Thesis Dissertation

ANALYZING SELFISH MINING STRATEGIES COMBINED WITH ECLIPSED ATTACKS USING STATISTICAL MODEL CHECKING

Diomidis Michael

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2024

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Analyzing Selfish Mining Strategies combined with Eclipse Attacks using Statistical Model Checking

Diomidis Michael

Supervisor

Dr. Anna Philippou

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor’s in computer science at University of Cyprus

May 2024

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Anna Philippou for her valuable guidance, understanding and support throughout the research work, to overcome any challenges that arose and the final preparation of this thesis.

Abstract

Bitcoin is the most known digital currency also known as cryptocurrency and has had a great impact on economy since its conception in 2008. It utilizes a decentralized network of miners whose purpose is to create blocks which are connected to each other like a chain. These blocks act as part of a database and hold records of transactions of this cryptocurrency, gaining income from transaction fees in return. However, there are vulnerabilities in such peer-to-peer networks. A type of attack that can happen in this network is *"selfish mining"*, where a pool can withhold its mined pools from the public and follow a set of actions which can lead to disproportionate amounts of revenue for the pool who utilizes them. Another attack is known as *"eclipse attack"* and is used by an attacker to cut off a set of nodes from the rest of the network, by controlling all outgoing and ingoing connections, thus being able to manipulate the nodes in ways that benefit the attacker. This thesis examines the subject of multiple attackers who adopt these strategies inside the same network. The tool used for modelling and analysing the system is UPPAAL. The use of UPPAAL was crucial for this work as it provides several advantages for simulations, especially for systems like the network of Bitcoin where timing and concurrency are critical. In the scope of our thesis we explored different scenarios. Some scenarios include miners who share the same properties and compete against each other. Other scenarios have a designated favourite attacker which is superior compared to the others, in order to study those cases. During these simulations we confirm previous findings, and we additionally present our own findings. Those include the relationship of each variable with the revenue of the attacker. We further extend the research by attempting to create realistic scenarios in which each attacker is expected to gain profit when behaving selfishly, assuming they are the only attacker in the network. These scenarios unveiled the possibility of two attackers benefitting from releasing their eclipsed pools. We proposed the term "riddance" to describe the action of dismissing part of the controlled resources.

Contents

Chapter 1 **Introduction**………………………………………………………… **1**

1.1 Introduction 1

1.2 Motivation and purpose 2

1.3 Methodology 3

Chapter 2 **Related Work**……………………….................…………………….. **5**

2.1 Bitcoin 5

2.2 Selfish Mining Strategy 6

2.3 Eclipse Attacks 7

2.4 UPPAAL – SMC 7

2.5 Previous work 9

Chapter 3 **Blockchain Network Modelling** …………………………………........... **11**

3.1 Overview 11

3.2 Modelling Parameters and Assumptions 11

3.3 Selfish Mining Strategy 14

3.4 Selfish Mining Strategy involving an Eclipse Attack 15

3.5 Complete Picture of the Network 16

Chapter 4 **Implementation** ....…………………………….………… **19**

4.1 Overview 19

4.2 Global Variables and Environment 19

4.3 Honest Miners Template 21

4.4 Selfish Miners Template 23

4.5 Eclipsed Miner Template 27

4.6 Complete Picture of the Network 28

Chapter 5 **Experimental Results and Evaluation……**……………………………… **31**

5.1 Overview 31

5.2 Multiple Selfish Attackers Competition 32

5.3 Designated Superior Pool 35

5.4 Realistic Scenarios with Profitable and Damaged Attackers 43

Chapter **6** **Conclusion** **51**

6.1 Multiple Miners Conclusion 51

6.2 Realistic Application Findings 52

6.3 Future work 55

References **56**

Appendix A.. **A-1**

Appendix B.. **B-3**

Appendix C.. **C-9**

Chapter 1

Introduction

1.1 Introduction 1
1.2 Motivation and Purpose 3
1.3 Methodology 4

1.1 Introduction

The digital currency known as Bitcoin was created in 2008 by an anonymous person or group going by the name Satoshi Nakamoto [1]. What sets it apart from conventional currencies, is its ability to operate independently of a single organisation, like a bank or a government.

The fundamental component of Bitcoin is a technology known as ‘blockchain’. A network of nodes known as 'miners' are trying to solve a problem. Whoever finds a valid solution creates a 'block' which is published to the rest of the network. Other miners then verify that the block is indeed valid, the miner who provided the block earns their reward and then everyone proceeds to solve a new problem based on the new block. Each block is connected to the previous block, like a chain, hence the name 'blockchain'. This blockchain ultimately serves as a database that keeps track of every Bitcoin transaction. The decentralized nature of Bitcoin derives from this blockchain, which is distributed throughout a network of nodes rather than being kept in a single location, and it is what makes it so valuable. Because of its decentralization, Bitcoin is immune to censorship and manipulation since a single entity is unable to control it.

However, there are certain difficulties with using Bitcoin. The possibility of self-serving mining tactics and eclipse attacks is one area of concern. ‘Selfish mining’ [2] is a set of strategies employed by some Bitcoin miners to gain a disproportionate share of rewards by withholding discovered blocks from the network, strategically releasing them later. By manipulating block propagation and network latency, selfish miners attempt to outperform the rest of the network, mainly by wasting their competitors' resources, thus increasing their chances of earning rewards at the expense of other miners. Another malicious activity which may take place are eclipse attacks [4], whose purpose is to cut off a miner from the rest of the network, or even manipulate their actions in ways that benefit the attacker. In much simpler words, eclipse attacks are used to influence transactions for a specific miner/mining pool, whereas selfish mining involves miners trying to trick the network to get rewards that they are not legally entitled to.

1.2 Motivation and Purpose

The purpose of this thesis is to enhance our knowledge of selfish mining tactics in real-world applications. This study has two main goals. Firstly, it aims to clarify the dynamics and consequences of these methods in practical contexts by simulating several scenarios where multiple mining pools use selfish mining strategies, combined with eclipse attacks and the control of honest mining groups. Through this series of simulations our target is to explain the significance of variables and their effect on the revenue of the attacker. The other purpose of this thesis is to examine realistic scenarios where attackers are allowed to mine selfishly only if their properties predict profitability when using the findings of [2]. The research of these scenarios is crucial as they pose a threat to the decentralized network.

The proposal of the selfish mining strategy [2] and the implementation of eclipse attacks in peer-to-peer networks [4] initiated the research on this subject. Significant contributions have been made ever since, with the demonstration of variations of the original selfish strategy, in conjunction with an eclipse attack in [3]. In [10] the existence of two attackers was studied and in [9] the success of multiple attackers, up to ten attackers. In [5] the dominant strategies for scenarios with a single attacker are presented, while [8] extended this subject to scenarios with two attackers.

Through the simulation, analysis, and formal verification of the network with the use of UPPAAL [7], we are going to demonstrate how the presence of multiple attackers who utilize eclipsed nodes influences the behaviour of the network. The decision to work with UPPAAL and its Statistical Model Checker extension derived from the various benefits it offers, as it is suitable for modelling real-time systems and its features allow us to easily create our models, detect potential errors and formally verify properties. The availability of previous UPPAAL models used in [5] and [8] was a factor that positively contributed to this decision. Through the SMC extension we unveiled how different parameters affect the revenue of the attackers.

We also viewed scenarios in which malicious mining pools can expect to gain a disproportionate amount of revenue. These are insights that show exploitation possibilities regarding the Bitcoin's blockchain protocol.

1.3 Methodology

The methodology employed in this thesis was organised into multiple pivotal phases. First, a comprehensive study and general knowledge collection on the subject of selfish mining strategies was carried out. This entailed going over previously published works, scholarly articles, and projects related to the topic. Previous studies and initiatives, such as the UPPAAL models used in [5] and [8], which offered insightful information about the dynamics of selfish mining, were given particular consideration. To analyze the behavior and impact of selfish mining, formal methods and model checking techniques using UPPAAL were employed. This approach was chosen as it allowed us to thoroughly verify the correctness and efficiency of mining strategies by using the Statistical Model Checker extension of the tool.

Expanding on the current models, which included the selfish and honest miner templates, we added new features and scenarios that were suitable for the research. To be specific, we modified the honest miner template to count their revenue, created the template for the eclipsed miner, we edited the selfish miner template in order to control their eclipsed victims and also added some transitions to the templates in order to maintain their correctness for multiple attackers. To guarantee that the models faithfully reflected actual conditions, this process required considerable thought and modification. Several sanity checks were carried out to confirm that the simulations behaved as predicted and generated accurate results, including that the sum of all mining pools' revenue added up to 100% of the blockchain's length and that the system never entered states in which blocks were published in wrong sequence. Those tests were necessary in validating the integrity and functionality of the extended models.

With the models in place, scenarios were designed to simulate various conditions and strategies, including selfish mining techniques and eclipse attacks, alongside controlled groups representing honest mining behavior. These scenarios were then executed through simulation and formal verification, allowing for the collection of data and observations.

Following the simulation phase, the results were analyzed, evaluated, and explained in detail. This involved comparing outcomes across different scenarios, identifying patterns, and drawing conclusions for the dynamics of selfish, honest, and controlled/eclipsed mining pools based on the observed behavior of the simulated networks. Through this process, we were able to gain information about the effectiveness and implications of selfish mining strategies, contributing to a deeper understanding of their impact on the security and integrity of blockchain networks.

Chapter 2

Related Work

2.1 Bitcoin 5
2.2 Selfish Mining Strategy 6
2.3 Eclipse Attacks 7
2.4 UPPAAL – SMC 7
2.5 Previous work 9

2.1 Bitcoin

Bitcoin was introduced to the world in 2008 by Satoshi Nakamoto[1], as a digital currency which does not require a regulator, such as banks or governments. This currency works in a peer-to-peer fashion, and it utilizes the efforts of multiple computers who are in charge of validating transactions and keeping the records of the balances, using the 'Blockchain' concept. Blockchains are exactly what one would expect, a chain of blocks, where each block contains the identity of its previous block, like a pointer in a linked list structure. A blockchain network using the Proof-of-Work (PoW) consensus mechanism functions as follows. In a network there are many nodes known as miners, whose aim is to solve a complex mathematical problem. The first miner who succeeds, is able to create a valid block which, in the case of cryptocurrencies like Bitcoin, holds transactions. The block is published so that the rest of the network can verify that the block is valid, they can update their records and use the new block as the head of the chain. The head of the chain is the last published block of the longest branch in the structure of the blockchain and its identity must be included in the next block that is mined, that is why when a new block is mined, the miners restart trying to solve the problem taking into account the newly set head.

Miners of course do not participate in this process as volunteers – their aim is to find as many blocks as possible, and gain revenue through transaction fees. However, the chance of a single node finding the solution is very slim, which has led to nodes forming groups also known as mining pools, in order to expand the chance of gaining revenue. Each member of the winner pool gains a portion of the revenue which usually corresponds to the resources they have provided to the pool. By joining a mining pool, a miner expands their chances of gaining revenue, for the price of only earning a small share of what they originally would if they had mined the block by themselves.

Each mining pool can follow different strategies in order to increase their revenue. There are various approaches that a mining pool can adopt when aiming to maximize their profits. Some of the most common practices is to include picking transactions with the highest transaction fees as part of the block to increase the potential revenue. However, some other practices are not so innocent and in fact are considered attacks.

2.2 Selfish Mining Strategy

In their work “Majority is not Enough: Bitcoin Mining is Vulnerable” [2], Ittay Eyal and Emin Gün Sirer present a practice commonly referred to as Selfish Mining. A selfish mining pool would not publish a block instantly but would instead keep it private. They would

```
1 on Init
2 public chain ← publicly known blocks
3 private chain ← publicly known blocks
4 privateBranchLen ← 0
5 Mine at the head of the private chain.

6 on My pool found a block
7 Δprev ← length(private chain) − length(public chain)
8 append new block to private chain
9 privateBranchLen ← privateBranchLen + 1
10 if Δprev = 0 and privateBranchLen = 2 then (Was tie with branch of 1)
11 publish all of the private chain (Pool wins due to the lead of 1)
12 privateBranchLen ← 0
13 Mine at the new head of the private chain.

14 on Others found a block
15 Δprev ← length(private chain) − length(public chain)
16 append new block to public chain
17 if Δprev = 0 then
18 private chain ← public chain (they win)
19 privateBranchLen ← 0
20 else if Δprev = 1 then
21 publish last block of the private chain (Now same length. Try our luck)
22 else if Δprev = 2 then
23 publish all of the private chain (Pool wins due to the lead of 1)
24 privateBranchLen ← 0
25 else (Δprev > 2)
26 publish first unpublished block in private block.
27 Mine at the head of the private chain.
```

Selfish Mining Algorithm as presented in [2] proceed to mine on their private head until one of the following happens: either the public network finds a block and catches up to the selfish mining pool or the selfish mining pool extends their lead by finding another block. In the first case the attacker would release their block in order to create a fork in the blockchain. In case of a fork, a third-party mining pool could decide to mine on either head, typically the block which has come to its attention first, so the selfish mining pool could use some help from the public, when another pool mines on the head of the attacker. In the second case, the attacker has increased their lead to 2 blocks. When surpassing the milestone of 2 blocks lead the attacker can keep their private chain secret until the public reduces the difference to 1. When that happens, the attacker reveals their private branch, marking the efforts of the rest of the network as wasted.

2.3 Eclipse Attacks

An eclipse attack [4] is a type of attack that targets a computer connected to a decentralized network. In this attack, the attacker tries to isolate the victim's computer from the rest of the network by controlling all of its incoming and outgoing connections. When a target becomes eclipsed by an attacker the attacker has many options about dealing with them. In “Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack” [3] a couple of ways to manipulate an eclipsed target mentioned. The attacker could either destroy the target, meaning that they would isolate them from the rest of the network, or they could collude with them and control what they publish to the rest of the network and when, as well as what they receive from it.

2.4 UPPAAL – SMC

UPPAAL [7] is a tool used for modeling, simulating, and verifying real-time systems based on timed automata. It is particularly powerful for systems that involve concurrent processes and timing constraints, such as communication protocols, embedded systems, and network protocols. UPPAAL allows users to create models, specifying the behavior of system components and their interactions. Once a model is constructed, UPPAAL provides analysis tools to simulate the system's behavior and check properties such as reachability, safety, and liveness using formal verification techniques like model checking. This enables designers and developers to identify potential issues early in the design process and ensure that their systems meet desired specifications and requirements.

As mentioned before, UPPAAL facilitates modelling, simulation, and formal verification of timed automata, enhancing the reliability of time-sensitive applications. Timed automata are a mathematical model used to represent systems where precise timing is crucial, extending finite automata with real-valued clocks that increase uniformly over time. These automata consist of states, transitions labelled with actions, clock constraints, also known as guards, that control when transitions can occur, and invariants that specify timing conditions for remaining in states.

The decision to work with UPPAAL, apart from the existence of previous models, was based on the advantages that it offers. First of all, it is based on timed automata which are suitable for modelling real-time systems, making it ideal for the purpose of modelling a Bitcoin mining network, where precise timing is crucial. Secondly, it comes with an extension known as Statistical Model Checker, which supports model checking, allowing formal verification for properties of the system to ensure that it works as expected under all possible scenarios. Additionally, this tool offers scalability by being able to handle complex systems and large state spaces efficiently. It is also user-friendly, as it offers graphical interface for the visualization of the models making it easy for the user to understand a system's behaviour, thus reducing errors. Another way it helps the user deal with possible errors is the simulation and trace analysis it offers, which allows for detailed simulation of the system and examination of specific execution paths, allowing the user to detect issues. Last but not least, the extensibility of the tool was an immense convenience, as it enabled us to integrate the verifier with scripts outside of the tool, in order to create the scenarios and run the queries efficiently, without manually editing the variables.

The UPPAAL Statistical Model Checker, whose tutorial is provided in [6], is an extension of this tool which allows the user to further examine the behaviour of such parallel systems, through formal verification and statistical model checking with the use of queries. Formal verification and statistical model checking are preferred over plain simulations for studying Bitcoin because they provide mathematical certainty about system behaviour and properties, offering evident proofs of correctness and security, which simulations cannot guarantee. Additionally, model checking enables exhaustive exploration of all possible system states and behaviours, ensuring comprehensive analysis and verification. However, when studying such complex networks where the state space is indefinite the exhaustive exploration is impractical. Statistical model checking, on the other hand, offers probabilistic guarantees about system properties by providing confidence intervals on the likelihood of a property being satisfied based on sampled data. That is why statistical model checking is more suitable in our case, since it relies on statistical analysis and simulation to assess the probability of a property being satisfied.

Queries in UPPAAL have their own language which is provided in the UPPAAL SMC tutorial. There are queries for (a) simulation, which are used to display the values of a variable as the time passes. Another type of query is the probability query which is used to (b) the probability of an expression being true. The last type of query is (c) the evaluation of expected maximums and minimums of a variable. The queries mentioned above include additional information such as probability distribution, probability density distribution, cumulative probability distribution, cumulative probability confidence intervals and frequency histogram. Examples for each type of query are presented below:

(a) simulate[<=10000;100] {var1,var2}

This query will simulate the system 100 times for 10000 time units and present the values of var1 and var2 for those 100 runs as a graph where y-axis is the value of the variables and x-axis is time.

(b) Pr[<=10000;100] (A<> var>100)

This query will run the system 100 times for 10000 time units and calculate the probability range of the expression being true, with 95% confidence interval.

(c) E[<=10000;100] (max: var)

This query gives us the average of the maximum value of var in 100 runs, again with 95% confidence interval. We could also use 'min' instead of 'max' to find the average minimum value of var.

2.5 Previous work

In this section we shortly present previous work that is related to this thesis, following the work of [2] and [4], which introduced the concepts of selfish mining and eclipse attacks.

First, the work in [3] demonstrated an extension of the selfish mining strategies which are also known as Stubborn, with some variations. It also demonstrated how an eclipsed mining pool would affect results, finding that in some cases the eclipsed victims can actually benefit from an eclipse attack.

Additionally, the work in [9] has examined the success of multiple attackers, finding that the common beneficial area for attackers is given by the range of [1/(n+2), 1/(n-1)], where *n-1* is equal to the amount of attackers and if each attacker's power percentage is in this range they can all profit from the simultaneous attacks. It also showed that this area practically exists for 7 attackers at most. Findings included that attackers who have knowledge of each other can benefit from merging in one united pool.

Further, the work [10] extends the work of selfish mining, using two attackers instead of the original work where only one attacker was considered. The main finding of this work was that for a game with two attackers the minimum profitable threshold drops down to 21.48% of the power controlled by each attacker from the initial 25% proposed in [2] for a single attacker.

In [5] UPPAAL was used for modelling the miners and assessing the properties of networks with a single attacker, with multiple variations of the selfish strategy, also known as stubborn strategies. Additional work about multiple attackers using UPPAAL and its Statistical Model Checker exists in [8] where previous findings where confirmed, including the minimum profitable threshold given in [10]. Results included in this work indicated that the higher the maximum allowed length of the private branch, the higher the expected revenue, that however comes with a higher risk factor.

The purpose of this thesis consists of two goals: the first is to extend the models of [8] to simulate multiple attackers in a network, similarly to [9]. Further, we implement eclipse attacks in our simulations and examine how different variables affect the revenue. We aim to target two specific cases: the cases where all attackers have the same power, meaning no one has an advantage and the cases where an attacker has a clear advantage over the rest of the attackers. The other purpose of this thesis is to examine how would a realistic sequence of events unfold, meaning that a mining pool is not allowed to behave selfishly unless the findings of [2] allow them to predict a higher revenue than their deserved shares.

Chapter 3

Blockchain Network Modelling

3.1 Overview 11
3.2 Modelling Parameters and Assumptions 11
3.3 Selfish Mining Strategy 14
3.4 Selfish Mining Strategy involving an Eclipse Attack 15
3.5 Complete Picture of the Network 16

3.1 Overview

In this chapter we will take a further look into the concepts of selfish mining and the eclipse attacks, while defining some variables that will help us deeply understand what is going on inside the network. This chapter offers a chance for the reader to grasp the necessary knowledge for the scope of this thesis.

We start by explaining some modelling parameters that play a role in the model, such as the number of pools, their power etc. We explain concepts that were introduced in previous works and list the assumptions we made in order to make the modelling of such a complex network possible. The essentials of the selfish mining strategy are explained in section 3.3, while an extension of the strategy which combines an eclipse attack is provided in 3.4. The following section aims to explain the coexistence of multiple attackers, providing us with the bigger picture of the network as a whole.

3.2 Modelling Parameters and Assumptions

This section is used to familiarize the reader with the model, its defined elements and the parameters which it consists of.

Entities:

Alice, Bob, Candice, Denise, Ellie, Fred, George, Helen, Ian, John	Selfish Miners
Charles	Unaffected Honest Miner
bCharlesA, bCharlesB, bCharlesC, … bCharlesJ	Controlled Honest Miners
eCharlesA, eCharlesB, eCharlesC, …eCharlesJ	Eclipsed Miners

Parameters:

a (or α), b, c, d, e, f, g, h, i, j	The computational power of each attacker
badCharlesA, badCharlesB, … badCharlesJ	The computational power of each controlled honest mining pool
eclipsedCharlesA, eclipsedCharlesB, … eclipsedCharlesJ	The computational power of each eclipsed pool
goodCharles	The computational power of the unaffected honest pool
γ	The percentage of total controlled honest miners, compared to the total honest miners
γ_a, γ_b, γ_c, … γ_j	The percentage of controlled honest miners that each attacker controls, compared to the total amount of controlled honest miners
n	Number of attackers

A total of ten selfish mining pools were created in the model, along with their controlled groups, an honest and an eclipsed pool for each attacker. The initial of each attacker appears in the end of their victim's name. Charles is the only honest mining pool which is not influenced by the attackers and always mines on the first known longest head. This means that when the public announces a block that becomes the head of the blockchain and a selfish mining pool, or a set of those, publish their blocks in order to create a fork in the chain, Charles will always pick the oldest head, meaning the one announced by an honest miner.

The mining power of each selfish entity is represented by the letters [a~j]. The power of their respective controlled pools is denoted by the variables badCharles[A~J] for controlled honest pools and eclipsedCharles[A~J] for eclipsed pools. There is also the variable goodCharles which represents the power of Charles, the unaffected honest mining pool. The sum of all variables is equal to 100, meaning that these variables represent the percentage of the power of each pool when compared to the total mining power.

In previous works, the concept of ‘γ’ was mentioned and used, both in the theoretical framework and in experimentation phase, and since we want to be consistent and compatible to previous findings, we need to define it. The variable ‘γ’ is the portion of Charles that is influenced by a selfish miner, meaning that it opts to mine on the attacker’s published head when a fork appears. In case of multiple attackers this γ is shared between the attackers, with $\gamma_{[a\sim j]}$ representing the portion of γ the attacker possesses. While we do not directly declare the variables γ and its subparts $\gamma_{[a\sim j]}$, we manipulate their values in the system using the variables badCharles[A~J], with their total sum being equal to γ. For miner x, their $\gamma_{\{x\}}$ is equal to badCharles{X} divided by the total sum of badCharles[A~J]. For example, assume that sum of badCharles[A~J] is 50. If Alice controls an honest mining pool whose power is badCharlesA=25, then γ_a=0.5 since that pool contains half the power of all the controlled honest mining pools.

At this point we have to list the assumptions that were made, in order to reduce the complexity of our model, while maintaining the essence of Bitcoin’s blockchain network. First of all, the revenue is counted in blocks mined. While in reality, the profit of a mining pool is affected from other factors such as transaction fees, we kept the revenue metric as simple as possible, and since the strategy’s purpose is to increase the blocks mined by the pool who adopts it, it made sense to count the gain of a pool by the amount of blocks they mined in the valid chain.

The next convention we used is that the entire set of honest miners is represented by one entity called Charles. This idea was presented in the original work for the Selfish Mining strategy and we kept it, since the behaviour of the model would not change at all, whether the collection of honest miners was divided into multiple pools or kept as one entity. The merge of unaffected honest miners in one unit reduces the complexity of our model, making it simpler and time saving when implementation takes place.

We also have to mention that the network’s topology and properties that derive from it, such as propagation delays and ways they could be taken advantage of, are not directly embedded in our model, however, through the controlled honest mining pools we simulate the behaviour of honest mining pools close to the attacker, that receive information in a timely manner that serves their neighbouring selfish mining pools.

When forks occur, if a pool has no incentive to mine on a certain block, meaning that there does not exist a block in the fork that has been mined either by them, or by a miner that controls them, the pools mine on the head that was published first. This convention was adopted to serve the unaffected honest mining pool, as it is usually the first pool who publishes their block when forks occur and as mentioned earlier, the attackers can only influence their controlled mining pools to choose the attacker's block as their head. By using the convention described above, the rest of the pools, meaning the attackers and the controlled mining pools who do not participate in the fork, will mine on the unaffected miner's head.

Last but not least, when modelling the eclipsed pools, we assume that their attackers collude with them, making them unofficially a part of the selfish mining pool. Our assumption is that the attacker cuts their victim off from the rest of the network and they share all their private information with them, making them mine on their private head, thus helping them extend their lead and win in situations when forks occur. When aiming to create a realistic scenario, we have to take into account that the total power of a selfish mining pool and their eclipsed victim can not exceed the 50% of the total mining power in the network, because in that case we are no longer concerned about selfish mining, but rather a 51% attack, a concept that we will not delve into in this thesis.

3.3 Selfish Mining Strategy

In this section we are going to explain the selfish mining strategy model as initially proposed in [2] and [8]. Its purpose is to maximize the pool's revenue, by wasting time and resources of other miners on orphaned blocks. Its state diagram is presented below.

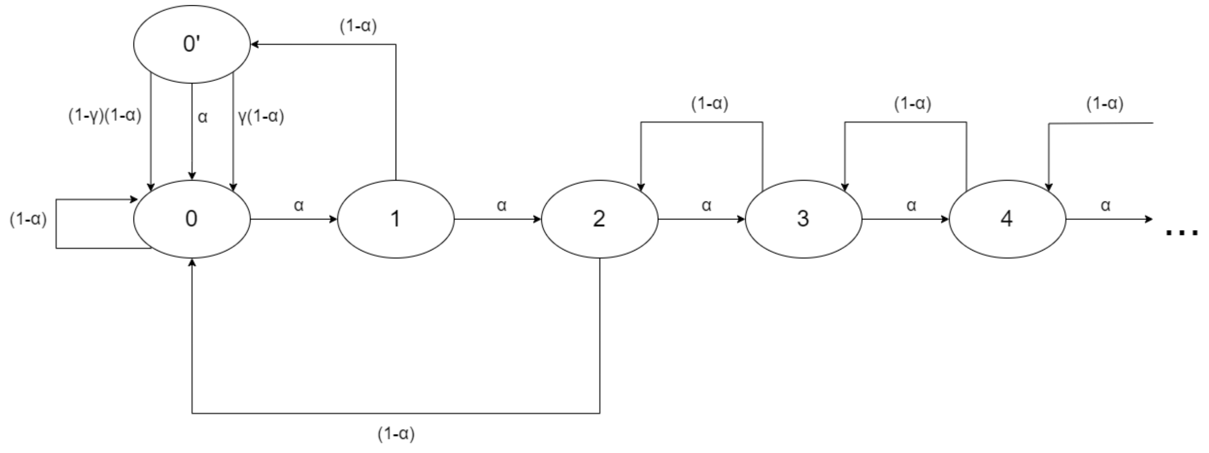


Figure 3.1 – Selfish Mining state

Remember from the previous section that α is the percentage of the mining power the attacker holds in the system and γ is the power of the honest miners who mine on the head of the attacker in case of a fork. The mining power of a pool describes the probability of that pool mining the next block. A single attacker follows the strategy as described by the above chart:

Each state is labelled by the number of blocks that describe the lead of the attacker. The lead of the attacker is equal to the length of their private chain minus the length of the public chain. State 0 describes the state where the attacker and the public have the same head. On the other hand, state 0' describes the state where a fork exists, the attacker has zero lead and the miner who finds the next block determines whose block is publicly accepted, depending on the block they used as their newly found block's predecessor.

The flow of the state diagram can be explained as follows. Let us assume that we are in state 0, where the attacker and the public share the same head. If the public finds a block in state 0, we are still in the state 0, with the new head being the recently published block. When the attacker finds a block in this state, instead of publishing the block, the selfish miner keeps it private in the hope of extending their lead, while the rest of the network is mining on the outdated head. When leading by 1, if the public equalizes finds a block, the selfish miner publishes their block, creating a fork, which is described by the 0' state. When the selfish miner mines the block that wins them the tie, they publish the block to gain the revenue for the two blocks and the procedure restarts. There is also the case where the public finds a new block, either on the miner's head, or on the public head. In the first case, the selfish miner's branch surpasses the public chain, establishing the former as part of the new public chain, thus validating the selfish miner's block and increasing their revenue. In the second case, the public branch remains the longest, thus invalidating the orphaned block of the attacker. Either way, the flow of events directs us to state 0. Back in state 1, if the attacker succeeds in finding a new block before the public does, their lead is increased to 2. Once the lead reaches the milestone of two blocks the strategy claims that the attacker should keep all new blocks private, no matter how many blocks they are leading by, until their lead decreases to 1. Once the lead drops to 1, the attacker reveals their private chain to the public, making their branch part of the blockchain while invalidating all other blocks mined in the meantime process.

3.4 Selfish Mining Strategy involving an Eclipse Attack

In this section we are going to expand on the previous model, to create a new version which involves an eclipse attack. Eclipse attacks [4] can be summarized as the action of isolating a node inside a network, controlling all their connections. This behavior can open many options for the attacker, regarding the ways of manipulation of their victim. They can simply destroy their victim, by cutting them off from the rest of the network. In our model, we are going to implement the strategy of collusion, in which the attacker shares a portion of information to the eclipsed pool, thus influencing them to behave in a manner that would benefit the attacker. In this particular scenario the selfish miner shares their private chain with the eclipsed miner, causing the miner to mine on it since it is the only branch of the chain visible to them, thus helping the attacker with the extension of their lead. In this model we have three variables determining the probability of each transition, a, c and γ, where a is the percentage of the attacker's power and c is the percentage of the eclipsed pool's power when compared to the total mining power, while γ determines the percentage of controlled miners. The modified model can be found below:

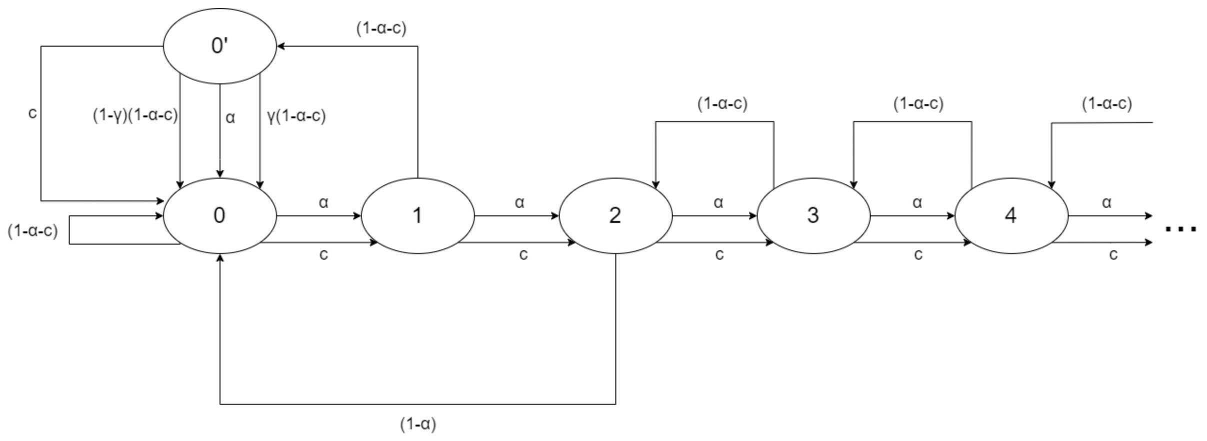


Figure 3.2 – Selfish Mining combined with Eclipse Attack state machine

3.5 Complete Picture of the Network

The network we are modelling consists of multiple attackers following the strategy described in the previous section, which complicates the situation we are investigating. To describe such networks, we would need a new model for each *n*, where *n* is the number of attackers, and by increasing n we exponentially increase the complexity of each state machine. An instance of such a state machine with n=2 and no eclipsed pools is presented in [8]. We omit the graphical representation of the state machines since this is not the case where a state diagram would clear things out but rather confuse the reader.

This section aims to explain the behavior of the network by examining a variety of different situations and assessing the sequence of events that would occur in each scenario. What is key to understand a network of this nature is realising that each miner sticks to the mindset of creating forks when their lead is equalized, and securing their revenue when their lead is dropped to 1. In the context of explaining the model without perplexing things, we merge the cases where the attacker and their eclipsed victim find a block by mentioning only the former, since the two events always lead to the same states, with the only difference being the pool which claims the revenue for the block.

The first scenario is the case where only one attacker finds a block, or a set of them, while their selfish competitors fail to find a block, with the only other miners contributing to the chain being the unaffected honest miners, until the attacker has to publish their private branch. In such a scenario, the functionality of the network is exactly the same with the one described in Section 3.4, since the other malicious mining pools are inactive during that time.

The following scenario involves a set of selfish miners being active in the network. There exists the possibility that a set of miners find a couple of blocks that they keep private from the rest of the network. At that point an honest miner finds and publishes a block. If there exist miners with a private branch of length 0, they adopt that block as their new head. If there are miners with a private branch of length 1, based on the proposed strategy they publish their block, creating a fork, possibly with multiple branches if there exist multiple miners with such private chains. In that case, each miner attempts to win over the fork, by getting either themselves or their controlled groups to mine on their published head. However, there might exist miners with even larger branches that are still kept private to waste their opponents time and resources even further. They will keep those chains a secret, until they are forced to publish them, when their lead drops to 1, in order not to risk their theoretically “secured” revenue. This scenario potentially reaches a state where all pools publish their private branch, having all pools agree on a common public head, when the selfish miner with the longest private chain has their lead decrease to 1, thus being forced to publish their entire private branch. If there exist multiple miners whose private branches share the same length, a fork will occur instead of a new commonly agreed public head.

When we originally viewed the selfish mining strategy, we observed that forks happen when a selfish miner has a secret branch of length 1 and the public finds a new block, equalizing the attacker and forcing them to create a fork. However, when multiple attackers play, there might occur forks, without an honest miner’s block being part of the fork. Let us examine the case where two attackers have a lead of two and a block is announced by an honest mining pool. The two attackers will both publish their private branches, thus creating a fork where none of the blocks are from an honest miner. In that case whichever miner manages to publish their private branch first gains the advantage, as our convention suggests that when a miner has no incentive when choosing a head to mine on, they mine on the oldest possible head. In the scenario described the miner with the most resources has the advantage, as when it comes to the implementation in Chapter 4, the more power rate a miner has, the greater the chance that they publish their branch first. This is an accurate representation of what would realistically happen, since the larger the mining pool, the greater the influence they are likely to have to the rest of the network. In a scenario where miners share the same power, each side shares the same chance of publishing their branch first as any other miner with that same power, making our model fair for every attacker.

Chapter 4

Implementation

4.1 Overview	19
4.2 Global Variables and Environment	19
4.3 Honest Miners Template	21
4.4 Selfish Miners Template	23
4.5 Eclipsed Miner Template	27
4.6 Complete Picture of the Network	28

4.1 Overview

In this chapter we are going to present the implementation of the templates described in the previous chapters, with the use of the UPPAAL modeling tool [7]. The Selfish and Honest miner templates were originally used in [8]. Since we wanted to continue the research on that subject, the same templates were used, with some changes taking place so that the system could function with multiple attackers. The automata were designed in a simple, abstract way, so that we could focus on our target which is the simulation of scenarios where attackers gain an unfair share of blocks mined. In other words, we want to compare the relative revenue of each miner to their relative power, find cases where this ratio exceeds realistic standards and find solutions to battle this phenomenon. In this chapter, one can find the implementations of Selfish, Honest and Eclipsed miners, as well as assumptions that were made in order for the implementation to be short and straightforward.

4.2 Global Variables and Environment

The very first thing our implementation needed to include was the global declaration page, where we defined global variables, structures and broadcast channels used for synchronization between distinct miner entities. This environment is listed below, with explanation for each element of the model’s ‘Declaration’ page.

- const int BLOCKLIM – The maximum amount of blocks to be mined. When the amount of blocks mined reaches BLOCKLIM, the miners stop mining and after synchronization they calculate their revenue.
- typedef int [-1,BLOCKLIM] BlockID – BlockID is a type defined by us, an integer in the range of [-1,BLOCKLIM]. It works as a safety measure, as it guarantees that blocks do not contain ids larger than the limit set by BLOCKLIM. If for any reason a BlockID variable is set out of the definition range the simulation terminates.
- int blockHash – This variable functions as a counter, and is used to set the ids of the blocks. Whenever a block is found, blockHash increases by 1.
- const int SLIM – The limit of Selfish miners in the simulation.
- const int HLIM – The limit of Honest miners in the simulation
- typedef int [0,100+HLIM] HMinerID – HMinerID is a type defined for the ids of Honest and Eclipsed Miners.
- typedef int [0,SLIM] SMinerID – SMinerID is a type defined for the ids of Selfish Miners.
- The block structure is defined as follows:
  typedef struct {
  BlockID blockID; - Its ID
  BlockID prevID; - The ID of its previous block in the chain
  HMinerID hMiner; - The ID of its miner (if the miner was honest/eclipsed)
  SMinerID sMiner; - The ID of its miner (if the miner was selfish)
  int [0 ,BLOCKLIM+1] length; - The length of its chain when it was assigned as the head
  } Block;
- broadcast chan newBlock – A broadcast channel used for synchronization when a miner publishes a new block
- broadcast chan newEclipsed – A broadcast channel used for synchronization between an attacker and an eclipsed miner who just found a block
- meta Block tempBlock – This variable is used for transferring information about the mined block to each miner when synchronization takes place.
- meta int eclipsedID – This variable contains the id of the eclipsed miner who found a block so that when synchronization takes place, the attacker who controls that mining pool can take their actions.

- bool outOfSpace – This variable is false throughout the simulation, until blockhash reaches the limit of BLOCKLIM. At that point the variable turns true, the mining stops and each pool calculates their revenue.
- int syns – This variable is used to count the amount of Selfish miners who are synchronized at the end of the simulation. Synchronized at the end of the simulation means that they have published all their blocks.
- bool lock , int slock – These two variables are used as a lock when selfish miners are in release states. If a miner has a priority (e.g. they started releasing their whole private branch), then the variable slock is set to the value of that miners id.
- meta bool nextNotFound, meta bool prevNotFound, meta bool wrongLength – These are variables used in sanity checks for the system, they should always be false.

4.3 Honest Miners Template

The template for honest miners is simple. Its instance variables are as follows:

- const int [1,HLIM] id – The identity of the honest miner. Its values can be in range of 1 to HLIM.
- const int rate – Rate of exponential controls the pace at which transitions happen in each state. We control this with the variable rate, which depicts the mining power of each mining pool. The rate of exponential at the 'Mine' state of each miner is equal to rate:1000, meaning that each miner mines approximately (rate:1000) blocks per time unit, each using their own rate. The sum of all rates is equal to 100. This means that if we assume that time units represent minutes, we can expect a block to be mined every 10 minutes in our system, like in the real life application of the Bitcoin network.
- const bool isBad – Honest miners might be controlled by selfish miners. The variable isBad is used to define whether that miner is controlled by a selfish miner. This makes them function differently when processing broadcasted blocks, favoring the attacker in case of a fork.
- int publicHead – the index of the public head in the chain.
- Block chain[BLOCKLIM+BLOCKLIM/2] – An array of Blocks which contains all published blocks.
- Variables that are used in the calculation of the revenue and comparison to other miners are:
    - int revenue
    - int revenueOthers

- double revenueDbl
- double revenueOthersDbl
- double relativeRev

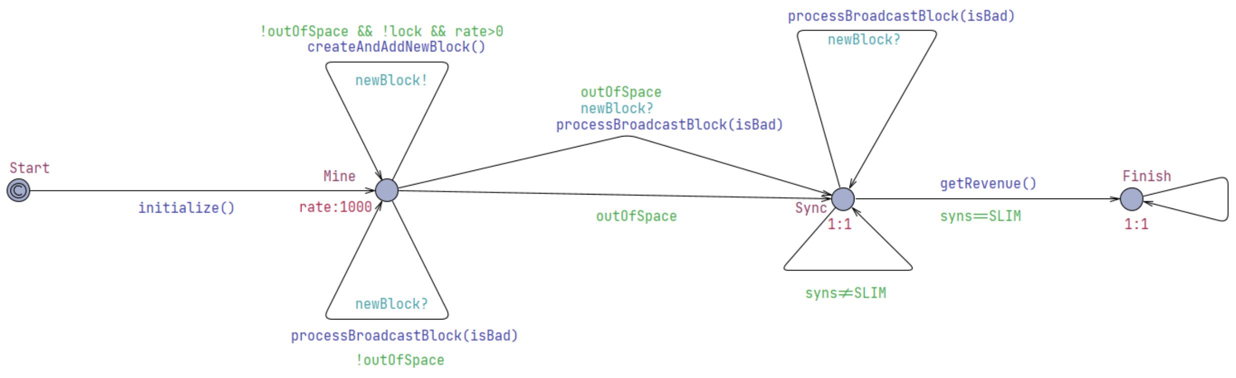


Figure 4.1 - Honest Miner Template

Let us take a look at the Honest Miner's template and explain each state and transition. All honest miners begin at the 'Start' state, and take the transition with the function 'initialize()'. This function creates the first block in the blockchain known as genesis block and sets the publicHead variable to 0.

Now the Miner is at the Mine state, where they try to mine as many blocks as they can. Using the upper edge on the Mine state, when the conditions of the guard are met, they create a new block and add it to their blockchain. They also set the tempBlock variable equal to that block and use the synchronization channel to let the rest of the network know that they have created a new block.

When at the Mine state, a miner can also receive information about a published block from another miner, again by the use of the newBlock synchronization channel. The synchronization edge makes the miner process the published block. An honest miner when receiving a new block compares its length with the head they are mining on and if the new block has greater length they switch the block they are mining on, making the new block the head of the public branch. However, an honest mining pool will also switch heads when the new block creates a fork, and that fork is mined by the selfish mining pool that controls them. This behavior is embedded in the method processBroadcastBlock(isBad), which takes into account whether the mining pool is controlled by an attacker or not. Even though in the real life application of the Bitcoin's blockchain this is not what would happen exactly, it is still an approach that simulates the influence that a mining pool would have on another pool, when exploiting the network, for instance taking advantage of the topology and propagation delays.

When the blockhash variable reaches the limit set by BLOCKLIM, the Honest Miner stops mining and transitions to the Sync state, where they process the blocks broadcasted by Selfish Miners. One can notice two edges for getting to the Sync state. The first edge exists simply for the transition. However that does not serve us when the outOfSpace variable is true, the Miner is in the Mine state and a new block is found from another miner, making the newBlock channel fire a synchronization. In the scenario described above, we need to utilize the second edge, as without it blocks will be lost. These scenarios were thoroughly tested and determined the significance of its presence.

Last, but not least is the Sync state, at which the Miner receives the blocks that were kept private by the selfish mining groups, in order to determine a final public head and make the revenue calculations based on the whole picture. When all selfish miners have published all their blocks, the Miner transitions from the Sync state to the Finish state. During that transition each miner calculates their revenue, as well as their relative revenue, which is the ratio of their revenue compared to the sum of all miners' revenues.

4.4 Selfish Miners Template

Selfish Miners are described by a much more complex template:

- const int [1,SLIM] id – The identity of the selfish miner. Its values can be in range of 1 to SLIM.
- const int rate – const int rate – Rate of exponential controls the pace at which transitions happen in each state. We control this with the variable rate, which depicts the mining power of each mining pool. The rate of exponential at the 'Mine' state of each miner is equal to rate:1000, meaning that each miner mines approximately (rate:1000) blocks per time unit, each using their own rate. The sum of all rates is equal to 100. This means that if we assume that time units represent minutes, we can expect a block to be mined every 10 minutes in our system, like in the real life application of the Bitcoin network.

- const int maxLength – This variable is a threshold. When the length of the private branch reaches the maxLength, the miner publishes all of their blocks, in order not to risk losing the revenue of those blocks.
- int publicHead – the index of the public head in the chain.
- int privateHead – the index of the private head in the chain.
- int indexLastPublic - the index of the last block that this pool mined and published
- Block chain[BLOCKLIM+BLOCKLIM/2] – An array of Blocks which contains all published blocks.
- int privateBranchLen – This is the length of the private branch of the pool
- bool indSync – This is a boolean flag used at the Sync location, which shows whether the miner is synchronized, or in other words they have published all of their private blocks.
- bool eclipsedMined – This is a boolean flag which indicates whether the eclipsed miner that found a block is controlled by this mining pool.
- int diffPrev – This is an integer used to store the difference between private chain and public chain when a new block is announced.
- int release – This is a variable which stores how many blocks the miner will release
- int counter, bool tieBreaker – Variables used to determine the winner when lead ties occur.
- Variables that are used in the calculation of the revenue and comparison to other miners are:
    - int revenue
    - int revenueOthers
    - double revenueDbl
    - double revenueOthersDbl
    - double relativeRev

A selfish miner starts at the ‘Start’ state. A transition to the ‘Mine’ state lets the miner initialize their local variables; they add the genesis block to the blockchain and initialize the public head, the private head, the index of last public and private branch length variables all to 0. If they miner hasn’t got a rate bigger than 0, that means that the miner practically does not exist, therefore the synchronization part of the code that would happen in the end happens at the beginning of the simulation.

We are now at the ‘Mine’ state, where there are lots of options. Let us start with the case of our miner finding a block. In that case, the miner calculates the value of diffPrev, creates the block and adds it to its private branch and increases the private branch’s length. Now the miner is located in ‘SelfishMine’ where one of the following will happen:

- The private branch’s length has reached the maxLength threshold, therefore the miner will release their whole private branch.
- A fork exists and this block wins the tie for the selfish miner, therefore they publish it instantly.
- If neither of the above is true, do nothing else.

The above possibilities describe the three edges that lead back to the ‘Mine’ state.

When returning to the ‘Mine’ location, the guard’s conditions for the edge responsible for releasing blocks might be true. In that case, the selfish miner takes the transition, once or multiple times, releasing the appropriate amount of blocks.

When at the ‘Mine’ location, another Miner might publish a block, using the newBlock synchronization channel. To deal with the blocks published, the selfish miner will take the transition associated with that channel. During this transition the miner will again calculate the diffPrev variable, they will add the new block to their chain structure, they will check for double lead tie and calculate the amount of blocks that they have to release according to the selfish mining strategy.

There is also the case of an Eclipsed Miner publishing a block. When that happens, the broadcast channel newEclipsed is used and therefore the selfish miner follows the edge which leads to the EclipsedMined location. Each selfish miner checks the id of the miner that enabled the broadcast channel which can be found in the meta variable eclipsedID. If that id is equal to the selfish miner’s id + 100, that means that the eclipsed miner is controlled by that mining pool. In that case the variable eclipsedMined is true. From the location EclipsedMined, if the flag is true, the selfish miner takes creates the new block, treating it as a block created by themselves. This means that the block is not published immediately, but it gets included in the pool’s private branch. The only way that it differs from a block mined by the pool is that the revenue for the block is claimed by the eclipsed pool who originally mined the block. It must be noted, that this flow of events is not what would realistically happen in a blockchain, the selfish mining pool would not create the block themselves, they would just withhold this information from the public. Nevertheless, this implementation is simpler and produces the same results, thus it was utilized to save time and resources.

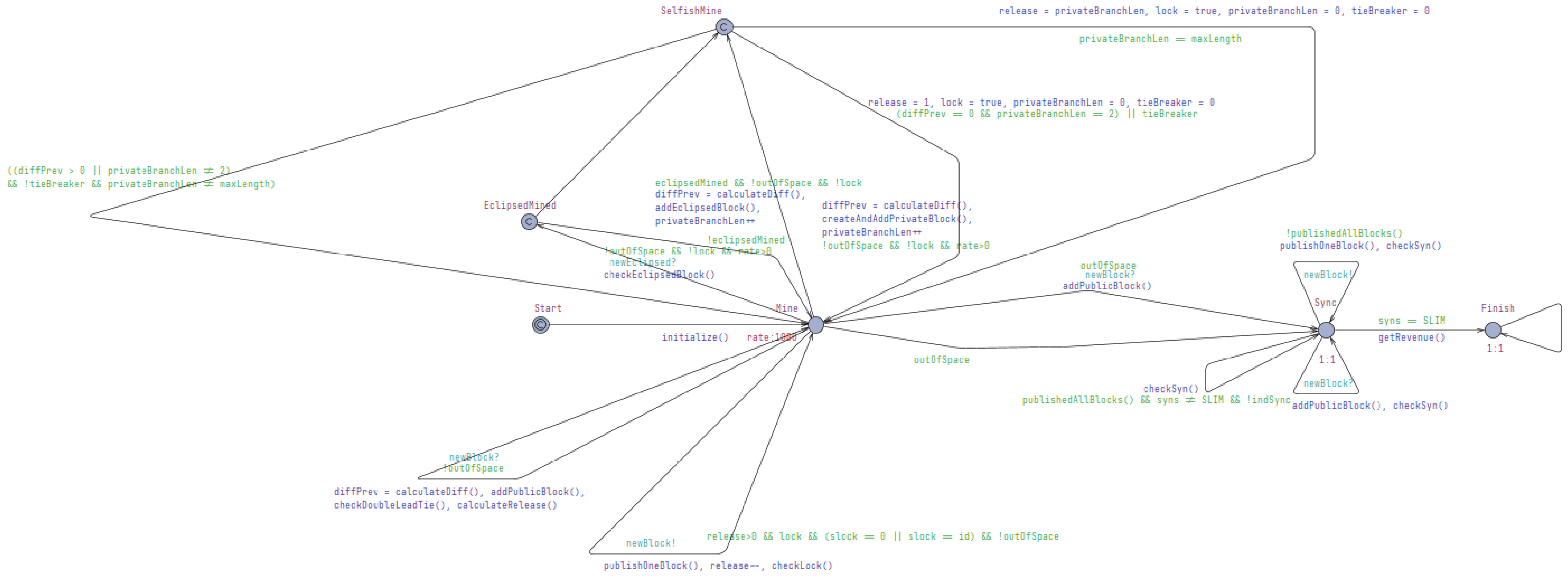

SelfishMine
release = privateBranchLen, lock = true, privateBranchLen = 0, tieBreaker = 0
privateBranchLen == maxLength
release = 1, lock = true, privateBranchLen = 0, tieBreaker = 0
(diffPrev == 0 && privateBranchLen == 2) || tieBreaker
((diffPrev > 0 || privateBranchLen != 2)
&& !tieBreaker && privateBranchLen != maxLength)
EclipsedMined
eclipsedMined && !outOfSpace && !lock
diffPrev = calculateDiff(),
addEclipsedBlock(),
privateBranchLen++
diffPrev = calculateDiff(),
createAndAddPrivateBlock(),
privateBranchLen++
!outOfSpace && !lock && rate>0
!eclipsedMined
!outOfSpace && !lock && rate>0
newEclipsed?
checkEclipsedBlock()
!publishedAllBlocks()
publishOneBlock(), checkSyn()
outOfSpace
newBlock?
addPublicBlock()
newBlock!
Sync
Start
Mine
initialize()
rate:1000
syns == SLIM
getRevenue()
Finish
1:1
1:1
outOfSpace
checkSyn()
publishedAllBlocks() && syns != SLIM && !indSync
newBlock?
addPublicBlock(), checkSyn()
newBlock?
!outOfSpace
diffPrev = calculateDiff(), addPublicBlock(),
checkDoubleLeadTie(), calculateRelease()
newBlock!
release>0 && lock && (slock == 0 || slock == id) && !outOfSpace
publishOneBlock(), release--, checkLock()

Figure 4.2 – Selfish Miner Template

When the variable ouOfSpace turns true, the miners jump to the Sync state where each miner publishes their unreleased blocks. When a miner publishes all of their blocks they “synchronize” with the rest of the network. The variable syns is responsible for counting the synchronized selfish mining pools. When all mining pools are synchronized, each mining pool is allowed to go to the Finish state and count their revenue.

4.5 Eclipsed Miner Template

Before we dive into the details of the implementation, we must provide an overview of what our aim was regarding the template. As mentioned in Chapter 2, there are many ways an attacker could manipulate their eclipsed victim. Since we are investigating ways in which an attacker exploits the network in order to maximize their revenue, we decided to design an eclipsed miner based on the "Collude with Eclipsed" strategy. When the attacker utilizes this tactic, the Eclipsed Miner simply assists the Selfish Miner who controls them, by mining on the head of their private chain, thus extending their lead. In order to achieve this without needing to synchronize each set of attacker and eclipsed mining pools, we used a simpler implementation; the Eclipsed Miner does not have an accurate image of the blockchain, but instead focuses on mining blocks. When they mine a block, they broadcast their *id* and the attacker takes care of the block's details. In reality, the eclipsed miner would have the information needed to mine the block. In this implementation the attacker is responsible for publishing the eclipsed miner's blocks. The eclipsed miner does not ever publish blocks, they only add a block in their chain structure when another mining pool publishes one.

Let us proceed with further details about the implementation. The eclipsed miner template is similar to the honest miner template. The only difference regarding the variables is that the upper bound of the ID is equal to SLIM+100, since a selfish miner controls the eclipsed miner whose id is greater by 100.

Like the rest of the templates, Eclipsed miners begin the execution at the Start location. The initialize() function creates the genesis block and the miner is ready to create the first block entering the ‘Mine’ location. When finding a new block, the createNewBlock() method will only set the eclipsedID meta variable equal to the eclipsed miner’s ID so that the synchronization can take place. After that they notify all Selfish miners using the newEclipsed broadcast channel. From the selfish miner template in section 4.3 we know that only the pool which controls this eclipsed miner will manipulate the block, while the rest will continue the execution as if nothing happened.

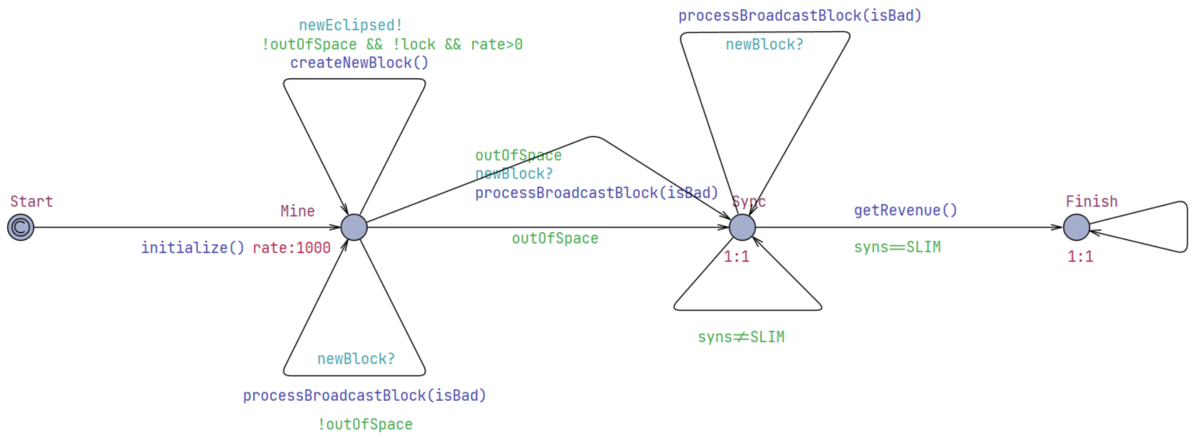


Figure 4.3 - Eclipsed Miner Template

There is also the possibility of blocks being published towards the eclipsed miner. While realistically would probably not occur, we allow eclipsed miners to learn about all blocks, so that they can calculate their revenue as the rest of the pools are able to. This detail does not have an impact on the functionality of the eclipsed miners.

After the system the threshold of BLOCKLIM has been reached, the miner transitions to the Sync location where Selfish miners publish their last blocks. The eclipsed miner adds these blocks to their blockchain and after the synchronization takes place they can calculate the revenue while shifting to the Finish state.

4.6 Complete Picture of the Network

Having the insights of each template we can now view the system as a whole. The system is defined in the System Declarations page of the UPPAAL tool. In this page we declare the power that each miner will have, the max length of the private chains and we create the miners using the templates mentioned above. After everything is set, we call the **"system"** instruction to emulate the system with the system elements created in the page.

A complete system consists of multiple attackers, their honest and eclipsed victims, as well as a set of honest mining pools which function exactly as the protocol suggests. Alice, Bob, Candice, Denise, and Ellie are the selfish miners, Charles represents the unaffected honest miners as a united entity, bCharles[A~E] are the controlled honest pools and eCharles[A~E] are the eclipsed pools. The relationship between attackers and controlled pools is displayed using the last letter in the name of the controlled pool. For example, bCharlesA is an honest miner controlled by Alice and eCharlesE is a pool eclipsed by Ellie. What follows is an example of the declaration page. Reoccurring patterns were omitted.

```
//Honest miner parameter bad or not
const bool isBad = true ;

//Mining Power parameters
const int alfa = 10;
const int beta = 10;
...

const int goodCharles = 30;
const int badCharlesA = 10;
const int badCharlesB = 0;
...

const int eclipsedCharlesA = 0;
const int eclipsedCharlesB = 10;
…
const int maxLength = 3;

Alice = Selfish(1, alfa, maxLength);
Bob = Selfish(2, beta, maxLength);
…

Charles = Honest(SLIM+1, goodCharles, !isBad);

bCharlesA = Honest(1, badCharlesA, isBad);
bCharlesB = Honest(2, badCharlesB, isBad);
…

eCharlesA = Eclipsed(101, eclipsedCharlesA, isBad);
eCharlesB = Eclipsed(102, eclipsedCharlesB, isBad);
…
```

```
// List one or more processes to be composed into a system.
system Alice, Bob, Candice, Denice, Ellie, Charles, bCharlesA, bCharlesB, bCharlesC,
bCharlesD, bCharlesE, eCharlesA, eCharlesB, eCharlesC, eCharlesD, eCharlesE;
```

In the above example five attackers were simulated along with their controlled groups and one honest mining pool. The maximum amount of attackers tested for the purposes of this thesis was ten. Another aspect of the network we took into account is the total mining resources. The total mining power in each scenario was equal to 100.

Another thing taken into consideration when designing and reformatting the templates was the scenario where two branches have the same length at the end of the simulation. In those cases, whichever branch is published first is considered the winning branch. This convention is fair as no pool gains any unfair advantage from it. We tested that the weighted sum of relative revenues, where the weight is the mining resource, is equal to 1, which is what we expected, and the tests verified that the weighted sum is indeed 1, thus the revenue calculation is correct.

Chapter 5

Experimental Results and Evaluation

5.1 Overview 31

5.2 Multiple Selfish Attackers Competition 32

5.3 Designated Superior Pool 35

5.4 Realistic Scenarios with Profitable and Damaged Attackers 43

5.1 Overview

In this chapter, we present a series of simulations to explore various scenarios in the context of Bitcoin and selfish mining. The primary focus was on two types of scenarios: (a) equal competition between selfish mining pools, where no attacker has a distinct advantage, and (b) scenarios featuring an attacker with a clear advantage. The objective was to analyse how different variables impact the revenue of the favoured attacker.

During the simulations, we examined the influence of each variable on the attacker's revenue. This involved adjusting the number of mining pools and manipulating the values of b and c to observe the resultant effects under varied combinations of α and γ in order to verify that our findings are consistent with all network variations. By understanding these dynamics, we were able to identify conditions under which an attacker could potentially gain a disproportionate share of mining rewards and ways that they could maximize their results. Another remark that derives from these simulations is that a selfish miner could take advantage of unwise mining pools who opt to mine selfishly without having the required power to back up this decision.

Moving from theoretical analysis to more realistic applications, the scenarios were further refined to involve two miners with various interaction dynamics. The settings for these cases required that each attacker should expect to be profitable based on the findings of [2]. Such scenarios were practically possible with a maximum of two attackers. The simulations were divided in three categories: cases where the two miners have the exact same properties, cases where one attacker directly holds more power than their opponent, while the latter controls more honest miners, and last, the cases where one attacker has a clear advantage by both holding more power and controlling more honest miners.

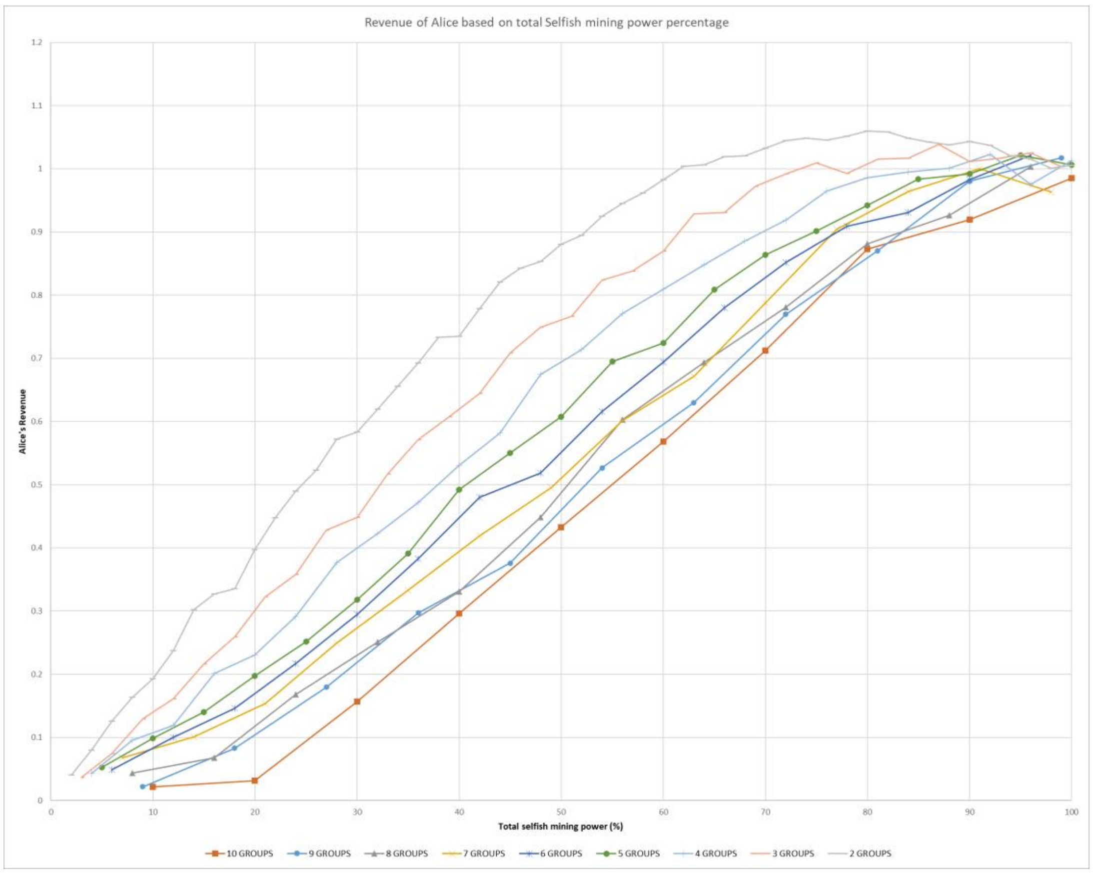
5.2 Multiple Selfish Attackers in Equal Competition

We want to expand on previous work, simulating scenarios where the selfish mining pools are more than two. We reused the templates after readjusting them for our purposes. After testing the system for two attackers and verified that our results are consistent with previous findings, we were able to proceed with the experimentation stage.

The first experiment aimed to examine how the amount of selfish mining pools influences the relative profitability of each pool, assuming equal power distribution among selfish miners. The underlying theory from [9] suggests that an increase in the number of selfish pools leads to a much more competitive network, making it increasingly challenging for these pools to find a common beneficial area and generate profit. Our main objective was to confirm this statement and provide evidence of its validity. At the same time, the experiment was repeated twice using controlled mining pools, the first time with controlled honest pools and the second time with eclipsed pools, aiming to compare the level of influence that each type of pool has.

We designed the experiment as follows. First of all, we created ten selfish mining pools and the honest miner known as Charles. Using Python and Windows PowerShell we were able to change the power of each pool and run the appropriate query for our results, by utilizing the 'verifyta' command. Each scenario was created by specifying two variables; the number of attackers, and the power of each selfish mining pool using a variable called 'alfa'. (e.g. groups = 8, alfa = 3). Then, the python script was responsible for assigning the power alfa to the specified amount of groups, as well as power equal to 100-alfa*groups to the honest mining pool, so that in each case the total power adds up to 100. The PowerShell script contains a loop which runs the query and increases alfa by 1 each time, with a=1 as the initial value. When alfa reaches a point where 100-groups*alfa would be a negative number the execution is stopped and the results are taken. To take results for all values of the variable 'groups' we ran the PowerShell script 9 times, one for each possible value in the range of [2-10], each time modifying the variable inside the script, as well as adjusting the Python script to work with that number of groups.

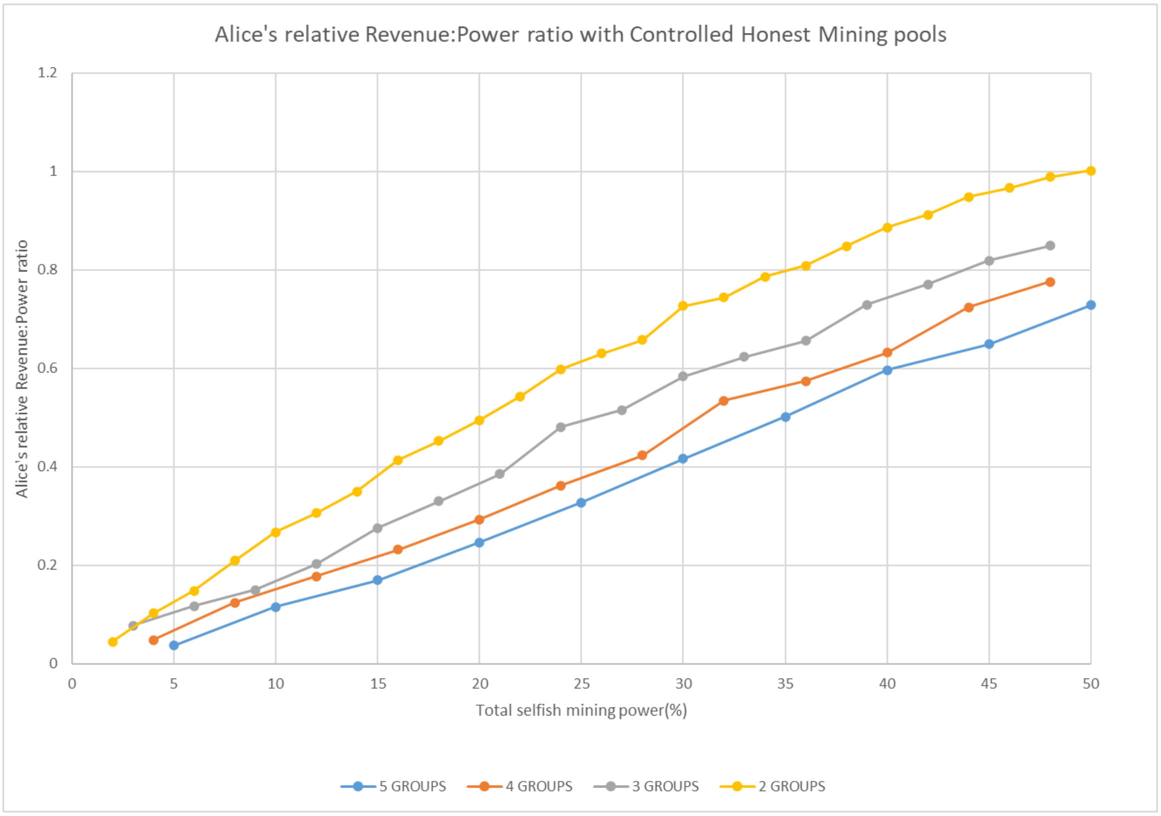
The query used for this experiment is E[<=T;100] (max: Alice.relativeRev / (1.0*alfa/100)). T is used to control the time units that each execution takes and was changed throughout the experiment as executions with more miners needed more time than executions with less amount of miners. The value after T indicates the amount of times the query will be executed. We ran each query 100 times since we wanted our results to be accurate and leave minimum space for error. Remember that Alice is an attacker in our system, while relativeRev is a field which is equal to Alice's revenue divided by the whole network's revenue; in other words it gives us the percentage of Alice's revenue counted in blocks, compared to the total blocks mined. The query gives us the average of the maximum value of the ratio of Alice's relative revenue to Alice's relative power. We could have used any miner instead of Alice, since they all compete under the same terms, same mining power and no controlled pools, therefore the values are identical for all mining pools. We used maximum since this value is produced at the end of the simulation therefore the maximum value it takes during a run is the calculated value at the end of each execution. The formula is self-explainable as well; we divide Alice's relative revenue by Alice's relative power – alfa is the power of Alice, while the total power in the system adds up to 100. In a normal scenario where everybody follows the protocol, this value should approach 1. However, when selfish mining takes place, this value can vary. The query is executed 100 times and gives us a mean average with 95% confidence interval, whose range is significantly low, enabling us to use these values.

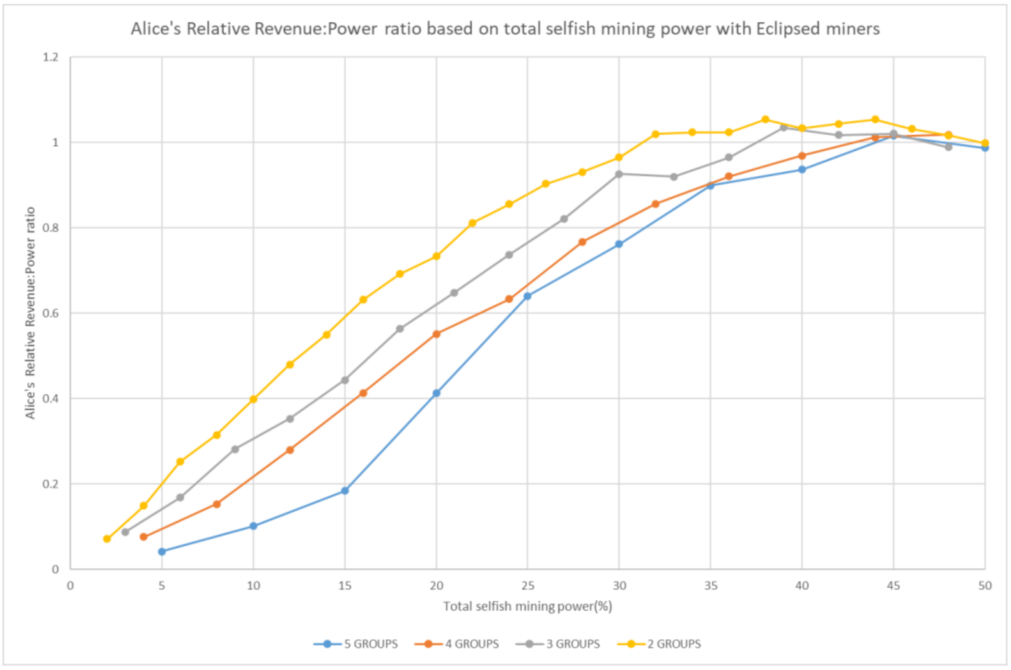


The above graph has two key features that gain our attention. Firstly, we can confirm the findings of [9] regarding the amount of pools was correct. It is clear that as the amount of attackers decreases, the expected revenue of the attacker increases, while the common beneficial area gets larger. However, the scenarios where the attackers can expect profit are limited.

After seeing these results, we decided to extend the previous test with the use of controlled miners. Since we witnessed the effects of the amount of groups has to the relative revenue we decided to continue the experiments with a maximum of 5 groups, focusing on other aspects of the network rather than the amount of pools.

In the next scenarios, the use of controlled honest mining pools and eclipsed mining pools was utilized. The power of each controlled pool is equal to the power of the attacker which controls them. For example, in the scenario where there are 5 attackers and the total selfish mining power is 25, each attacker holds 5% of the network's power and each controlled pool also has 5% of the power, while the rest of the power is managed by Charles, who is the unaffected honest miner.





The two experiments presented in the graphs above confirm that controlling other groups can significantly increase the expected revenue of a mining pool, as opposed to controlling none, with eclipsed groups being better than controlled honest pools, as expected. Nevertheless, in most cases presented, the miners are still unable to profit from an uneven share of revenue.

A further analysis on the effect of each is variable will be demonstrated in the following section, while the examination of multiple miners in an equal competition with the use of eclipsed and controlled honest mining pools will follow in section 5.4, where we will examine the circumstances that allow profit maximization in a variety of realistic conditions.

5.3 Designated Superior Pool

Our aim is to design scenarios in which a pool is able to exploit the system and gain an unfair amount of profit, compared to its power percentage. In the previous section we observed that when multiple attackers share the same amount of mining resources they are not likely to succeed in this task, apart from some exceptions, in which the attacker still does not increase their expected revenue significantly.

Therefore, we had to come up with a new scenario in which the attacker has the odds to their favour. This new scenario consists of 5 selfish mining pools, where Alice holds a major percentage ‘a’ whose range is in [30,50], while each of the other 4 mining pools holds power ‘b’. The selfish mining pools can control honest mining pools as γ indicates, according to what has been mentioned in section 3.2. The only selfish mining pool that controls an eclipsed miner in this scenario is Alice. The power of the eclipsed miner is described by ‘c’. The amounts of good and bad honest miners are determined by the variables a, b and c as follows:

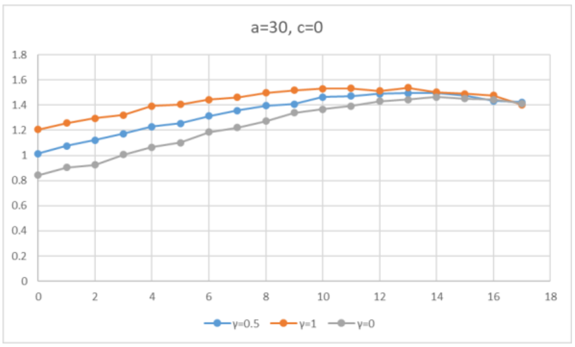
```
Charles = int((100-a-4*b-c))
badCharles = math.floor(Charles*gamma/100)
badB = math.floor(badCharles*b/(a+4*b))
badA = badCharles-badB*4
```

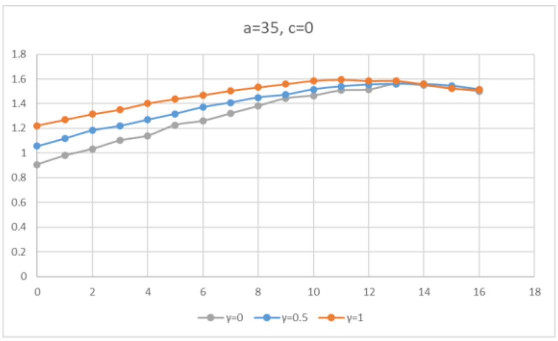
The above is part of the python script that is responsible for assigning values to the variables in the xml file used in UPPAAL.

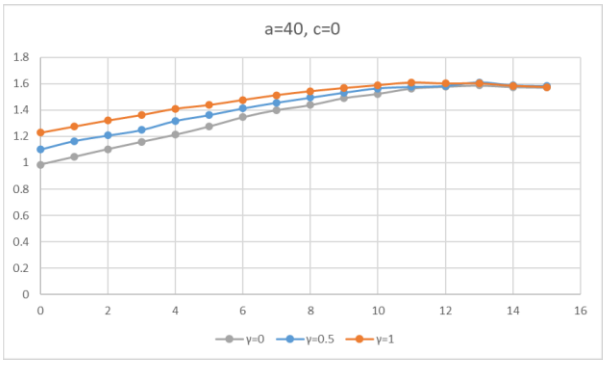
While these scenarios are still not realistic, since the other 4 attackers do not have an incentive to behave selfishly, they still have to offer significant information, since we are able to observe the effect that these less-favoured pools and the eclipsed miner have on Alice’s revenue, compared to their absence, as initially assessed in previous work.

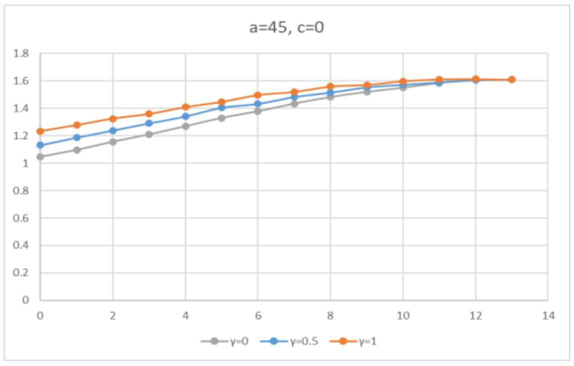
Let us start from the effect of the weak selfish pools. Their effect can be measured through the variable ‘b’. Our first simulation will help us observe how the power of the ‘outsiders’ affects Alice’s revenue, keeping every other variable constant, throughout each simulation. Of course, we also want to examine whether the correlation of the two variables is consistent with all sets of combinations of the other variables (a, c, and γ), thus we created a variety of scenarios to test this. Each chart contains three lines: γ=0, γ=0.5, γ=1, while a changes with each chart. We grouped the charts using the variable c. Our results are depicted in the following graphs:

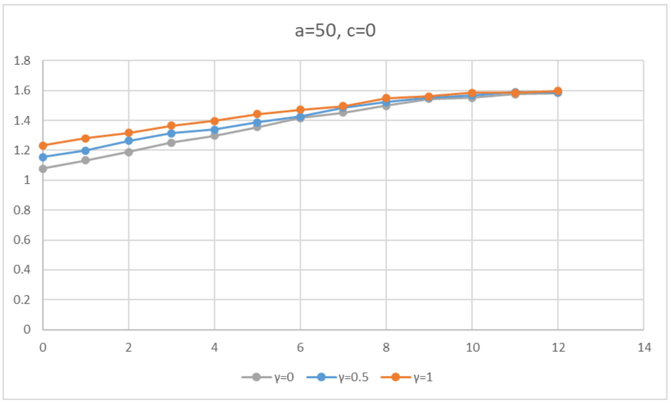
a) c=0



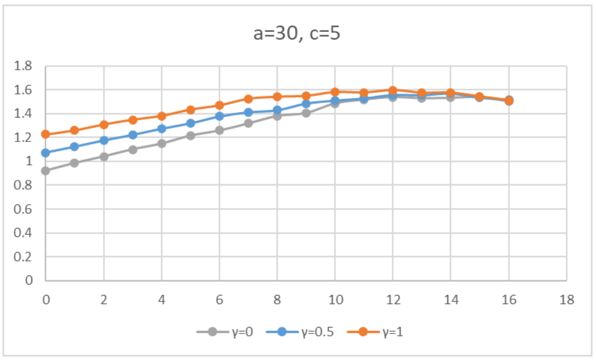


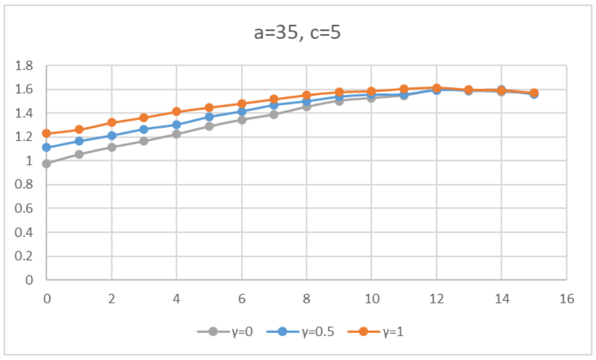

a=40, c=0
γ=0
γ=0.5
γ=1

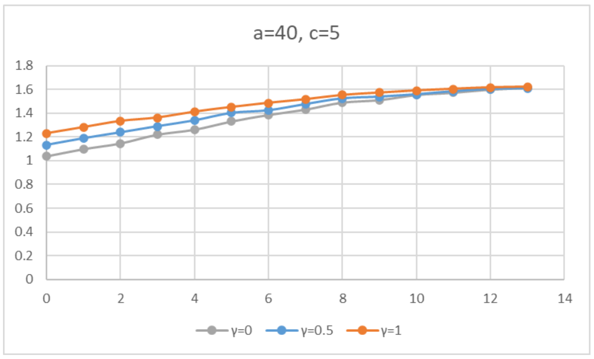

a=45, c=0
γ=0
γ=0.5
γ=1

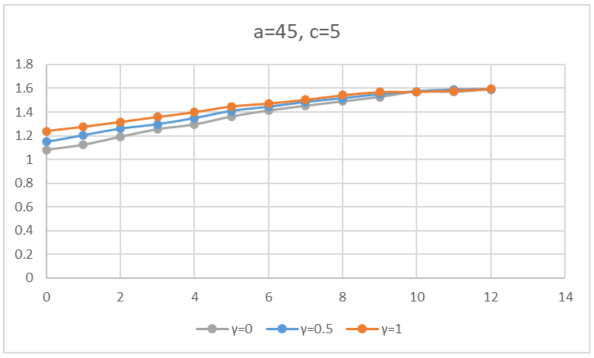

a=50, c=0
γ=0
γ=0.5
γ=1

b) c=5

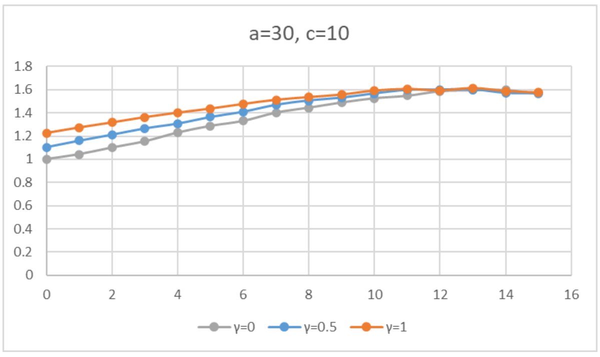

a=30, c=5
γ=0
γ=0.5
γ=1

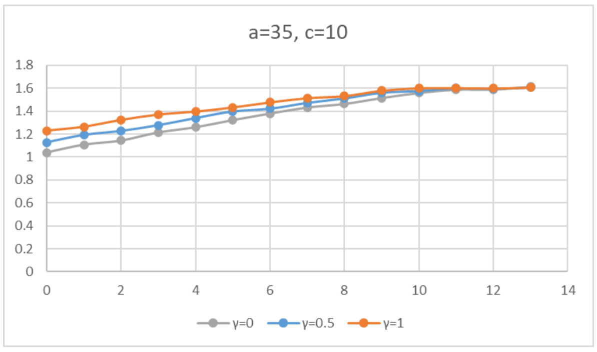

a=35, c=5
γ=0
γ=0.5
γ=1

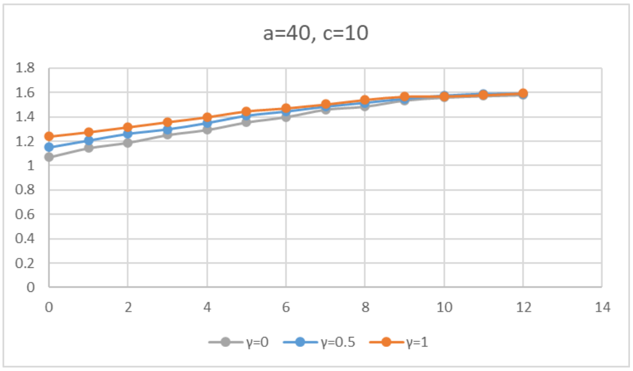

a=40, c=5
γ=0
γ=0.5
γ=1


a=45, c=5
γ=0
γ=0.5
γ=1

c) c=10







As we can observe from the above graphs, Alice benefits from the existence of weak selfish miners in the system. We get a clear increase when values of b are low and this effect seems to be saturated towards the end of the graphs where b reaches the maximum value it can get in each case, indicating the ideal value of b for each scenario. When examining the case with four weak attackers, the relative revenue:power ratio of the strong attacker is maximized when each weak attacker controls around 12~13% of the network's power. This observation is consistent throughout all scenarios that were simulated.

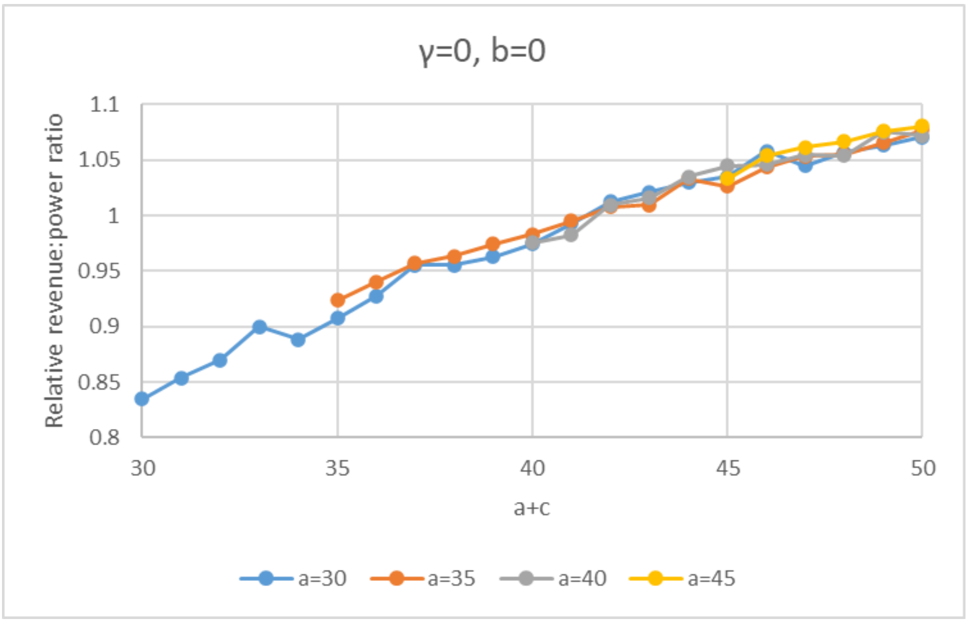
The explanation for the effect depicted above is simple. The decision of the small mining pools to follow a selfish mining strategy is unwise, since their mining power does not back up their behaviour, turning the odds against them and minimizing their expected revenue. Alice, who is the selfish miner with a considerable amount of mining resources, not only benefits from adopting the selfish mining strategy, but also exploits the other miners. When the small mining pools' power approaches zero, their activity in the network is almost irrelevant. As the variable b increases, the activity of the selfish miners who Alice might take advantage of increases, thus the advantage of Alice increases accordingly. This effect seems to be maximized towards the end of each graph, at about 12-13%. This is the point where the increase of the outsider's power does not make them more likely to be taken advantage of by Alice, but rather makes them more likely to be a worthy competitor against them. Although the nature of the experiment prevents this from actually happening, we know it is true, since in the experiments from section 5.2 where all selfish miners share the same power, Alice ends up having no advantage over the rest of the attackers.

The next effect we want to examine is the effect that c has on Alice’s revenue. Similarly to the previous tests, we created various scenarios and by using the increment of c we aimed to experimentally observe the effect that the eclipsed pool has on Alice’s revenue. Theoretically, we would expect that increasing the power of the eclipsed victim would have the same effect as increasing the power of the attacker themselves, since the blocks that the eclipsed miner finds are controlled by the attacker. The relative revenue:power ratio we use as a metric would be calculated by (the percentage of valid blocks mined)/(the percentage of power). The percentage of valid blocks mined by a mining pool with power percentage a_1 which controls an eclipsed victim with power percentage c_1 is expected to approach the relative revenue:power of a pool which has power $a_2 = a_1+c_1$ and does not control any eclipsed pools. This is true, since the two pools operate as one, with the only difference being that each pool claims the blocks they mined in the process. If they operate as one pool, we expect the total mined blocks from these two pools to be equal to those mined by a single pool controlling equal power to the pair’s total. We would then expect that the two pools claim their respective revenue from that total, with the percentage of blocks claimed approaching the percentage of power between the pair (e.g. 100 blocks mined from the pair, power of attacker=30, power of eclipsed = 10 would have us expecting 75 blocks for the attacker and 25 blocks for the victim). We could mathematically express this as follows, where expected(x) is the function which returns the expected relative revenue power ratio for a mining pool with power percentage x, for the scenario that we are investigating:

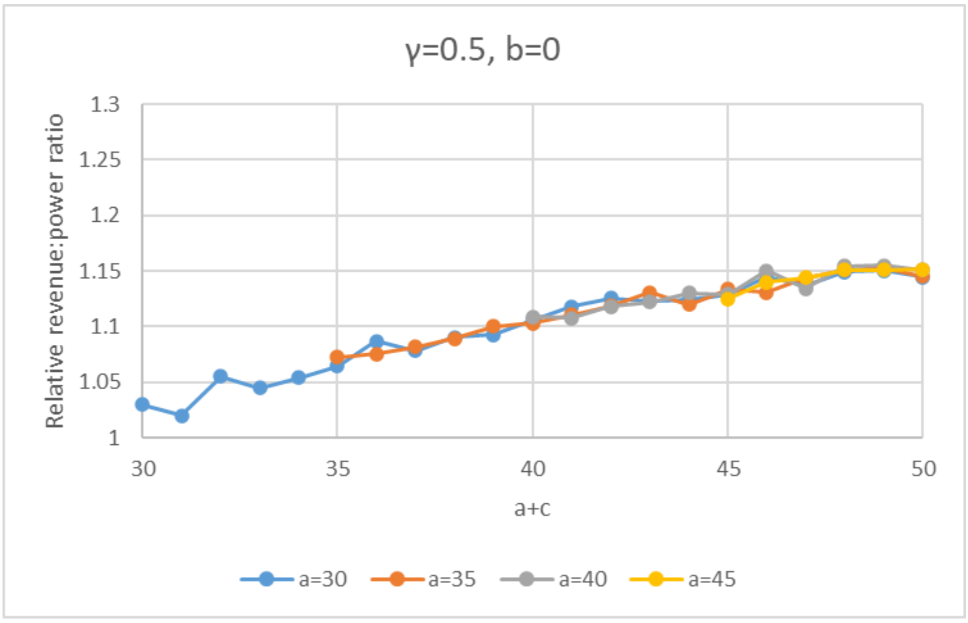
$$\begin{aligned}&\text{Rel\_rev\_power\_ratio}_1\\&= \text{expected}(a_1+c_1)*[a_1/(a_1+c_1)]/a_1\\&= \text{expected}(a_2)/a_2\\&= \text{Rel\_rev\_power\_ratio}_2\end{aligned}$$

To generalize the above, for each set of valid variables which satisfy $a_1+c_1=a_2+c_2$:

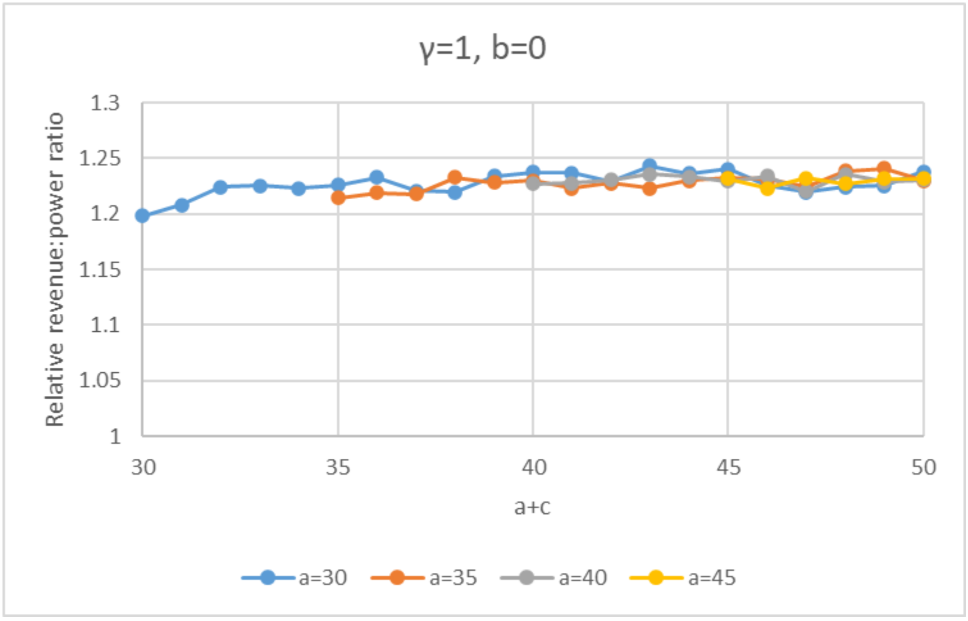
$$\begin{aligned}&\text{Rel\_rev\_power\_ratio}_1\\&= \text{expected}(a_1+c_1)*[a_1/(a_1+c_1)]/a_1\\&= \text{expected}(a_2+c_2)/(a_2+c_2)\\&= \text{expected}(a_2+c_2)*[a_2/(a_2+c_2)]/a_2\\&= \text{Rel\_rev\_power\_ratio}_2\end{aligned}$$



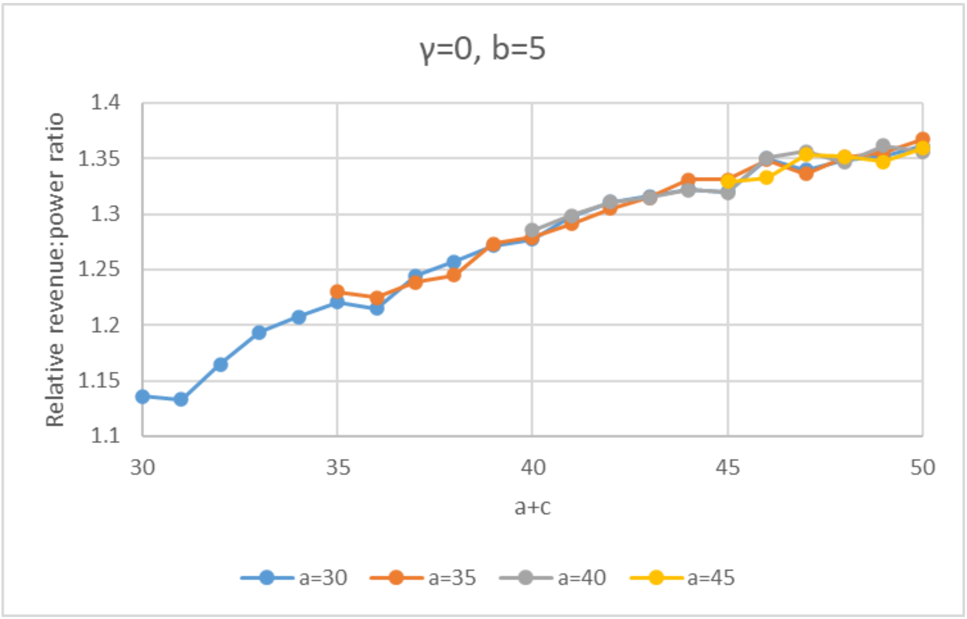
γ=0, b=0
Relative revenue:power ratio
1.1
1.05
1
0.95
0.9
0.85
0.8
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45

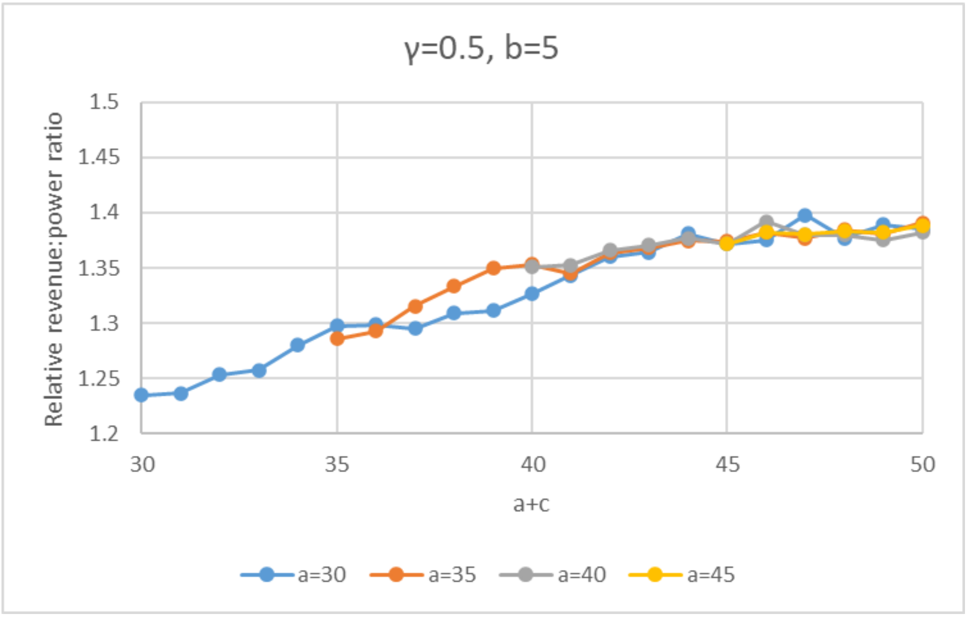


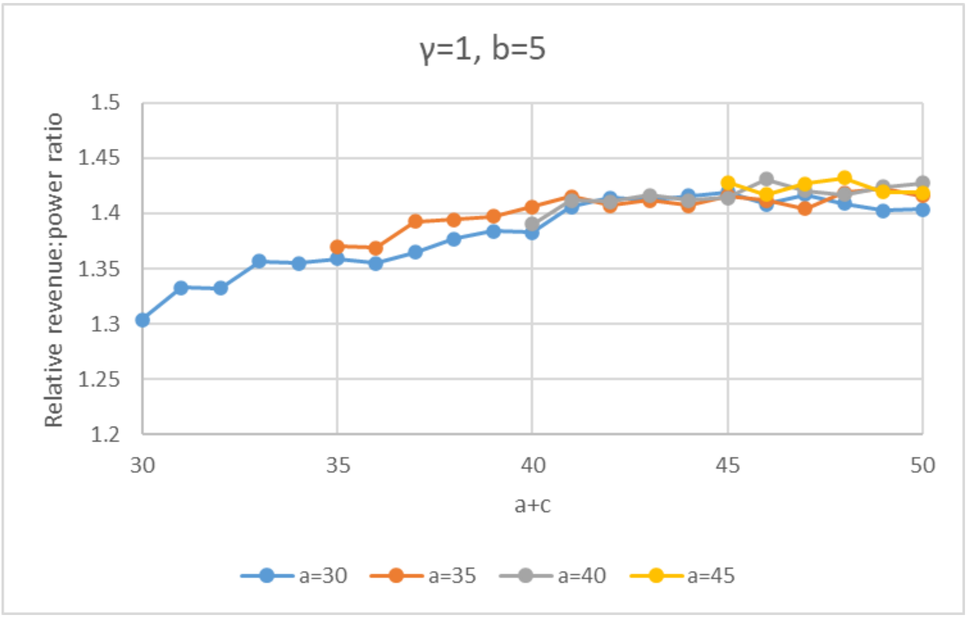
γ=0.5, b=0
Relative revenue:power ratio
1.3
1.25
1.2
1.15
1.1
1.05
1
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45

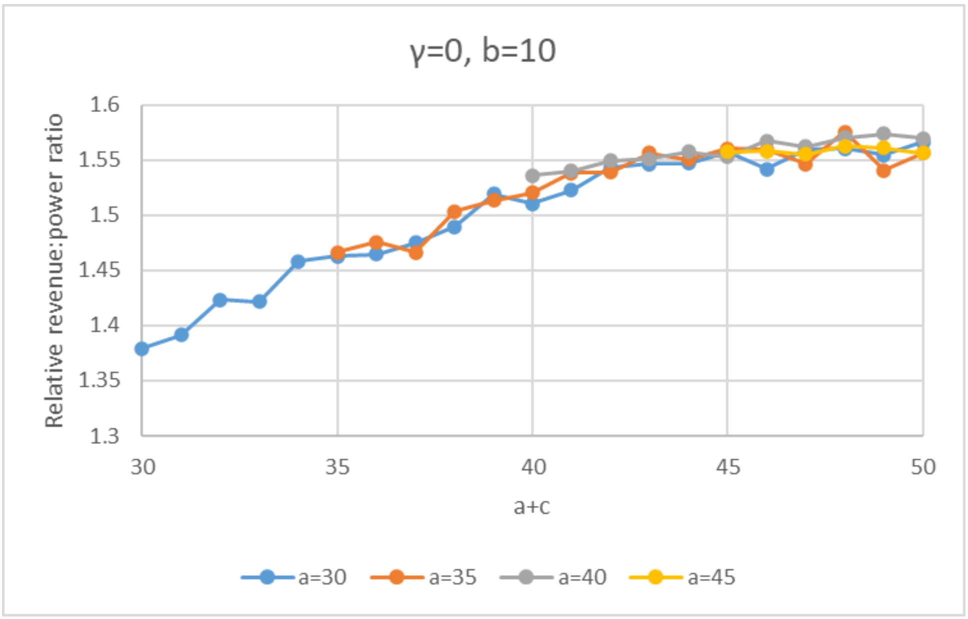


γ=1, b=0
Relative revenue:power ratio
1.3
1.25
1.2
1.15
1.1
1.05
1
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45

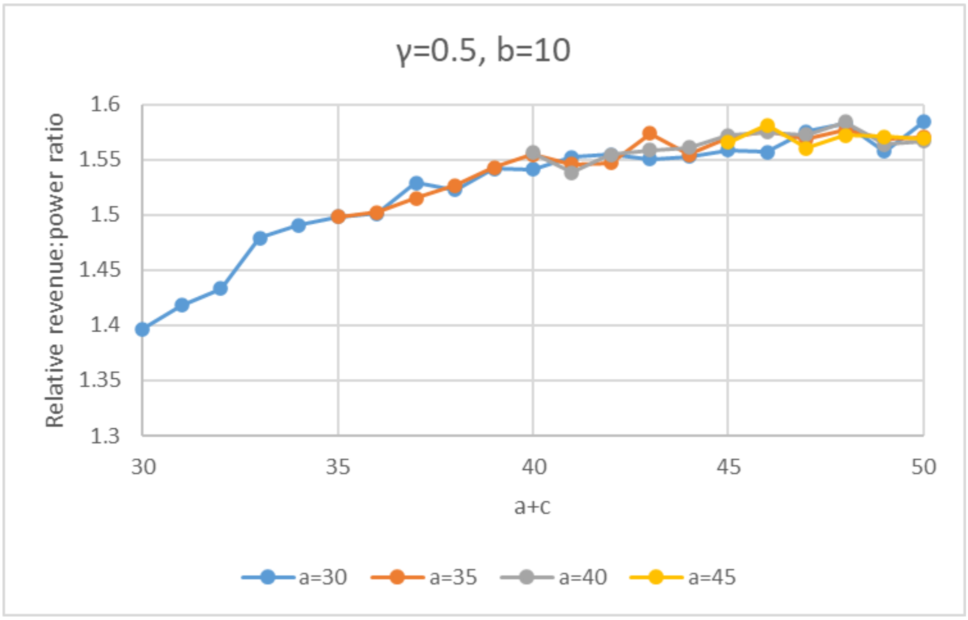

γ=0, b=5
Relative revenue:power ratio
1.4
1.35
1.3
1.25
1.2
1.15
1.1
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45


γ=0.5, b=5
Relative revenue:power ratio
1.5
1.45
1.4
1.35
1.3
1.25
1.2
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45

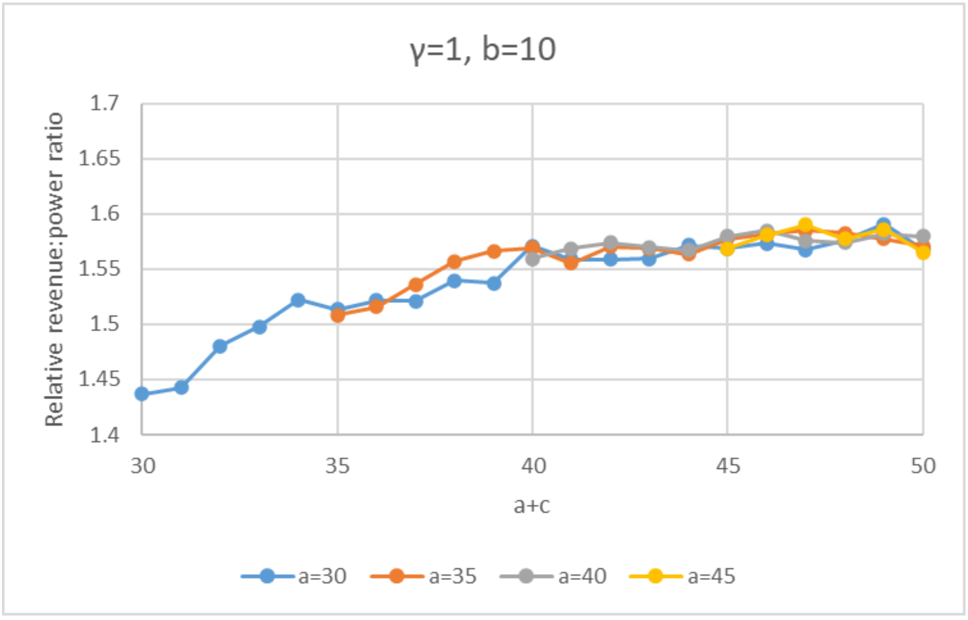

γ=1, b=5
Relative revenue:power ratio
1.5
1.45
1.4
1.35
1.3
1.25
1.2
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45



γ=0, b=10
Relative revenue:power ratio
1.6
1.55
1.5
1.45
1.4
1.35
1.3
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45



γ=0.5, b=10
Relative revenue:power ratio
1.6
1.55
1.5
1.45
1.4
1.35
1.3
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45



γ=1, b=10
Relative revenue:power ratio
1.7
1.65
1.6
1.55
1.5
1.45
1.4
30
35
40
45
50
a+c
a=30
a=35
a=40
a=45

By examining the results, we can observe that the theoretical framework has accurately predicted the shared properties of a and c. This finding implies that, for practical purposes, the relative revenue:power ratio can be accurately calculated by considering the sum of a and c as a single variable. As we increase either a or c, there is a corresponding increase in the ratio, highlighting their direct and proportional impact. This relationship between a and c and the ratio is impacted by the values of γ and b. By observing the charts, the pattern is visible in all scenarios, but there are cases where effect of a and c on the relative revenue:power ratio is reduced. When examining all available information and details throughout the various scenarios, we were able to conclude that this phenomenon occurs due to the fact that γ has a significant impact in the equilibrium where two strong attackers compete against each other, thus reducing the effects of a and c to a certain extend. This is evident when comparing the case where γ=0 and b=0 to the case where γ=1 and b=0. We can see that in the former the pattern is a linear-like increase which starts from a ratio of 0.835 and ends at 1.07, while in the latter the difference is barely noticeable, with the ratio starting from 1.20, peaking at 1.245.

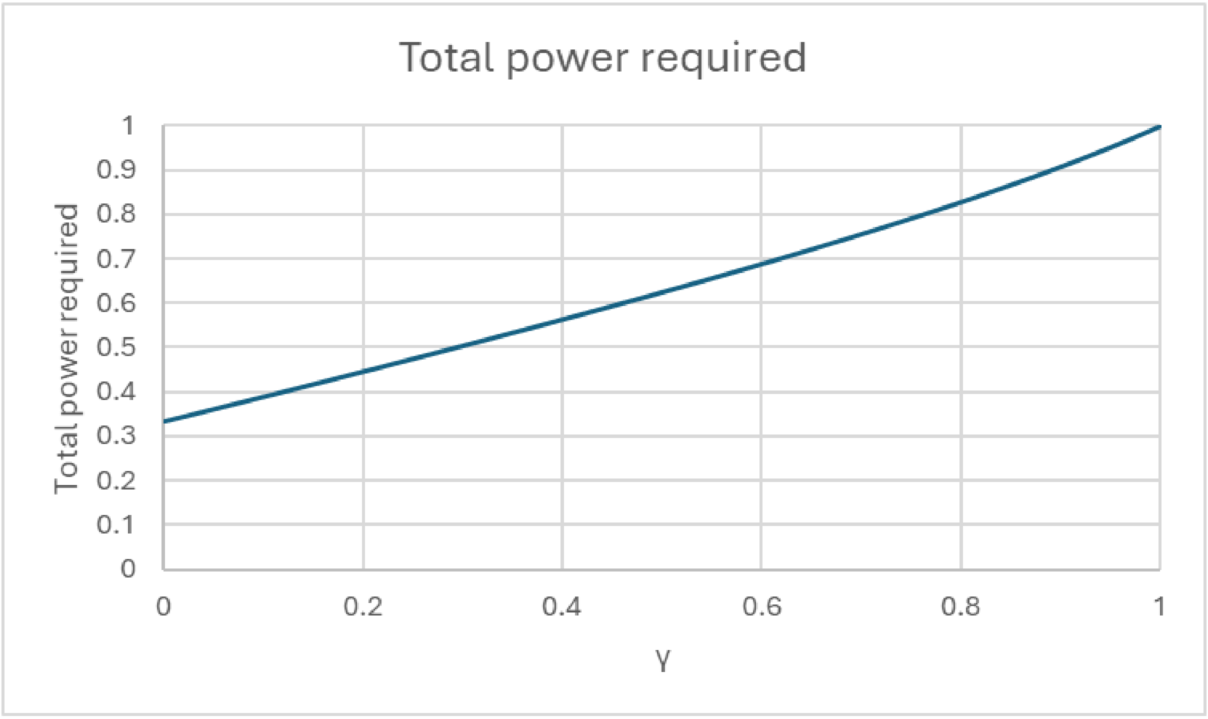
5.4 Realistic Scenarios with Profitable and Damaged Attackers

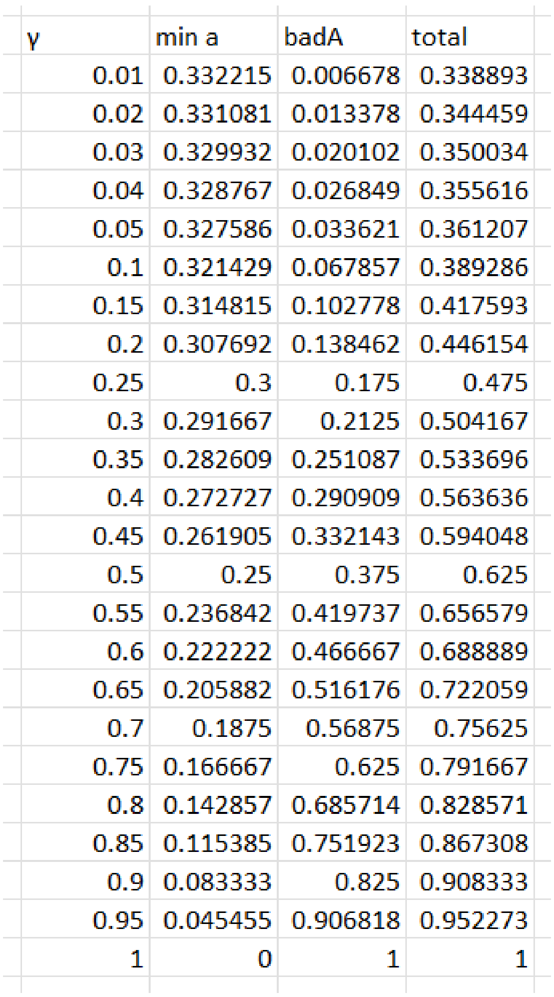
So far, we have experimented with the system to enhance our understanding of the network and how each parameter influences the income of Alice. However, the previous scenarios lack the element of realism. In reality, if a group is not expected to generate more income, they will simply not adopt this strategy. However, we can use information from previous sections to develop some specific test cases with multiple attackers, where each attacker has an expected ratio of income to power greater than 1. We can observe how an attacker would either benefit from the existence of other attackers or suffer an unexpected loss.

First of all, we have to define what a realistic scenario is, meaning the conditions a situation has to meet in order to classify as realistic. We have already mentioned that the sum of the pool and their eclipsed victim’s power percentage has to be below the threshold of 50% of the total power, since anything above 50% is considered a 51% attack. Another criterion is that the total power of each attacker allows them to expect a higher revenue by behaving selfishly rather than honestly, based on the findings of [2], since we assume that attackers do not know that other attackers exist in the network. Cases where miners assume the existence of other miners can be found in [9] where it was proven that the threshold of minimum power for profitability decreases with the addition of new attackers in the network.

We will approach the challenge of unveiling these realistic scenarios mathematically. From [2], we notice that a bigger γ helps the mining group, however γ_i=0.5 means that that the miner 'i' controls honest miners with at least 25% of the total power of the network, which is not particularly helpful when designing a profitable scenario. A logical step taken to resolve this obstacle was the reduction of the γ_i variable for each miner. The total power percentage required to make an attacker profitable, when no other attackers are present in the network, is presented below, both as a graph, as well as in table form, specifically indicating the percentage which the attacker directly controls ('a'), as well as the power percentage of honest miners that the attacker controls (γ*(1-α)).

As we can observe from the data presented above, the minimum power required for each selfish miner in order to start gaining unfair levels of profit is around 33~34%. This essentially means that the only case we would ever get 3 attackers behaving selfishly is depicted in section 3.2 with the triple equal competition. Recall that in that case the three attackers' revenue:power ratio is approximately 1, since they all play an equal game, thus the properties are mirrored, leading to a tie between the three attackers.

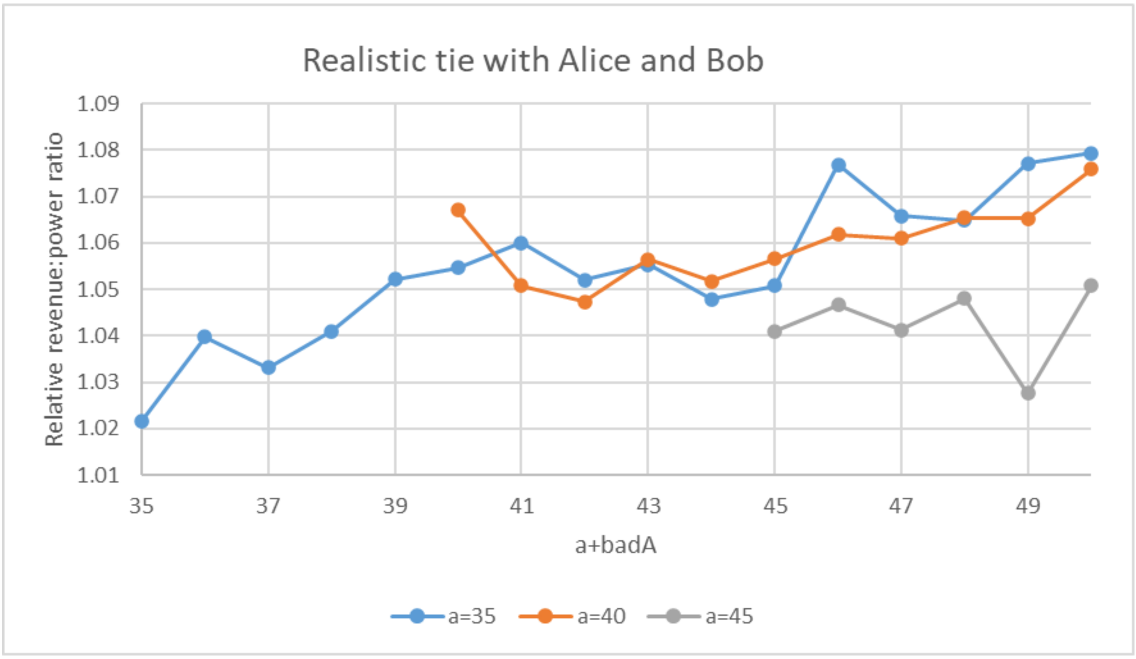




γ	min a	badA	total
0.01	0.332215	0.006678	0.338893
0.02	0.331081	0.013378	0.344459
0.03	0.329932	0.020102	0.350034
0.04	0.328767	0.026849	0.355616
0.05	0.327586	0.033621	0.361207
0.1	0.321429	0.067857	0.389286
0.15	0.314815	0.102778	0.417593
0.2	0.307692	0.138462	0.446154
0.25	0.3	0.175	0.475
0.3	0.291667	0.2125	0.504167
0.35	0.282609	0.251087	0.533696
0.4	0.272727	0.290909	0.563636
0.45	0.261905	0.332143	0.594048
0.5	0.25	0.375	0.625
0.55	0.236842	0.419737	0.656579
0.6	0.222222	0.466667	0.688889
0.65	0.205882	0.516176	0.722059
0.7	0.1875	0.56875	0.75625
0.75	0.166667	0.625	0.791667
0.8	0.142857	0.685714	0.828571
0.85	0.115385	0.751923	0.867308
0.9	0.083333	0.825	0.908333
0.95	0.045455	0.906818	0.952273
1	0	1	1

Since the scenarios where three attackers exist in the networks are limited and they do not significantly influence the expected revenue of the attackers, let us examine some realistic cases with two attackers instead. Using the possible combinations, we are led to 3 categories of such possible scenarios. The first category regards scenarios where Alice and Bob face each other on equal terms, having equal portions of power, while controlling the same amount of honest miners. The second category is similar to the first, the two pools control the same amount of power, however Alice holds more power within their pool, while Bob controls more honest miners to make up for their handicap. The third type of scenarios is as one would expect, the cases where Alice has a clear advantage over Bob.

When examining the first type of realistic scenarios we want to evaluate whether the two groups are able to make a significantly larger amount of revenue than expected and at the same time cause damage to the rest of the network by reducing their income. We constructed simulations to take these results.



We ran some simulations which would realistically generate revenue for the attackers. Each point on the chart represents a simulation with the following set of variables:

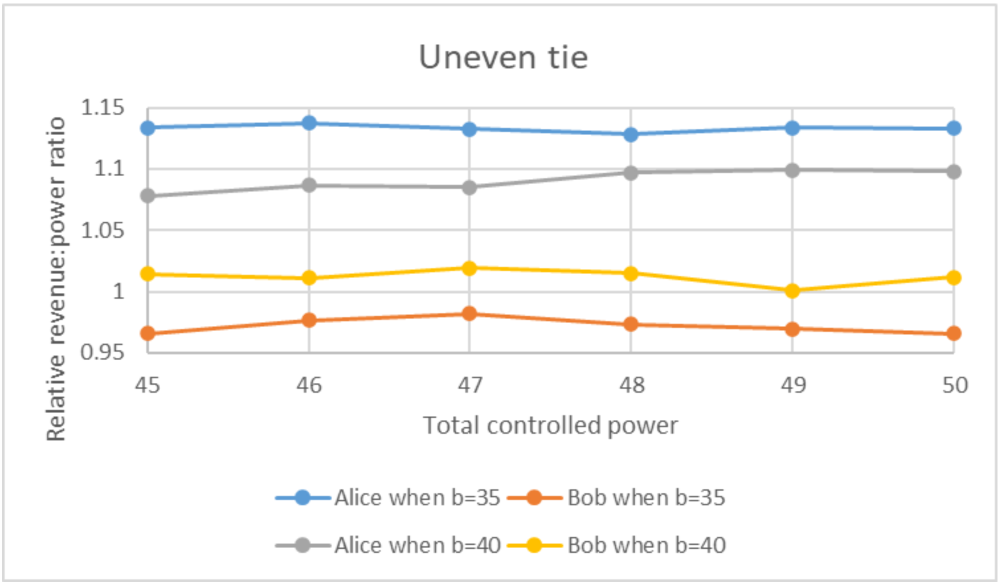
- a = Alice’s total power = Bob’s total power: is shown by the line
- badA = Alice’s controlled honest power = Bob’s controlled honest power: is equal to the value of the point’s x component minus ‘a’
- Unaffected honest miners’ power: 100-2*a-2*badA
- As for the eclipsed miners, according to the previous section we can assume that they are part of a.

The results verify that we can have two profitable attackers at the same time in a realistic scenario as defined previously. When analysing those we can see that having more power can be proven to be harmful in cases where the two pools end up controlling the same amount of resources. Specifically, from points with x values in [46-50] we can presume that in scenarios of ties the two pools would actually benefit from dismissing a small portion of their pool. While there exists the argument of how would that group behave once dropped from a selfish mining pool, we must not forget the possibility of eclipse attacks taking place in the network. We could assume that each attacker holds 35% of the power and has eclipsed a pool, with each eclipsed victim holding 10% of the total power. The behaviour of this network does not differ at all from a network where each attacker holds 45%, according to the previous section. However, the existence of the eclipsed pools unlocks the possibility of safe ‘riddance’.

By the term riddance, we mean that the attacker releases some of their controlled mining resources and forces them to behave as a new mining pool. Even though this seems paradoxical, in cases where attackers share a large portion of the networks' power, releasing some controlled miners to exploit them as an honest pool benefits the attacker. This action would be unwise when the mining resources were consciously part of the attacker, since they would have knowledge of the attacker's existence and therefore will be likely to report them or punish them by cooperating with other groups. However, when the pool that the attacker gets rid off is an eclipsed victim, this means that even though they terminate the eclipse attack they can still use them as a controlled honest mining pool, further exploiting the pool.

From the above derives a set of interesting questions, regarding the concept of riddance. A pool would have to know that they are involved in a situation where they have equal power with another attacker in order to proceed with dismissing their eclipsed victim. Even if the two pools eventually found out about each other's existence, unless they communicate directly with each other, the synchronization of the pools' riddance sequence poses a significant challenge. If the two pools communicated found a way to synchronize the release of their victims, would they be able to trust each other about being honest with their riddance if they were not acting honestly during mining in the first place? What would happen if Bob was deceived by Alice to dismiss their eclipsed victim as part of the riddance events, while Alice gets to keep their eclipsed miners?

The cases where an attacker has a higher amount of power than their opponent, while their opponent has more controlled honest miners belongs in the second category of realistic scenarios with two attackers. We simulated the case where a miner performs riddance while the other keeps their eclipsed victim and took the following results:

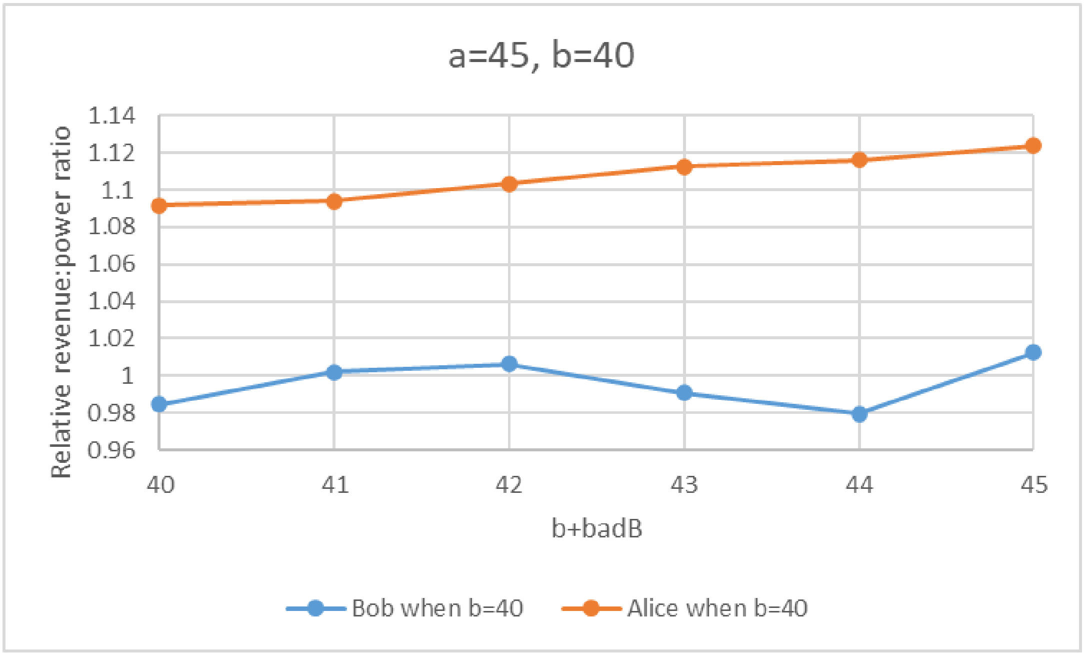


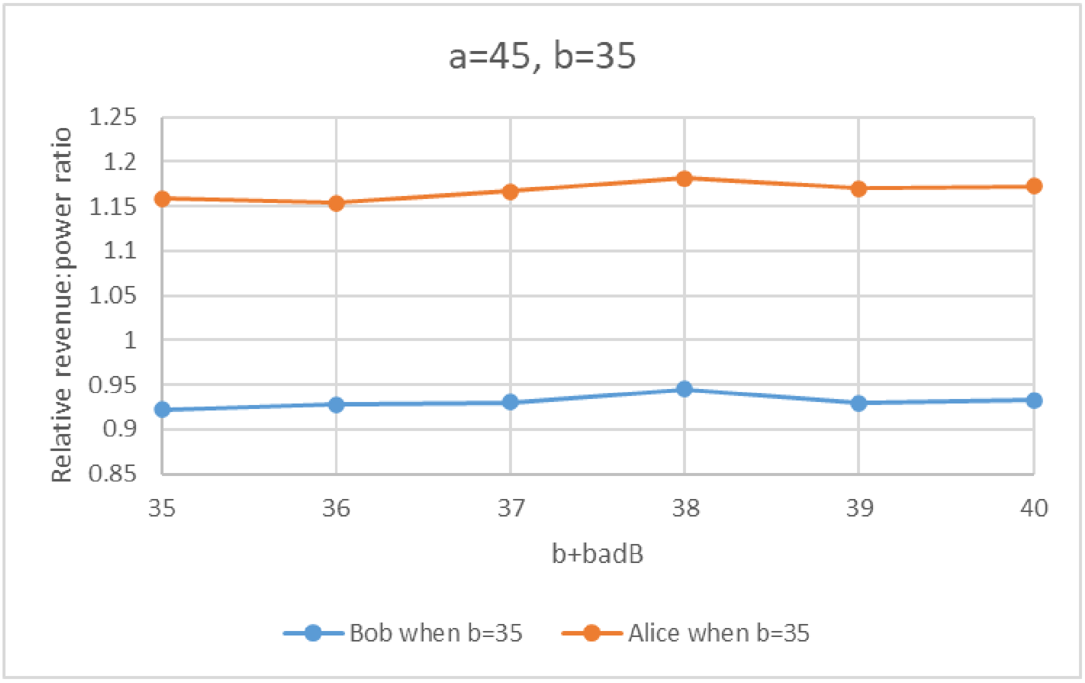
When speaking about the total controlled power we mean the sum of the pool's power, plus the power of their eclipsed victim and the power of their controlled honest pool. Each point represents a scenario where the two attackers had the exact same amount of resources, and Alice has deceived Bob into releasing their eclipsed pool but still uses them as a controlled honest pool, while Alice changes nothing. What one would expect in such a scenario was for Alice to increase their expected revenue and Bob to suffer a decrease in their expected income. As we can see this speculation has been backed up by the simulations. When given enough time, Bob can judge whether their decision was correct, by their relative revenue:power ratio; if it approaches 1 this means that they have been tricked by their competitor and it would be wise to retry eclipsing the pool they released. However, if they clearly generate more revenue than they should this means that both groups have let go of their eclipsed pools and can enjoy the revenue they make by sacrificing their victims, as displayed in the previous graph.

We used two different values for the power of the eclipsed pool which Bob is dismissing; the first value is 5% and the second is 10%. This lead to b=40 and b=35 respectively, since Bob was initially holding 45% of the power. In the first case, the pools are expected to have a profitable revenue:power ratio, with Alice's ratio being about 1.09, while Bob can expect a ratio of 1.02 in these scenarios. In the second case, only Alice can expect to be profitable with a ratio of 1.13, while Bob suffers a minor loss, by having a ratio smaller than 1, at about 0.97.

The results presented above offer a theoretical framework through which the two pools can communicate and synchronize their riddance. The two pools can either ignore, suspect, or know the existence of each other in the network. If a pool suspects the existence of the other attacker, they can opt to temporarily release their eclipsed victim, thus with enough time given, their competitor will eventually notice the change of their income and suspect the existence of another pool. If a time frame is reached and the second pool does not respond by releasing their eclipsed pool, the former pool can eclipse their victim once again and the revenues will be restored at their initial expected values from that point onward. This sequence of events will further act as an indicator of the existence of an attacker for the second pool, which can repeat the process if they are willing to cooperate with the other selfish mining pool. As mentioned earlier, dismissing a pool of will have an effect on both attackers; by increasing the size of the released pool, the attacker might risk losing some of their potential revenue, in order to have a greater impact on the other attacker's revenue, thus making it easier for them to notice the first attacker and cooperate with them.

When examining the occurrence of ‘riddance’ we must also look into cases where Alice not only does not release their eclipsed miner, but also attempts to steal their opponents released group as a controlled honest mining pool. The scenarios in which Alice succeeds belong in the third category, where a miner has a clear advantage over their opponent.





Each point in the above charts represents a different simulation, whose variables can be determined as follows:

- a and b are mentioned in the chart’s title
- badB is equal to the value of the point’s x component minus b

- badA is equal to badB + a – b. The reason why we set this is because we assumed scenarios were Bob and Alice shared the same mining power and controlled honest pools of equal size. Bob then released their eclipsed victim, leaving Bob with power b, however the victim (whose power is equal to a-b) is now controlled as an honest miner by Alice. Therefore badA = badB + (a-b).

In the case that Alice steals Bob's released pool of power 5%, their relative revenue:power ratio can range from 1.09 to 1.12, depending on the power of the controlled honest miners. At the same time, Bob's ratio drops to a range from 0.98 to 1.01, thus making the expected revenue of Bob fair to their power.

When Bob drops a larger pool of 10% and Alice manages to control the behaviour of that pool on forks, the revenue of each pool is dramatically changed. Alice's expected relative revenue:power ratio starts from 1.16 and can reach up to 1.18. On the other hand Bob is unable to make profit from such scenarios, since their ratio's max value is 0.95.

From the charts we can conclude that if Alice is able to fool Bob in order to steal their eclipsed victim and use them as a controlled pool, apart from the obvious increase of Alice's expected revenue, Alice can also cause damage to the other attacker, making their expected revenue:power ratio to approach 1, and even drop lower than that, depending on the size of the released pool. In the original case of tie, before the riddance of the eclipsed pool, the ratio for each pool was around 1.05, while in the case of cooperation and synchronized riddance their expected revenue:power ratio can reach 1.08.

The risk of facing such outcomes is arguably not worth the ratio's increase from 1.05 to 1.08, however there is a secure way of getting rid of an eclipsed pool, that neutralizes this risk. Instead of releasing the eclipsed pool, the attacker should use part of his resources to maintain the eclipse attack, making sure that the victim can not be used against them. At the same time, they should terminate the collusion with their victim. By terminating the collusion, the attacker passes all of the eclipsed pool's outgoing and ingoing information, causing the target to act as an honest mining pool, while still controlling them, thus making sure that their victim can not be used against them.

Chapter 6

Conclusions

6.1 Multiple Miners Conclusion 51
6.2 Realistic Application Findings 52
6.3 Future work 55

6.1 Multiple Miners Conclusion

Bitcoin [1] and its vulnerabilities to malicious strategies such as selfish mining [2] and eclipse attacks [4] has gained significant attention and has been extensively studied. A variety of research efforts have explored these threats and their implications. These include [3] which examined different strategies along with an eclipse attack and work [5] which compared and assessed the dominant strategies. However, the concern over networks with multiple attackers has particularly worried the scientific community. In [9] the success of multiple selfish mining pools in the same network has been examined, while in [10] the power threshold for two attackers was found to be smaller for each pool than the threshold of one single attacker.

One of the purposes was to examine the effect that each variable has on the relative revenue of the attackers. To achieve this, we conducted experiments that simulated two types of scenarios. The first scenario was the competition of equal selfish mining pools against each other, whereas the second had a designated superior pool which was expected to dominate the network.

The first scenario did not offer many possibilities for us to experiment with different combinations of variables, since the attackers rarely made profit in such circumstances. Our results confirmed the findings of [9] which suggested that pools will benefit from merging with one another, since the revenue results for many attackers were smaller compared to revenue results for fewer attackers. We also viewed that controlling pools can benefit the attackers, with eclipsed mining pools having better results than controlled honest mining pools. Nevertheless, the conclusion was that in scenarios where pools have the exact same properties, the pools can benefit only in cases that were predicted by [9] and the proposed common beneficial area.

We then moved on to the next type of simulations which included a stronger attacker and four attackers who shared the same properties. In these scenarios we tested how the power of the minor attackers affects the rewards of the main attacker, as well as how an eclipse attack would help the attacker. We found that the presence of weaker attackers in the network benefits the favourite pool, which supports the findings of [9] regarding the existence of a weaker attacker in the network. We also viewed the impact that the power of the weaker attackers had on the revenue of the stronger attacker as a continuous function, reaching a maximum towards the end of its possible range. This can be explained due to the importance of the small mining pools in the network. The results indicate that extremely weak mining pools have insignificant effect on the main attacker’s revenue, and as their power grows, their activity increases, therefore the level of advantage that the attacker gains from the smaller mining pools increases as well. This trend seems to slow down towards the end of the range of this variable, where the “outsiders” start growing to a level where they become more of a threat to the protagonist, rather than a tool to increase their revenue. In the scenarios studied using four weak attackers, the revenue of the main attacker was maximized when each weak pool controlled about 12-13% of the network’s total power, while the main attacker’s power ranged from 30% to 50%.

Another aspect of the environment that we examined was the variable c which indicates the power of the eclipsed group that the main attacker controls. Analysis of our data lead to the realisation that when the attacker colludes with their victim, they act as one united pool, with each pool gaining the revenue which corresponds to the total mining power they offer to the greater union. Therefore, the total power of the two pools can act as a single value which can predict their expected revenue in each scenario. Further examination of the results indicated that the total power that the attacker controls is positively correlated to the rewards they claim, with each scenario being described by this relationship to its own level; scenarios with high γ tend to reduce the effect that a and c have on the expected revenue, compared to scenarios with low values of γ.

6.2 Realistic Application Findings

The previous experiments were particularly useful in explaining the relationship of the expected revenue:power ratio with each variable, however, they lack an important aspect –

realism. There is no reason for a miner with low amount of mining resources to mine selfishly, since research on the subject has shown that they are not expected to profit from such behaviour. Previous studies have shown that the threshold of minimum power in order to be profitable diverges with the addition of new attackers in the network [9], however, it does not make sense for the pools to attempt mining selfishly with hopes of others doing so, when their power is not enough to predict profitability in the absence of other selfish mining pools. In other words, we can not accept that a mining pool would attempt selfish mining, unless they fulfill the requirements for profitability presented in [2]. With this strict definition, we are practically left with two attackers, since the only other case is 3 attackers with 1/3 of the network's power each, ultimately leading to a tie between them.

The game involving two attackers was divided in three categories of scenarios: (a) the two attackers control the same power, (b) the two attackers control the same power but Alice has more direct control, while Bob controls more honest miners and (c) Alice has a clear advantage. We started by examining the scenarios where the two attackers are equal. Those indicated that there are cases where two attackers would gain more revenue if some of the power they controlled was neither a part of the pool nor an eclipsed miner, but rather a controlled honest pool. This occurs due to the domination of the network by the two pools, leaving small margin for them to take advantage of other pools.

At this point we introduced the term "riddance", which is the act of releasing a small portion of the pool in the network as a new independent pool in order to exploit them. This practice is unsafe when it involves releasing miners which belong in the pool and have knowledge of the attacker's selfish mining strategy. However, the existence of eclipsed mining pools makes this practice possible, as they can be released in the network, while still being manipulated by the attacker as a controlled honest pool. If the two attackers communicate successfully and perform this "riddance" simultaneously they can increase their expected relative revenue:power ratio.

However, an attacker might attempt to release their eclipsed attacker while the other does not, for whatever reason. This scenario belongs to the wider spectrum of the second category, where Alice directly controls more power than Bob, while Bob controls his previously eclipsed victim as an honest pool, making the total controlled power of the two pools equal. The results in these scenarios are close to one would expect; the relative revenue:power of Alice increases while Bob's relative revenue decreases. The greater the size of the pool released, the larger the change of the rewards for each pool. This effect could be utilized by the pools for synchronization purposes; The two attackers can be in a variety of states when it comes to the knowledge of each other. The two pools might not be aware of the presence of another attacker in the network, or depending on their rewards they might suspect about it. In order to communicate their existence which will lead to the acknowledgement of each other, and possibly to their cooperation, the suspecting pool will release some of their eclipsed miners, in order to cause this effect described above. The other pool will eventually be able to recognise the existence of the other attacker at which point they will be presented with two options; cooperate with the other attacker by dismissing their eclipsed miners or ignore the request.

The last scenario we examined was the scenario where Alice has a clear advantage, and to be specific we used the previous cases where Bob got rid of their eclipsed mining pool. In the scenarios examined, Alice attempted to steal the eclipsed pool from Bob and use them as their controlled mining pool instead. In those scenarios, the results were as predicted – Alice increased their revenue even more, while Bob reached a point where their relative revenue:power ratio was marginally greater than 1 at best. The revenue of Alice and, respectively, the damage that Bob suffered varied with the amount of miners Bob released, as well as the power that Bob controlled as honest mining pools.

The risk of losing the released pool can be averted by not actually terminating the eclipse attack. Instead of releasing the pool, the attacker allows the pool to communicate with the rest of the network, thus acting as a normal honest mining pool. This lets the attacker control the pool in cases of forks, while eliminating the risk of the pool being stolen by the other attacker.

In conclusion, we confirmed the findings of previous studies and explored the impact of weaker attackers on the revenue of stronger attackers. We found that as the power of smaller selfish mining pools increases, the main attacker benefits from higher revenue. However, when the power of these weaker attackers grows too significant, they start to pose a threat to the main attacker. Additionally, colluding with an eclipsed mining pool has the same effect as directly controlling their power within the pool. In scenarios involving two attackers, the mutual release of their eclipsed pools can help them increase their relative revenue:power ratio. However, terminating the eclipse attacks can pose a threat to the attacker, as they risk losing the control of the pool to the other attacker. Instead of releasing the pool, the attacker should maintain the eclipse attack, making sure that the other attacker can not steal their victim. At the same time, they will allow the target to communicate with other pools, so that they act as a controlled honest mining pool, which would increase each attacker’s revenue, if the two attackers manage to cooperate.

6.3 Future work

Future work related to this subject could take a variety of directions. One direction it could potentially take is the inclusion of Stubborn mining strategies as they were presented in [3]. The comparison between those for two players was presented in [8], however this could be expanded by combining elements of our work and [8] in a new study.

The concept of “riddance” could be further explored in future work, as there exists a variety of things to study. Future work may include the further analysis of this concept, diving deeper in this idea, by rigorously examining all possible scenarios where two pools might benefit from it. What could also be interesting and useful is the concept of synchronization between attackers. The idea that two attackers could communicate their existence to each other should be further looked into, since the proposed synchronization technique is just a theoretical framework that practically faces many obstacles, such as the required time that it would need to work effectively, as well as the safety of it, since the actions of the attackers also influence the revenue of the rest of the network. Another way that eclipse attacks could be studied in this context, is the strategy of the attacker. In this thesis we focused on the ‘collusion’ strategy, but the scenarios could be expanded by examining the ‘destroy the victim’ strategy, which makes the target appear offline.

Finally, the exploration of this subject on other blockchain implementations such as Ethereum’s could be of use, since the two cryptocurrencies share basic similarities but are also quite different when it comes to their details; their key difference is that Ethereum uses the Proof-Of-Stake consensus mechanism instead of Proof-Of-Work which has been implemented in our model. It also utilizes the concept of “uncle” and “cousin” rewards which mitigate damage in forks.

References

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, 2008.

[2] 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.

[3] 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.

[4] Atul Singh, Tsuen-Wan "Johnny" Ngan, Peter Druschel and Dan S. Wallach. “Eclipse Attacks on Overlay networks: Threats and Defenses” Ngan et al., Infocom, 25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies. IEEE, 2006.

[5] Andreas Tsouloupas, “Conservative Stubborn Mining: Extending, modeling and verifying selfish mining strategies on Bitcoin,” Undergraduate Thesis, University of Cyprus, January 2021.

[6] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen. “Uppaal SMC Tutorial,” International Journal on Software Tools for Technology Transfer volume 17, pages 397–415, 2018.

[7] UPPAAL, “https://uppaal.org/”

[8] Panos Marinou, “Modeling and analyzing selfish and stubborn mining strategies on Bitcoin with multiple attackers” Undergraduate Thesis, University of Cyprus, January 2023.

[9] Shiquan Zhang, “Analyzing the Success of Selfish Mining with Multiple Players” Master Thesis, McGill University, June 2020

[10] Qianlan Bai, Xinyan Zhou, Xing Wang, Yuedong Xu, Xin Wang and Qingsheng Kong, “A Deep Dive into Blockchain Selfish Mining,” IACR Cryptol. ePrint Arch. 2018: 1084, 2018

Appendix A

UPPALL’s implementation of Selfish Miner is presented below:

```
// Length of Private Chain
int privateBranchLen;

//Boolean that shows that the selfish miner performed synchronization
bool indSync = false;
bool eclipsedMined = false;


//Track of each fork head
int publicHead;
int privateHead;

//Index of last public block that this pool mined and published or that was inserted in chain
int indexLastPublic;

//Variable to store difference between private chain and public chain (when a new block is announced)
int diffPrev;

//Variables to store various miners revenue at the end. Initialized at zero
int revenue = 0;
int revenueOthers = 0;
double revenueDbl = 0.0;
double revenueOthersDbl = 0.0;
double relativeRev = 0.0;

//Variable to store how many blocks to release
int release = 0;

```

```
//Variables used in Double Lead Tie breaker
int counter = 0;
bool tieBreaker = false;

//Array that will hold the mined blocks
Block chain[BLOCKLIM+BLOCKLIM/2];

//Initialize blockchain with genesis block
void initialize(){
    chain[0].blockID = 0;
    chain[0].prevID = -1;
    chain[0].hMiner = 0;
    chain[0].sMiner = 0;
    chain[0].length = 0;

    //Start private and public chains
    publicHead = 0;
    privateHead = 0;
    indexLastPublic = 0;

    //Initialize local variables
    privateBranchLen = 0;

    //if the miner has no hashpower then no synchronization is needed
    if(rate == 0){
        syns++;
        indSync=true;
    }
}

//Find the first empty spot in the private chain
int getChainFreeIndex() {
    int i , index = -1;
    for ( i = 0; i < BLOCKLIM + 1; i ++){
        if (chain[i].blockID == 0 && chain[i].prevID == 0){
            index = i;
            i = BLOCKLIM + 1;
        }
    }
    return index;
}

//Find the index of a given block in chain
int getBlockChainIndex(BlockID bID){
    int i, index = -1;
    for(i = 0; i < BLOCKLIM + 1; i++){
        if(chain[i].blockID == bID){
            index = i;
            i = BLOCKLIM + 1;
        }
    }
    return index;
}

//Find the index of the next block in chain
int getNextBlockChainIndex(BlockID bID){
    int i, index = -1;
    for(i = 0; i < BLOCKLIM + 1; i++){
        if(chain[i].prevID == bID && (chain[i].sMiner == id || chain[i].hMiner == 100+id)){
            index = i;
            i = BLOCKLIM + 1;
        }
    }
    return index;
}

//Add new block to private chain mined from selfish
void createAndAddPrivateBlock() {
    int index;
    Block b;

    index = getChainFreeIndex();

    if (index != -1){
        b.blockID = blockHash;
        b.prevID = chain[privateHead].blockID;
        b.hMiner = 0;
        b.sMiner = id;
        b.length = chain[privateHead].length + 1;

        chain[index] = b;
        if (blockHash == BLOCKLIM){
            outOfSpace = true;
        }
        else {
            blockHash++;
        }

        privateHead = index;
    }
    else {
        outOfSpace = true;
    }
}


void checkEclipsedBlock(){
    int eID=eclipsedID;
    if (eID == id+100){
        eclipsedMined = true;
    }
}

void addEclipsedBlock() {
    int index;
    Block b;
    index = getChainFreeIndex();
    if (index != -1){
        b.blockID = blockHash;
        b.prevID = chain[privateHead].blockID;
        b.sMiner = 0;
        b.hMiner = id+100;
        b.length = chain[privateHead].length + 1;
        chain[index] = b;
        if (blockHash == BLOCKLIM){
            outOfSpace = true;
        }
        else {
            blockHash++;
        }
        privateHead = index;
    }
    else {
        outOfSpace = true;
    }
    eclipsedMined=false;
}

//New block announced from public
void addPublicBlock() {
    int index, prevIndex, headLen;
    Block b;
    b = tempBlock;

    index = getChainFreeIndex();
    prevIndex = getBlockChainIndex(b.prevID);

    //check if error exist with previous block, the variable prevNotFound will become true
    if (prevIndex == -1){
      prevNotFound = true;
      return;
    }

    //check if error exist with previous block length, the variable wrongLength will become true
    if (chain[prevIndex].length + 1 != b.length){
      b.length = chain[prevIndex].length + 1;
      wrongLength = true;
      return;
   }

    //add block  to chain if it has longer length
    if (index != -1){
        chain[index] = b ;
        headLen = chain[publicHead].length;
        if(headLen < b.length){
            publicHead = index;
        }
    }
    else {
        outOfSpace = true;
    }

    //reset tieBreaker boolean when the miner loses
    if(tieBreaker){
       tieBreaker = false;
    }
}

//Reset private fork
void resetPrvChain(){
    privateHead = publicHead;
    indexLastPublic = publicHead;
    privateBranchLen = 0;
}

//Returns true if all the blocks in private chain were published
bool publishedAllBlocks(){
   return (indexLastPublic == privateHead) ? true : false;
```

```
}

//Publish the earliest block of the chain to the public
void publishOneBlock(){
    int nextIndex, currentID;

    if(!publishedAllBlocks()){
        currentID = chain[indexLastPublic].blockID;
        nextIndex = getNextBlockChainIndex(currentID);

        if(nextIndex != -1){
            tempBlock = chain[nextIndex];
            indexLastPublic = nextIndex;

            if(chain[publicHead].length < chain[privateHead].length && publishedAllBlocks())
                publicHead = privateHead;
        }
        else{
            nextNotFound = true;
        }
    }
}

//Calculate difference of private and public chains after a block is mined by selfish (true) or by honest (false)
int calculateDiff(){
    int diff;
    diff = chain[privateHead].length - chain[publicHead].length;
    if(diff < 0){
        diff = 0;
    }
    return diff;
}

//check whether the lock status should change at the end of the release
```

```
void checkLock(){
  if(release > 0){
    lock = true;
    slock = id;
  }
  else{
    lock = false;
    slock = 0;
  }
}

//Calculate how many blocks to be released depending on the calculate difference with the public chain
int calculateRelease(){
  //calculate number of blocks to release initially based on algorithm
  if(release == 0){
    if(diffPrev>2){
      release = 1;
      lock = true;
      privateBranchLen--;
    }
    else if(diffPrev==2){
      release = 2;
      lock = true;
      privateBranchLen = 0;
    }
    else if(diffPrev==1){
      release = 1;
      lock = true;
    }
    else {
      if(chain[privateHead].length < tempBlock.length){
        resetPrvChain();
      }
      release = 0;
    }
  }
  else {
     //if SM has to not release all his private blocks but he lost, now he must reset
     if(chain[privateHead].length < tempBlock.length){
       resetPrvChain();
       release = 0;
     }
         //if not lost, remaing to lock state so he continue releasing blocks based on algorithm
     else{
       lock = true;
    }
  }
  return release;
}

//Function to check if it is needed to release all blocks in case of tie with two blocks
void checkDoubleLeadTie(){
  if(release == 2){
    counter++;
    if(counter == 2){
      tieBreaker = true;
      counter = 0;
    }
  }
}

//Calculate revenue - number of blocks that were mined by selfish and mined by others
void ownRevenue(){
  int index = publicHead;
/*
   if (chain[publicHead].length<=chain[privateHead].length){
    index = privateHead;
    testFlag=true;
  }
*/
```

```
  while(index > 0){
    if(chain[index].sMiner == id){
      revenue++;
      revenueDbl = revenueDbl + 1.0;
    }
    else{
      revenueOthers++;
      revenueOthersDbl = revenueOthersDbl + 1.0;
    }
    index = getBlockChainIndex(chain[index].prevID);
   }
}

//Calculate revenue in percentage form
void getRevenue(){
  ownRevenue();
  relativeRev = revenueDbl/(revenueDbl + revenueOthersDbl);
}

//Check if the miner is synchronized after finishing the cycle
void checkSyn(){
  if(publishedAllBlocks() && !indSync){
    syns++;
    indSync = true;
  }
}
```

Appendix B

UPPALL’s implementation of Honest Miner is presented below:

```
//Head of Public chain
int publicHead;

//Array that will hold the mined blocks
Block chain[BLOCKLIM+BLOCKLIM/2];

//Variables to store various miners revenue at the end. Initialized at zero
int revenue = 0;
int revenueOthers = 0;
double revenueDbl = 0.0;
double revenueOthersDbl = 0.0;
double relativeRev = 0.0;

//Initialize blockchain with genesis block
void initialize(){
    chain[0].blockID = 0;
    chain[0].prevID = -1;
    chain[0].hMiner = 0;
    chain[0].sMiner = 0;
    chain[0].length = 0;
    publicHead = 0;
}

//Find the first empty spot in the public chain
int getChainFreeIndex() {
    int i, index = -1;
    for (i = 0; i < BLOCKLIM + 1; i++){
        if (chain[i].blockID == 0 && chain [i].prevID == 0){
            index = i;
            i = BLOCKLIM + 1;
        }
    }
    return index;
}

//Find the index of a given block in chain
int getBlockChainIndex(BlockID bID){
    int i, index = -1;
    for(i = 0; i < BLOCKLIM + 1; i++){
        if(chain[i].blockID == bID){
            index = i;
            i = BLOCKLIM + 1;
        }
    }
    return index;
}

//Add new block to public chain
void createAndAddNewBlock() {
    int index;
    Block b;

    // Find the place of the new block in the chain structure
    index = getChainFreeIndex ();

    // If index is not -1 it means there exists a place for the block in the chain
    if (index != -1){
        b.blockID = blockHash; // Create its block ID
        b.prevID = chain[publicHead].blockID; // Its previous block is the block with index 'publicHead' (not necessarily the same as the previous block index-wise in the chain array)
        b.hMiner = id; // The miner who added this block is an honest miner with ID = 'id'
        b.sMiner = 0; // This block was not mined by a selfish miner => sMiner = 0;
        b.length = chain[publicHead].length + 1; // the length is now equal to the length of the public head increased by 1.

        chain[index] = b; // Add the new block in the chain in place 'index'
        // If the ID of the block has reached the limit
        if (blockHash == BLOCKLIM){
            // We've ran out of space
            outOfSpace = true;
        }
        else {
            // Increase the variable blockHash by 1
            blockHash++;
        }

        // The head of the public chain is now the block in the place 'index'
        publicHead = index;

        // tempBlock is the new block which is going to be broadcasted to the rest of the mining pools
        tempBlock = b;
    }
    else {
        outOfSpace = true;
    }
}

//Process a block broadcasted in the network
void processBroadcastBlock(bool isBad){

    // GET THE NEW BLOCK THAT HAS BEEN FOUND
    int index, prevIndex, headLen;
    Block b;
    b = tempBlock;
    index = getChainFreeIndex(); // the index that the block will be placed at
    prevIndex = getBlockChainIndex(b.prevID); // The index of its previous block

    //check if error exist with previous block, the variable prevNotFound will become true
    if (prevIndex == -1){
      prevNotFound = true;
      return;
    }

    //check if error exist with previous block length, the variable wrongLength will become true
   if (chain[prevIndex].length + 1 != b.length){
     b.length = chain[prevIndex].length + 1;
     wrongLength = true;
     return;
  }

   //add block to the chain. if the miner is bad and block was received from the selfish that controls the miner will be added
   if (index != -1){
       // Add the new block in the blockchain in the place indicated by 'index'
       chain[index] = b ;

       // Headlen = the length of the public chain before adding the new block
       headLen = chain[publicHead].length;

       // Two reasons to change the public head which we are mining on:
       // a) The length of the new block's branch is greater than the existing head's length
          // b) The two lengths are equal, but the miner is a selfish miner and the honest mining pool has the same ID as the selfish miner (which means they mine on their block when forks exist)
       if((headLen == b.length && isBad && id == b.sMiner)|| headLen < b.length){
           publicHead = index;
       }
   }
   else {
       outOfSpace = true;
   }

}

// REVENUE
void ownRevenue(){
  int index = publicHead;

  while(index > 0){
    if(chain[index].hMiner == id){
      revenue++;
      revenueDbl = revenueDbl + 1.0;
    }
    else{
      revenueOthers++;
      revenueOthersDbl = revenueOthersDbl + 1.0;
    }
    index = getBlockChainIndex(chain[index].prevID);
   }
}

//Calculate revenue in percentage form
void getRevenue(){
  ownRevenue();
  relativeRev = revenueDbl/(revenueDbl + revenueOthersDbl);
}
```

Appendix C

UPPALL’s implementation of Eclipsed Miner is presented below:

```
//Head of Public chain
int publicHead;

//Array that will hold the mined blocks
Block chain[BLOCKLIM+BLOCKLIM/2];

//Variables to store various miners revenue at the end. Initialized at zero
int revenue = 0;
int revenueOthers = 0;
double revenueDbl = 0.0;
double revenueOthersDbl = 0.0;
double relativeRev = 0.0;

//Initialize blockchain with genesis block
void initialize(){
    chain[0].blockID = 0;
    chain[0].prevID = -1;
    chain[0].hMiner = 0;
    chain[0].sMiner = 0;
    chain[0].length = 0;
    publicHead = 0;
}

//Find the first empty spot in the public chain
int getChainFreeIndex() {
    int i, index = -1;
    for (i = 0; i < BLOCKLIM + 1; i++){
        if (chain[i].blockID == 0 && chain [i].prevID == 0){
            index = i;
            i = BLOCKLIM + 1;
        }
    }
    return index;
}

//Find the index of a given block in chain
int getBlockChainIndex(BlockID bID){
    int i, index = -1;
    for(i = 0; i < BLOCKLIM + 1; i++){
        if(chain[i].blockID == bID){
            index = i;
            i = BLOCKLIM + 1;
        }
    }
    return index;
}

//Add new block to public chain
void createNewBlock() {
    eclipsedID = id;
}

//Process a block broadcasted in the network
void processBroadcastBlock(bool isBad){

    // GET THE NEW BLOCK THAT HAS BEEN FOUND
    int index, prevIndex, headLen;
    Block b;
    b = tempBlock;
    index = getChainFreeIndex(); // the index that the block will be placed at
    prevIndex = getBlockChainIndex(b.prevID); // The index of its previous block


    //check if error exist with previous block, the variable prevNotFound will become true
    if (prevIndex == -1){
      prevNotFound = true;
      return;
    }

    //check if error exist with previous block length, the variable wrongLength will become true
    if (chain[prevIndex].length + 1 != b.length){
      b.length = chain[prevIndex].length + 1;
      wrongLength = true;
      return;
    }

    //add block to the chain. if the miner is bad and block was received from the selfish that controls the miner will be added
    if (index != -1){
        // Add the new block in the blockchain in the place indicated by 'index'
        chain[index] = b ;

        // Headlen = the length of the public chain before adding the new block
        headLen = chain[publicHead].length;

        // Two reasons to change the public head which we are mining on:
        // a) The length of the new block's branch is greater than the existing head's length
            // b) The two lengths are equal, but the miner is a selfish miner and the honest mining pool has the same ID as the selfish miner (which means they mine on their block when forks exist)
        if((headLen == b.length && isBad && id == 100+b.sMiner)|| headLen < b.length){
            publicHead = index;
        }
    }
    else {
        outOfSpace = true;
    }

}

// REVENUE
void ownRevenue(){
  int index = publicHead;

  while(index > 0){
    if(chain[index].hMiner == id){
      revenue++;
      revenueDbl = revenueDbl + 1.0;
    }
    else{
      revenueOthers++;
      revenueOthersDbl = revenueOthersDbl + 1.0;
    }
    index = getBlockChainIndex(chain[index].prevID);
   }
}

//Calculate revenue in percentage form
void getRevenue(){
  ownRevenue();
  relativeRev = revenueDbl/(revenueDbl + revenueOthersDbl);
}
```