************/
/** Check if there is a path to a valid strategy.
*
*  @param lead Indicates if lead is activated.
*  @param slead Indicates if slead is activated.
*  @param fork Indicates if fork is activated.
*  @param sfork Indicates if sfork is activated.
*  @param trail Indicates if trail is activated.
*  @return It returns true if there is a path to a valid strategy.
*/
bool validTrail(bool lead, bool slead, bool fork, bool sfork, bool trail){
    int i = 0;
    bool l = false, sl = false, f = false, sf = false, t = false;

    for (i = 0; i < 17; i++){
        l = false; sl = false; f = false; sf = false; t = false;
        if (disable[i]){
            if (strategies[i][0] == 1){
                l = true;
            }

            if (strategies[i][1] == 1){
                sl = true;
            }
            else if (strategies[i][1] == 2) {
                sl = slead;
            }
```

```
56
57              if (strategies[i][2] == 1){
58                  f = true;
59              }
60
61              if (strategies[i][3] == 1){
62                  sf = true;
63              }
64              else if (strategies[i][3] == 2) {
65                  sf = sfork;
66              }
67
68              if (strategies[i][4] == 1){
69                  t = true;
70              }
71
72              if (l == lead && sl == slead && f == fork && sf == sfork
73                      && t == trail){
74                  return false;
75              }
76          }
77      }
78
79      return true;
80  }
81
82  /** Check if there is a path to a valid strategy.
83  *
84  *  @param lead Indicates if lead is activated.
85  *  @param slead Indicates if slead is activated.
86  *  @param fork Indicates if fork is activated.
87  *  @param sfork Indicates if sfork is activated.
88  *  @return It returns true if there is a path to a valid strategy.
89  */
90  bool validFork(bool lead, bool slead, bool fork, bool sfork){
91      return validTrail(lead, slead, fork, sfork, true)
92              || validTrail(lead, slead, fork, sfork, false);
93  }
94
95  /** Check if there is a path to a valid strategy.
96  *
97  *  @param lead Indicates if lead is activated.
98  *  @param slead Indicates if slead is activated.
99  *  @return It returns true if there is a path to a valid strategy.
100 */
101 bool validLead(bool lead, bool slead){
102     return validFork(lead, slead, true, false)
103             || validFork(lead, slead, true, true)
104             || validFork(lead, slead, false, false);
105 }
```

Listing A.3: Strategy automaton local declarations UPPAAL STRATEGO.

```
1  /** Honest miner's local declarations.
2  *
3  *  This automaton implements the operation of an honest miners who are either
```

```
4  *  well connected with the selfish miners coalition or not. You can specify
5  *  this with the automaton's parameter controlled.
6  *
7  *  @param id Honest miners coalition's id.
8  *  @param rate_b Honest miner coalition's hash rate of the entire network.
9  *  @param controlled Selfish miner is well connected with this honest coalition
10 *                    if true.
11 */
12
13
14 // This array will maintain all mined blocks.
15 Block chain[BLOCKLIM+1];
16
17 int publicHead;
18
19
20 /*****************************************************************************/
21 /** Find the first free slot in chain array.
22 *
23 *  @return It returns the index of the first free slot in the chain array.
24 *          It returns -1 if there is no free slot.
25 */
26 int getChainFreeIndex(){
27     int i, index = -1;
28     for (i = 0;i < BLOCKLIM + 1; i++){
29         // First free slot.
30         if(chain[i].blockID == 0 && chain[i].prevID == 0 ){
31             index = i;
32             i = BLOCKLIM;
33         }
34     }
35     return index;
36 }
37
38 /** Find the index of the given block in the chain.
39 *
40 *  @param id Block's id to search for its index.
41 *  @return It returns the index of the given block in the chain.
42 *          It returns -1 if there is no such block.
43 */
44 int getBlockInChainIndex(BlockID bid){
45     int i, index = -1;
46
47     for (i = 0; i < BLOCKLIM + 1; i++){
48         if (chain[i].blockID == bid){
49             index = i;
50             i = BLOCKLIM;
51         }
52     }
53     return index;
54 }
55 /*****************************************************************************/
56
57
58 /** Initialize blockchain with the genesis block.
59 *
60 *  This function also do some initializations on local variables.
```

```
61 */
62 void initialize(){
63     // Initialize blockchain with genesis block.
64     chain[0].blockID = 0;
65     chain[0].prevID = -1;
66     chain[0].hMiner = 0;
67     chain[0].sMiner = 0;
68     chain[0].length = 0;
69 
70     // Public chain starts from the genesis block.
71     publicHead = 0;
72 }
73 
74 /** This function is responsible to add a block that was broadcasted from
75 *  newBlock broadcast channel to the chain array.
76 *
77 *  It can give priority to the latest block in a case of a length tie. This is
78 *  used to differentiate honest controlled miners from honest uncontrolled.
79 *
80 *  @param latestPriority In case of chain length tie latest block wins if true
81 */
82 void processNewBlock(bool latestPriority){
83     int index, prevIndex, headLen;
84     Block b;
85 
86     b = tempBlock;
87     prevIndex = getBlockInChainIndex(b.prevID);
88 
89     // Previous block does not exist, something is wrong.
90     if (prevIndex == -1){
91         prevNotFound = true;
92         return;
93     }
94 
95     if (chain[prevIndex].length + 1 != b.length){
96         b.length = chain[prevIndex].length + 1;
97         wrongLength = true;
98     }
99 
100    index = getChainFreeIndex();
101    if (index != -1){
102        chain[index] = b;
103 
104        // Check if the new block forms a longer chain.
105        headLen = chain[publicHead].length;
106        if ((headLen == b.length && latestPriority) || headLen < b.length){
107            publicHead = index;
108        }
109    }
110    else{
111        outOfSpace = true;
112    }
113 }
114 
115 /** Add a new block to the chain and then broadcast.
116 */
117 void createAndAddNewBlock(){
```

```
118     int index;
119     Block b;
120 
121     index = getChainFreeIndex();
122     if (index != -1){
123         b.blockID = blockHash;
124         b.prevID = chain[publicHead].blockID;
125         b.hMiner = id;
126         b.sMiner = 0;
127 
128         b.length = chain[publicHead].length + 1;
129 
130         chain[index] = b;
131         if (blockHash == BLOCKLIM){
132             outOfSpace = true;
133         }
134         else{
135             blockHash++;
136         }
137 
138         publicHead = index;
139 
140         // Broadcast the new block.
141         tempBlock = b;
142     }
143     else{
144         outOfSpace = true;
145     }
146 }
```

Listing A.4: Honest miners automaton local declarations UPPAAL STRATEGO.

```
1  // Place template instantiations here.
2 
3  // Honest miners parameters
4  const bool controlled = true;
5 
6  // Selfish miner parameters
7  const int trail_len = 1;
8  // Mining power parameters
9  const int alfa = 40;
10 const double a = 0.4;
11 const int goodBob = 54;
12 const int badBob = 6;
13 
14 // Disable strategies.
15 const bool disable[17] = {
16                         false, // LSFST
17                         false, // LSFT
18                         false, // LSFS
19                         false, // LSF
20                         false, // LST
21                         false, // LS
22                         false, // LFST
23                         false, // LFT
24                         false, // LFS
```

```
                        false, // LF
                        false, // LT
                        false, // L
                        false, // FST
                        false, // FT
                        false, // FS
                        false, // F
                        false // T
                        };
AliceStrat = Strategy(1, disable);
Alice = SMiners(1, trail_len, alfa);
GoodBob = HMiners(1, goodBob, !controlled);
BadBob = HMiners(2, badBob, controlled);

// List one or more processes to be composed into a system.
system AliceStrat,Alice,GoodBob,BadBob;

/** TEST_FILENAME test_ */
/** TEST_FILEEXT .java */
/** TEST_PREFIX
package test;
import test.App;

public class Test {
    public static void main(String[] args){
*/
/** TEST_POSTFIX
    }
}
*/
```

Listing A.5: System declarations UPPAAL STRATEGO.

Appendix B

In this appendix, we present UPPAAL SMC's automata. More specifically, we provide in Figure B.1, the automaton of selfish miners, and in Figure B.2, the automaton of honest miners.

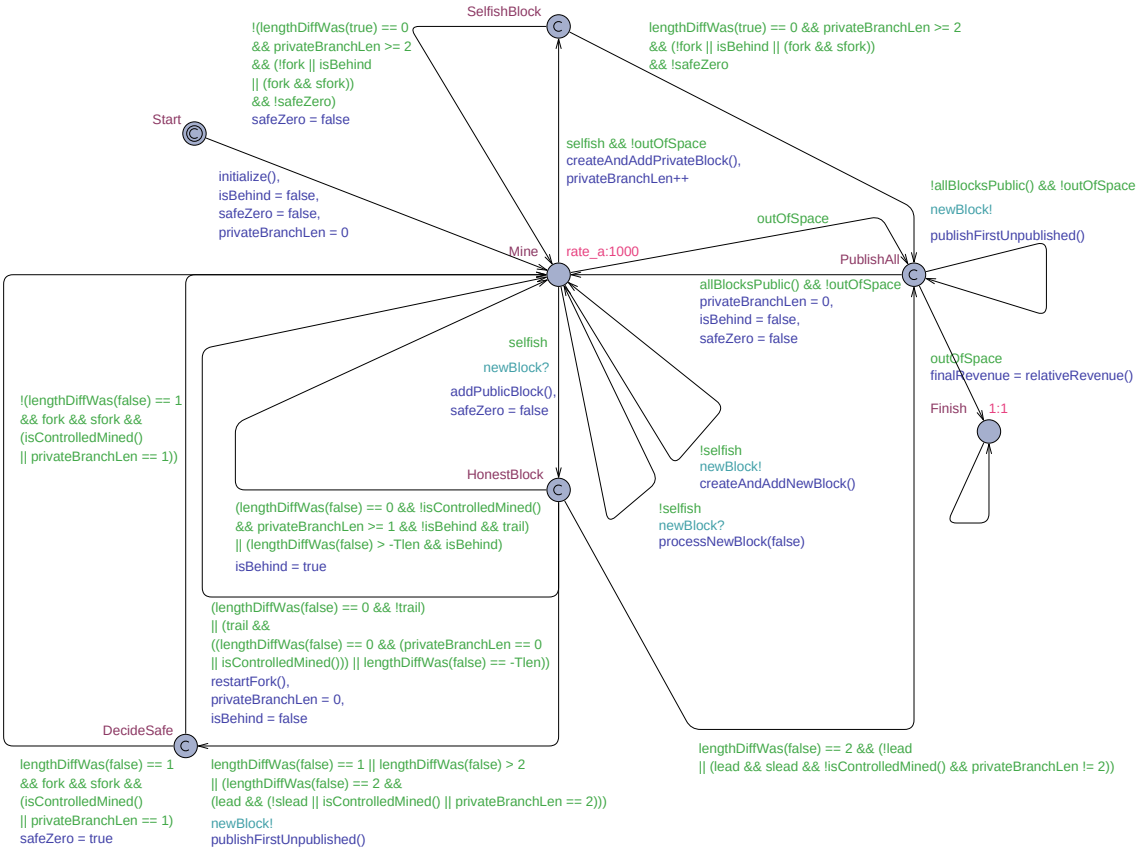


Figure B.1: Selfish miners automaton UPPAAL SMC.

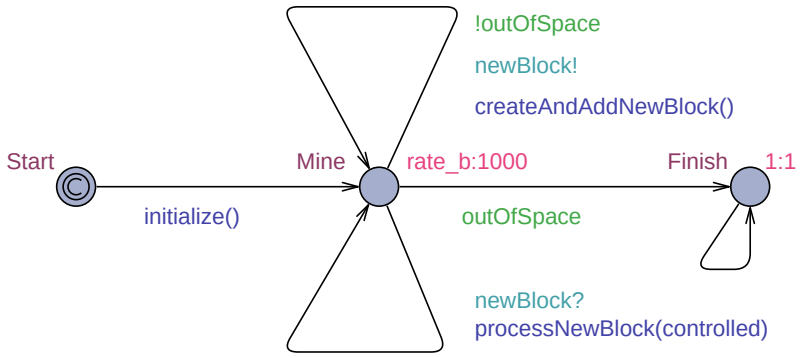


Figure B.2: Honest miners automaton UPPAAL SMC.

Appendix C

In this appendix, we quote the java implementation of the testing code used by test cases. More specifically, we provide in Listing C.1, the enum class of actions, in Listing C.2, the Block class which implements the block's structure used in UPPAAL, in Listing C.3, the class of selfish miner which implements the pseudocode of Algorithm 1, and in Listing C.4, the class which provides functions that are used on edges and locations of UPPAAL automata.

```
1  package test;
2
3  /**
4   * All feasible actions of a miner.
5   */
6  public enum Action {
7      NA{
8          public String toString() {
9              return "No action - NA";
10         }
11     }
12     , ALL{
13         public String toString() {
14             return "Publish all - ALL";
15         }
16     }, FIRST{
17         public String toString() {
18             return "Publish first - FIRST";
19         }
20     }, RESTART{
21         public String toString() {
22             return "Restart fork - RESTART";
23         }
24     };
25 }
```

Listing C.1: Test cases Action enum.

```
1  package test;
2  /**
3   * Implementation of a blockchain block.
4   */
5  public class Block {
6    public int blockID;
7    public int prevID;
8    public int hMinerID;
9    public int sMinerID;
10   public int length;
```

```
11
12    public Block() {
13      this(0,-1,0,0,0);
14    }
15
16    public Block(int blockID, int prevID, int hMinerID, int sMinerID,
17        int length) {
18      this.blockID = blockID;
19      this.prevID = prevID;
20      this.hMinerID = hMinerID;
21      this.sMinerID = sMinerID;
22      this.length = length;
23    }
24
25      // Overriding equals() to compare two Blocks
26      @Override
27      public boolean equals(Object o) {
28
29        if (o == null) {
30          return false;
31        }
32
33          if (!(o instanceof Block)) {
34              return false;
35          }
36
37          // typecast o to Complex so that we can compare data members
38          Block b = (Block) o;
39
40          // Compare the data members and return accordingly
41          return b.blockID == this.blockID
42              && b.prevID == this.prevID
43              && b.hMinerID == this.hMinerID
44              && b.sMinerID == this.sMinerID
45              && b.length == this.length;
46      }
47
48      /**
49       * String representation of a block.
50       */
51    public String toString(){
52      return "Block#"+this.blockID+"(prevID:"+this.prevID+",␣hMinerID:"
53          +this.hMinerID+",␣sMinerID:"+this.sMinerID+",␣length:"
54          +this.length+")";
55    }
56  }
```

Listing C.2: Test cases Block class.

```
1  package test;
2
3  import java.util.ArrayList;
4
5  /**
6   * Implementation of a selfish miner behavior (algorithm).
7   *
```

```
 * The behavior of the selfish miner depents on his strategy defined
 * by lead, fork, trail, tlen and selfish instance's members.
 */
public class SelfishMiner {
    private int publicHead;
    private int privateHead;
    private boolean isBehind;
    private boolean safeZero;
    private int privateBranchLen;
    private int indexLastPublic;

    private int id;
    private int lead;
    private int slead;
    private int fork;
    private int sfork;
    private int trail;
    private int tlen;
    private int selfish;

    private ArrayList<Block> chain = new ArrayList<>();

    private int currentID;

    private boolean lastSelfish;

    public SelfishMiner(int id, int lead, int slead, int fork, int sfork,
            int trail, int tlen, int selfish) {
        this.publicHead = 0;
        this.privateHead = 0;
        this.isBehind = false;
        this.safeZero = false;
        this.privateBranchLen = 0;
        this.indexLastPublic = 0;

        this.id = id;
        this.lead = lead;
        this.slead = slead;
        this.fork = fork;
        this.sfork = sfork;
        this.trail = trail;
        this.tlen = tlen;
        this.selfish = selfish;

        this.currentID = 1;
        // genesis block
        this.chain.add(new Block());
    }

    public void incCurrentID(){
        this.currentID++;
    }

    public boolean checkState(int publicHead, int privateHead,
            boolean isBehind, boolean safeZero, int privateBranchLen,
            int indexLastPublic) {
        // ignore when miners are acting like honest.
```

```
65          return (this.publicHead == publicHead
66                  && this.privateHead == privateHead
67                  && this.isBehind == isBehind
68                  && this.safeZero == safeZero
69                  && this.privateBranchLen == privateBranchLen
70                  && this.indexLastPublic == indexLastPublic)
71                  || this.selfish == 0;
72      }
73
74      public Block addSelfishBlock() {
75          Block b = new Block(this.currentID,
76                  this.chain.get(this.privateHead).blockID,0,
77                  this.id,chain.get(this.privateHead).length+1);
78
79          this.chain.add(b);
80          this.privateHead = this.chain.size()-1;
81          this.privateBranchLen++;
82          this.currentID++;
83
84          this.lastSelfish = true;
85          return b;
86      }
87
88      public void addHonestBlock(Block b) {
89          this.chain.add(b);
90          this.publicHead = this.chain.size()-1;
91
92          this.lastSelfish = false;
93      }
94
95
96      /********************/
97      /*      ALGORITHM     */
98      /********************/
99
100
101     /**
102      * Algorithm's decisions.
103      *
104      * @return It returns the expected action
105      */
106     public Action algorithmExpectedAction() {
107         if (this.lastSelfish) {
108             return this.onSelfishMiner();
109         }
110         else {
111             return this.onOthers();
112         }
113     }
114
115     /**
116      * on My Miners found a block.
117      *
118      * @return It returns the expected action
119      */
120     private Action onSelfishMiner() {
121         int d = this.lengthDiffWas();
```

```
122
123         if (d == 0 && this.privateBranchLen >= 2 && !this.isBehind) {
124             return SM1();
125         }
126         else if (d == 0 && this.isBehind) {
127             this.publishAll();
128             this.privateBranchLen = 0;
129             this.isBehind = false;
130             return Action.ALL;
131         }
132         else {
133             return Action.NA;
134         }
135     }
136
137     /**
138      * on Others found a block.
139      *
140      * @return It returns the expected action
141      */
142     private Action onOthers() {
143         int d = this.lengthDiffWas();
144
145         if (d > -this.tlen && this.isBehind) {
146             return Action.NA;
147         }
148         else if ((d == 0 && this.privateBranchLen == 0) || d == -tlen){
149             this.restart();
150             this.privateBranchLen = 0;
151             this.isBehind = false;
152             return Action.RESTART;
153         }
154         else if (d == 0 && this.privateBranchLen >= 1 && !this.isBehind) {
155             this.safeZero = false;
156             return SM2();
157         }
158         else if (d == 1) {
159             SM3();
160             this.publishFirst();
161             return Action.FIRST;
162         }
163         else if (d == 2) {
164             return SM4();
165         }
166         else {
167             this.publishFirst();
168             return Action.FIRST;
169         }
170     }
171
172     /**
173      * SM1
174      * if (fork and not sfork) or safeZero then
175      *     safeZero <- false
176      * else
177      *     publish all of the private chain
178      *     privateBranchLen <- 0
```

```
179         *
180         * @return It returns the expected action
181         */
182        private Action SM1() {
183            if ((this.fork == 1 && this.sfork != 1) || this.safeZero) {
184                this.safeZero = false;
185                return Action.NA;
186            }
187            else {
188                this.publishAll();
189                this.privateBranchLen = 0;
190                return Action.ALL;
191            }
192        }
193
194        /**
195         * SM2
196         * if trail and not controlled() then
197         *     isBehind <- true
198         *     do nothing
199         * else
200         *     private chain <- public chain
201         *     privateBranchLen <- 0
202         *
203         * @return It returns the expected action
204         */
205        private Action SM2() {
206            if (this.trail == 1 && !(controlled())) {
207                this.isBehind = true;
208                return Action.NA;
209            }
210            else {
211                this.restart();
212                this.privateBranchLen = 0;
213                return Action.RESTART;
214            }
215        }
216
217        /**
218            * SM3
219            */
220        private void SM3() {
221            if (this.fork == 1 && this.sfork == 1 && (controlled()
222                    || this.privateBranchLen == 1)) {
223                this.safeZero = true;
224            }
225        }
226
227        /**
228         * SM4
229         * if lead and (not slead or controlled() or privateBranchLen = 2) then
230         *     publish first unpublished block in private chain
231         * else
232         *     publish all of the private chain
233         *     privateBranchLen <- 0
234         *
235         * @return It returns the expected action
```

```
236         */
237         private Action SM4() {
238             if (this.lead == 1 && (this.slead != 1 || controlled()
239                     || this.privateBranchLen == 2)) {
240                 this.publishFirst();
241                 return Action.FIRST;
242             }
243             else {
244                 this.publishAll();
245                 this.privateBranchLen = 0;
246                 return Action.ALL;
247             }
248         }
249 
250         /**
251          * @return It returns true iff the miner who mined the honest block is well
252          *         connected with the selfish miners.
253          */
254         private boolean controlled() {
255             int id = this.chain.get(publicHead).prevID;
256 
257             for (Block b: this.chain) {
258                 if (b.blockID == id) {
259                     return b.sMinerID == this.id;
260                 }
261             }
262             return false;
263         }
264 
265         private void restart() {
266             this.privateHead = this.publicHead;
267             this.indexLastPublic = this.publicHead;
268         }
269 
270         private void publishAll() {
271             this.publicHead = this.privateHead;
272             this.indexLastPublic = this.privateHead;
273         }
274 
275         private void publishFirst() {
276             int id = this.chain.get(this.indexLastPublic).blockID;
277 
278             for (Block b: this.chain) {
279                 if (b.prevID == id && b.sMinerID == this.id) {
280                     this.indexLastPublic = this.chain.indexOf(b);
281                 }
282             }
283         }
284 
285         private int lengthDiffWas() {
286             int priv = this.chain.get(this.privateHead).length;
287             int publ = this.chain.get(this.publicHead).length;
288             int d = priv - publ;
289             if (this.lastSelfish) {
290                 d--;
291             }
292             else {
```

```
293             d++;
294         }
295         return d;
296     }
297
298
299     /********************/
300     /* GETTER FUNCTIONS */
301     /********************/
302
303
304     public int getPublicHead(){
305         return this.publicHead;
306     }
307
308     public int getPrivateHead(){
309         return this.privateHead;
310     }
311
312     public int getIsBehind(){
313         return (this.isBehind) ? 1 : 0;
314     }
315
316     public int getSafeZero(){
317         return (this.safeZero) ? 1 : 0;
318     }
319
320     public int getPrivateBranchLen(){
321         return this.privateBranchLen;
322     }
323
324     public int getIndexLastPublic(){
325         return this.indexLastPublic;
326     }
327 }
```

Listing C.3: Test cases selfish miner class.

```
1  package test;
2
3  /**
4   * Implementation of the functions returned from UPPAAL states and edges
5   * during test cases.
6   *
7   * Assertions will catch all problems of the implementation on UPPAAL if any.
8   *
9   */
10 public class App {
11
12   private static SelfishMiner sm;
13
14   public static void initialize_sm(int id, int lead, int slead, int fork,
15       int sfork, int trail, int tlen, int selfish) {
16     sm = new SelfishMiner(id,lead,slead,fork,sfork,trail,tlen,selfish);
17   }
18
```

```
19    public static void selfish_after_action(int publicHead, int privateHead,
20        int isBehind, int safeZero, int privateBranchLen,
21        int indexLastPublic) {
22      boolean isbehind = false;
23      boolean safezero = false;
24      if (isBehind == 1) {
25        isbehind = true;
26      }
27
28      if (safeZero == 1) {
29        safezero = true;
30      }
31
32      try{
33        assert sm.checkState(publicHead, privateHead, isbehind, safezero,
34            privateBranchLen, indexLastPublic) : "Unexpected state";
35      } catch (AssertionError e){
36        System.out.println("[UPPAAL] State{Public Head:"
37                  +publicHead+", Private Head:"+privateHead
38                  +", Is Behind:"+isBehind+", Safe zero:"
39                  +safeZero+", Private Branch Length:"
40                  +privateBranchLen+", Last published index:"
41                  +indexLastPublic+"}");
42        System.out.println("[ Java ] State{Public Head:"
43                  +sm.getPublicHead()+", Private Head:"
44                  +sm.getPrivateHead()+", Is Behind:"
45                  +sm.getIsBehind()+", Safe zero:"
46                  +sm.getSafeZero()+", Private Branch Length:"
47                  +sm.getPrivateBranchLen()
48                  +", Last published index:"
49                  +sm.getIndexLastPublic()+"}");
50        e.printStackTrace();
51
52        System.exit(1);
53      }
54    }
55
56    public static void new_selfish_block(int blockID, int prevID, int hMiner,
57        int sMiner, int length) {
58      Block b = new Block(blockID,prevID,hMiner,sMiner,length);
59
60      Block expected_b = sm.addSelfishBlock();
61
62      try{
63        assert b.equals(expected_b) : "Unexpected new selfish block";
64      } catch (AssertionError e){
65        System.out.println("[UPPAAL] "+b);
66        System.out.println("[ JAVA ] "+expected_b);
67        e.printStackTrace();
68
69        System.exit(1);
70      }
71    }
72
73    public static void new_honest_block(int blockID, int prevID, int hMiner,
74        int sMiner, int length) {
75      Block b = new Block(blockID,prevID,hMiner,sMiner,length);
```

```
76
77      sm.addHonestBlock(b);
78    }
79
80    public static void expect_publish_all() {
81      Action expected = sm.algorithmExpectedAction();
82      try{
83        assert Action.ALL == expected :
84          "UPPAAL Unexpected action(publish all)";
85      } catch (AssertionError e){
86        System.out.println("[UPPAAL] "+Action.ALL);
87        System.out.println("[ JAVA ] "+expected);
88        e.printStackTrace();
89
90        System.exit(1);
91      }
92    }
93
94    public static void expect_no_action() {
95      Action expected = sm.algorithmExpectedAction();
96      try {
97        assert Action.NA == expected :
98          "UPPAAL Unexpected action(no action)";
99      } catch (AssertionError e){
100       System.out.println("[UPPAAL] "+Action.NA);
101       System.out.println("[ JAVA ] "+expected);
102       e.printStackTrace();
103
104       System.exit(1);
105     }
106   }
107
108   public static void expect_restart() {
109     Action expected = sm.algorithmExpectedAction();
110     try {
111       assert Action.RESTART == expected :
112         "UPPAAL Unexpected action(restart)";
113     } catch (AssertionError e){
114       System.out.println("[UPPAAL] "+Action.RESTART);
115       System.out.println("[ JAVA ] "+expected);
116       e.printStackTrace();
117
118       System.exit(1);
119     }
120   }
121
122   public static void expect_publish_first() {
123     Action expected = sm.algorithmExpectedAction();
124     try {
125       assert Action.FIRST == expected :
126         "UPPAAL Unexpected action(publish first)";
127     } catch (AssertionError e){
128       System.out.println("[UPPAAL] "+Action.FIRST);
129       System.out.println("[ JAVA ] "+expected);
130       e.printStackTrace();
131
132       System.exit(1);
```

```
133         }
134     }
135
136     public static void inc_currentID(){
137         sm.incCurrentID();
138     }
139 }
```

Listing C.4: Test cases App class.