b
UNIVERSITÄT
BERN

Implementing and Evaluating Protocol Π3 in BlockSim

Bachelor Thesis

Lawrence Chiang
from
Köniz, Switzerland

Faculty of Science, University of Bern

23. June 2023

Prof. Christian Cachin
Orestis Alpos, Ignacio Amores-Sesar
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

A common attack on decentralised exchanges is the so-called sandwich attack. It involves a malicious miner placing a victim transaction between two of their own transactions in a block. The attacker frontruns and then backruns the victim, resulting in a profit for the attacker and worse exchange rates for the victim. Alpos, Amores-Sesar, Cachin, and Yeo propose a new protocol called 3 to prevent such attacks. In this thesis, we implement a simplified version of the protocol in BlockSim, a blockchain simulator written in Python. The simulator is also extended to support sandwich attacks and all changes are shown as pseudocode. Benchmarks demonstrate that the protocol is highly effective in preventing sandwich attacks if all participants of the protocol stay honest. Even if they collude, results show that 99.97% of attacks from October 2022 would have been unprofitable under the protocol. Attacks from the rest 0.03% are still profitable because the amounts transferred in these attacks are large. However, 3 also significantly reduces the profits of such attacks.

Acknowledgements

I would like to thank my supervisors Orestis Alpos and Ignacio Amores-Sesar for explaining every part of their protocol whenever anything is not clear to me. The almost weekly meetings with Orestis made me feel supported during the writing of this thesis and kept me motivated. I would also like to thank Prof. Christian Cachin for letting me write this thesis in his research group and for providing valuable feedback.

Contents

1 Introduction **2**

2 Background **3**
2.1 Blockchain . . . 3
2.2 Automated Market Maker (AMM) . . . 3
2.3 Sandwich Attacks . . . 4
2.4 3 Protocol . . . 5
2.5 BlockSim . . . 7

3 Design **9**
3.1 Design choices . . . 9
3.2 Sandwich attack simulation . . . 11
3.3 AMM Simulation . . . 12

4 Implementation **14**
4.1 Implementation of 3 in BlockSim . . . 14
4.2 Partial Seeds Commitments and Permuting . . . 15
4.3 Chunking . . . 16
4.4 Implementing the new incentives . . . 16

5 Evaluation **18**
5.1 Benchmarks of protocol with honest leaders . . . 19
5.2 Benchmarks of the protocol with colluding leaders . . . 19

6 Conclusion **22**

Chapter 1

Introduction

A rich person wants to buy a lot of ETH with their dollars. They go to the most used decentralised exchange that executes the transaction. But when the transaction is executed, they received less ETH than they expected. Little did they know, they have been subjected to a *sandwich attack*. Such attacks are a form of *Maximal Extractable Value (MEV)*, which is when miners maximize their gains beyond the block reward and transaction fees [17]. MEV has cumulatively cost more than 675 million dollars [11].

There are groups developing solutions to the MEV problem, some by encrypting the transactions or having a trusted third party bundle the transactions for miners to mine [17]. These solutions have some drawbacks as encryption may impact scalability without completely solving the MEV problem [10] and utilizing a trusted third party diminishes the trustless nature of blockchains. The 3 protocol, proposed by Alpos, Amores-Sesar, Cachin, and Yeo [2], not only makes sandwich attacks no longer profitable, but requires few resources and does not require a third party. It can be implemented on all types of blockchains and the security of the underlying blockchain is preserved.

Before a miner can start mining a new block, they must fill a new block with new transactions. Miners have the ability to choose which transactions to put in their block and also in what order. However, this freedom becomes problematic as malicious miners can exploit it to strategically position their transactions to gain a profit. 3 revokes the miner's ability to choose the order of the transactions. It works on the consensus level, namely after a block is mined. During delivery is where the protocol first comes into play: all transactions of the block are *chunked*. This is a process where the protocol replaces each transaction with many copies of the same transaction. Each copy, or *chunk*, is altered to transact a fraction of the original transaction value in relation to the amount of chunks. It is made sure that if the values of all chunks are added up, then the summed values is the same as the original value transacted by the original transaction. E.g. if 10 chunks are made for each transaction, then the values transacted on each chunk is one tenth of the original transaction. The protocol then randomly permutes all chunks. As a result of the chunks being randomly ordered, it is often the case that a frontrunning chunk from the attacker comes before or after a backrunning chunk from the attacker. When this happens, there is no victim chunk between these chunks, so there is no profit for the attacker. An hypothesis from Alpos, Amores-Sesar, Cachin, and Yeo is this leads to lower profits for the attacker. Another hypothesis states that as the number of chunks increases, the profit for attackers approaches zero. Two plots were taken from their paper, which can be seen in Figure 2.5. These plots depict the average profits of attackers with different behaviours [2]. The goal of this thesis is to replicate these two plots with data collected from an extended version of BlockSim and also to verify the two aforementioned hypotheses.

Chapter 2 provides the background of sandwich attacks, the protocol, and BlockSim. Chapter 3 shows the design choices made for the implementation and outlines the simulation framework that was used to simulate sandwich attacks. In Chapter 4, the pseudocode of the implementation of 3 is presented and explained. Chapter 5 showcases the results of various benchmarks which illustrate the performance of the protocol under normal conditions, as well as the profits obtained by actors not abiding with the rules of the system. Finally, the thesis ends with a conclusion in Chapter 6.

Chapter 2

Background

2.1 Blockchain

Blockchain is the technology behind cryptocurrencies like Bitcoin and Ethereum. It is a decenteralised, public ledger which is usually used to keep track of how much tokens each user has. Users of a blockchain own accounts which are associated with the number of tokens the user has. Users can send an amount of tokens to other users by broadcasting a transaction on the internet.

The blockchain is maintained by people called *miners*. These people put transaction data into a list called a *block*. In Bitcoin, the list is hashed in a process called mining. Miners compete against each other and the first person to find a certain hash is said to have successfully mined the block. If this is the case, the protocol delivers the block to other miners. The block is added to the preexisting chain of blocks and the mined block is usually accepted by all other miners with time. These blocks then form a chain, hence the name *blockchain*. The miner is rewarded with tokens and transaction fees. After a block has been mined, the miner collects more transactions into a new block to mine and the cycle begins again [6].

Some blockchains such as Ethereum allow code to be run on the blockchain with a functionality called *smart contracts*. Smart contracts, just like users, can own an amount of tokens. Users of the blockchain can interact with the code. Every interaction with a smart contract changes the state of the blockchain at every node [19].

2.2 Automated Market Maker (AMM)

Blockchains like Ethereum gave form to *Decentralised Exchanges* (DEX). DEXes utilise the smart contract functionality of blockchains like Ethereum to run code on the nodes of the network. They let users exchange one amount of a token for another token without the need of a central authority. An advantage of using DEXes instead of a centralised exchange is users they do not have to worry about their funds being stolen by the people running the exchange.

The simplest implementation of a DEX is an *Automated Market Maker (AMM)*. An AMM usually possesses a huge amount of a token A and another equally huge amount of a token B. Let $amountA$ be the amount of token A and $amountB$ be the amount of token B the AMM has under its disposal. When an AMM is initialised, a constant k calculated by multiplying $amountA$ and $amountB$. At all times, $amountA$ multiplied by $amountB$ must equal k.

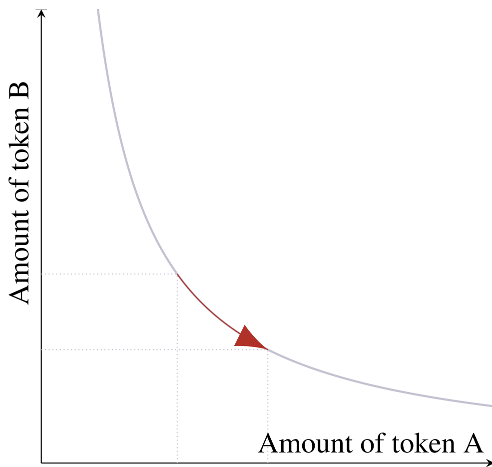
If a user wants to swap an amount of token A for token B, the user must send the amount of token A to the AMM. Upon receiving the token A, the smart contract adds the amount of token A from the transaction to $amountA$. As the $amountA$ has increased but $amountB$ has not, the constant k has not been respected. As a result, the smart contract calculates the amount of token B it must send back to the user so that k is restored. The amount of token B is calculated in the following way:

$$amountB = amountA = k$$

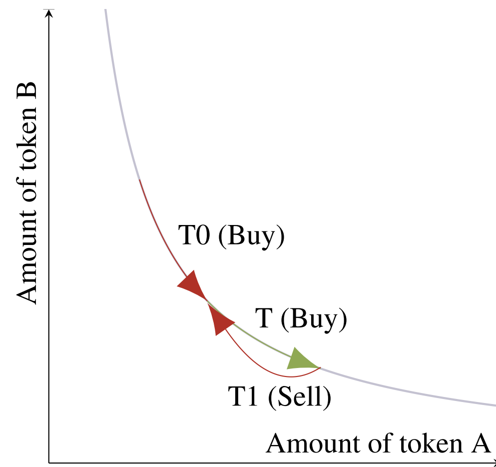
The AMM then sends the user the exact amount of token B so that k is restored. The same principle is used if the user wants to exchange an amount of token B for token A. As k is fixed, the amount of token A is inversely proportional to the amount of token B at any given time. As a result, a curve like Figure 2.1a can be plotted to represent states of an AMM. The exchange rate between the tokens is determined by the state of the AMM at the time of the transaction. [4]. A vulnerability of an AMM is big transactions can change the internal state by a lot and malicious actors can take advantage of this.

Figure 2.1. AMM

(a) The curve shows the possible states of the amounts of tokens under the control of the AMM. The red arrow is the change of state of the AMM because a user swapped token A for token B.



(b) The three arrows depict the changes of the state of the AMM caused by the three sandwich attack transactions. The two red arrows depict transactions of the attacker and the green arrow is the transaction of the victim.



2.3 Sandwich Attacks

A prerequisite for sandwich attacks is a user who wants to exchange a big amount of a token A for a token B. They broadcast a transaction T, which is the victim transaction of the sandwich attack. As transactions of a blockchain are usually public and the code of an AMM can be read by anyone, anyone can calculate how a given transaction would affect the constant of a given AMM [14]. An attacker, who is a malicious miner, may see an opportunity for a profit. The attacker adds T to their new block alongside two transactions of their own, $T0$ and $T1$: $T0$ in the position before T and $T1$ in the position after T. Transaction $T0$, just like T, exchanges an amount of token A for token B. $T1$ exchanges an amount of token B for token A. The victim transaction is sandwiched between the two attacker transactions, hence the name sandwich attack (see Figure 2.2, top left).

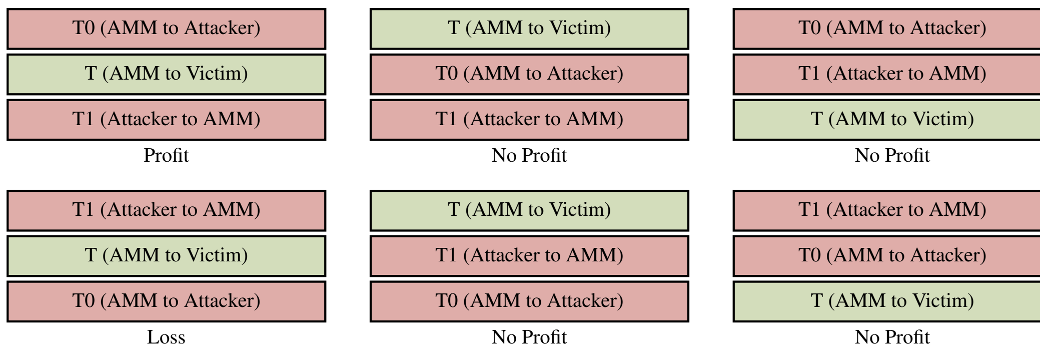
If the block is mined, the AMM processes the transactions sequentially, so $T0$ is processed before T, and T is processed before $T1$. $T0$ exchanges token A for token B, so the price of token B goes up, otherwise the constant k of the AMM is violated. This is the frontrunning portion of the attack. Transaction T, as stated above, also exchanges token A for token B. Because T follows $T0$, the exchange rate for T is worse than had there been no $T0$. As for transaction $T1$, the attacker sells token B after the price of token B has increased because of transactions $T0$ and T. Thus the attacker sells token B for a better exchange rate than before $T0$ and T have executed. This is the backrunning portion of the attack [12]. The transactions $T0$, T and $T1$ are depicted on the AMM curve in Figure 2.1b.

2.4 Π3 Protocol

For the scope of this thesis, 3 has been simplified to four main components: Permuting, Partial seeds commitments, Chunking and Incentivising. The following paragraphs will explain each component.

Permuting The main mechanism of the protocol is the random permutation of transactions. The three transactions in a sandwich attack can be scrambled in six different ways, see Figure 2.2. Only when the transactions are in the normal order ($T0$, T, $T1$) does the attacker gain a profit. If both transactions from the attacker are switched ($T1$, T, $T0$) the attacker incurs a loss. In the other four cases, there is no victim transaction between the attacker transactions to move the state of the AMM. As a result, the attacker neither gains a profit nor suffers a loss. Because the likelihood of each permutation is the same, the average profit for an attacker is around zero. This claim is evaluated in Section 5.1. To achieve the random permutation, randomness is required. Generating random numbers on a blockchain is problematic because blockchains are by their nature deterministic whereas non-determinism is required for randomness [16]. An off-chain pseudorandom number generator cannot be relied on as the single source of randomness because its outputs can be predicted or it involves a trusted party. Hence, malicious miners can predict the next permutation and apply the inverse of the predicted permutation on the transactions. The protocol permutes the inversely permuted transactions and the transactions are ordered the way the attacker wanted. This is the reason a random number generator with a random seed as an input is required.

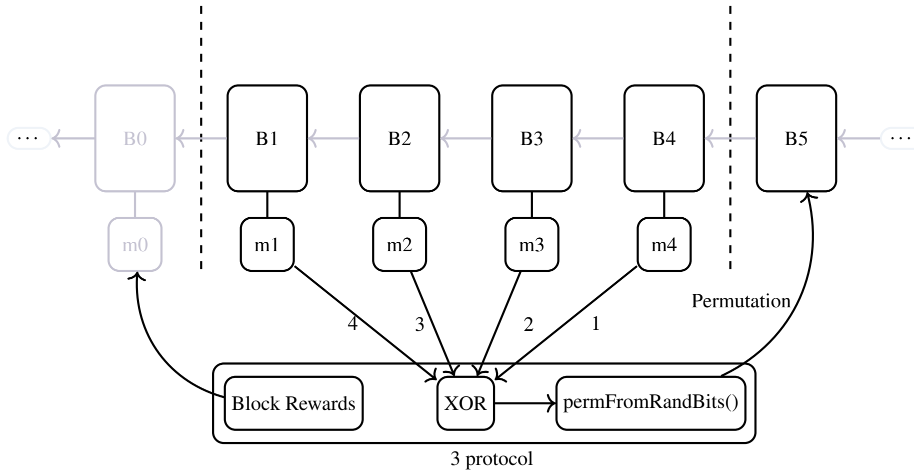
Figure 2.2. The six permutation possibilities of sandwich attack transactions. The order the attacker intended is on the top left. It is the only permutation where they get a profit. The worst permutation for the attacker is bottom left, because they incur a loss. The other permutations are neutral for the attacker.



Partial Seeds Commitments If the seed for the permutation of the transactions is solely provided by the miner of the newest block, then the miner can apply the inverse of the permutation on the transactions of their block before the block is delivered. On delivery, the protocol permutes the transactions in the tampered order to the order that the miner wanted and the sandwich attack is executed. Thus more actors must be involved to provide more randomness in the form of partial seeds. Let nl be a positive integer that is the number of actors needed. 3 finds actors in the last nl miners of the last nl blocks. The nl miners are the *leaders* and together they build the *leader set*. When a new block is mined, the miner of the block becomes a new leader and the oldest leader is no longer part of the leader set. The protocol requires all leaders to generate nl partial seeds. Each seed is hashed and the hashes are added to the blockchain as so-called *commitments*. The *commitment opening phase* is the period of time for leaders to reveal their seeds so the seeds can be used. The leaders are each obligated to reveal a seed with the property that if the seed is hashed, then the hash is the same as the prior commitment from the leader. This is called the *commitment scheme* and is done to prevent the leaders from later changing their seeds to bias the permutation. If the leader cannot or refuses to do what is required from them, they are penalised. The revealed partial seeds are merged together using the XOR function to form the final seed, which is what

is used to permute the transactions of the newest block of the chain (see Figure 2.3). The commitment scheme and the commitment opening phase have not been implemented in this thesis because they do not alter the results of the benchmarks run in Chapter 5.

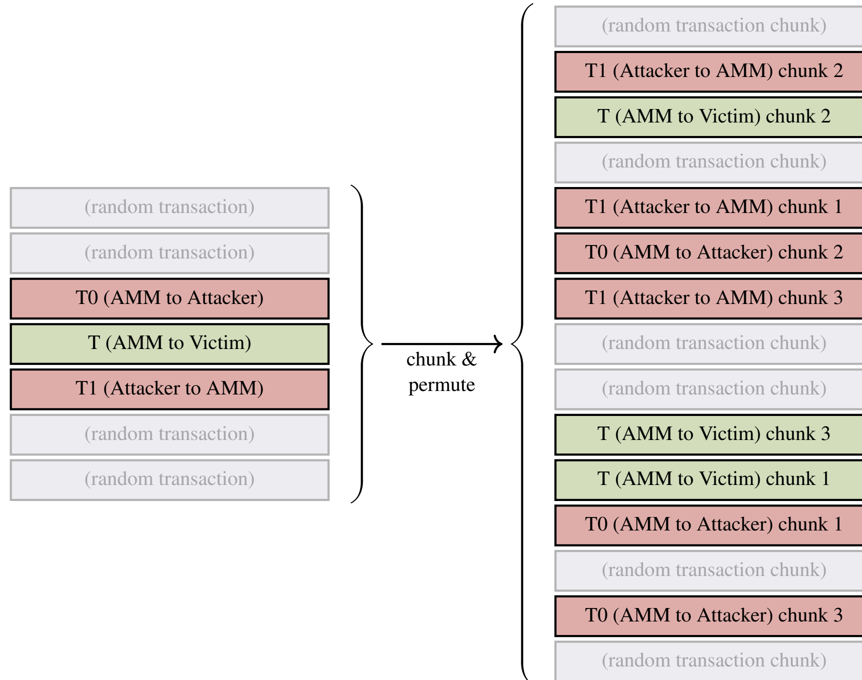
Figure 2.3. The top row of boxes is the blockchain with blocks *B0* to *B5*. Connected to the blocks *B0* to *B4* are the miners *m0* to *m4*. The four miners between the dashed lines are the leaders. The symbol denotes the partial seed published by each of the four leaders. The seeds are XORed and used in permFromRandBits(), a permutation function. The resulting permutation is applied to the transactions of the new block *B5*. On the left, miner *m0* is not a leader any more and receives their block reward.



Chunking The process of chunking is dividing each transaction in a block into many smaller transactions. This is done because the central part of 3 is the utilisation of a permutations and in order to achieve more possible permutations, all transactions are *chunked* on delivery. Let m be a positive integer that is the number of chunks set for the protocol. In this thesis, this number will also be referred to by *chunk size*. Transaction T, a transaction from the block, instantiates m number of chunks, all identical to T except for the amount transacted. Each chunked transaction is treated as an individual transaction sending a *m-th* amount of the original amount of ETH in T. For example if m is 10, a transaction T is split into ten distinct transactions $T0$, $T1$, ... , $T8$, $T9$. The value transacted in each of the chunks is a tenth of the original transaction T. If the total number of transactions is n, then chunking allows for $(n \quad m)!$ permutations, which is much bigger than the original $n!$. A visualisation of chunking and permuting can be seen in Figure 2.4. Chunking does not affect normal transactions from one person to another because the amounts transacted in the chunks add up to full amount transacted in the whole original transaction. However, if a person is transacting with an AMM, the permuted chunks change the state of the AMM by less overall compared to the original unchunked T. Chunking reduces the probability of a profitable sandwich attack as we will see in Chapter 5.

Incentivising To incentivise all leaders to abide by the rules of the protocol and commit truly random partial seeds, the block rewards and transaction fees of each block are temporarily held in custody. If miner $M1$ mined block $B1$ and $M1$ fails to reveal its seeds for the blocks $B1 + nl$, $M1$ loses the block reward for $B1$. If $M1$ is malicious and wants to successfully execute a sandwich attack, they must know the seeds of all other leaders in the leader set. This is because even if a large coalition of leaders collude, if one leader stays honest and provides a random seed, then the end seed is also random, by virtue of the XOR function. To further disincentivise leaders from colluding, the protocol allows leaders to claim the

Figure 2.4. On the left is a visualisation of a block containing a sandwich attack. On the right is the same block after chunking with the amount of chunks set to 3 and then permuted. The result is a list of transactions three times as large and randomly permuted. The end state of the transactions of a block when 3 is used has been reached. For better visualisation some chunks of the random transactions have been omitted.



rewards of other leaders if a leader guesses the partial seed of another leader. Here one can see the bigger nl is, the more secure 3 is, because the chance of a leader defecting is bigger and the block reward is held for longer. However, a trade off is miners must wait longer to receive their block rewards. Alpos, Amores-Sesar, Cachin, and Yeo [2] have shown that the protocol is sound in a game theoretical sense. To come to such a conclusion, the effect of colluding leaders was analysed. In the paper all sandwich attacks in October 2022 were analysed and it was determined that 99.97% of sandwich attacks had a profit of 6.37 ETH or less. The highest recorded profit of a sandwich attack in October 2022 was 109 ETH. For both of these values, plots of the probability of profitable permutations with different amounts of chunks (m) have been created. Figure 2.5a is the plot in their paper for 6.37 ETH vast majority case and Figure 2.5b for the 109 ETH edge case. The aim for this thesis is to recreate these plots with the simulator and generate more plots of the profits of leaders that are honest or collude. The results can be seen in Chapter 5.

2.5 BlockSim

BlockSim [1] is an open-source blockchain simulator written in Python that is designed for easy extensibility to test new ideas. It functions as a discrete event simulator with events accompanied by timestamps. In BlockSim, the energy-intensive mining process involving hashing is substituted with a probability distribution that is modeled after a miner's likelihood of mining a block at a given time. This is calculated by comparing the miner's hash power relative to the available hash power on the network. To simulate mining, a random number is chosen on the distribution that depicts the time that the node needed to mine a block. Many networking problems of real blockchain systems have also been abstracted away in a similar fashion. To simulate propagation time of blocks and transactions, similar probability distributions have been implemented. As a result of these design choices, simulations that are run on different computers with different hardware specifications give similar outputs.

Figure 2.5. The probability of profitable permutations with regards to chunk size (m). Both graphs are from the paper from Alpos, Amores-Sesar, Cachin, and Yeo. It is the goal of this thesis to recreate the graphs using BlockSim.

(a) Potential profits lost was 6.37 ETH

(b) Potential profits lost was 109 ETH

The simulator can be set to a few different modes like Base, Bitcoin, Ethereum and Appendable block. If Bitcoin or Ethereum are chosen, different variables of simulated blockchain are set to match constants the real life counterparts, such as block size and block propagation delay. If one wants to change these values, one can go to the configuration file of BlockSim, which is the *InputsConfig* file. In the file one can set parameters such as the number of miners, also called nodes, and their hash power. The block interval time, the block propagation delays, the block and transaction sizes, the block rewards, the transaction fees etc can also be set.

When running BlockSim, it first initialises nodes, which each have their id, hash power and their own internal blockchain. Each node generates initial block events and their own genesis block for their internal blockchain. In BlockSim, events are objects and are added in a list called event list. As long as there are events in the event list and the simulation time has not ended, BlockSim sorts the events by event time and executes the most recent one. An event can either be a *generate_block* event or a *receive_block* event. If the event is a generate_block event, BlockSim creates transactions for a block and propagates the block to other nodes. After this is done, a new generate_block event for the miner is added in the events list. Other nodes receive the block with receive_block events. They each see if the block is built on the last block of their internal blockchain. If that is the case, the block is added to the internal blockchain of the miner. If the depth of the received block is higher than the length of the miner's internal blockchain, then the miner updates its internal blockchain to match the longer chain.

At the end of the simulation, BlockSim looks for the longest internal blockchain of each node and calls it the global chain. This global blockchain, alongside the number of blocks mined and the rewards gained by each miner and other useful information from the simulation are printed in an excel file. What is printed in the file can also be easily changed to fit one's needs.

Chapter 3

Design

3.1 Design choices

Python was chosen for the implementation of the simulation of 3 because it is a widely used programming language and it is often used to create pseudocode examples. BlockSim [1] was chosen because it is written in Python and is easily extensible.

The mode of the simulator was set to Ethereum which causes BlockSim to build models of real past Ethereum transactions. The models are sampled to simulate more realistic transactions with regards to transaction size, gas fees and gas prices. To avoid the lengthy retraining process of the models every time the simulation is initiated, the models for the Ethereum transactions were saved. Such changes were made on BlockSim to speed up the development process. For more efficient debugging, the excel output has been extended so that it contains the transactions of all blocks concatenated. A sheet was added in the excel to list the profits of the sandwich attacks of all blocks, alongside the average and standard deviation of the profits. This is done for a speedy analysis of the sandwich attacks.

A conscious effort was made to keep the implementation lean and unbloated so the simulation runs efficiently. To avoid overhead, BlockSim was extended to contain one sandwich attack per block. Another example of prioritizing run speed is the choice between *LightTransactions* and *FullTransactions*, which is the choice of how complex the transaction objects are in BlockSim. The main difference lies in if all nodes can see the same transactions at any given time. If FullTransactions is enabled instead of LightTransactions, each node has its own pool of transactions that it is aware of. Each transaction object must be propagated to all other nodes. In LightTransactions all nodes see the same transactions, namely the transactions in a list of the LightTransactions object. LightTransactions instead of FullTransactions was chosen because there was no need for such detail of normal transactions as the main focus of the project were the sandwich transactions. This choice also makes it makes the simulation faster. A new variable to increase the amount of chunks after each run was added. The change was made so simulations testing different chunk sizes can be run back to back.

3 Implementation in BlockSim As described in Section 2.5, event objects in BlockSim can either be a generate_block event or a receive_block event. If the event is a generate_block event, then the generate_block() function of BlockSim is called. 3 is implemented in the function because it, alongside receive_block(), is the part of BlockSim that most closely resembles a live blockchain system. This is so that the simplified version of the protocol mimics the real protocol better. A drawback of implementing in generate_block() is because there is a sandwich attack in each block, when the blockchain forks and the fork is resolved, a sandwich attack is thrown away.

Sandwich attack simulation In this thesis we simplified sandwich attacks by adding a constraint: the amount of ETH transacted in each of the sandwich transactions *T0*, *T* and *T1* is the same. There are two ways to identify if a transaction is a sandwich transaction: the *id* variable of the transaction object is either 1, 2 or 3 and either the *sender* or *to* attribute is set to the string "amm". To ensure the sandwich

transactions are included in the block, the three transactions are given above average gas fees to ensure that they are executed before all other transactions. To ensure that all three transactions stay in the same order, *T0* must execute before *T* and *T* must execute before *T1*. Thus the gas fee of *T0* is set higher than *T* and the gas fee for *T* is set higher than *T1*. To keep track of the profits of each sandwich attack, a sandwich object is instantiated each time an attack takes place. These objects each have a *pro t* attribute that gets updated over the course of the simulation. The sandwich objects are then collected and used for analysis.

AMM Implementation Whenever a sandwich object is instantiated, an AMM object is also instantiated. The starting amounts of the tokens in the AMM are set to certain amounts that can be configured in the InputsConfig file. This is done instead of a single persistent AMM object handling all sandwich attacks. For cleaner data, each sandwich object interacts with an AMM with the same starting state. This is done so a sandwich attack that has been executed in the past does not affect the profits of another sandwich attack that is executed in the future. The AMM is modeled after the Uniswap USD/ETH smart contract, which is one of the most used DEXes in the world [7]. As a result, ETH and USD is the token pair used. The amounts of ETH and USD are easily settable in the configuration file. Fees for using the AMM were abstracted away as they are small amounts.

Permuting design Bacher, Bodini, Hwang, and Tsai present the algorithm FYKY which takes a list as an argument and returns the list with its elements randomly permuted [3]. In the paper from Alpos, Amores-Sesar, Cachin, and Yeo and this thesis, the function is renamed to *permFromRandBits()* and is used to permute transactions. Note that the paper from Bacher, Bodini, Hwang, and Tsai uses different indexes for elements of a list compared to this thesis [3]. In the original 3 protocol the miner gives only a random seed which is fed into a pseudorandom generator to generate a random binary string. The implementation in this thesis pretends that the seed has already been put into a random number generator that already outputted a random binary string.

Partial Seeds Commitments design All pseudorandom number generators produce cycles, which occur when the same sequence of numbers is generated periodically [13]. To avoid small cycles, the implementation used Python *secrets*, a library that generates cryptographically strong random numbers. An alternative design for permFromRandBits() was to directly source the random numbers from the *secrets* library instead of getting the randomness from partial seeds. However, the idea was scrapped because partial seeds were needed for the simulation of the *biased permutation attack*. This is when the worst-case scenario for the protocol is simulated because it simulates the case where all leaders know each other's seeds and are working with each other in order to execute the sandwich attack. They try to find the best combination of partial seeds to reveal to maximise their profit. The attack involves picking between the partial seeds of leaders, so each miner must generate its own seeds. An alternative would have been to write a reveal_seeds() function that internally generates some seeds in the generate_partial_seed() function for each leader. The internal seeds are then sent to permFromRandBits(). Both implementations are similar and at the end the option with each leader generating their own seed was chosen because it most closely resembles the real protocol. The number of leaders *nl* is easily settable in the InputsConfig file.

Chunking design The function chunk() is one of the functions implemented in this thesis that takes a block as a parameter only to cycle through the transactions of the block and send the transactions to another function of a similar name. The function chunk() takes a block as a parameter and feeds every transaction of the block in chunkTransaction(). The project could have been designed to work more on the transaction level, but it was decided to stay on block level so the project is easier to understand as a whole. The original implementation of chunkTransaction() used the deepcopy() function to recursively copy the transactions. The implementation was changed to instantiating many new transaction objects

because using deepcopy() takes a long time. The amount of chunks is easily settable in the InputsCon g le.

Implementing the new incentives design The biased permutation attack involves simulating millions of sandwich attacks and is the part of the simulation that takes the most time. Cython was utilised to compile Python code to C code to make it run faster [8]. Another measure taken so that this part of the simulation does not take much time is the avoidance of the deepcopy() function in the simulate_sandwich() function. Similarly to how the chunking is designed, new block objects are instantiated instead of using deepcopy(), which saves a lot of time. The rule allowing leaders to claim the block rewards of other leaders has been abstracted away.

3.2 Sandwich attack simulation

Algorithm 1 Generating and Executing Sandwich Transactions

```
 1: function generateSandwichTransactions():
 2:     attacker   random.choice (NODES)
 3:     victim   random.choice ([x for x in NODES if x.id != attacker.id])

 4:     T0   Transaction(id  1, sender  "amm", to  attacker.id, value  sandwichTransactionValue)
 5:     T   Transaction(id  2, sender  "amm", to  victim.id, value  sandwichTransactionValue)
 6:     T1   Transaction(id  3, sender  attacker.id, to  "amm", value  sandwichTransactionValue)

 7:     return [T0, T, T1]

 8: function executeSandwichTransactions(self, block):
 9:     for tx in block.transactions:
10:         if tx.to = "amm" or tx.sender = "amm":
11:             if tx.id = 1 or tx.id = 3:
12:                 self.pro t   self.pro t + self.amm.executeTransaction(tx)
13:             else
14:                 self.amm.executeTransaction(tx)
```

The algorithm rstly chooses a random attacker node and a random victim node from the list of nodes objects NODES. It is made sure that the victim is not the same as the attacker (lines 2 and 3).

We model the sandwich transactions the same way as described in Section 2.3 in the background. On lines 4 to 6, three new transaction objects are instantiated: T0, T and T1. The sandwichTransactionValue variable holds the amount of ETH transacted in the attacker sandwich transactions as well as the victim sandwich transaction. In Section 3.1 it is mentioned that all sandwich transactions transact the same amount of ETH in this thesis. This variable can be found in InputsCon g. Transaction T0 has the id 1 and sends sandwichTransactionValue amount of ETH from the AMM to the attacker. Transaction T has the id 2 and sends sandwichTransactionValue amount of ETH from the AMM to the victim. Transaction T1 has the id 3 and sends sandwichTransactionValue amount of ETH from the attacker to the AMM. A list containing the three transaction objects is returned (line 7).

The sandwich transactions generated from generateSandwichTransactions() are processed in executeSandwichTransactions(). The latter function can only be called from an instantiated sandwich object like on line 46 of Algorithm 3. In this example, the function is called by the sandwich object sw which was instantiated on line 39 of Algorithm 3. Whenever a function is called from an instantiated object in Python, the object itself is the rst parameter of the function [15]. The sw from the line 39 of Algorithm 3 is the self parameter of the function. The other parameter block is a block object containing normal transactions as well as sandwich transactions. A loop is initialised to go through all the transactions in the block looking for sandwich transactions. Such transactions are recognised by checking if

they are either sent to or received by the AMM (lines 9 and 10). If the *id* attribute of a transaction is either 1 or 3, then it belongs to the attacker (line 11). This was determined in generateSandwichTransactions() on lines 4 and 6. Each sandwich object has been defined to have an AMM object and have a profit variable. For these two transactions, the state of the AMM object and the *profit* variable of *self* are updated. The AMM of the sandwich object is accessed by *self.amm* and the transaction is executed on the AMM with the executeTransaction() function. This function is described in greater detail on line 21 of Algorithm 2. The return value is added to the *self.profit* variable (line 12). Because the AMM object and *profit* variable are part of *self*, the changes to the state of the AMM and the changes in the *profit* variable persist even after leaving the context of the executeSandwichTransactions() function. The *profit* variable is used by other parts of the simulation for analysis. If a transaction involves the AMM and is neither 1 or 3, then *id* must be 2, which is the victim transaction. This transaction is also processed by executeTransaction() from the AMM but the return value is ignored (line 14).

3.3 AMM Simulation

Algorithm 2 AMM Simulation

```
15: function setPair(self, EthSupply, UsdSupply):
16:     self.EthSupply ← EthSupply
17:     self.UsdSupply ← UsdSupply
18:     self.Constant ← EthSupply * UsdSupply

19: function getEthPrice(self):
20:     return (self.Constant / (self.EthSupply - 1)) - self.UsdSupply

21: function executeTransaction(self, tx):
22:     usdSupplyBefore ← self.UsdSupply

23:     if tx.to = "amm":
24:         self.EthSupply ← self.EthSupply + tx.value
25:         self.UsdSupply ← self.Constant / self.EthSupply
26:         priceDifference ← usdSupplyBefore - self.UsdSupply

27:     if tx.sender = "amm":
28:         self.EthSupply ← self.EthSupply - tx.value
29:         self.UsdSupply ← self.Constant / self.EthSupply
30:         priceDifference ← usdSupplyBefore - self.UsdSupply
31:     return priceDifference
```

The function setPair() is called immediately after an AMM object is instantiated. It is called by the AMM object itself so the first parameter of the function is *self*. It takes in two numbers *EthSupply* and *UsdSupply* to set the amount of the ETH and the amount of USD under the possession of the AMM object (line 15). The two variables are from in the InputsConfig file. The function then multiplies the two numbers to create the constant, as described in the background. All three numbers are saved as attributes of the AMM object (line 16 to 18).

The function getEthPrice() is used to determine the exchange rate for ETH in USD (line 19). If one recalls the calculation in Section 2.2 and plugs *EthSupply* as *amountA* and *UsdSupply* as *amountB*, then *UsdSupply* is the same as *Constant* divided by *EthSupply*. To calculate the cost of 1 ETH in USD, the effect on *UsdSupply* if 1 ETH is subtracted from *EthSupply* is calculated. The effect can be expressed as $Constant/(EthSupply - 1)$. This number is subtracted from the untouched *UsdSupply* variable and the difference is returned.

The function executeTransaction() is another function that must be called through an AMM object, so the first parameter is *self*. The other parameter *tx* is the transaction which is to be executed on

the AMM. The function works by firstly saving the current value of the AMM's USD supply in the variable *usdSupplyBefore* (line 22). The AMM must then determine if it is on the sending or receiving end of the transaction, so it sees if the *to* or *sender* attribute of the transaction is set to string “amm” (lines 23 and 27). These attributes were set on lines 4 and 6 in Algorithm 1. If the transaction is sent to the AMM, the AMM adds the value transacted to its ETH pool, which is its *EthSupply* variable (line 24). Because *EthSupply* multiplied by *UsdSupply* is now not the same as *Constant*, UsdSupply must decrease because *EthSupply* increased. *UsdSupply* goes down as much as *Constant* divided by the new *EthSupply* value (line 25). To determine how much *UsdSupply* went down compared to before line 25, the difference between *usdSupplyBefore* and *UsdSupply* is used to determine the change in price in USD. This value is saved in the *priceDifference* variable (line 26). Similarly to line 23, if the sender is the string “amm”, the ETH is removed from the *EthSupply* (line 28). The *UsdSupply* must also change due to the changes in *EthSupply*, but this time the amount in the variable increases because ETH was removed from the supply. The *priceDifference* variable is calculated the same way as line 26 and in both cases, the variable is returned (line 31).

Chapter 4

Implementation

4.1 Implementation of 3 in BlockSim

Algorithm 3 Block Creation Event

```
32: function generate_block(event):
33:     block    event.block
34:     miner    block.miner
35:     if block.previous = miner.last_block().id:
36:         block.transactions    block.transactions + generateNormalTransactions()
37:         block.transactions    block.transactions + generateSandwichTransactions()

38:         chunk(block)

39:         sw = Sandwich()

40:         leader_blocks    miner.blockchain[-numberOfLeaders:]
41:         block.leaders    [b.miner for b in leader_blocks]
42:         partialSeeds    [leader.generate_partial_seed() for leader in block.leaders]
43:         partialSeeds    sw.reveal_seeds(block, partialSeeds)

44:         block.seed    xor(partialSeeds)
45:         block.transactions    permFromRandBits(block.transactions, block.seed)

46:         sw.executeSandwichTransactions(block)
47:         SANDWICHES.append(sw)

48:         generate_next_block(block, miner)
```

The function generate_block() is the only function listed here as pseudocode that already existed in BlockSim prior to the implementation. The function takes an event object as a parameter. The block and the miner of the block are extracted from the event object and saved as variables (line 33 and 34). The algorithm then checks if the block is built on the last block of the internal blockchain of the miner (line 35). Normal block transactions without sandwich transactions are generated and added to the block transactions. The function generateSandwichTransactions() creates the three sandwich transactions and adds them to the block transactions (lines 36 and 37). See Algorithm 1 for a closer inspection of generateSandwichTransactions(). Now the first element of 3 comes into play: chunking. The block is chunked on line 38. To see how this is done, see Algorithm 5. On line 39, a new sandwich object sw is instantiated. The object has a pot attribute which is set to zero. The sandwich object will be used on lines 43, 46 and 47, so over the course of generate_block(), the value of the attribute changes. The leader blocks are determined by looking back in the last n blocks of the internal blockchain of the miner

and determining the miners of the last *n* blocks (40). The variable *n* is set in the InputsConfig file. The miners are added to a list called *leaders*, which itself is an attribute of the block object (line 41). Each of the leaders generate partial seeds using the function generate_partial_seed() on line 49 in Algorithm 4 (line 42). The partial seeds are used as arguments in reveal_seeds(), which is the function where the biased permutation attack is simulated (line 43). This function is specified in Algorithm 6. The sandwich object *sw* is needed to call reveal_seeds() because the function may change the *sw.profit* attribute. This is because on line 88 of Algorithm 6, one can see *sw.profit* is set to the *disincentive* variable from reveal_seeds(). This means the *sw.profit* variable starts with a negative value because of the disincentive and the profit of the sandwich attack is added to the *sw.profit* on line 46. The partial seeds that are revealed on line 43 are then XORed into the seed of the block (line 44). The seed is used in permFromRandBits() to permute the chunked transactions (line 45). This function is specified on line 51 from Algorithm 4. To calculate the profit of the sandwich attack, executeSandwichTransactions() from Algorithm 1 is called on line 46. The function adds the profit from the sandwich transactions to the *sw.profit* variable. This is the final change for *sw.profit* because the disincentive for not revealing seeds has already been subtracted from *sw.profit* on line 43 in reveal_seeds(). On line 47 the sandwich object sw that was instantiated on line 39 is appended to the SANDWICHES list of InputConfig. The list collects sandwich objects over many blocks and when the BlockSim simulation is over, the profits of each sandwich object is extracted and put into an excel file. This is done so the profits of the sandwich attacks can be analysed. In the line 48, the block is propagated to other miners and the miner starts generating a new block.

4.2 Partial Seeds Commitments and Permuting

Algorithm 4 Partial Seeds Commitments and Permuting

```
49: function generate_partial_seed():
50:     return secrets.randbits(seedSize)

51: function permFromRandBits(array, randBits):
52:     for i in reversed(range(2, len(array)+1)):
53:         j ← KnuthYao(i, randBits) + 1
54:         array[i-1], array[j-1] ← array[j-1], array[i-1]
55:     return array

56: function KnuthYao(n, randBits):
57:     u ← 1
58:     x ← 0
59:     while (True):
60:         while u < n:
61:             u ← 2*u
62:             r ← randBits.pop() if len(randBits) > 0 else secrets.randbits(1)
63:             x ← 2*x + r
64:         d ← u - n
65:         if x >= d:
66:             return x - d
67:         else u ← d
```

The function generate_partial_seed() is called on line 42 of Algorithm 3. The variable *seedSize* is a positive integer that dictates how many random bits a leader must generate. It is given in the InputsConfig file. Its size is dependent on the chunk size because the bigger this number is, the more transaction objects there are and the more random bits are needed to permute them. The generate_partial_seed() function uses the Python *secrets* library to generate a number of *seedSize* size and returns the number.

The function permFromRandBits() takes an array and a list of random bits as parameters. The array is the one that is to be permuted. The function then iterates through the array in reverse, down to the second element (line 52). Firstly, the index of the last element of the array is taken and fed alongside the randBits parameter in KnuthYao(). This helper function returns a random number between zero and the index (line 53). The last element of the array is switched with the element with the index of the random number (line 54). The function then moves on to the index of the second last element and the same procedure is executed. When the array is iterated through to the second index, the permuted array is returned (line 55).

The function KnuthYao() has been extended such that if the provided randBits list does not have a sufficient amount of bits, then it takes random bits from the Python secrets library (line 62). This is to make sure the function always finishes executing.

4.3 Chunking

Algorithm 5 Chunking a Block

```
68: function chunk(block):
69:     chunkedTransactions ← []
70:     for tx in block.transactions:
71:         chunkedTransactions.extend(chunkTransaction(tx))
72:     block.transactions ← chunkedTransactions

73: function chunkTransaction(tx):
74:     chunksOfTx ← []
75:     for c in range(amountOfChunks):
76:         chunk ← Transaction(id ← tx.id, sender ← tx.sender, to ← tx.to,
                        value ← tx.value / amountOfChunks, chunkId ← c )
77:         chunksOfTx.append(chunk)
78:     return chunksOfTx
```

The function chunk() takes a block object as a parameter. An empty list named chunkedTransactions is initialised on line 69. On line 70 a loop is initiated to cycle through each transaction in the block. Each transaction is given as a parameter to chunkTransaction(). The chunkedTransactions list is extended by the list returned from chunkTransaction() (line 71). At the end, the transactions of the block is replaced with chunkedTransactions (line 72).

The function chunkTransaction() takes a transaction tx as a parameter. It initialises an empty list called chunksOfTx where it will store the chunks of tx (line 74). A new transaction object is initialised for the number of amountOfChunks (lines 75 and 76). The amountOfChunks variable can be set in the InputsConfig file. All parameters of the new transaction object is copied from tx, except for the value, which is divided by the amount of chunks. A new attribute of the transaction object, chunkId, is utilised to identify each chunk. Because the transaction id of each chunk is identical to tx and the chunkId attribute is always distinct, each chunk is uniquely identifiable. The chunks initialised on line 76 are all appended to chunksOfTx (line 77). The list chunksOfTx is returned (line 78).

4.4 Implementing the new incentives

The function revealSeeds() can only be called by an sandwich object, so the first parameter is self. The second parameter is a block object and the third parameter is a list of partial seeds. The function starts by initializing a very negative number as the highestProfit variable. The variable serves a placeholder for later in the function (line 80). The itertools library then produces all possible combinations of the seeds in the seeds list and the function loops through all of the subsets (line 81 and 82). The seeds of

Algorithm 6 Biased Permutation Attack Simulation

```
79: function reveal_seeds(self, block, seeds):
80:     highestProfit ← -1000000000
81:     for L in range(1, len(seeds) + 1):
82:         for subset in itertools.combinations(seeds, L):
83:             simulatedProfit ← simulate_sandwich(block, xor(subset))
84:             disincentive ← (numberOfLeaders - len(subset)) * blockReward * amm.getEthPrice()
85:             if simulatedProfit - disincentive > highestProfit:
86:                 highestProfit ← simulatedProfit - disincentive
87:                 revealedSeeds ← subset
88:                 self.profit ← -disincentive
89:     return revealedSeeds

90: function simulate_sandwich(block, seed):
91:     simulatedBlock ← Block()
92:     simulatedBlock.transactions ← list(block.transactions)
93:     simulatedBlock.transactions ← permFromRandBits(simulatedBlock.transactions, seed)
94:     sw = Sandwich()
95:     sw.executeSandwichTransactions(simulatedBlock)
96:     return sw.profit
```

each subset are XORed together and used in simulate_sandwich(), which is described on line 90. The return value is saved in the *simulatedProfit* variable (line 83). Part of the incentives system of 3 is taking custody of block rewards and only returning them if the miner stays honest. The *disincentive* variable quantifies the amount of USD forfeited by the coalition of leaders if some of the leaders have not revealed their seeds. The *disincentive* is calculated by the current number of colluding leaders times the number of ETH rewarded per block times the price of ETH (line 84). The *blockReward* variable used in the calculation can be found in the InputConfig file. The getEthPrice() function is described on line 19 of Algorithm 2. The variable *simulatedProfit* minus *disincentive* is the profit of the subset and this profit is compared to the *highestProfit* variable. If *simulatedProfit* minus the *disincentive* is higher than *highestProfit*, then a new combination of seeds has been found that leads to the highest profit has been found. The *highestProfit* variable is updated (lines 85 and 86). The subset used to achieve the highest profit is saved in the *revealedSeeds* variable (line 87). The *self.profit* variable is updated with the negative of *disincentive*. This is the *profit* attribute of the sandwich object that called reveal_seeds(). The reason this is done is explained in Algorithm 3. On line 89 the *revealedSeeds* variable is returned.

The function simulate_sandwich() is solely used in the reveal_seeds() function. It takes a block and a seed as parameters. On line 91, a new block object called *simulatedBlock* is instantiated. The transactions of the original block are copied to the transactions of *simulatedBlock* (line 92). The function List() was used to create a new list object that is not the same as transactions of the block parameter. These precautions were implemented so the simulation run in simulate_sandwich() does not change the transactions of the actual block that was given as a parameter. The transactions of *simulatedBlock* are then permuted with permFromRandBits() using the seed parameter (line 93). A new sandwich object is instantiated which calls executeSandwichTransactions(). This is done to calculate the profit of the transactions in the new order (line 94 and 95). Only the profit of the sandwich attack is of interest which is why solely the profit of the sandwich object is returned on line 96. Because *sw* is a simulated sandwich, it is not added to the *SANDWICHES* list like on line 47 from Algorithm 3.

Chapter 5

Evaluation

In this chapter the extended BlockSim programme is utilised to examine the behaviour of the simpli ed version of 3 . The two modes of leader behaviour are simulated: honest and colluding. In the rst mode, we simulate the situation where all leaders adhere to the rules of 3 and provide truly random seeds. It is the expected way the protocol is used. In the other mode, the pro ts of leaders executing the biased permutation attack are observed.

Before running the simulation, some parameters of 3, the sandwich attacks and BlockSim itself had to be con gured. Five nodes were initialised, each with the same hash power. A small number was chosen for the number of nodes because adding many to the system would complicate the system without any gain in the quality or quantity of the data. In real blockchain systems, it is common for many small miners to come together in a group to mine in mining pools, so the ve nodes can also be seen as ve mining pools of the same size [5]. The number of leaders, n, was set to 10. This number was chosen for easier comparison to the plots from the paper [2]. The amount of ETH and USD under the possession of the simulated AMM had to be set. The real life counterpart chosen to provide the values was the Uniswap USD/ETH contract, which was the top DEX ranked by 24 hour trading volume as of early 2023 [7]. During March 2023 there was 71'620 ETH and 133'730'000 USD locked in the AMM. So at that time 1 ETH was exchanged for 1867.24 USD.

Let sandwich size be the amount of ETH transacted in each of the three sandwich transactions in a sandwich attack. The sandwich sizes chosen in the evaluation were 100, 472, 1000, 1897, 5000, 10000 ETH. At rst, the sandwich sizes chosen were 100, 1000 and 10000 ETH to test small, big and extremely big sandwich attacks. Let potential pro t be de ned as the pro t of any given sandwich attack had there been no protocol to diminish the pro t. The sandwich sizes 472 and 1897 ETH were chosen because the potential pro ts were 6.35 and 109 ETH respectively. These sandwich sizes were chosen because the pro ts are close to the values chosen in Figures 2.5 from the paper from Alpos, Amores-Sesar, Cachin, and Yeo. Note that there is a small difference between 6.35 ETH potential pro t simulated in this thesis and 6.37 ETH used in the paper. After running the experiment, it has been observed that the plots of 1897 ETH and 10000 ETH were similar to each other despite the big difference in size. A value between these amounts was chosen for further investigation. To this end, the sandwich size 5000 ETH was also chosen to be part of the experiment.

The chunk sizes chosen to test were 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 100 and 200. 5 up to 50 chunks were chosen to see the behaviour of the protocol with small, linear increases in chunk size. 100 and 200 were chosen to see how far the protocol can go if the chunk size is taken to the extreme.

The simulator was ran in both honest and colluding mode with different chunk sizes and with different sandwich sizes. The values listed in the following graphs are the average of the pro ts from 1000 runs. The error bars are standard deviations values from the 1000 runs. All graphs below are relative to no protocol for easier comparison.

Figure 5.1. The average profit of honest leaders. Even though the error bars are big, it was observed that the higher the number of chunks, the closer to zero the values become.

5.1 Benchmarks of protocol with honest leaders

Figure 5.1 depicts the average profits of the attackers if they execute sandwich attacks of size 472 and 10000 ETH and do not collude. The variable that is changed in the x axis is the chunk size. One can observe that the simulations of both sandwich sizes behave nearly exactly the same in these conditions: all values are very close to zero. The case without the use of the protocol (m = 0) is omitted to see the data from 5 chunks on more clearly. The highest profit was achieved by sandwich attacks of size 10000 ETH at 5 chunks, which was less than 1.5% of the potential profit. Upon closer inspection, a trend was observed where higher chunk sizes correspond to lower average profits. However, since the error bars are very big, this finding is to be interpreted with caution. In both tests, the average profits with the protocol from 10 chunks on are nearly zero when compared to the case where 3 was not used. This is to be expected because the random bits provided to permFromRandBits() were truly random because no colluding leaders were present to bias the permutation. Because there are around the same amount of profitable chunks as loss chunks for the attacker, most chunks cancelled each other out, as described in the “permuting” paragraph in Section 2.4. The error bars are especially big at 5 chunks. However, one can observe that the size of the error bars decreases as m increases. This is due to the fact that the more chunks there are, the more frequent the case that an attacker buy chunk comes directly before or after an attacker sell chunk. When this happens, there is no victim chunk between the attacker chunks, so there is no profit for the attacker. All in all, the results show that 3 works very well in making sandwich attacks unprofitable provided the leaders are honest.

5.2 Benchmarks of the protocol with colluding leaders

This simulation presupposes that all leaders trust each other enough to reveal their partial seeds to each other. The leaders must trust that no leader would defect from the group and steal all the block rewards for themselves.

In both Figure 5.2a and Figure 5.3a, one can see many colourful lines plotted representing the amount leaders out of the 10 leaders are working together to execute an attack. In this thesis the line that is compared to is the blue line, which is the case where all 10 leaders are colluding. There are also differences in what is measured in the plots. The y axis from the theory is the upper bound for a profitable permutation. The simulation results compares the average profit of 1000 runs. With enough sampled data, the

results from the simulation are bound to be equal or below the values from the theory. In Figure 5.2, one can see that the simulated pro ts are much lower than in theory. This is to be expected because the line from the theory is an upper bound. Reasons for the difference could lie in the amounts of tokens in the AMM or the ratio of the amounts when the sandwich transactions are executed. Another reason could be the assumption that all three sandwich transactions in a sandwich attack transact the same amounts does not hold in the paper. The curves in Figure 5.3 are much more similar, even though there is a small difference between 6.35 and 6.37 ETH. In both graphs, one can see that the 6.35 ETH potential pro t has been reduced to around zero at around 10 chunks. These ndings suggest that if 3 had been used in October 2022, 99.97 percent of attacks would have been reduced to nearly 0 with 10 chunks, even with colluding leaders.

Whilst observing Figure 5.4, one can see that the plots of sandwiches of size 100 and 472 ETH are very close to 0 after 15 chunks, because the potential pro ts of these small sandwiches are too small to give up block rewards for. Once the average pro t is around zero, it stays at zero for all successive chunk size increases. The biggest difference between Figure 5.4 and Figure 5.1 is some sandwich attacks are still pro table on average, namely sandwich attacks of size 1000, 1897, 5000, 10000 ETH. In there cases, one can see curves that steeply go down at rst but then gradually converge to 0. 15 chunks reduces 1000 ETH attacks by 85 percent and with 200 chunks the 1000 ETH attacks are reduced by more than 99.5 percent. The 10000 ETH attacks are reduced to around a third of the case without a protocol at 15 chunks. These decreases are caused by the increase in amounts of chunks, which increase the amount of combinations of chunks there are that cancel the movement in the AMM of each other out. One can see that bigger sandwiches like 10000 and 5000 ETH have similar curves to each other. This is the case where the sandwich pro ts to be gained far succeed the maximum disincentive of the protocol, which is 20 ETH. The maximum disincentive is calculated by the amount of ETH lost if all ten leaders collude and since the block reward is 2 ETH each, the maximum disincentive is 20 ETH. As seen before, the potential pro t lost for 472 ETH is 6.35 ETH, which is less than the disincentive, so the curve is close to 0. We have also seen that the potential pro t for 1897 ETH is 109 ETH, which is much more than the disincentive, so the curve is close to the plots of much bigger sandwiches. The sandwich size 1000 ETH has the potential pro t of 29.13 ETH, which is relatively close to the 20 ETH disincentive. As a result, the curve on the graph is between the two aforementioned cases of 472 and 1897 ETH. Another observable trend is, like in Figure 5.1, the error bars decrease the higher the chunk number is. The reason for this is the same as described in the last section: the higher n is, the higher the chance that attacker chunks cancel each other out, resulting more pro ts around zero for the attacker.

Figure 5.2. Comparison between the theory and the simulation results with potential pro t = 109 ETH

(a) The plots for the potential pro t of 109 ETH from the original paper from Alpos, Amores-Sesar, Cachin, and Yeo. Only the blue line is relevant for the comparison as it depicts the case where all leaders collude.

(b) The sandwich attacks were set to the size 1897 ETH because this number gives a potential pro t of 109 ETH like in Figure 5.2a. Solely k = 10 was simulated. One can see that the simulated results are lower than the values provided in the theory.

Figure 5.3. Comparison between the theory and the simulation results with the potential pro t set to 6.37 ETH and 6.35 ETH

(a) The plots for the potential pro t of 6.37 ETH from the original paper from Alpos, Amores-Sesar, Cachin, and Yeo.

(b) For this plot, the sandwiches have the size 472 ETH because this value gives a potential pro t of 6.35 ETH. Solely k = 10 was simulated. One can see that this graphs is similar to Figure 5.3a.

Figure 5.4. Average pro t of colluding leaders. 100, 472, 1000, 1897, 5000, 10000 ETH represent the different amounts of ETH transacted in each of the three sandwich transactions in a sandwich attack. 0 chunks represents no protocol, so the pro t at 0 chunks in proportion to no protocol is always 1. All values converge to 0. Here one can see 100 and 472 ETH are very close to 0 after 15 chunks. One can also observe that the values of 1897, 5000 and 10000 ETH are close to each other.

Chapter 6

Conclusion

In this bachelor thesis, a simplified version of [3] has been implemented in BlockSim. The simulator has been extended to support sandwich attacks of many different sizes. An AMM has been simulated as well as the biased permutation attack. Many simulations and benchmarks were run to evaluate the performance of the protocol under different conditions. The results showed that the protocol solved all sandwich attacks when the leaders were not colluding and 99.97% of all attacks even if leaders collude. For the 0.03% of sandwich attacks that are unsolved, [3] reduced the profits by a significant margin. The comparisons of the simulated results with the plots from the paper from Alpos, Amores-Sesar, Cachin, and Yeo showed that the simulated results were the same or better than in theory. Implementing the protocol took less time than building the frame of the project such as sandwich execution, AMM implementation and benchmark of colluding leaders.

However, there are limitations to inferring results using the simplified version of the protocol. In this thesis we did not implement the commitment scheme of [3]. The fees in AMM were disregarded. We also did not look at sandwiches where the values transacted in the attacker transactions and victim transactions are different from each other. All these simplifications may need to be addressed in further research. If this has been looked into, a next step would be to implement [3] in a real blockchain system.

A future work would be to determine mathematical functions that approximate the curves of the average profit of colluding leaders. The function would use the sandwich size, the chunk size and the size of the holdings of the AMM to calculate the average profit of sandwich attacks under these parameters. Finding this out would help in determining the reason there is an upper bound in Figure 5.4. It can also be used to help determine the optimal chunk size. A trade-off to consider is the bigger the chunk size, the longer the function KnuthYao() takes. As some nodes on the Ethereum blockchain must store all chunks, if the number of chunks is high, then the amount of storage space needed to store the chunks increases. A solution to this would be to selectively chunk the transactions going to or coming from an AMM and leaving all other transactions unchunked. The effectiveness of the protocol is not affected and space is saved.

Bibliography

[1] M. Alharby and A. van Moorsel, "Blocksim: An extensible simulation tool for blockchain systems," Frontiers Blockchain, vol. 3, p. 28, 2020. https://github.com/maher243/BlockSim.

[2] O. Alpos, I. Amores-Sesar, C. Cachin, and M. Yeo, "Life is good when you do not have a sandwich." Unpublished Manuscript, 2023.

[3] A. Bacher, O. Bodini, H. Hwang, and T. Tsai, "Generating random permutations by coin tossing: Classical algorithms, new analysis, and modern implementation," ACM Trans. Algorithms, vol. 13, no. 2, pp. 24:1–24:43, 2017.

[4] M. Bartoletti, J. H. yu Chiang, and A. Lluch-Lafuente, "A theory of automated market makers in de ," 2022.

[5] BTC.com, "Pool distribution (calculate by blocks)." https://btc.com/stats/pool. Accessed: 7. June 2023.

[6] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, Introduction to Reliable and Secure Distributed Programming (2. ed.). Springer, 2011.

[7] CoinGecko, "Top decentralized exchanges ranked by 24h trading volume." https://www.coingecko.com/en/exchanges/decentralized. Accessed: 3. June 2023.

[8] Cython, "About cython." https://cython.org/. Accessed: 22. May 2023.

[9] S. D'Aprano, "secrets — generate secure random numbers for managing secrets." https://docs.python.org/3/library/secrets.html. Accessed: 22. May 2023.

[10] Flashbots, "Frp-18: Cryptographic approaches to complete mempool privacy." https://collective.flashbots.net/t/frp-18-cryptographic-approaches-to-complete-mempool-privacy/1210. Accessed: 5. June 2023.

[11] Flashbots, "Mev-explore v1." https://explore.flashbots.net. Accessed: 22. May 2023.

[12] A. Gervais, "Lecture 13.7: Sandwich attacks." https://www.youtube.com/watch?v=Om6Fqf7IRKQ. Accessed: 5. June 2023.

[13] D. Johnson and D. M. Ceperley, "Generation of random numbers." https://courses.physics.illinois.edu/phys466/sp2013/lnotes/random_numbers.html. Accessed: 5. June 2023.

[14] M. Musharraf, "How to read smart contract data." https://www.ledger.com/academy/how-to-read-smart-contract-data. Accessed: 5. June 2023.

[15] G. S. Panwar, "Self in python class." https://www.geeksforgeeks.org/self-in-python-class/. Accessed: 3. June 2023.

[16] O. Pomerantz, "How to build a random number generator for the blockchain." https://blog.logrocket.com/build-random-number-generator-blockchain/. Accessed: 5. June 2023.

[17] C. Smith, J. Cook, and P. Pettinari, "Maximal extractable value (mev)." https://ethereum.org/en/developers/docs/mev/. Accessed: 3. June 2023.

[18] Uniswap, "Usdc - eth." https://info.uniswap.org/#/pools/0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640. Accessed: 22. May 2023.

[19] P. Wackerow, M. Zoltu, and P. Jadhav, "Introduction to smart contracts." https://ethereum.org/en/developers/docs/smart-contracts/. Accessed: 22. May 2023.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 23.06.2023
Ort/Datum

Lawrence Chiang
Unterschrift