Informatics

Ethereum Attack Simulator

A Discrete Event Simulation Engine for Attacks against the Ethereum Blockchain

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Alexander Maitz, BSc
Matrikelnummer 01426069

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl
Mitwirkung: Mag.rer.soc.oec. Nicholas Stifter

Wien, 28. November 2022

Alexander Maitz   Edgar Weippl

TU WIEN Informatics

Ethereum Attack Simulator

A Discrete Event Simulation Engine for Attacks against the Ethereum Blockchain

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Alexander Maitz, BSc

Registration Number 01426069

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl
Assistance: Mag.rer.soc.oec. Nicholas Stifter

Signiert von:	Alexander Maitz
Datum:	01.12.2022 12:01:12

A TRUST einfach sicher

Dieses Dokument ist digital signiert! Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Prüfinformation: Informationen zur Prüfung der elektronischen Signatur finden Sie unter: www.handy-signatur.at

Signiert von:	Edgar Weippl
Datum:	01.12.2022 11:49:50

A TRUST einfach sicher

Dieses Dokument ist digital signiert! Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Prüfinformation: Informationen zur Prüfung der elektronischen Signatur finden Sie unter: www.handy-signatur.at

Vienna, 28th November, 2022

Alexander Maitz Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Alexander Maitz, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Signiert von: Alexander Maitz
Datum: 01.12.2022 12:02:14

A-TRUST einfach sicher

Dieses Dokument ist digital signiert!
Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument hat gemäß Art. 25 Abs. 2 der Verordnung (EU) Nr 910/2014 vom 23. Juli 2014 ("eIDAS-VO") die gleiche Rechtswirkung wie ein handschriftlich unterschriebenes Dokument.

Prüfinformation: Informationen zur Prüfung der elektronischen Signatur finden Sie unter: www.handy-signatur.at

Wien, 28. November 2022

Alexander Maitz

Danksagung

Ich möchte ein großes Dankeschön an meinen Betreuer Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl für seine Beratung und Unterstützung auf dieser Reise aussprechen.

Vielen Dank auch an Assistenz-Betreuer Mag.rer.soc.oec. Nicholas Stifter, der immer ein offenes Ohr für mich und meine Bedenken hatte, während er mir beim Entwickeln großartiger Ideen half.

Auch würde ich gerne meinen Großeltern, meiner Freundin, und jedem aus meiner Familie und von meinen Freunden danken, die von Beginn an an mich geglaubt haben, eure Unterstützung ist das Wertvollste, das ich mir vorstellen kann.

Acknowledgements

I want to say a big thank-you to my supervisor Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl for his guidance and support throughout this journey.

Many thanks to my assistant supervisor Mag.rer.soc.oec. Nicholas Stifter, who always had a friendly ear for me and my concerns while developing some great ideas along with me.

I also want to thank my grandparents, my girlfriend, and everybody of my family and friends who believed in me from the beginning. Your support was the most valuable thing I could imagine.

Kurzfassung

Blockchains sichern heutzutage einen erstaunlichen Wert von über einer Billion Euro ab. Es ist entscheidend, sowohl derzeit eingesetzte als auch zukünftige Blockchain Implementierungen im Hinblick auf Attacken evaluieren zu können, um ein besseres Verständnis von deren Richtigkeit, Sicherheit und Leistung zu bekommen. Das schnellebige Umfeld von Blockchains macht genau das sehr schwierig, und mangels ausgereifter Werkzeuge im Hinblick auf abweichendes Verhalten, ist es nicht möglich die Forschungslücke zwischen Theorie und Praxis, was Angriffe betrifft, zu schließen.

Diese Arbeit präsentiert einen ereignisdiskreten Simulator für Ethereum - die am weitesten verbreitete Blockchain im Bezug auf Applikationen - der sowohl dazu imstande ist, gewisse Angriffsszenarien zu evaluieren, als auch weitreichend konfigurierbar ist, um die Auswirkungen dieser Attacken auf die Blockchain Stabilität zu untersuchen. Um die Entwicklung dieser Simulationsumgebung zu ermöglichen wird zuerst eine Kategorisierung von Angriffen vorgenommen und zusätzlich der State of the Art, bezüglich existierenden Simulatoren, ausgearbeitet. Ein tieferes Verständnis für das Ethereum Protokoll wird ermöglicht, indem derzeit benutzte Ethereum Clients untersucht werden, während außerdem eine tiefgehende Analyse des de facto Standard-Clients für Ethereum, Geth, dargelegt wird. Hierdurch wird gezeigt, dass bestehende Simulatoren die Anforderungen nicht erfüllen, und folglich, basierend auf der Analyse der Clients und der Simulatoren, ein Konzept für den neuen Simulator entworfen, gefolgt von dessen Implementierung und Validierung gegen reale Daten aus dem Ethereum Mainnet. Angelehnt an die vorangehende Kategorisierung von Angriffen wird eine Teilmenge von relevanten Attacken ausgewählt und deren Auswirkungen hinsichtlich möglicher finanzieller Belohnungen und Destabiliserung von Ethereum mittels der neu entwickelten Simulationsumgebung ausgewertet. Hier wurde die Selfish Mining und die Verifier's Dilemma Attacke aufgrund der vergleichbaren Eigenschaften und Auswirkungen auf das Ethereum Netzwerk ausgewählt.

Die Simulation der Attacken führt zu interessanten Einblicken in die Auswirkungen dieser auf Ethereum. Diese betreffen sowohl mögliche Belohnungen für Angreiferinnen und Angreifer, als auch die Protokollsicherheit, z.B. durch Ausnutzen einer Fehlanpassung von Transaktionskosten an den tatsächlichen Rechenaufwand. Weiters wird die Notwendigkeit der Neubewertung von Sicherheitsrisiken durch den Vergleich von Simulationsauswirkungen vor und nach der Einführung von EIP1559 aufgezeigt.

Abstract

Blockchains are used to secure staggering amounts of over 1 trillion euro worth of digital assets. With this in mind, it is crucial to be able to evaluate both currently deployed and future implementations concerning different attack scenarios, in order to gain deeper insights into their correctness, security and performance. However, the fast-paced environment of cryptocurrencies renders exactly these aspects challenging to achieve in practice, and mediocre tool support concerning the simulation of adversarial behaviour leads to a research gap between theory and practice when it comes to evaluating attacks against real-world blockchain protocols.

This thesis addresses this research gap by presenting a discrete event simulation engine for Ethereum - the most widely used blockchain related to smart contract applications - that is capable of evaluating various attack scenarios, as well as being highly customizable in order to aid investigations into the impact these attacks can have on the overall system's security. To facilitate the development of such a simulation engine, a categorization of known attacks against blockchains, as well as an overview of the state of the art of existing blockchain simulators, is provided. In addition, the currently utilized Ethereum client implementations are examined while also providing a deep dive into the code and processes of the de-facto state-of-the-art Ethereum client, namely Geth. Hereby it is shown that current simulators do not offer the desired functionality. Hence a novel attack simulator is first designed and subsequently implemented as well as validated against real-world Ethereum mainnet data. Based on the previous attack categorization, a subset of relevant attacks is selected and investigated regarding the possible monetary rewards and Ethereum destabilization an adversary could achieve, using the newly developed simulation environment. Specifically, the selfish mining and the verifier's dilemma attacks were chosen because of their comparable properties and effects on the Ethereum network.

The results of this thesis show that the simulation of attack scenarios can lead to interesting insights into the sometimes significant impact attacks can have on Ethereum, e.g., by exploiting a mismatch between actual transaction execution costs and transaction pricing. Furthermore, this work highlights the need for continuous re-evaluation of the security impact of protocol changes through a comparison of the simulated implications of pre- and post-EIP1559 parametrizations.

Contents

CHAPTER 1

Introduction

Every day, millions of digital coins and tokens resembling billions of euros and US dollars circulate in the opaque markets of blockchain-based cryptocurrencies. At the time of writing, the entire cryptocurrency universe has a market capitalization of 1 trillion dollars and a daily trade volume of approximately 70 billion dollars [Coi22c]. The two biggest cryptocurrencies, Bitcoin and Ethereum, have a combined market capitalization of about 616 billion dollars with a daily volume of roughly 49 billion dollars [Coi22a, Coi22b].

A variety of studies and scientific publications have been presented to disclose potential vulnerabilities of blockchains. However, while research has focussed on structural attacks against blockchains, the outlined attacks have remained largely theoretically feasible, with few works that investigate if the attack strategies are also practical. Often, the attacks are shown to be feasible using only formal reasoning or approaches, such as the Markov Decision Process (MDP), that require a high level of abstraction and therefore do not necessarily capture their real-world feasibility, which requires taking protocol-specific implementations into account. This is further emphasized by the lacking or incomplete tools necessary for testing and developing [CPNX19, SGD20].

The value of distributed ledger technologies and the digital assets and tokens they secure is – among other factors – highly based on people's trust and belief in their correct and secure operation. It is crucial to build or keep up trust in this regard through sufficiently advanced and well-researched, secure protocols and the corresponding software implementations. The problem is that a multitude of works, e.g., the Ethereum yellow paper [Woo14], suggest that security can be guaranteed if and only if the main premises hold and that, under certain adverse circumstances, the protocol may not be secure at all. In particular in the cryptocurrency space, there can often exist a discrepancy between theoretically possible attacks and real-world scenarios where such adversarial strategies are rarely observed in practice, even if they are believed to be feasible. More importantly, there remains a research gap between formally describing attack strategies and empirically analyzing real-world systems in order to identify if participants are currently engaging in such adversarial behaviour.

Consider, for example, the case of mining pools. While there exists theoretical work that shows miners could gain an unfair advantage by deviating from protocol rules (e.g., by executing a selfish mining attack [ES13, NF19]), and it was suspected that miners actually engage in such behaviour, there was little to no empirical research done to undermine this assumption until the recent work from Yaish et al., which discovered some details in block timestamps that effectively show that pools act maliciously to some extent [YSZ22].

In order to investigate if hypothetical attack scenarios against blockchain technologies presented by academia can readily be translated into practice, various approaches exist. First of all, an exclusively analytic approach can be followed that tries to formally model the entire protocol stack and the complex execution environment. This is quite an expensive approach, taking a lot of time and effort, even more, if there should be the possibility to change certain variables and factors easily. A second approach would be a simulation environment that mirrors real-world behaviour as closely as possible and makes it easy to simulate various attack scenarios. This is the approach that this work will follow to help to close the research gap between theory and practice, particularly by developing a simulator suitable to evaluate attack scenarios, which is also available on github.com and open source [Mai22]. In 2019, a potential security vulnerability due to inadequate contract execution pricing in the Ethereum Virtual Machine (EVM) was intentionally concealed until its fix by the Ethereum foundation because it could have been used to attack the Ethereum protocol [Fou21b]. While this attack was obviously considered feasible by the developers and therefore was concealed, the existence of such vulnerabilities highlights the usefulness of a realistic simulation environment in providing methods for analyzing and testing protocol behaviour for novel attack strategies. Additionally, such simulation environment could be beneficial for examining implications that follow from protocol changes, like the transaction fee market changes that come with Ethereum Improvement Proposal (EIP)1559 or the change of the underlying consensus model via the merge[1] [eth22, Fou22].

Blockchains are peer-to-peer systems where the goal is a high degree of decentralization. That is the case for network capacity, mining decentralization and client software diversity. However, when it comes to Ethereum, there is one piece of software that is used by a worryingly large majority of peers (about 86.93 % of the synchronized nodes) who participate in the network, namely GoEthereum (Geth) [eth20b]. This is the de facto standard Ethereum client software that nodes use to connect to the network, find peers, send and receive messages, mine[2] blocks and validate blocks and transactions, among others [eth20a]. Due to the lack of good documentation and because of frequent code changes, it is difficult to obtain and maintain a complete picture of the concrete

[1]Ethereum's transition from Proof of Work (PoW) to Proof of Stake (PoS)

[2]After the merge, mining is not possible anymore, but block validation and construction are still performed

functionality without being highly involved. This work also addresses this issue by providing an overview of how Geth works to the interested reader, however, it is clearly not a long-term solution to this open issue that emerges from the current code development practices of Geth.

1.1 Objectives & Research Questions

Before approaching our main objective of providing an Ethereum attack simulation environment, preceding groundwork is done to build a solid foundation. At first, this work aims to give an overview and a categorization of attacks and adversarial behaviour in the context of blockchain technologies, as well as to describe some relevant attacks for this thesis in more detail, resulting in the first research question (RQ):

RQ_1 = What is the current state of the art in regard to attacks against blockchain systems and how can they be categorized?

This work further focuses on outlining different available blockchain simulators, showing their advantages and disadvantages, and ensuring that the proof-of-concept simulator will be based on the most appropriate technologies to overcome issues that current simulators may have. This leads to the formulation of our second research question:

RQ_2 = What is the current state of the art regarding blockchain simulators?

One significant point that also helps with developing the simulator later on is to provide an overview of how exactly the most widely used Ethereum client (Geth) works. This will not only help develop a simulator but also help researchers quickly understand the behaviour of Geth for future research, resulting in the third research question:

RQ_3 = How does the most widespread client implementation, Geth, behave under various conditions?

The main aim of this thesis is to provide an easy-to-use but complex enough proof-of-concept Ethereum simulation environment, where the level of abstraction is tailored to be more suitable for the Ethereum blockchain than generalized state-of-the-art simulators and, in particular, *is capable of simulating attack scenarios* to address the existing research gap between theory and practice by enabling effective and parametrizable simulations of attacks, which leads to our fourth and main research question:

RQ_4 = How can realistic Ethereum network setups and attacks against them be simulated effectively?

Finally, the simulator is used to examine and outline the influence of attacks on blockchain network security and attacker revenue in order to determine if certain hypothetical attack scenarios against blockchain technologies presented by academia translate into practice. For this step, the following attack scenarios are considered, namely the verifier's Dilemma [LTKS15, PTS19] and the selfish mining attack [NF19, SSZ17, NKMS16]. This leads to the formulation of our fifth and last research question:

RQ_5 = What impacts do the selfish mining attack and the verifier's dilemma have on real-world Ethereum setups?

1.2 Outline

1.2.1 Methodology

The methodological approach of this thesis includes the following steps:

1. **Literature Review**

   During this stage, the state of the art of blockchain attacks and simulators is examined with a focus on the Ethereum network by systematically gathering state-of-the-art literature from conferences and journals as well as grey literature (see Chapter 2 *Attacks* and Chapter 3 *Simulators*). This stage focuses on research questions RQ_1 and RQ_2. The data to define the hypotheses for attack simulation later in this work is collected via literature review too.

2. **Technology Review**

   For developing an Ethereum simulator, it is crucial to gain deeper insights into the most widely used and de facto standard implementation of the Ethereum client. This is done by reverse engineering and reviewing the source code as well as summarizing important features, properties and behaviour (see Chapter 4 *Ethereum Clients*). Another part of this step is to compare and analyze existing simulator frameworks to see what features are needed to fit the goals of this thesis (see Chapter 3 *Simulators*). This step concentrates on research questions RQ_2 and RQ_3.

3. **Empirical Evaluation**

   a) *Development of a Proof-of-Concept Ethereum Simulator Implementation*

      A proof-of-concept simulator that enables the user to quickly simulate realistic Ethereum network setups, including attackers or malicious nodes, is implemented based on the results from technology review (see Chapter 5 *Ethereum Attack Simulator*). This part focuses on research question RQ_4.

   b) *Comparison & Evaluation of Simulator Results*

      The previously designed simulator solution allows simulating different forms of adversarial behaviour and attack scenarios under various conditions, allowing the user to adjust relevant parameters. To ensure that the simulator delivers results comparable to the real-world Ethereum network behaviour, it is validated against real-world empirical data to verify that it produces realistic results. To achieve this, properties, such as the block propagation time, are measured and compared to real-world data, again concentrating on research question RQ_4 (see Chapter 5 *Ethereum Attack Simulator*).

   c) *Analysis of Attack Scenarios*

      Hypotheses are defined based on the results of papers from literature review that cover the theoretical outcomes of selected attacks, followed by an extension of the simulator application to enable the simulation of attacker-specific behaviour. After conducting different attack simulations, the thereby gathered data and results form the basis of discussion, where the proposed hypotheses are analyzed regarding the feasibility, the consequences and the likelihood of their realization (see Chapter 6 *Attack Simulation*). The focus here is on research question RQ_5.

1.2.2 Structure of Work

1. **Introduction**

   This section gives a short introduction to this work and its objectives.

2. **Blockchain Attacks**

   A comprehensive overview of blockchain attacks will be given in this section. This includes categorization and a detailed explanation of selected attacks that are important for this work.

3. **Blockchain Simulators**

   An introduction to simulation and emulation is given here, but the central part will be a comparison of various blockchain simulator frameworks to see if they fit the goals for this thesis of efficiently simulating realistic Ethereum networks with the possibility of adding malicious nodes and attackers and without the need for immense computational power.

4. **Ethereum Client**

   This part starts by showing an overview of different available Ethereum client software used today and their role in the Ethereum network. Afterwards, the de-facto standard Ethereum client (Geth) is examined, and its most essential features, properties and behaviours are described in detail.

5. **Proof-of-Concept Simulator Implementation**

   The steps of creating a proof-of-concept Ethereum attack simulator implementation will be presented, together with a detailed explanation of its architecture and behaviour, as well as how to extend the simulator if needed. Furthermore, the validation results for the implementation are presented in this section.

6. **Attack Simulation**

   Hypotheses based on theoretical attack scenarios are defined for selected attacks against the Ethereum system, followed by the implementation and description of the attacker in the context of the simulator. Finally, the hypotheses are tested against the simulator results, which are discussed afterwards.

7. **Conclusion**

   The results are summarized based on the research questions, limitations are discussed and incentives for future research are given in this section.

CHAPTER 2

Attacks

With new technology comes new threats, and if this new technology secures digital assets with potentially high valuations, the incentives to attack the system in order to profit from the attacks are very strong. The goal when designing and implementing a blockchain system is to minimize the attack surface because the success of such system stands and falls with the trust in the technology used. Research has identified many potential vulnerabilities, however, most of them have yet to be observed in the real world against major cryptocurrencies. The problem here is that a multitude of works, e.g., the Ethereum yellow paper [Woo14], suggest that security is given if and only if main premises hold and that under certain adverse circumstances, such as an attacker finding mispriced transaction fees, it is not secure at all, which makes it – in theory – less trustworthy. However, especially in the cryptocurrency space, there is a large discrepancy between theoretically feasible attacks and real-world scenarios, where some of those attacks have little to no effect at all [CPNX19, SGD20].

In this chapter, an overview of the state of the art in research that focuses on attacks against blockchain systems is provided and the attacks are systematically analyzed and categorized. Subsequently, a closer look at selfish mining and verifier's dilemma attacks is taken, which forms the basis for modelling these attacks in the simulation environment. For an overview of past successful attacks, please refer to Section A.1 *Successful Attacks against Blockchain Systems.*

2.1 Categories

Depending on the properties that are used to categorize attacks on blockchains and depending on the level of detail used for classification, different attack categories are used to describe attacks. For example, Chia et al. [CHH+19] introduce three main categories, namely (i) *Operations Security (OPSEC)*, where the targets are individuals or organizations and the goal is to gain access to critical information, (ii) *Smart Contracts*, containing attacks that target applications running on the blockchain rather than the blockchain itself and (iii) *Consensus Protocol*, which contains attacks that try to maliciously gain advantages over other miners by using the flaws of the distributed consensus mechanism of the blockchain.

Wang et al. [YWL+19] and Mosakheil [Mos18] consider the architecture of the blockchain for their classification of attacks leading to the following five categories (i) *Application Layer*, consisting mainly of attacks against central nodes, i.e., exchanges and mining pools, software used to interact with blockchain systems and also traditional flaws such as tampering a wallet address used for, e.g., an Initial Coin Offering (ICO), (ii) *Execution Layer*, where vulnerabilities of software running on the blockchain, e.g., smart contracts, are exploited, (iii) *Data Model Layer*, attacking the data model of a blockchain system such as signature and encryption, or using the unerasable nature of information written to a blockchain to distribute dangerous data, (iv) *Consensus Layer*, which again contains attacks that try to maliciously gain advantages over other miners by using the flaws of the distributed consensus mechanism of the blockchain and (v) *Network Layer*, containing attacks that use the properties of a distributed system for their advantages. Mosakheil [Mos18] also proposes categories based on the affected blockchain processes and the primary targets of attacks, but these categories will not be further discussed here.

Saad et al. [SSN+19] show how to categorize attacks based on the primary affected targets as well, but they also suggest utilizing the following categories for classification (i) *Blockchain Structure*, containing attacks related to potential flaws in the blockchain structure such as forks, (ii) *Peer-to-Peer (P2P) system*, where attacks using the distributed nature and related protocols of blockchains, like, e.g., selfish mining, are pooled together and (iii) *Blockchain Application*, consisting of attacks on applications utilizing the blockchain systems.

Based on the categories defined in various sources above, this thesis introduces a slightly modified and supplemented set of categories previously defined by Saad et al. and Chia et al. [SSN+19, CHH+19], explained in the following:

- Development & Operations Security (DevOpSec)
- Blockchain Structure
- P2P System
- Blockchain Applications

2.1.1 DevOpSec Attacks

All incidents that occur because some adversary gains access to confidential information by compromising an organization or individual fall into this category [CHH+19]. Additionally, security threats that emerge from software development not following the highest security standards, social engineering tactics and misappropriation are also contained in this category. This type of attack is neither newly emerged nor limited to blockchain systems. The simplest example here is someone unintentionally exposing a private key of a cryptocurrency wallet and therefore having an adversary obtain control of the assets within. Due to the nature of DevOpSec attacks, exploring them is not easily done through simulation.

2.1.2 Blockchain Structure Attacks

Every implementation of a blockchain uses more or less different structures and constructs by design, e.g., linked lists or trees, and those constructs might have their own vulnerabilities. Forks and orphaned blocks, for example, are possible consequences [SSN+19]. Blockchain security is strongly dependent on cryptography, such as hash functions and signatures, which are generally considered secure for now, but with further development and research, they might not be safe in the future. Nevertheless, there are some known attacks that belong in this category, like brute-forcing, collision attacks and pre-image attacks [YWL+19, Mos18, SSN+19]. One example of a collision attack is, e.g., the self-designed cryptographic hash function Curl-P-27 used by IOTA, which was later shown to be insecure by Heilman et al. [HNT+20]. Because it is possible to store arbitrary information on blockchains and due to the persisting, unerasable nature of the blockchain, potentially problematic data can be stored on blockchains and such attacks belong to this category too [YWL+19]. Analysis of some blockchain structure attacks can be done exceptionally well through the usage of simulation (e.g., by changing various limits and upper/lower bounds), others may not readily be analyzed through simulation at all, e.g., hash collisions or brute force approaches, which may be again more in fields of cryptography, formal analysis or statistics.

2.1.3 P2P System Attacks

Attacks on the P2P system are either directed towards structures and protocols common in distributed systems such as the Domain Name System (DNS) or distributed ledger protocol specific (e.g., consensus mechanism). This category includes all attack vectors that affect the participants' behaviour in a blockchain environment, as well as the communication between various parties, e.g., DNS hijacks, eclipse attacks or selfish mining attacks [SSN+19, Mos18]. P2P system attacks appear particularly suitable to being explored and analyzed through simulation because, according to Aristizabal et al., "Despite of the recent interest in P2P systems and applications, little work has been done in the formal analysis of P2P protocols" [ALRV05] and the replication of a live system with lots of participating nodes generally appears difficult, making simulation seem like the best-suiting approach.

2.1.4 Blockchain Application Attacks

As blockchain systems can be used by different applications working on top of it or are strongly related to it, more potential vulnerabilities are introduced. Applications working on top of blockchain systems include, e.g., smart contracts or cryptocurrencies and token systems [SSN+19, Mos18, YWL+19]. On the other hand, an example of an application that is strongly related to blockchain technology is a cryptocurrency exchange [YWL+19]. Simulation of attacks on blockchain applications is rather difficult and may not be worthy because, e.g., exchanges run their own proprietary software and attacks on them are primarily not using flaws of a blockchain protocol itself.

2.1.5 Attack Categorization

Table 2.1 was created by systematically reviewing available literature about known attacks and assigning them to the related categories based on their flaws used for execution. However, this may not be a complete list as some attacks may be missing. An attack can fall into multiple categories concurrently. The table shows that most known attacks fall into the P2P system and consensus category. This may not be a coincidence because this category provides one of the most extensive attack surfaces. The second most known attacks are from the application category and with increasing applications, this attack surface also increases. The least known amount of formally described attacks belong to the categories DevOpSec and blockchain structure, however, that does not indicate their importance in real-world scenarios, as Section A.1 *Successful Attacks against Blockchain Systems* shows. The reason for this may be that it is difficult to analyze and describe these attacks formally.

2.2 Attack Types Considered Within This Thesis

Table 2.1 shows that most attacks can be assigned to the P2P category, which is also the category that is conducive to being analyzed through simulation, highlighting its importance for the practical evaluation of those attacks. As there are a lot of possible attacks in the world of blockchain systems, the upcoming section focuses only on the selfish mining and verifier's dilemma attacks from the P2P category and elaborates on their execution in detail, as well as describes their variants.

Attacks	DevOpSec	Structure	P2P	Application
Selfish Mining [NF19, SSZ17, NKMS16]			✓	
Social Engineering [SGD20]	✓			
Wallet Theft [SGD20, BDWW14]	✓	✓		
Flawed Key Generation [Llo15]		✓		
Exchange [SGD20]	✓			✓
Collision [GCR16]		✓		
Finney [SSN+19]			✓	
Double Spending [KAC12, Lei15]			✓	
Bribery [JSZ+19, Bon17]	✓			
DNS Hijack [AZV17]			✓	
EVM Bytecode [Mos18]				✓
Brute-Force [YWL+19]		✓		
Vector76 [SZ16]			✓	
EREBUS [TCM+20]			✓	
Consensus Delay [EGSvR16]			✓	
Goldfinger [Bon17, JSZ+19]			✓	
Majority / 51% [Bas15, SGD20]			✓	
Pool Hopping [Ros11]				✓
BGP Hijacks [AZV17, TCM+20]			✓	
Eclipse [MHG18, HKZG15, TCM+20]			✓	
Block Withholding [CB14, SSZ17]			✓	✓
Fork After Withholding [KKS+17]			✓	✓
DDoS [SGD20, Mos18]			✓	
Cryptojacking [ELMC18]			✓	✓
Sybil [LSM05, SGD20]			✓	
Overflow [SMGC20, Gri17]				✓
Delay Routing [AZV17]			✓	
Verifier's Dilemma [LTKS15, PTS19]			✓	
Refund [MSH17]				✓
Forks [Eya14]		✓		
IMA [JSZ+19]			✓	
Partition [SCN+19]			✓	
Orphaned Blocks [DW13]		✓		
Timejacking [Vya14, SGD20]			✓	
Replay [CPNX19]				✓
Alternative History [Lei15]			✓	
Transaction Malleability [SGD20, ADMM15]			✓	
Reentrancy [SMGC20, Gri17]				✓
Balance [SSN+19]				✓
Vulnerable Signature [BHH+14]		✓		
Partition Routing [AZV17]			✓	
Quantum [ABL+18]		✓		

Table 2.1: Categorization of blockchain attacks

2.2.1 Selfish Mining

The selfish mining attack is used by PoW miners that want to maximize their rewards through a special strategy by exploiting the cryptocurrency's incentive scheme through preventing other honest peers from receiving their fair share. For the most part, this strategy consists of a simple trick that forces other honest miners to waste computation power on stale branches or blocks that do not make it into the final blockchain [ES13, CPNX19, SGD20]. The trick is that selfish miners - in contrast to the standard mining protocol - do not unveil a newly found block immediately but keep it secret so that the honest miners still mine to find a block that will possibly be dismissed, while the selfish miners work on top of the secret block, giving them a strategic advantage. This behaviour results from the honest miners following the heaviest chain, the chain with the most cumulative proof of work done. The moment the honest miners find a block of the same chain height, the selfish miners release the secret block, leading to a block race in which the adversary's success probability depends on the tie-breaking selection rule in case of two blocks occurring with the same height, on the attacker's network connection and on its ability to control or even censor the P2P network [ES13, NF19, SSZ17, LLW+19]. There are various selfish mining strategies available in literature that mainly differ in the exact timing of the attacks, e.g., when to release a block under specific circumstances [ES13, NF19, SSN+19, LRDJ18, SSZ17]. Figure 2.1 illustrates the selfish mining strategy, with M$_h$ being the honest miner and M$_s$ being the selfish miner. The selfish mining attack starts when M$_s$ finds block b$_{n+1}$.

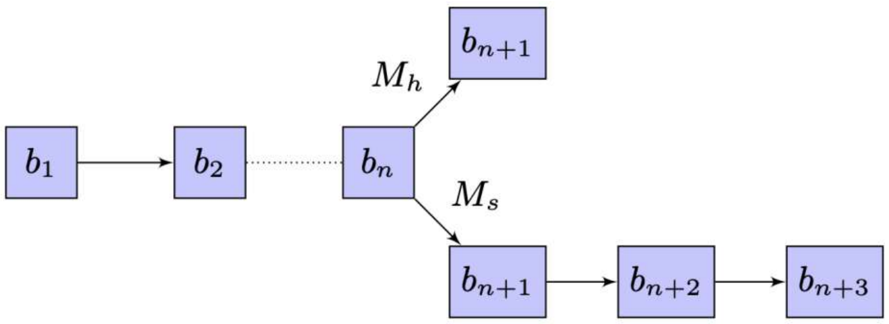


Figure 2.1: Selfish mining illustration [SSN+19]

The fraction of mining power is relevant for selfish miners because, with a fraction too low, it is not likely for the miners to profit from the slight advantage they have through selfish mining. The fraction of mining power needed for selfish miners to be profitable in Bitcoin is about 33.3%, in Ethereum, the threshold is about 16.3%, primarily due to its uncle rewards [ES13, NF19]. Additionally, the placement in the network and the importance of peers in the P2P network, modelled by a parameter γ that denotes the percentage of peers deciding to mine on top of the attacker's block, is vital to the selfish miner, as this helps winning block races [SSZ17]. Eyal and Sirer [ES13] show that a selfish mining attack can be detected by monitoring the stale block rates because they rise in case of an attack. A second indicator may be a minimal time gap between the releases of two consecutive blocks [ES13].

This attack is particularly interesting because it does not only impact a miner's profitability but could possibly have major impacts on the security assumptions of the protocol [ES13]. Simulation of this attack could gain deeper insights into this issue.

Variants

The **block withholding attack** is a variation of the selfish mining attack but is targeted against mining pools. Mining pools keep track of provided hash rate through partial PoWs or sub-puzzles that have to satisfy a smaller difficulty than the block's target difficulty and are, therefore, easier to find. The nonce to create such a sub-puzzle is called share, which has a certain probability of solving the main PoW. The share is also submitted to the pool manager to verify the participation of the miner. If a block is found by a pool member, the miners are paid a share according to the number of valid shares submitted [LV17]. The simplest way of performing such an attack is to only publish shares or partial PoWs to the pool manager but never publish a found block. This will harm the revenue of the pools that pay per share [ES13, KKS$^+$17, Ros11]. Another version of the block withholding attack is started by the manager of a mining pool, who registers as a member of the victim pool and forwards tasks he receives to some of his own miners, called infiltrating miners. If an infiltrating miner finds a partial PoW, he forwards it to the victim pool, full PoWs are discarded. This way, the attacking pool loses some of its mining power but additionally receives rewards from the victim pool and also harms the outcome of the victim pool [Eya14, Mos18]. In the last variation of the block withholding attack, called lie in wait, the attacking miner holds back his found full PoW to submit further partial PoWs and extend his revenue but eventually submits the full PoW too [Ros11].

A **fork-after-withholding attack** is an extension of the block withholding attack combined with elements from the selfish mining attack that guarantees a higher outcome for the attacker in any case [SCN$^+$19, KKS$^+$17, Mos18]. The attacker uses a part of its mining power to mine for a victim pool and the other part for mining himself. If the attacker finds partial PoWs, he submits them to the pool and if the attacker finds a full PoW for the pool, he withholds it initially. There are now three different scenarios available, (i) the attacker finds a block itself, so he receives the full block reward and discards the full PoW of the victim pool, (ii) another miner that is not part of the victim pool, publishes a block, then the attacker immediately releases the full PoW to the pool which creates a fork / block race and (iii) another miner of the victim pool finds a full PoW, making the attacker collect the rewards for its prior shares. There is now only one scenario where the attacker does not receive a reward for a block, namely if the submitted full PoW does not win the block race [KKS$^+$17, Mos18].

2.2.2 Verifier's Dilemma

Another attack that aims at wasting computational resources of victims is the so-called verifier's dilemma which was first presented by Luu et al. [LTKS15]. In blockchain systems such as Ethereum and Bitcoin, miners must verify a new block before appending it to the blockchain and starting to mine a new block on top of it. However, this verification process lacks remuneration for the miners in the protocol. The only incentive for miners to verify all of the transactions and state transitions is that they will not waste computation in case any transaction in the block is incorrect, which would lead to other peers not accepting their block, and to keep the integrity so that the blockchain system does not loose reputation if, e.g., all miners stop verifying blocks [LTKS15, TR19]. Problems start to arise if not all miners verify each and every transaction in a block. Some miners could try to take advantage by not verifying the whole block, which gives them the benefit of saving time and starting to mine on a new block earlier than others while taking the risk of mining on top of an incorrect block. The risk of mining on top of an incorrect block could also be exploited by an attacker sending incorrect blocks to victims on purpose [PTS19, LTKS15].

Another problem arises in blockchain systems like Ethereum, which allows for arbitrary computation through, e.g., smart contracts, by design. Although Ethereum tries to limit the maximum computational effort for all state transitions per block through the use of gas, which was introduced to meter execution costs and includes it in transaction fee calculation, an adversary could deploy a smart contract that is purposely designed to take as long as possible to be executed. With this smart contract, the attacker can fill up a new block he mined with a transaction that uses the block's whole gas (there is an upper bound all transactions of a block combined can use at most, adjusted by protocol rules [Com21c]), which leads to exceptional high verification effort to be taken on the side of the miners receiving the block, without the need of monetary expenditure on the side of the attacker, although he loses the rewards related to the gas of transactions the block would usually be filled up with. This, in turn, leads to the adversary having an advantage in mining the next block because he is able to start mining on top of the block right away. This advantage decreases with the average block time of a blockchain system [PTS19, ACLAvM20, LTKS15]. Pontiveros et al. [PTS19] showed that a smart contract designed for delaying verification based on the EXP opcode of the Ethereum EVM takes up to 1.21s on the Geth client with a 2.9 GHz Intel i5 processor and with a maximum of 8 million gas. At the time of writing, the gas limit was as high as 12.5 million, extending the time to about 1.9 seconds. The fraction of hash power for the verifier's dilemma attack to be viable with a block reward of 2 Ether and an average transaction fee of 0.08 Ether when running the Geth client is 35% [PTS19].

As a consequence, miners could start to skip verification to overcome their disadvantage from high verification times and this could open the door to other attacks and decrease overall security of the blockchain system [LTKS15, ACLAvM20]. To mitigate the risk of divergence from the protocol, different solutions were suggested (i) the miners could validate non-conflicting transactions in parallel, decreasing the overall verification time of a block [ACLAvM20], (ii) adjust incentives like the average transaction fee and the block reward [PTS19], (iii) a solution called the *verification game*, that outsources arbitrary computational effort to TureBit, an off-chain solution for trustless computation [TR19] and (iv) intentionally wrong blocks can be created to recognize and penalize nodes not verifying a block [ACLAvM20].

2.3 Summary

In this chapter, categories of attacks on blockchain systems were proposed and detailed explanations of each category were given. In succession, a categorization of known attacks from literature was performed based on the flaws used for their execution. Finally, a closer look at the selfish mining and verifier's dilemma attacks was given, explaining their implications in detail. These two attacks were chosen because of their comparable properties and effects on the Ethereum network and this choice will lead to these two attacks being evaluated with the simulator in Chapter 6 *Attack Simulation*.

CHAPTER 3

Simulators

To develop systems and to further evaluate and test different design parametrizations and configurations, without the need to adapt or even implement the whole system or resource-heavy deployments, modelling is crucial. A model represents the system that is to be tested in a way that is similar to it to a certain extent but considerably more abstract and enables detailed analytics to help engineers build or change a system while keeping the risk as low as possible [BCNN10, KB12]. Maria defines a good model in a very clear way: "On the one hand, a model should be a close approximation to the real system and incorporate most of its salient features. On the other hand, it should not be so complex that it is impossible to understand and experiment with it." [Mar97]. Additionally, a good model takes the properties of the underlying system into consideration and is selected according to the tools used in and the goals of a simulation study. Simulation itself is the execution of a model [KB12, McG02].

To summarize, the situations modelling and simulation, in general, should be used include but are not limited to:

- simulation of implementation changes of a system under test,
- evaluate a system that is not yet implemented,
- simulate different inputs,
- estimation of complex systems,
- develop knowledge related to a system (e.g., flight simulators for pilots),
- evaluation of the interaction between numerous actors,
- or entirely for fun purposes (e.g., games) [Whi19, BCNN10, Mar97].

According to Banks et al., modelling and simulation should not be used if one of these rules apply:

- The problem is solvable using common sense,
- the issue can be solved using analytics,
- running the experiments directly on a real-world system is less expensive,
- the possible savings are below the simulation costs or the resources regarding time and costs are simply unavailable [BCNN10].

The scope of this chapter is to give an introduction and to examine the different types of simulation and emulation, as well as to explore various available blockchain simulation frameworks to give an overview and select the perfect fit for the targeted needs.

3.1 Simulation vs. Emulation

To begin with, there are two major ways of evaluating a model, simulation and emulation. Although the term simulation is widely used as a supercategory for simulation and emulation, there are major differences as well as advantages and disadvantages coming with each possibility, and this section will guide through them.

3.1.1 Simulation

The abovementioned simulation model is the base for a simulation study and has to represent the system under test in the closest way possible, it has to imitate the system. Therefore, to create the model, certain assumptions are made and some details are abstracted, e.g., implementation details or inputs and outputs from sensors and actuators. Another essential property of a simulation study is repeatability. That means recurring experiments with the same parameters should lead to the same results of the simulation run. However, this is rather difficult to achieve for simulation models containing random variables [McG02, Whi19, BCNN10]. According to McGregor, simulators are often used to "[...] test and develop different solutions in order to arrive at a best solution, based on an accepted set of pre-defined metrics." [McG02]. An example of simulation application is a modern high-tech factory that should be built in the most optimal way possible. To evaluate this, it is impossible to build the factory upfront and move parts of it afterwards if needed. Hence, a simulation model is created, including, e.g., all the assembly lines and their inputs and outputs, and through reordering them, changing their timing or other input parameters in the simulation study, the best way to build the factory can be developed.

3.1.2 Emulation

Emulation, on the other hand, can be carried out either against a real system that is virtualized on a different host system or against an emulation model that works similarly to a simulation model but includes parts of the real system under evaluation [McG02, JH11]. Fully virtualized emulation describes, e.g., running software meant to be used on thousands of nodes controlled by different entities in a distributed system on a single host with a modified interface to emulate the entire network. Related to software, an emulation model could be described as a simulation model, but instead of abstracting away the full implementation, core parts of the original implementation of the system under test are preserved and executed as part of the simulation. According to McGregor, emulation is useful to "[...] test the operation of the control system under different system loading conditions, and as a risk-free means of training system operators and maintenance staff" [McG02]. With these two examples in mind, the advantages of emulation are clear, they are meant to be as accurate as possible by keeping the original implementation intact or preserving at least some parts of the system under test, but this accuracy leads to very low scalability compared with simulation. In the above example of testing a distributed system, an emulator could only run a tenth or even a hundredth of nodes compared to a simulator [CFS06, McG02, JH11]. A good example of emulation is a computer system that is tested for random user inputs. The system itself remains unchanged and is then emulated using, for example, a fuzz tester.

3.2 Types of Simulators

Simulators and models can further be categorized according to various properties. This section presents and explains these properties.

3.2.1 Time & State Progression

The most basic types of simulators can be categorized in are *discrete* or *continuous* models or systems. Discrete models, on the one hand, change their state only at specific, discrete times. The state of a continuous model, on the other hand, changes steadily in time. It should be noted that not always a continuous model is used to model a continuous system. Likewise, that applies to discrete models [BCNN10].

3.2.2 Static vs. Dynamic

Models that are static express a single point in time. Dynamic models express a system's change over time [BCNN10].

3.2.3 Stochastic vs. Deterministic

If a model does not work with random variables, it is called *deterministic* and the same input will lead to the same output every time. The opposite is called a *stochastic* model, which leads to results only being statistical estimates because of the use of random variables [BCNN10].

3.2.4 Implementation Strategy

Simulators can also be categorized based on their implementation strategy. Here the most important ones are listed and explained. It is to be noted that not every implementation strategy can cope with all the abovementioned simulation types, so some implementation strategies are instead bound to specific types.

Discrete Event Simulation

Discrete simulation changes its state only at specific points in time. With *discrete event simulation*, this happens through the occurrence of predefined events. Such events can be, e.g., a customer arriving at a shop or a factory worker picking up a workpiece. Discrete event simulation is a dynamic, mostly stochastically used and, as its name implies, discrete simulation method [BW10, BCNN10]. According to Banks et al. and White et al., the basic structure for discrete event simulation systems are:

- *Inputs & outputs.* Inputs, such as parameters, specify how the system is interacted with, and outputs are the results of a simulation derived from the state at the end of any given point.
- *State*, a collection of resources, global variables, time, etc., that describes a system at a given point in time.
- *Entities* and their attributes, which describe objects like, e.g., humans, a server node, etc. and their attributes like, e.g., system memory, download speed, etc.
- *Events*, predefined occurrences that change the state. They are kept in an event queue throughout the simulation run, ordered by the time of future occurrence.
- *Activities* that are generated by events and possibly lead to the occurrence of new events. Such activities include, e.g., a computation task whose need arises after the occurrence of an event.
- *Resources*, a collection of limited capacity that can be either globally available to all entities or bound to specific entities.
- *Random number generator* that generates pseudo-random numbers according to a predefined distribution.

- *Clock*, a simple timestamp variable representing the current time within a simulation run.
- *Statistics Collectors* that record events, activities, their duration, system state at given intervals or any statistically relevant number that arises from the simulation run and is needed to create the desired results. [BCNN10, Whi19].

Agent-Based Simulation

The idea behind *Agent-Based Simulation* is not to model the whole system under test but instead model the specific core entities of the system – so-called agents – together with their interaction with one another, the agent-to-environment interaction as well as their internal behaviour that leads to such interactions or follows interactions with other agents or the environment. The environment is responsible for keeping global resources and variables, advancing time and visualization [KB12, CQHM19]. This simulation implementation can execute discrete, continuous, stochastic and deterministic dynamic models. Agent-based simulation has a wide field of applications and according to Chitra et al., it “is used by practitioners and researchers in algorithmic trading, artificial intelligence, autonomous vehicles, cybersecurity, economics, energy allocation, and by the US Commodities and Futures Trading Commission to detect fraudulent market activity” [CQHM19]. Because this simulation variant does not model the whole system but instead generates it from agent interactions, one of the main advantages is that local interaction can be observed better and, although the agent’s behaviour can be complex, it can be visualized easier. The downsides of agent-based simulation are a rather complex implementation, as well as difficult validation and verification [KB12].

Trace-Driven Simulation

The goal of *Trace-Driven Simulation* is to use logs of a small part of a real-world application, so-called traces, and develop a model based on this to simulate a system of larger scope. The first step in this kind of simulation strategy is to collect traces from the system under test. An example goal would be to simulate the network traffic of the whole internet, but traces are only collected from some nodes. These traces are now edited in step two to be able to use them as input to the simulation model. In step three, the model is generated so that it takes the traces of a small part of the internet traffic as input and is able to simulate the whole internet traffic based on this [FP94, OL04]. Due to its nature of requiring trace collection, this method is limited to the simulation of computer systems. [HN05]

Monte Carlo Simulation

Monte Carlo Simulation is another word for stochastic simulation. Any simulation approach that uses stochastic methods is also a Monte Carlo Simulation [Bon01]. It describes the usage of one or more Probability Density Functions (PDFs) in the simulation process to simulate variability. These PDFs have to be defined a priori, which is rather difficult in the case of not yet existing systems under test, but expert knowledge from past simulations can be used to estimate the PDFs in this case. After defining the PDFs, the model and its inputs, the simulation runs a certain number of times which gives a probability for the respective outcomes after computing the statistics [HGL10, Bon01]. An example would be a simulator rolling two dice, executed, e.g., a hundred times, which most probably assigns the highest probability to the number 7 as outcome for such a dice roll.

System Dynamics Simulation

The approach of *System Dynamics Simulation* is to make reliable simulations of sparsely structured problems possible. It combines the "traditional management theory, cybernetics and computer simulation" to do that [Ła17]. Cybernetics is built on feedback theory, which also takes delayed reactions to events into account and additionally differentiates between important and irrelevant data related to the context. System dynamics simulation is often combined with user interfaces to adapt inputs and visualize the different effects immediately [Ła17]. An example of a system dynamics simulation is, e.g., the adoption of a newly launched product with regard to the potential of the product, the early adopters and the amount and pace of imitators on the market.

3.3 Simulation Study

According to Banks et al. and Maria, the steps for a simulation study are the following:

1. *Identify and formulate the problem*, make clear what shall be achieved and how this will be done.

2. *Collect real-world data* to set correct parameters, e.g., for random variable distributions, and if a production system exists, to validate the simulation model later on.

3. *Develop a model*, this includes the conceptualization and translation of the real-world system into a model.

4. *Verify, validate and document the simulator* to lay the foundations for correct results. Verification means ensuring the simulator does what is specified, if it is programmatically correct. Validation means to actually ensure the proper specifications exist. The latter is done by comparing the first simulator results to real-world data, if existing.

5. *Design the experiment* according to the problem specification. Decide what parameters should be used and the length and amount of runs, etc.

6. *Run the experiment.*

7. *Analyze* the results of the experiments, go back to step 5 if more runs or other simulation parameters are needed.

8. *Interpret and report results* [BCNN10, Mar97].

3.4 Existing Simulators

There are a number of blockchain simulators available to this date and most of them are based on discrete event simulation. The three most recent implementations are Blocksim, created by Faria and Correira, another simulator named Blocksim, created by Alharby and van Morsel and SimBlock by Aoki et al. [FC19] [AvM20] [AOK$^+$19]. They all use a high level of abstraction, making it possible to create scalable simulations of PoW blockchains with the possibility to adjust different input parameters. The option to simulate a variety of PoW blockchains requires them to ignore or abstract away some of the specific functionality and unique properties of the client applications, though, which is likely to impact accuracy. This has to be done because it is not feasible to simulate the computationally expensive part of solving hash puzzles. Emulation, on the other hand, is not commonly used in scientific literature, as the required resources hamper scalability and simulation speed, although Miller et al. appear to address this issue through their Bitcoin simulator, which directly executes the Bitcoin reference client implementation [MJ]. Another approach was taken by Rosa et al. in 2019 with their agent-based LUNES-Blockchain simulator, which makes use of parallel and distributed simulation for a considerable performance boost [RDF19]. Within this section, existing blockchain simulators that appear relevant to this work are presented and their characteristics and properties are compared to one another.

Simulators are examined for the following properties:

- Suitability for simulating *Ethereum*, which is crucial for this thesis,
- simulation of *attacks*, so that adversaries can be integrated into the simulation,
- *network* modelling, which is a vital feature to simulate the implications of certain attacks,
- authenticity of the *consensus model*, to get the most detailed approximation of the real Ethereum consensus model,
- *extensibility*, again essential to integrate attackers into an existing simulation framework, but also crucial for simulator extension in general and
- *resource intensiveness*, so one can execute simulations on a single machine.

3.4.1 Blocksim (Faria & Correira)

The BlockSim (Faria & Correira) (Blocksim (F)) simulator is a stochastic, dynamic Discrete Event Simulation Engine (DESE) created to simulate various blockchain systems and is powered by Python and the underlying framework PySim. As part of the blockchain simulator analysis, the author examined the codebase of Blocksim (F) and found it to be well-structured and clear to understand. The nodes are categorized as miners and non-miners. All nodes are interconnected, so every node has all other nodes as peers. It further includes a network model, allowing the configuration of different network characteristics, such as latency and throughput, that are based on real-world measurement data. A model of Ethereum's block validation is implemented by adding randomized delays, but the granularity of the simulator does not add execution overhead based on the transaction level and hence does not simulate gas. The block creation is controlled centralized, meaning that the simulation world decides which node will create the next block based on a normal distribution. The transaction creation is also controlled centralized and the simulation engine chooses a node that broadcasts them in batches. Blocksim (F) claims to require approximately 27 minutes to simulate a network with 400 nodes and 2000 transactions. In local simulation experiments, the author of this thesis was able to confirm these runtime estimates [FC19, Far].

However, Blocksim (F) does not include some of the details of block propagation, for example, if a new block is mined by a node or received from another node on the network, it only broadcasts the block hashes to its peers, whereas in Ethereum the execution client Geth broadcasts the whole block to some of its peers and the hashes to the rest. This is important when simulating attacks on the Ethereum network, where the reception order of a new block is crucial. Furthermore, the missing peer selection does not reflect a live Ethereum network since, in the simulated network, the communication graph is fully connected, giving every node a distance of 1. Additionally, centralized block creation is not suitable for the goal of this work because it is essential for attack simulation, such as the selfish mining attack, that the block creation depends on the local node state. A problem found during the code review conducted by the author of this thesis may be the difficulty calculation of Blocksim (F). The difficulty is usually calculated in a way that blocks created in a shorter amount of time have a higher difficulty, which is crucial for block ordering. The way it is implemented here assigns a higher difficulty to a block that is created in more time, which appears to break the expected properties of a PoW blockchain (see blocksim/models/consensus.py#calc_difficulty [Far21]) [FC19].

3.4.2 Blocksim (Alharby & van Morsel)

BlockSim (Alharby & van Morsel) (Blocksim (A)) is a light-weighted, stochastic, dynamic DESE and is, much like Blocksim (F), also powered by Python but does not use a discrete event simulation framework. Instead, it implements the event scheduler and queue from scratch and allows easy extension and simulation of different blockchains, all while providing a well-structured codebase. Nodes are categorized into full and light nodes, which has an effect on transaction handling. Within this simulator, the communication graph is also fully connected, giving every node a distance of 1. A highly simplified network is modelled within Blocksim (A), only including a delay based on an exponential distribution that is configurable as well as abstracting away the details of block propagation in Ethereum. In regard to configurability, Blocksim (A) allows configuring the most interesting properties like the block interval, size and propagation delay, transactions per second, size and average fee, as well as the number of nodes in the network and uncle block properties. The block creation is modelled on a node level by using an exponential distribution. Transaction gas usage and block verification are not modelled in any way, which unfortunately makes Blocksim (A) impractical to use for this thesis [AvM20, AvM19]. All those findings were inferred from a code review the author has conducted.

3.4.3 SimBlock

Another DESE for simulating various blockchains is SimBlock by Aoki et al., however, it is only verified using Bitcoin data. This is, again, an event-driven, stochastic, dynamic and discrete simulator. This is the first simulator considered in this thesis that actually uses a peer selection algorithm. The network model, in general, is sophisticated within SimBlock, also including throughput. Block creation is modelled on the node level, whereas transaction creation and propagation, based on the code review, appear not to be modelled at all and block verification seems to be missing too. Configuration possibilities include block size and mining interval, number of nodes and peers, location of the node and hash power, as well as network bandwidth and propagation delay. The paper introducing SimBlock offers no details about the speed of execution [AOK+19]. Although SimBlock offers the unique feature of peer selection, the missing model of block verification and its dependence on transaction gas usage, together with the missing validation of Ethereum simulations, renders the simulator ineligible for the goals of this work.

3.4.4 LUNES

Up to now, all simulators outlined in this section were for single-threaded use only. The agent-based, discrete, Large Unstructured Network Simulator (LUNES) by Rosa et al. is a Parallel and Distributed Simulation (PADS) simulator and the first of its kind that manages to execute parallel simulation. This is achieved using the Advances RTI System (ARTIS) middleware and a software layer on top of it named Generic Adaptive Interaction Architecture (GAIA). Blockchain nodes are represented as agents and this should ease the development of a simulation model and increase extensibility, according to Rosa et al.. Although the LUNES paper claims a significant speedup in comparison to single-threaded DESEs, it admits that the parallel execution on different execution units (e.g., CPU cores) leads to "[...] a relevant amount of execution time is spent in delivering the interactions between the model components." [RDF19]. Together with a focus of this work on a sophisticated message-passing model between nodes, this may lead to a significantly higher execution time compared to single-threaded simulators.

Additionally, the segmentation of a simulation model for the use in a PADS is much more complex than with a DESE, which leads to more complex simulator verification. The paper about LUNES focuses on Bitcoin only and unfortunately, it does not disclose much more information about simulation details the implementation provides with regard to the level of sophistication of the underlying models [RDF19].

3.4.5 Shadow-Bitcoin

The Shadow-Bitcoin simulator is a plugin developed by Miller and Jansen for an existing simulator framework named Shadow, capable of parallel and discrete event simulation and consisting of a simulator core and virtualized software executed by plugins. Miller and Jansen developed the Shadow-Bitcoin plugin, which is capable of emulating the Bitcoin core software by providing interfaces for inputs and outputs as well as application hooks to the Shadow core. This makes it the first blockchain simulator providing real emulation capabilities [MJ]. However, while this adds much precision to the simulation results, it is not suitable for a wide variety of applications and tests because adapting the core implementation of a blockchain client is often not feasible. Furthermore, emulation remains resource-intensive and the adaption of this plugin to fit the Ethereum protocol is challenging to implement, making it not ideally suited for the envisioned goals of this thesis.

3.5 Summary

Sim.	ETH [1]	Consensus [2]	Network [3]	Attacks [4]	Ext. [5]	Resources [6]
Blocksim (F)	+	−−	~	−−	+	++
Blocksim (A)	+	−−	−	−−	+	++
SimBlock	−	−−	+	−−	+	++
LUNES	−	+	+	−−	+	−−
Shadow-Bitcoin	−	++	++	−−	−	−−

[1] Suitability for simulating Ethereum
[2] Authenticity of consensus model
[3] Network modelling
[4] Simulation of attack scenarios
[5] Extensibility of the simulator
[6] Resource intensiveness

Table 3.1: Comparison of existing simulator frameworks

In conclusion, simulators and emulators are used to reach different goals. Simulators are best suited to, e.g., quickly test different internal parameters and their impact on the overall system behaviour and characteristics. It is also the only possible choice if the system under test does not yet exist. Emulation, on the other hand, is useful, e.g., if the system itself shall stay intact and only external influences like load, network conditions or human interaction should be modelled, or if there are enough resources available to use the most precise method. Finally, if there is a need to test a system consisting of a significant amount of instances of the system under test – e.g., a distributed system with a lot of nodes and communication between nodes – without having many resources available, simulators are the way to go.

Table 3.1 gives an overview of examined simulators and rates their properties on a rank ranging from −− to ++. First thing that can be stated is that none of the simulator frameworks is prepared to simulate adversarial behaviour, which is a base requirement for the purposes of this thesis. However, this could be bypassed by offering good extensibility by design. The collected data further shows that the simulators claiming to be able to simulate Ethereum have rather restricted consensus and network models, rendering it difficult to simulate anything but normal network behaviour. They offer extensibility up to a certain amount through their well-structured codebase and are lightweight to execute. The multi-threaded simulators LUNES and Shadow-Bitcoin, on the other hand, offer sophisticated consensus and network models but have neither shown to enable simulating the Ethereum network nor offer great extensibility. Additionally, they are resource-intensive to run, especially when handling a significant amount of inter-node messages.

Regarding the existing simulators, we concluded that there is not a single simulation framework available that implements or models all details needed for sufficiently detailed attack simulation against the Ethereum network. The reasons include the inability to offer attack simulation out of the box, combined with either a consensus and network model lacking necessary detail for single-threaded simulators or combined with resource-intensiveness and insufficient or burdensome extensibility for multi-threaded simulators. Extension of existing simulator frameworks is considered infeasible because no framework offers preconditions that are near our requirements. Building a new simulator with proper design decisions for Ethereum attack simulation made upfront is considered viable. Details of our simulator will be discussed in Chapter 5 *Ethereum Attack Simulator*).

CHAPTER 4

Ethereum Clients

According to ethernodes.org, Geth is the most used Ethereum client at the time of writing. Out of 11232 full nodes connected to the mainnet, 9187 (81.79 %) use Geth, followed by openethereum, together with its predecessor parity with only 1505 (13.40 %). If only the 8438 fully synced nodes are taken into account, geth holds a share of 86.93 % and second-placed openethereum only 10.70 % (see Figure 4.1) [eth20b, eth20a, Com20]. Simulator implementations proposed by research make assumptions on how the Ethereum client software behaves, however, in practice, there can be discrepancies, e.g., the Blocksim (F) application assumes that a client never receives a full block within one message, but actually, when using the geth client, a full block is sent to at least some connected peers upon block mining or block reception [Com20] [FC19]. The main problem here is that although there is a whitepaper and a yellowpaper showing the technical specifications, the implementation sometimes differs slightly from that or the specifications leave out some details necessary in some models needed for Ethereum simulation [Woo14, But21].

This thesis takes a deep look at Ethereum clients and their behaviour by giving an overview of the available clients, their tasks and also systematically reviewing the code of the most popular and de-facto standard client, Geth, to help researchers and other interested readers quickly understand its behaviour, as well as to do the groundwork for modelling the simulator implementation later in this thesis.

4.1 Clients

Ethereum clients have a broad area of application when it comes to their tasks in the Ethereum blockchain environment. They are not only used to connect to the Ethereum network, they themselves make up the network. This implies that if a client is slightly changed so that, at some point, a contradiction to the original client is created, then a new network is created, called a hard fork. Available Ethereum clients at the time of writing include but are not limited to:

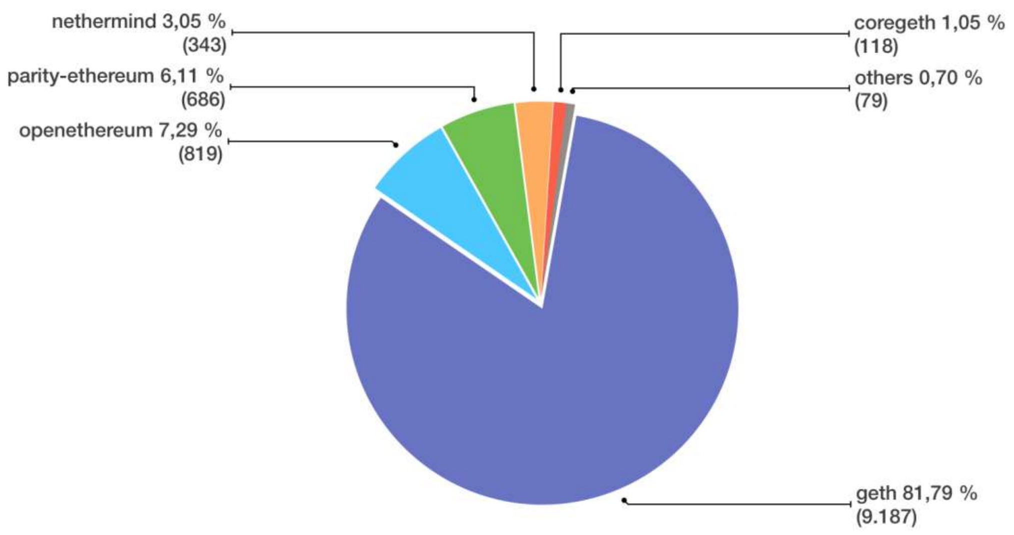


Figure 4.1: Ethereum clients distribution [eth20b]

- Geth [eth20a]
- Openethereum (former parity) [Com20]
- Nethermind [Net21]
- Coregeth [Com21a]
- Besu [Fou21c]
- Trinity [eth21b]
- Aleth (former cpp-ethereum) [Com21b] and
- Ethereumj [Com21c].

When comparing the amount of available Ethereum clients with the distribution of clients (see Figure 4.1), it clearly shows that although many different clients are available, most participants use the Geth client. This may be because Geth is one of the original Ethereum implementations, but it may be a problem regarding the goal of decentralization, as the need for decentralization also applies to client software to mitigate the possible errors in a single client application.

The tasks a client has to manage within the Ethereum network are:

- connect to the P2P network
- select peers
- help peers to select peers

- sync blockchain state with peers
- define the consensus mechanism
- propagate blocks
- propagate transactions
- verify blocks and transactions
- mine blocks
- keep the blockchain state
- create and manage accounts/wallets
- light client mode
- provide a Javascript Object Notation (JSON) Remote Procedure Call (RPC) Application Programming Interface (API).

There is also another difference between clients' usage because a client can be used to start (i) a *full node* or full client that fulfills all of the abovementioned tasks and (ii) a *light node* or light client that is designed to run the minimal functions to be able to have a high certainty about the Ethereum blockchain state with minimal space and computational power requirements to run it on, e.g., mobile phones by downloading only block headers and verifying just enough to meet this demands. Furthermore, light clients can be divided into *light clients*, which process all blocks, *partially light clients*, which process all blocks but have limited storage and *fully light clients*, which mainly process nothing and are bound to, e.g., transactions that affect a single account [Fou, Woo14]. However, this work will not go into more detail regarding light clients.

4.2 GoEthereum (Geth)

Geth is one of the three original Ethereum protocol implementations, together with cpp-ethereum, which is now called Aleth [Com21b] and Pyethapp, which has been officially deprecated since 19.07.2018 [eth21a]. Geth is written in Golang (Go), a compiled language created in 2009 by employees of Google Inc. [PTG09]. With a share of >80% (see Figure 4.1), Geth is currently the most widely used Ethereum client available. The upcoming sections focus on systematically reviewing the code of Geth and giving an overview of its behaviour with a focus on the communication between nodes. The basis for the code review in the upcoming sections of the thesis is version v1.9.21 of Geth with revision number d81c9d9b from 09.09.2020 [eth20a].

4.2.1 Difficulty Calculation

The difficulty calculation serves multiple purposes. First of all, it helps selecting the longest chain (the chain with the highest total difficulty, which is the accumulated difficulty of all blocks in the main chain), which is crucial for the Ethereum consensus algorithm. Additionally, it is adjusted with every mined block to keep the time between mined blocks consistent [Woo14]. This is done by increasing the difficulty of the block when it was mined earlier than 9 seconds after the last block, decreasing it if it was mined later or equally 18 seconds after the last block, and keeping it the same when mined after 9 to 18 seconds. If the block includes uncles, the range where the difficulty stays the same is shifted to between 18 and 27 seconds[Woo14]. This prefers blocks containing uncles by assigning them a higher difficulty having a similar timestamp. Uncle blocks are stale or orphaned blocks not contained in the longest chain, e.g., because they lost a block race, that are referenced by a block in the longest chain, also called nephew block, and thus are also compensated by the Ethereum protocol [LHXL20]. The minimum difficulty a block can have is 131072, which is also the genesis block's difficulty. The so-called difficulty bomb is another thing that adjusts difficulty towards a higher value. This is designed to increase the difficulty at an exponential rate every 100000 blocks once a given block is passed, with the idea of forcing the transition to a PoS consensus. The current difficulty bomb starts at block 9000000, this block was already mined. That, and the uncle rate of the Ethereum network, is the reason why higher block times than ten seconds can be observed in practice [Woo14]. Algorithm 4.1 shows how the difficulty of a block is calculated in detail. It should be noted that divisions in pseudo code presented in this section are integer divisions.

4.2.2 Peer Selection

When an Ethereum node connects to the network for the first time, it uses hardcoded bootnodes to find an entrypoint. The sole purpose of bootnodes is to enable nodes to bootstrap their P2P network, i.e., to discover their first peers. When the first actual peer node is found, other peers are discovered using a Kademilia-like P2P protocol [MHG18, ethc, ethd, ethb]. For a node operator, it is also possible to add static or trusted nodes, while a connection to the former is always kept, the latter can be replaced by other nodes in the discovery process, but they are always allowed to connect [MHG18, ethc, etha]. As the network and discovery processes are comprehensive parts of the Ethereum system that would need strong knowledge, examining this would go far beyond the scope of this thesis. Interested readers may use one or more of the following references to have a look at this [MHG18, ethc, ethd, ethb] or lookup the processes within Geth code themselves (see p2p/server.go#Start, p2p/discover.go#loop, p2p/server.go#listenLoop, p2p/server.go#SetupConn, p2p/server.go#startDial [eth20a]).

Algorithm 4.1: Ethereum difficulty calculation (see consensus/ethash/consensus.go#CalcDifficulty [eth20a])

```
Input: The timestamp of the block $\lambda$, the parent block header $\rho$, the bomb
       delay $\chi$
Output: Block difficulty $x$
$parentTime \leftarrow$ Time($\rho$);
$x \leftarrow \lambda - parentTime$;
$x \leftarrow x/9$;
if HasUncles($\rho$) then
    $x \leftarrow 2 - x$;
else
    $x \leftarrow 1 - x$;
end
$x \leftarrow$Max($-99, x$);
$y \leftarrow$ Difficulty($\rho$)$/2048$;
$x \leftarrow x * y$;
$x \leftarrow$ Difficulty($\rho$)$+x$;
$x \leftarrow$Max($131072, x$);
$fakeBlockNumber \leftarrow 0$;
if BlockNumber($\rho$) $\geq \chi$ then
    $fakeBlockNumber \leftarrow$ BlockNumber($\rho$)$-\chi$;
end
$periodCount \leftarrow fakeBlockNumber/100000$;
if $periodCount > 1$ then
    $y \leftarrow periodCount - 2$;
    $y \leftarrow 2^y$;
    $x \leftarrow x + y$;
end
return $x$;
```

4.2.3 Communication with Peers

Ethereum uses a protocol called *Ethereum Wire Protocol* for communication between nodes, which is based on Ethereum's version of the Recursive Lenght Prefix (RLP) protocol, called *RLPx* [etha]. This section elaborates on the Geth implementation of this protocol.

Block Hashes Received

If a node receives a *New Block Hashes Message* containing one or more block hashes, it filters out all known block hashes and schedules all blocks that remain unknown for fetching using a *Get Block Headers Message*, which is transmitted to the peer sending the unknown hashes (see Section 4.2.3 *Get Block Headers*). All blocks are marked as known by the sending peer (see eth/handler.go#687-705 [eth20a]). Figure 4.2 shows the related state diagram.

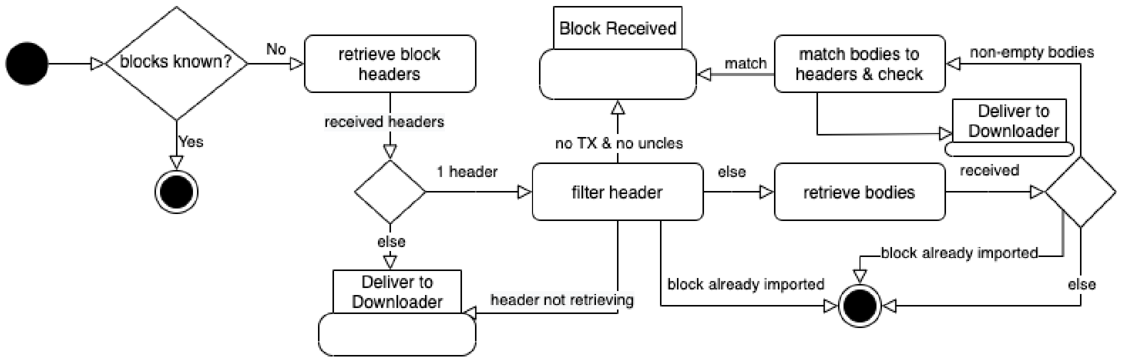


Figure 4.2: Block hashes reception state diagram

Transaction Hashes Received

Once a node receives a *Transaction Hashes Message*, which may consist of one or more transaction hashes, the protocol marks the transaction as known by the peer it has received it from, checks if the transaction is already known and marks all unknown transactions for fetching, using the *Get Transactions Message* (see Section 4.2.3 *Get Transactions*). If the transaction is received from other origins while fetching, the fetching process is stopped (see eth/handler.go#743-757 [eth20a]).

Get Transactions

Ethereum nodes receive *Get Transactions Messages* with an attached array of hashes that the peer node wants to obtain the related transactions for. The receiving node decodes the hashes, and for each hash it knows the corresponding transaction for, the transaction is added to the response object. Unknown transactions are skipped. After all hashes have been processed, the node answers with a *Pooled Transaction Message* (see eth/handler.go#759-793 [eth20a]).

Transactions Received

There are two types of messages available for receiving new transactions, (i) a *Transaction Messages*, which is sent to a subset of one node's peers after a new transaction is received and verified and (ii) a *Pooled Transaction Messages*, which is only sent when a node specifically requested transactions (see Section 4.2.3 *Get Transactions*), but both of the messages are handled similarly. On reception, all transactions are marked as known by the sending peer and each transaction undergoes basic validation, where the hash, the size, the intrinsic and maximum gas, sender nonce and balance, the gas price and the transaction value are checked. After successful validation, the transaction is marked ready to be mined and broadcasted to connected peers that do not know about it, sending the whole transaction object to a subset of the peers and announcing it to the others as explained in Section 4.2.3 *Broadcasting Blocks & Transactions* (see eth/tx_fetcher.go#Enqueue, eth/tx_pool.go#addTxs, eth/tx_pool.go#runReorg [eth20a]).

Get Block Headers

A *Get Block Headers Message* contains either the hash or the number of the origin block (the block the query starts with) and additionally contains the amount of headers to query, a flag named *reverse*, indicating if the query is iterating towards the genesis block (if true) or towards the latest block (if false). The last property of this message is an integer variable named *skip* that denotes how many blocks should be skipped between consecutive headers during the retrieval process. If the origin block is unknown, the process is stopped, the same is true if an end of the chain is reached while querying. (see eth/handler.go#398-483 [eth20a]). The answer to such a message is a *Block Headers Message* (see Section 4.2.3 *Block Headers Received*).

Block Headers Received

After the reception of a *Block Headers Message*, the node verifies that it only received one header, otherwise, the node is currently out of sync or in sync mode and the headers are delivered to the downloader (see Subsection *Synchronize with Peer*). If that is true, the headers are divided into empty blocks, incomplete blocks and unknown blocks. Empty blocks do not contain any transactions or uncles and are scheduled for chain import immediately (see Section 4.2.3 *New Block Received*). Incomplete blocks are scheduled for body retrieval (see Section 4.2.3 *Get Block Bodies*). Unknown blocks are the ones that the node did not explicitly query before. Such headers are delivered to the downloader to check for possible asynchronicity on the next run, but that should only occur in a sync process. The process is pictured in Figure 4.2 (see eth/handler.go#485-536, eth/fetcher/block_fetcher.go#501-591 [eth20a]).

Get Block Bodies

Get Block Bodies Messages contain one or more hashes a querying node wants to retrieve the bodies to. (see eth/handler.go#538-563 [eth20a]). The answer to such a message is a *Block Bodies Message* (see Section 4.2.3 *Block Bodies Received*).

Block Bodies Received

Once a *Block Bodies Message* is received, the receiving node tries to match the tuples of transactions and uncles to previously dispatched block body retrievals by computing and comparing the transaction and uncles hash to the ones from the header. If a match is found, the new block is assembled and scheduled for chain import Section 4.2.3 *New Block Received*). If there are remaining tuples of transactions and uncles that have not been requested, they are delivered to the downloader to check for a possible asynchronicity during the next cycle (see eth/handler.go#565-589, eth/fetcher/block_fetcher.go#593-659 [eth20a]).

New Block Received

If a new block is received, which either happens on the reception of a whole new block or after a block body has successfully been matched to a block header, the validation begins. If any validation steps fail with an error, the transmitting peer is dropped. In case we came here after matching a header to a body, the first step of checking the transaction and uncle hash and marking the block known by the peer is skipped. If now the block number is in the future (i.e., if the block number is more than one ahead of the current chain's head), the block is scheduled for later import, which is done by adding it to the end of the import queue. Afterwards, a check is done if the block has either an unknown parent or is too old to possibly become an uncle, and if that check fails, the block is dismissed. Now the header is verified, and if everything is fine to this point, the whole block is broadcasted to a subset of peers (see Section 4.2.3 *Broadcasting Blocks & Transactions* and Algorithm 4.1 with propagate set to true). Afterwards, the node tries to insert the block into the chain (see Section 4.2.4 *Insert Block*), and only if that terminates without an error, the block hash is propagated to the rest of the peers not knowing the block(see Section 4.2.3 *Broadcasting Blocks & Transactions* and Algorithm 4.1 with propagate set to false). Figure 4.3 shows a comprehensive state diagram modelling this process (see eth/handler.go#707-741, eth/fetcher/block_fetcher.go#348-373, eth/fetcher/block_fetcher.go#importBlocks [eth20a]).

Broadcasting Blocks & Transactions

To keep block and transaction propagation delays throughout the network low while concurrently minimizing traffic, Ethereum broadcasts a whole block or transaction to a subset of a node's peers and only announces them to the rest, briefly known as push-pull. This is done via a simple algorithm that broadcasts the whole block or transaction to the square root of the total number of the node's peers or announces the hash of it to all its peers depending on a boolean flag (see Algorithm 4.2). If the respective function is then consecutively called twice, once with the flag set to true and once set to false, the block or transaction is propagated to all of the peers.

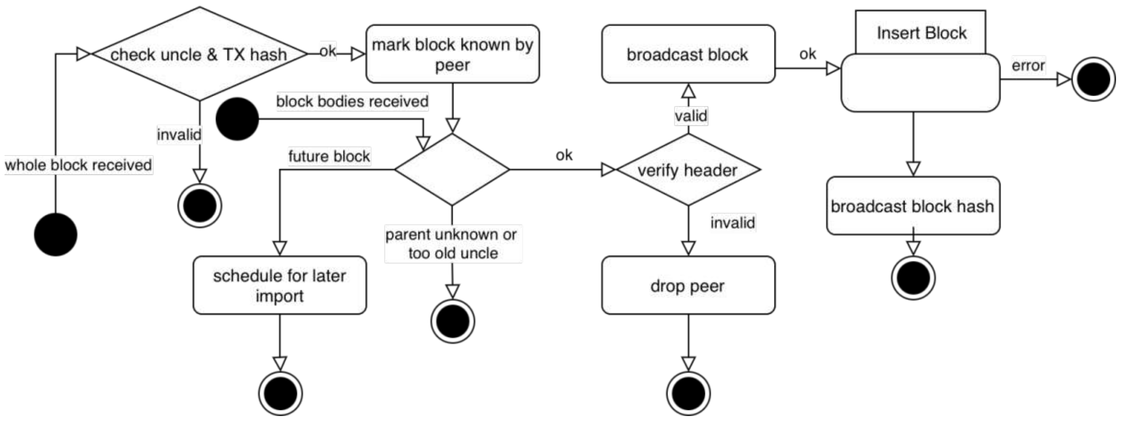


Figure 4.3: Block reception state diagram

Algorithm 4.2: Ethereum block & transaction broadcast (see eth/handler.go#822-891 [eth20a])

```
Input: A block or transaction $\beta$, a set of connected peers $\rho$, a boolean $propagate$
1  $hash \leftarrow$ Hash($\beta$);
2  $peers \leftarrow$ PeersWithoutBlockOrTx($\rho$);
3  if $propagate$ then
4      $peers \leftarrow peers[0 :$ int($\sqrt{\texttt{Size}(peers)}$)];
5      forall $peer$ of $peers$ do
6          Send($\beta$, $peer$);
7          MarkKnown($\beta$, $peer$);
8      end
9      return;
10 end
11 forall $peer$ of $peers$ do
12     Announce($hash$, $peer$);
13     MarkKnown($\beta$, $peer$);
14 end
```

Get Node Data

Nodes use a *Get Node Data Message* during the sync process to request arbitrary data from the state trie by passing hashes that represent them. If the request is successful, a requesting node receives a *Node Data Message* in response (see 14). A receiving node checks if the given trie node is present locally and returns all requested trie nodes (see eth/downloader/statesync.go#loop, eth/handler.go#591-627 [eth20a]).

Node Data Received

A *Node Data Message* contains an array of byte arrays that are related to the trie node. Once it is received, the data is passed to the downloader, which is responsible for the sync process (see eth/downloader/statesync.go#loop, eth/handler.go#629-638 [eth20a]).

Get Transaction Receipts

Get Receipts Messages are again only used in the sync process. On successful requests, a *Receipts Message* is received in response (see 14). On reception, a node searches for the given receipt hash, returning all receipts found locally to the requesting node (see eth/handler.go#640-674 [eth20a]).

Transaction Receipts Received

Receipts Messages contain all necessary data that describes the interaction of a transaction with the blockchain state, including the gas used, the transaction and block hash, events that occurred during execution and more. The received receipts are delivered to the downloader handling the sync process (see eth/handler.go#676-685 [eth20a]).

Synchronize with Peer

The downloader is responsible for keeping the node in sync with the network. This is done by recurrently (every time a new block is received or every 10 seconds) comparing the total difficulty of the best peer – the peer with the highest total difficulty – to the local total difficulty. If the remote total difficulty is higher than the local one, a sync process that uses the messages described earlier is started. Once the sync is finished, a chain reorganization may be necessary. A sync with a peer primarily occurs at the startup of a node or if the node lost network connection for some time (see eth/downloader/downloader.go, eth/sync.go [eth20a]).

4.2.4 Insert Block

A block insert can consist of multiple successive blocks, depending on if a single block was received or multiple blocks were received during, e.g., a sync process. First of all, all block headers are checked in parallel, and if any header contains an error, the whole insert is stopped. Then the first block's body is verified, and depending on the error given, there are multiple outcomes. If the first block is a future block, all blocks are pushed to the future block queue. If it has a pruned ancestor, meaning the ancestor is sufficiently old not to have a related state saved locally, the process goes on with inserting into a sidechain (see Section 4.2.4 *Insert to Sidechain*). If the first block is a known block, the blocks to insert are left-trimmed until the next block is unknown, and if no error occurs, the state of the first block is computed and verified before it is written to the chain. When an error is thrown on state validation, the whole process is stopped. Afterwards, if no error occurred and there are remaining blocks to insert, the process starts over with verifying the next block's body. If one of the consecutive blocks is considered a future block, again, all remaining blocks are added to the future block queue, if one is considered a known block, a special function to insert a known block is called described in Section 4.2.4 *Insert Known Block* (see core/blockchain.go#insertChain [eth20a]). Figure 4.4 shows the state diagram of the complete process of inserting.

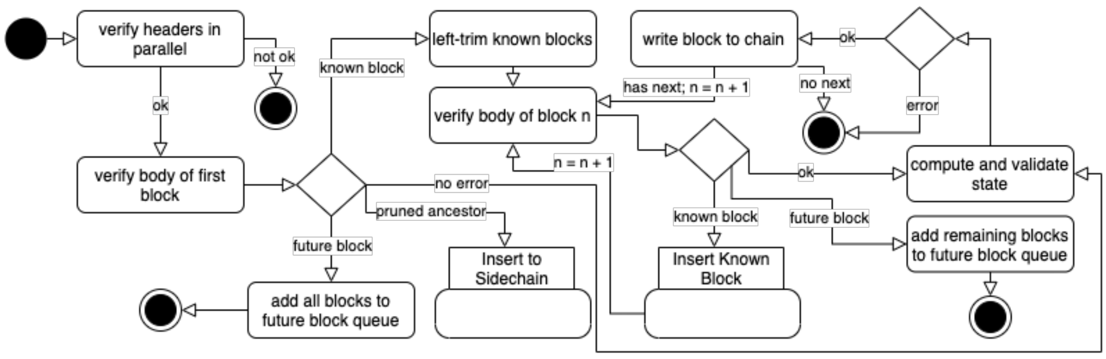


Figure 4.4: Block insertion state diagram

Insert Known Block

Inserting a known block is a shortcut that circumvents the state computation and validation because that was already done before. Here it is checked if the known block is the successor of the current head, if yes, it is written to chain as the new head, if not, a chain reorganization described in Section 4.2.4 *Reorganize Chain* occurs (see core/blockchain.go#writeKnownBlock [eth20a]).

Insert to Sidechain

A sidechain insert can only happen if the first block has a pruned ancestor when inserting a block or a sequence of blocks. A pruned ancestor is a sufficiently old block that is not in the currently longest chain and therefore has no related state. The process of inserting into the sidechain at first loops through all blocks to insert computing the total difficulty and writing the block to the database without its state. After all blocks are processed, the local total difficulty is compared to the external total difficulty (the one that was computed in the first step), and the process is finished. However, if the external total difficulty is equal or higher, a list of all predecessor blocks that lead to the blocks to import is built, beginning with the first one that has a parent, which has a locally saved state. This list is then passed to the insert block process described earlier (see core/blockchain.go#insertSideChain [eth20a]).

Write Block to Chain with State

On writing a block to the chain with its state, at first, the block with its related state is written to the database. Afterwards, the total difficulty of the block to insert is calculated, and if the total difficulty is lower than the local total difficulty, the process ends here. If the difficulty is higher, the block is added to the current chain, either simply as a head – if the predecessor is the current head – or a chain reorg has to be done (see Section 4.2.4 *Reorganize Chain*). In situations where the difficulties are equal, other criteria are used to decide which chain should be the new main chain. First of all, if the two chains differ in length, the shorter chain is selected and a chain reorg is initiated. If they do not differ, the chain with a head considered a local block is selected to be the main chain or the current main chain is preserved if both are local. A block is considered a local block if either the author equals the given etherbase (the address given in config to use for receiving mining rewards) or the author is an address specified in the transaction pool's local addresses. If neither of them is local, a random variable decides fifty-fifty if the block to insert should be the head of the new main chain or if the old head should be preserved (see core/blockchain.go#writeBlockWithState [eth20a]).

Reorganize Chain

A chain reorganization happens every time a new head is added to the current main chain, but the head can not simply be appended to the current head. The process needs two arguments, the current head and the new head, searches for the first common ancestor and rebuilds the chain to the point where the new head is the current main chain's head (see core/blockchain.go#reorg [eth20a]).

4.2.5 Mining

The mining process consists of four steps (i) *selecting a parent* that will be mined on top of, (ii) *selecting uncles* that should be included, (iii) *selecting transactions* that should be included and (iv) *solving a hash puzzle*, which is the actual process called mining. These steps will be elaborated further in the upcoming sections. Figure 4.5 shows a state diagram explaining the transitions between the different states. Once the miner is started, all four steps are executed. If a new head event occurs, meaning we added a new head to the chain, the process starts again with selecting the new head as a parent. Additionally, by default, every three seconds, a recommit timer fires, which is responsible for adding newly received, possibly higher-priced transactions to the current mining process. The recommit timer checks if new transactions have arrived since the last time and if yes, the process starts over with selecting uncles and transactions to be included in the new block. The last and possibly most important transition appears if a block is found, which immediately induces a broadcast to all of the peers as explained in Section 4.2.3 *Broadcasting Blocks & Transactions*, adds the block to the chain and starts a new mining process (see miner/worker.go#newWorkLoop [eth20a]).

Uncle Selection

When a block is added to the sidechain, this block is also considered to be a possible uncle. Possible uncles are tracked in two separate lists, one for local uncles and one for remote uncles. An uncle is considered a local uncle if either the author of the block equals the given etherbase (the address given in config to use for receiving mining rewards) or the author is an address specified in the transaction pool's local addresses. Local pool addresses are used to privilege selected addresses when mining blocks. Remote uncles are all other uncles not meeting this criterion. The uncle selection is a plain process that first tries to fill the block with local uncles and then, if there is space left (a maximum of two uncles are allowed per block), fills the block with remote uncles (see miner/worker.go#912-933, miner/worker.go#435-448 [eth20a]).

Transaction Selection

The transaction selection mechanism at first retrieves all pending transactions from the transaction pool and divides them into local and remote transactions according to the same criteria described in Section 4.2.5 *Uncle Selection*. Afterwards, the local and remote transactions are sorted ascending by nonce and descending by price. The gas limit for a block is calculated in a way that it stays above the provided floor and is increased if the previous block is full. If the provided ceil gas limit is exceeded, the limit will be decreased (default for floor and ceil gas limit is 8000000). Then the block is filled with local transactions at first and the rest is filled with remote transactions (see core/block_validator.go#CalcGasLimit, miner/worker.go#941-973 [eth20a]).

Solving the Ethash Proof of Work

The actual PoW process tries to find a PoW solution that is smaller than the target difficulty threshold. To be able to compute multiple PoW solutions – otherwise, a block would have exactly one solution – a nonce is used, which is incremented for every try. In conclusion, the PoW process tries to find a nonce that creates a PoW solution that satisfies the target difficulty (see 4.2.1, consensus/ethash/sealer.go#Seal, consensus/ethash/sealer.go#mine, miner/worker.go#taskLoop, miner/worker.go#resultLoop [eth20a]).

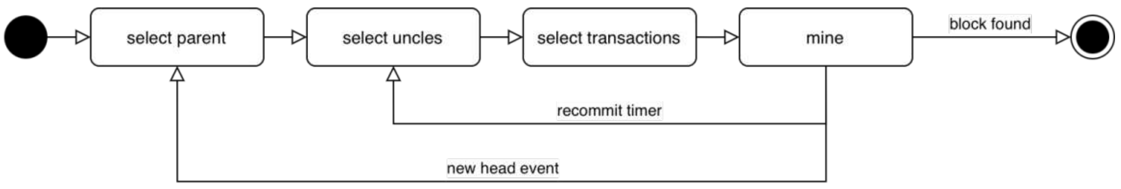


Figure 4.5: Mining state diagram

CHAPTER 5

Ethereum Attack Simulator

This chapter is about the process of developing a suitable simulator that fits our needs according to our research questions and the analysis of simulator frameworks in Chapter 3 *Simulators*, that did not provide a suitable candidate. This chapter will follow the steps 1-4 of a simulation study according to Banks et al. and Maria as described in Section 3.3 *Simulation Study*. It starts with step 1, the identification and formulation of the problem in Section 5.1 *Idea*, where the idea behind our simulator is labelled. Step 2 of a simulation study, the collection of real-world data, is done in Section 5.4.1 *Collecting and Processing Data*, where the exact processes are stated. A simulation study's step 3 of developing a model is elaborated further in Section 5.2 *Architecture*. The last part of this chapter, Section 5.6 *Validation*, corresponds to step 4 of a simulation study, focussing on the validation of the simulator's correctness by comparing it to previously collected real-world data.

5.1 Idea

The idea behind Ethereum Attack Simulator (EthAttackSim) is to make it possible to simulate the selfish mining and the verifier's dilemma attack against the Ethereum network and their implications on its stability while also enabling the simulation of other attack scenarios. This will be done by creating a new simulation engine using the insights and outcome of Chapter 3 *Simulators* as well as modelling the most important parts of an Ethereum client implementation of Geth revealed by Section 4.2 *GoEthereum (Geth)*.

The key objectives of our simulator are defined as follows:

1. *Model key parts* of the Geth client that are necessary for attack simulation.

2. *Extensibility* regarding various attacker implementations should be given.

3. *Configurability* for key properties is given through configuration files.
4. *Randomness* needed for stochastic random number generators should be based on real-world data and deliver the same results on similar runs based on a seed.

The most important key objective here is the modelling of the key parts of Geth, such as the network model, peer selection and P2P communication, difficulty calculation and mechanisms like inserting a block to the chain or reordering the chain on a new longest chain, which are crucial for simulating attacks like the verifier's dilemma and selfish mining and obtaining meaningful results. Configurability ensures that the operator of the simulation environment can easily modify the experimental setup without searching or actually having to edit the code. The extensibility should be given as much as possible with regard to attacker implementation, which should be designed using interfacing. An essential part is the pseudo-random number generator, which has to be configured based on parameters resulting from real-world data analysis. This ensures that the results of the simulation give a good statistical overview of possible outcomes. Ensuring repeatability of the results on similar runs based on a seed for the pseudo-random number generator – although important for reproducibility – may be challenging to achieve due to, e.g., possible inconsistent event ordering, but is, of course, a targeted objective too.

5.1.1 Simulation Framework

We decided, exactly like Blocksim (A) (see Section 3.4.2 *Blocksim (Alharby & van Morsel)*), to not use a simulation engine for our needs. It will be sufficient to implement and use an event queue in the shape of an ordered list which will be looped through instead of using a framework like PySim or GoDES.

5.2 Architecture

EthAttackSim is designed as a modular DESE consisting of two major parts. First, there is the simulation world and related modules that focus on providing the simulation environment and setup, the event queue, configuration abilities, metrics calculation and logs collection, randomness, as well as timekeeping. The second major part is the node module and its associated modules that represents a single participating node in the Ethereum network and provides networking abilities, a consensus engine and the ledger storage, which all are tied to one node. By design, every node has access to all simulation world data available, which is necessary to simulate attackers and their different abilities later on. Figure 5.1 gives an overview of the architecture of EthAttackSim and this section focuses on giving an understanding of the modules used.

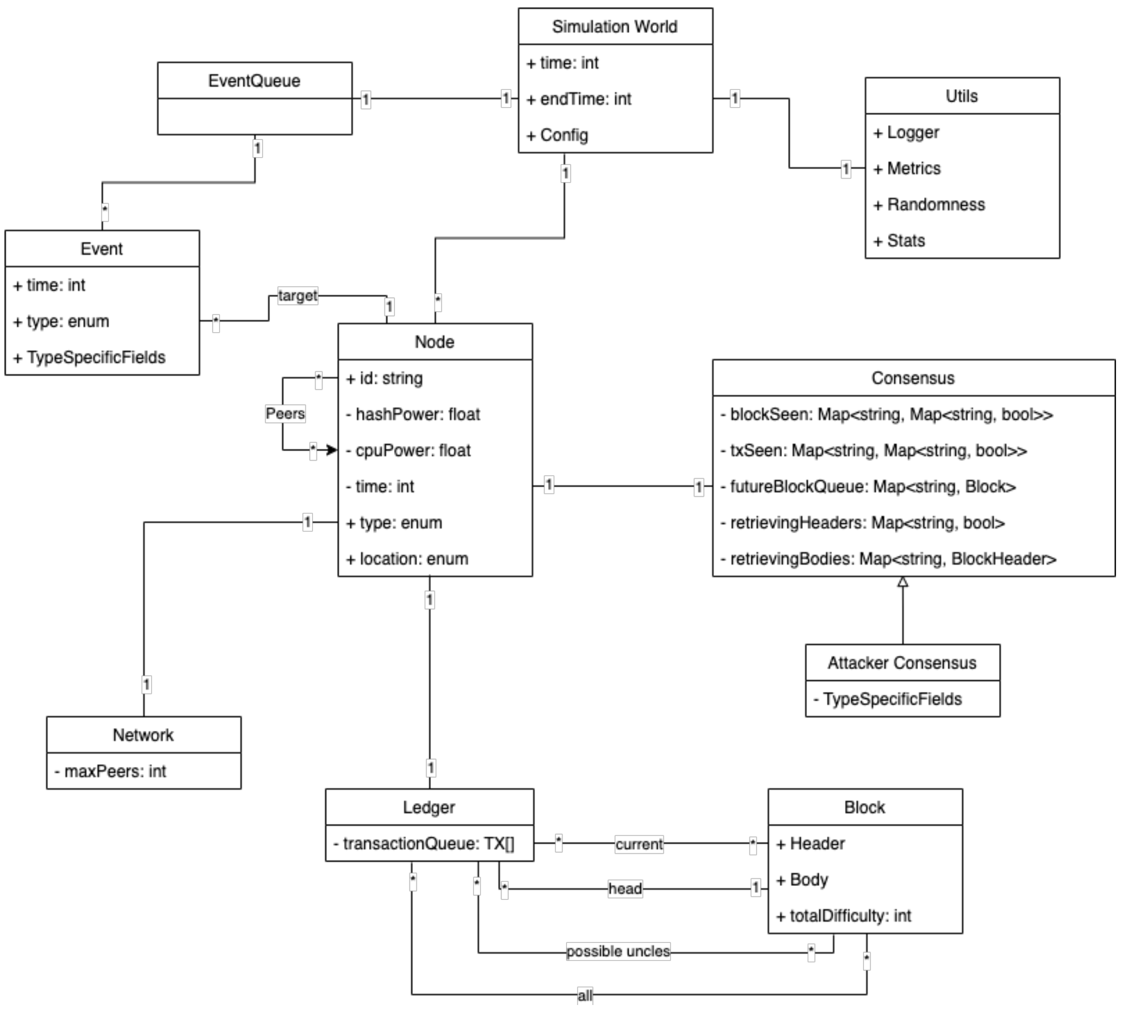


Figure 5.1: EthAttackSim architecture overview

5.2.1 Simulation World

The simulation world is the key part of the simulation itself. It is responsible for running the simulation event by event, timekeeping and providing configuration for all other modules used in the simulation. Further, it is a gathering point for all nodes the simulation contains, but besides that, there is no additional functionality. The responsibility of the main package, together with the simulation factory, which is somehow part of the simulation world, is to initialize configuration objects, randomness utilities and nodes together with their peers, as well as hosting metrics and statistics calculation utilities.

5.2.2 Events

As the simulator is event-based, the flow is controlled by the occurrence of various blockchain events. The upcoming events are stored in the so-called main event queue, which is a simple list of events sorted by timestamp ascending. Then there is another list for newly added events that is kept unsorted, additionally, the earliest timestamp of this list is kept. On calling the NextEvent function, there are two possibilities to occur. First, the case that the earliest timestamp of the new event list is after the first event in the sorted main queue, which leads to simply returning the first event of the sorted main queue. The second case is the one where the earliest timestamp of the new event list is before the first event in the main queue, which leads to the new event list being sorted via the merge sort algorithm, followed by the merge of the main event queue with the then already sorted new event list, which leaves the new event list empty, and afterwards again simply returning the first event of the main queue.

Event types occurring during the simulation are listed and described in the following.

- *Genesis Event*, the first event every node in the simulation network receives that also starts the simulation and mining process,
- *New Block Event*, represents the mining of a new block that is discovered by a node,
- *New Transaction Event*, expresses the reception of a signed transaction received by a user (not by a node),
- *Received Block Bodies Event*, the reception of a previously requested set of block bodies,
- *Received Block Event* the reception of a whole block from a peer in the network,
- *Received Block Hashes Event*, expresses the reception of a block announcement from a peer,
- *Received Block Headers Event*, the reception of a previously requested set of headers from a peer,
- *Received Transaction Hashes Event*, expresses the reception of a transaction announcement from a peer,
- *Received Transactions Event*, represents the reception of a bunch of whole transactions,
- *Retrieve Block Bodies Event*, an event that depicts a message from a peer requesting bodies certain blocks,
- *Retrieve Block Headers Event*, an event that depicts a message from a peer requesting headers of certain blocks,

- *Retrieve Transactions Event*, an event that depicts a message from a peer requesting certain transactions,
- *Transaction Creation Event*, a helper event representing non-node users submitting a signed transaction.

5.2.3 Node

A node is a simple assembly point for all the modules it uses, i.e., consensus, network and ledger. Additionally, a node keeps its own time and stores the configuration such as hash power, CPU power, node type and location, as well as a list of the peers connected to it.

5.2.4 Ledger

Each node, in analogy to the real Ethereum network, keeps its own ledger, which stores a node's view of the blockchain, i.e., the current longest chain, the current head, uncles and generally all blocks that it has ever seen. Furthermore, the ledger module also stores a map of possible uncle blocks it can include in the following block to mine as well as a queue of transactions, the so-called transaction pool. Besides storing data, the ledger functionality includes methods to append a new block to the current longest chain, write blocks to the ledger that are not appended or inserted in the longest chain, reorganize the longest chain, which is the same as setting a new head that is not simply appended, as well as retrieving a list of sorted transactions to include in the next mined block.

5.2.5 Network

Networking is a vital part of a node's activity. For example, the network module connects peers to a node when initializing the simulation. But the primary task of the network module is sending or broadcasting messages to other nodes, which in the case of this simulator is the same as adding newly created events to the event queue. This functionality is outsourced to the network module to change a node's behaviour quickly, e.g., send a new block to some selected peers only.

5.2.6 Consensus

The consensus module can be considered the heart of a node, where most of its consensus behaviour is located at during the process. This module defines the behaviour of a node for all possible messages or events it receives, from receiving a new block, over verifying this received block and including it into the ledger, to mining a new block itself. It also keeps track of the blocks and transactions it has already seen, as well as of requests waiting to be responded to and future blocks where the parent has not yet occurred in their sight.

5.2.7 Monitoring & Reporting

This is not a standalone module but rather part of utilities. All events occurring during the simulation, together with the originator, the receiver and additional data if needed, are recorded into an audit log CSV file. After the simulation is finished, the whole simulation world will be written to a JSON file containing a complete view of the world, including the view of the ledger for every single node. To provide a quick overview of the simulation output without digging through the whole simulation world by hand, the simulation prints an overview JSON file at the end that contains the key results of a simulation, e.g., how many blocks every node has mined, how much rewards each node gathered, the peers each node had, and additional statistics like transaction throughput, overall rewards, uncle rate, etc. Furthermore, a simulation also outputs a comprehensive list of metrics to be evaluated that show statistics, e.g., the time a specific event needed to be transmitted, how long it took to insert blocks into the ledger, and even how many blockchain reorganizations happened per node.

5.3 Configuration

According to our key objective number 4, every possible important property should be configurable via a configuration file to be easily adjusted during runs. The following configurations can be made via config.yml and delays.yml files for a simulation run of EthAttackSim:

- *hash power* overall and for mining pools/nodes,
- *CPU power* for mining pools/nodes, which expresses an abstract value of computational speed related to, e.g., transaction verification,
- *block time**, the time between blocks expressed as statistical distribution,
- *transaction gas**, a distribution for the amount of gas a transaction uses,
- *latency** for different locations to other locations, again expressed as distribution,
- *throughputs** for different locations to other locations represented as distribution,
- *time it takes to compute the transaction state update*, in gas/MHz/second a node can compute,
- amount of *mining pools* and amount of *nodes* overall,
- *sizes of different parts of a block and messages* between peers,
- *gas price**, a distribution for the gas price a transaction uses,
- *time it takes to verify transactions, headers and bodies*,

- *maximum uncle distance*,
- *block reward*,
- *nephew reward*,
- *initial gas limit*, the blocks initial gas limit set at simulation start,
- *minimum transaction gas*,
- deterministic *seed* for generating pseudo-random numbers to ensure repeatability,
- ability to output *metrics* to get insight into more data,
- *simulation time*, the amount of time a simulation shall run,
- ability to *transaction creation and propagation* should be enabled/simulated, or if a new block should be filled with random transactions,
- various configuration possibilities according to *logs* and
- the ability to define additional properties belonging to *different attackers*.

For variables that will be drawn using the random number generator during runtime, denoted by an asterisk*, please refer to Section 5.4 *Properties, Distributions and Randomness* to get insights into how the values of different configuration properties were selected for this work.

5.4 Properties, Distributions and Randomness

In this section, the data collection for different properties is comprehensibly summarized, and the sources of distributions and their creation are shown. It is subdivided into data collection and evaluation of collected data for the use in EthAttackSim.

5.4.1 Collecting and Processing Data

Data collection happens via various sources, starting with research papers, web-based tools like etherscan.io and miningpoolstats.stream, collecting own data directly on-chain or events extracted from Geth source code. Data like the hash power of the most important mining pools is collected via miningpoolstats.stream, which shows the hash rate of the top Ethereum mining pools. On-chain data collected by the author of this thesis can be found in the published repository related to this work (see [Mai22]). Block reward, maximum uncle distance, nephew reward, initial gas limit and minimum transaction gas are extracted from Geth source code and match the Ethereum protocol. These properties are relatively easy to extract, the parameters that need a detailed explanation are described in the following.

Block Time

The block timestamp may not match the actual time. For retrieving the actual time, it is necessary to monitor the network actively. However, due to consensus rules, it is difficult for block timestamps to deviate arbitrarily from the actual time, and therefore the block timestamp is the most accurate value available. The base data can be gathered from real-world block data relatively easy by running a script that retrieves the last n blocks and computing the differences of their timestamps from block n to block n + 1 (see [Mai22]). We did this for the blocks of about 60 days (roughly 6500 blocks per day, etherscan.io shows an average block time of about 13.40 seconds in November 2021), so we received data of 390000 blocks, including uncles. The computed mean without uncles during the last 60 days was 13.63533 seconds. Including all uncle blocks, the collected data spans 411906 blocks, the computed mean therefore is 12.91017 seconds per block. Figure 5.2 and Table 5.1 show the distribution of block times. 50% of block times occurring on the network are between 4 and 19 seconds, 75% of blocks occur between 1 and 19 seconds after the last one, the last 25% stretch from 19 seconds to a maximum of 170 seconds.

However, as we want to simulate the mining process on a per-node basis and with regard to the fraction of the hash power a node owns, the computed Pareto distribution does not fit the needs of the simulator because it does not map the long time it can take a node to find a block and cannot be weighted by hash power. The best fit in our case is the use of an exponential distribution with a rate parameter

$\lambda = fractionOfHashPower * (1/meanBlockTime)$

which weighs in the hash power owned by a specific node. The expected value is $1/\lambda$. As the simulator also models the creation of uncle blocks, the mean block time selected is the one including uncle blocks.

Message Sizes

According to Faria and Correira, the important message or message parts have the following sizes:

- *Hash:* 42 byte
- *TX:* 200 byte
- *Get Headers Message:* 54 byte
- *Header:* 90 byte [FC19]

Using these four sizes, the size of every message utilized in the simulator can be composed.

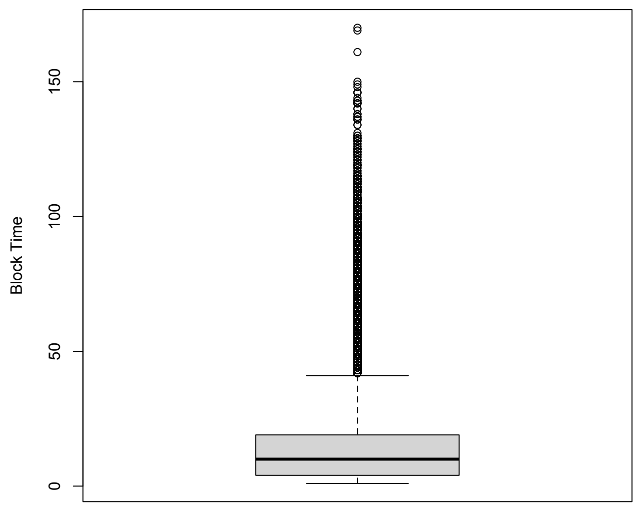


Figure 5.2: Ethereum block times distribution (s)

Min.	1st Quantile	Median	Mean	3rd Quantile	Max.
1	4	10	13.64	19	170

Table 5.1: Distribution summary of Ethereum block times (s)

Node Count

Counting all active nodes and miner nodes on the Ethereum network is unfortunately not easily possible. The online service miningpoolstats.stream [Min22] lists 84 different mining pools on 9.12.2021. The on-chain data shows that in the last approximately two months (390000 blocks), there were only 89 different miners, including the found uncle blocks there were 91 miners. So for the simulation runs, a node count of about 100 to 200 nodes should be sufficient to imitate the real-world distribution of nodes. Figure 5.3 shows an overview of all contributing Ethereum nodes that found a block or an uncle during the observed 390000 blocks period, where exactly 21906 uncles were found.

Latency & Throughput

These attributes, namely latency or ping as well as send and receive throughput, used for computing the time a message needs to arrive at another node, have to be measured on different locations in the world to be able to include the nature of distributed systems into our considerations regarding the simulator. Fortunately, Faria and Correira did precisely that [FC19, Far]. All that has to be further done is to take the measurements and fit

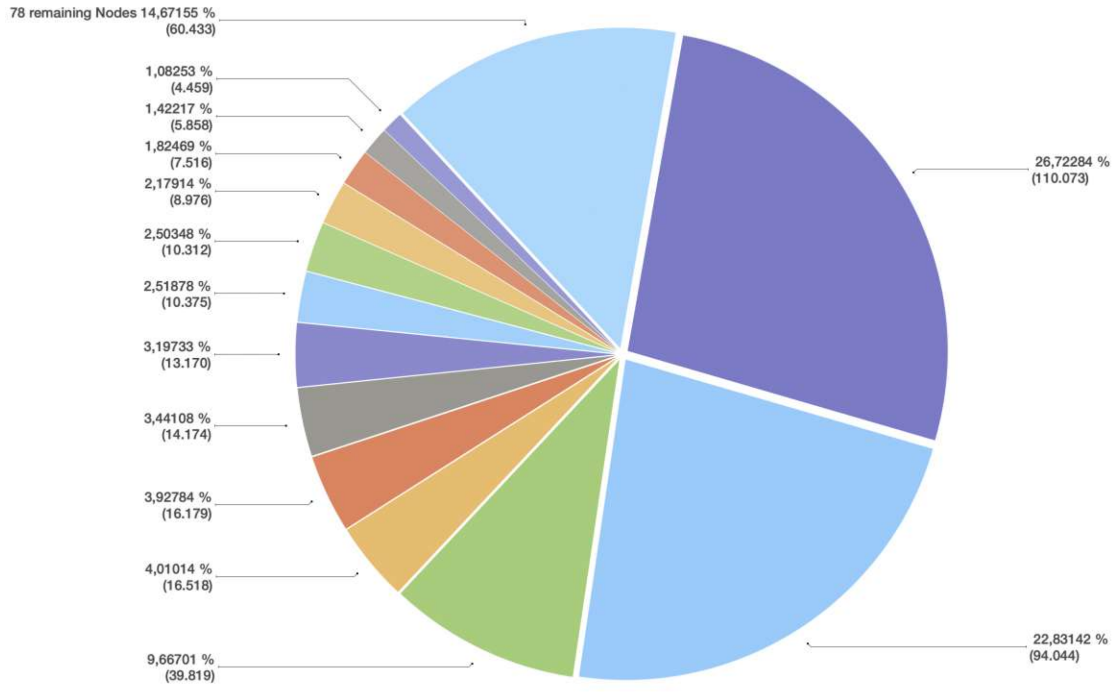


Figure 5.3: Consensus-contributing Ethereum nodes

them to the pseudo-random number generator used in this work. This is again done with our distribution fitting script, which is explained further in Section 5.4.2 *Distribution Script*.

Transaction Gas

The minimum transaction gas amount, according to the Ethereum protocol, is 21000 gas, meaning there will never be a transaction that uses less gas. To collect data for the distribution fitting, we selected to use on-chain data too, so the gas utilized by transactions in the last 1000 blocks is used, which sums up to about 200000 transactions. Figure 5.4 and Table 5.2 show the distribution of transaction gas used per transaction throughout the observed period. The data shows a minimum of 21000 gas used, as this is the limit set by the protocol. 75% of data points are observed with between 21000 and 74974 used gas, the rest stretches from 74974 up to a maximum observed used gas of 28 million with a mean of 78042 gas used.

For the distribution fitting, 21000 was deducted from the collected data because that is the minimum and will be added to the value drawn from the pseudo-random number generator. The normal distribution is the best-fitting one, with an error rate of

$F = 0.1912$

and parameters

$\mu = 40574965011$

$\sigma = 91293671277.$

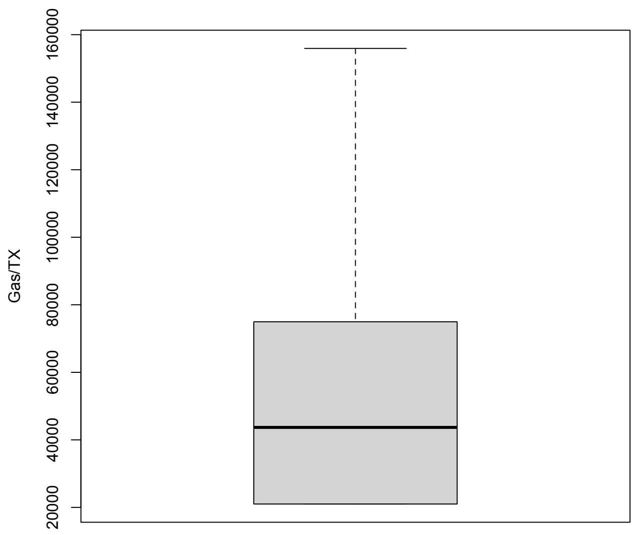


Figure 5.4: Ethereum gas used per transaction distribution

Min.	1st Quantile	Median	Mean	3rd Quantile	Max.
21000	21000	43694	78042	74974	28610220

Table 5.2: Distribution summary of Ethereum gas used per transaction

Gas Price

Gas price varies a lot, depending on the utilization of the Ethereum blockchain, leading to a very high range of possible values if inspected over an extensive timeframe. We selected to also use on-chain data for the purpose of defining gas price values for simulation, so the gas price of the transactions in the last 1000 blocks, totalling to about 200000 transactions, is collected and used for distribution fitting. Figure 5.5 and Table 5.3 show that during the observed timeframe, the observed mean gas price was at 74.48 GWEI, with the lower 75% of the data points spanning from a minimum of 40.58 GWEI up to 83 GWEI and the remaining 25% stretching from 83 GWEI up to a maximum of 12434.60 GWEI.

The best suiting distribution for the data is a uniform distribution with an error rate

$F = 0.1476$

and parameters

$min = 40574965011$

$max = 91293671277$

which is roughly 40 to 91 GWEI.

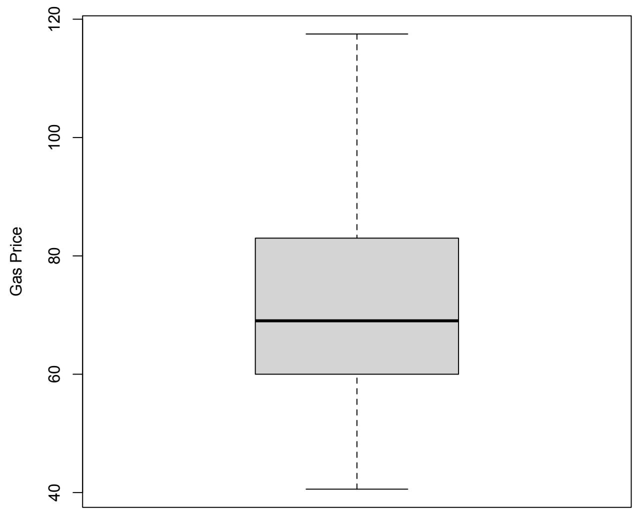


Figure 5.5: Ethereum gas price per transaction distribution (GWEI)

Min.	1st Quantile	Median	Mean	3rd Quantile	Max.
40.58	60	69.03	74.48	83	12434.60

Table 5.3: Distribution summary of Ethereum gas price per transaction (GWEI)

Base Verification Time

We specified a base verification time for a block header and body, as well as for transactions. This verification time is dependent on the CPU and hash power that is assigned to the mining nodes and includes the base verification of aforementioned entities, containing a check of the block hash, the block time and various limits of a block for the base header verification, and checking the number of uncles contained in a block for the base body verification. This excludes the time for the actual state computation, which takes significantly more time because of necessary plausibility checks (e.g., regarding the balance of an account before and after the transaction) and the computation of smart contract calls. As the verification – depending on the entity – does not need only computational power but also memory lookups, it is known that the actual verification time is not only dependent on the CPU and hash power of a node, but for reasons of simplification, the CPU and hash power are the only variables of a node regarding their hardware that are used within our simulations. In future work, it may be interesting to add more dimensions to the simulation of required resources. The verification time is computed by using some estimated value concerning the actual amount of headers, bodies and transactions a node can verify per MHz per second and additionally using the hash power of a node to compute the time it needs to compute the number of hashes necessary to do such base verification. Altogether, the base verification time has a shallow impact on the simulation as there are only minor differences between strong and weak hardware (CPU & hash power).

Transaction State Computation

This is an essential property of the simulator, as it represents the main delay a node is subject to when verifying a transaction, e.g., the time a node needs to execute a smart contract call. The higher the value, the less time the state computation takes. Additionally, reasonable values for this delay are needed for proper simulation and modelling of the verifier's dilemma attack. To model the transaction state computation delay with regard to a node's CPU power, we use the unit gas/MHz/s, which describes a node's ability to compute the state for transactions worth a specific amount of gas per MHz CPU power per second.

To obtain the most reasonable value for such delay, we take recourse to three resources, the first is a paper by Alharby et al. [ACLAvM20]. They collected the time it takes on average to compute the state for different amounts of gas with a 3.4GHz Intel i7 CPU with 8GB RAM. The second one is a blog article of the Ethereum foundation, they did not state which processor or RAM they used, but for simplicity, we assume the same 8GB amount of RAM as Alharby et al. and used a slightly slower CPU with 3GHz as our calculation base [Fou21b]. The third one is a paper by Pontiveros et al., who used a 2.9 GHz Intel i5 processor with 8GB RAM to collect their data [PTS19]. For all calculations, we used the mean execution time of the Geth client. Table 5.4 shows an overview of the results from Alharby et al., the Ethereum Foundation and Pontiveros et al. and includes the outcome of the calculation required for our Gas/MHz/s parameter used in the simulator.

It is clear that the data points are widely apart because it is a mixed table of regular execution times and contracts designed to exhaust the state computation engine to the maximum. For the base simulation and its verification, the 10230 gas/MHz/s will be taken, as it is the best assumption of a less extreme value and provides the best fit for simulator validation (see Section 5.6 *Validation*).

Gas	Mean Execution Time (s)	Gas/MHz/s	Source
10M	70	47,61	[Fou21b]
8M	1.21	2280	EXP [PTS19]
8M	0.23	10230	[ACLAvM20]
8M	0,15	18390	SHA3 [PTS19]
8M	0,03	91954	SLOAD [PTS19]
8M	0,02	137931	SSTORE [PTS19]

Table 5.4: Overview of transaction state computation time

5.4.2 Distribution Script

We wrote a small Go script that extracts the best-fitting statistical distribution from a sample data set based on the Go packages optimize, stat and distuv. This script uses the sample data set and randomly creates the same amount of entries for different, well-known distributions like, e.g., Normal, Gamma, ChiSquare, Uniform etc. For every distribution, a minimization problem is created and every time such a random data set is created with slight variations of parameters, it is tested against the original data set by a Kolmogorov-Smirnov-Test (KS-Test), which gives the lowest values for data that is thought to come from a common distribution. After many optimization runs, the optimization comes to an end and returns the distribution with its best parameters and the lowest failure rate F. Once all the optimization problems are complete, we have the best parameters for every distribution under test and their failure rate, of which we select the distribution with the lowest failure rate to be the best-suiting one we use in our simulator (see [Mai22]).

5.5 Modelling Attacks

A significant part of the simulator implementation is the need for extensibility in order to be able to simulate attacking nodes with different capabilities. This is solved by overriding different parts of the implementation specifically for one attack or a set of attacks. For example, if an attacker should, for whatever reason, not verify transactions of a received block, a new consensus module class for this specific attacker skill is created that overrides the implementation for the respective methods to simply return the signal for valid. This new module class also has to be implemented in the factory to be selected on initialization of the simulation world.

It is possible that for some attack scenarios, methods have to be extended and not just overridden. This could be the case when, e.g., a method has to take a new or changed parameter. For the verifier's dilemma, for example, the attacker needs to have the ability to create computationally extensive transactions, and as the simulator does not model the computational effort in terms of time for different types of transactions, the transaction class itself had to be extended to host an adapted parameter (see also Section 5.4.1 *Transaction State Computation*) that makes it possible to make specific transactions harder. Additionally, the delays utility had to be extended to take the new parameter into account when computing delays for transactions. This should always be done in a way that does not affect the base simulator implementation, a default value should be provided that disables this feature when not explicitly set.

Examples of attacker implementation will be covered in Chapter 6 *Attack Simulation*.

5.6 Validation

For obtaining valid results in attack simulation later on, EthAttackSim has to be validated against previously collected Ethereum network data (see Section 5.4.1 *Collecting and Processing Data*) when run without attack simulation. The validation happens by selecting various significant parameters and statistically comparing them to the real-world data and subsequently analyzing and interpreting it.

The simulator configuration parameters used for validation are elaborated in Section A.2 *Simulator Configuration Parameters*.

The parameters that were selected for validating the EthAttackSim are:

- block time (block interval)
- uncle rate (uncles per day) and
- percentage of blocks found per node.

The *block time* and *uncle rate* are the most critical parameters of the simulation because of their nature of illustrating the completeness of the simulated Ethereum network and the P2P communication between the nodes. The configuration parameter for time between blocks, together with a valid implementation of the P2P network, will lead to real-world data simulator output. The percentage of *blocks found per node* shows if the consensus mechanism and randomness are correctly implemented, i.e., if the output reflects the hash rate configuration.

5.6.1 Collecting Data

For the validation of EthAttackSim the simulation has been executed 30 times for two days of simulation time, each with an amount of 200 mining nodes with a hash rate percentage according to the distribution of miningpoolstats.stream [Min22]. The top 59 pool hash rates were taken and the rest of the Ethereum network hash rate was equally distributed to the 141 other nodes used in the simulation. The abstract configuration of CPU power for each mining node was selected in the range of 3600 to 4500 MHz.

5.6.2 Simulator Results & Evaluation

This section reveals the results of the simulator runs as well as analyzes and discusses their implication on the correctness of the simulator implementation.

When dealing with Goodness of Fit (GoF) tests, two-sampled and one-tailed KS-Tests are used to compare the on-chain data to simulator results for similarity of distribution. A significance level $\alpha = 5\%$ is used for hypothesis testing. The null hypothesis H_0 always denotes that the samples are drawn from a common distribution, while the alternative hypothesis H_1 stands for the two samples originating from a different distribution.

Block Time

The first thing that is done is to compare the distribution summaries and the boxplots of on-chain data against simulator data. The summaries can be seen in Table 5.1 and Table 5.5, as well as the boxplot in Figure 5.6. One can see the nearly perfect overlap, there are only two minor differences, with mean block time that is +0.02 seconds coming from the simulator and with maximum block time that differs by about -7 seconds from on-chain data. Both of them seem negligible, and the 7 seconds higher maximum block time on-chain may be coming from network outages or similar that are not considered in the simulator.

Min.	1st Quantile	Median	Mean	3rd Quantile	Max.
1	4	10	13.66	19	163

Table 5.5: Distribution summary of EthAttackSim block times

The next graphic (Figure 5.7) to look at is a side-by-side comparison between the histograms of on-chain and simulator data, segmented into 1-second buckets, starting with the bucket of blocks created with a timestamp of parent timestamp + 1. The figure shows that EthAttackSim outputs blocks with a timestamp of +2 most often, steadily decreasing from then on. On-chain data depicts a block time of 2 seconds occurs most frequently too, but although it decreases from then on most of the time, there are spikes at every ninth-second bucket, starting from the 8-second bucket. Also, 1-second blocks occur more often on-chain than in the simulator data.

There are multiple differences appearing in the data sets, but before examining them, it has to be mentioned that the block time derived from the block header does not accurately reflect the real block time because the block timestamp is set prior to mining a block and updated only once in a while so that it is possible that a block mined after 3 seconds still has a timestamp of parent timestamp + 1. This, in combination with the difficulty calculation algorithm (see Section 4.2.1 *Difficulty Calculation*), is also the explanation for the spikes occurring on-chain. Algorithm 4.1 line number 3 shows that an important difficulty factor x is computed by the time difference to the parent block divided by 9. So the difficulty of a mined block decreases significantly every 9 seconds, which could potentially incentivize miners to keep the lower timestamp before the threshold (9, 18, 27,

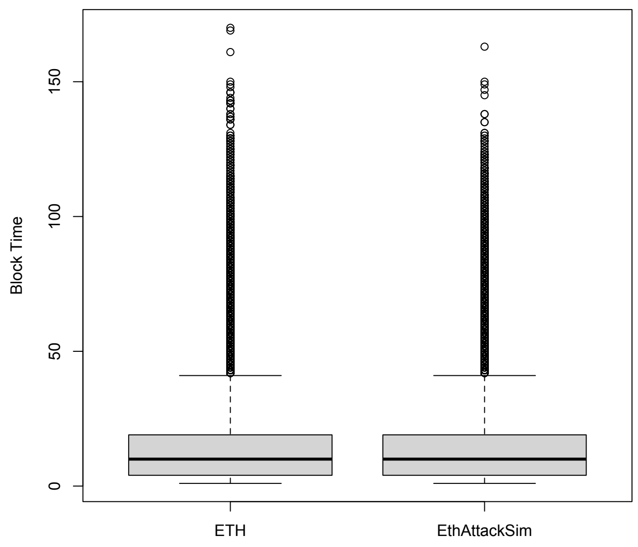
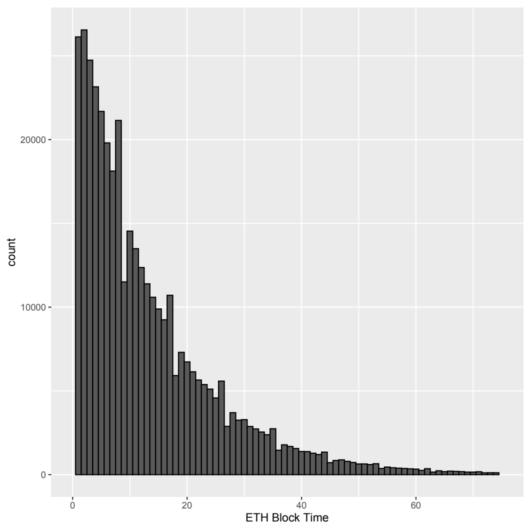


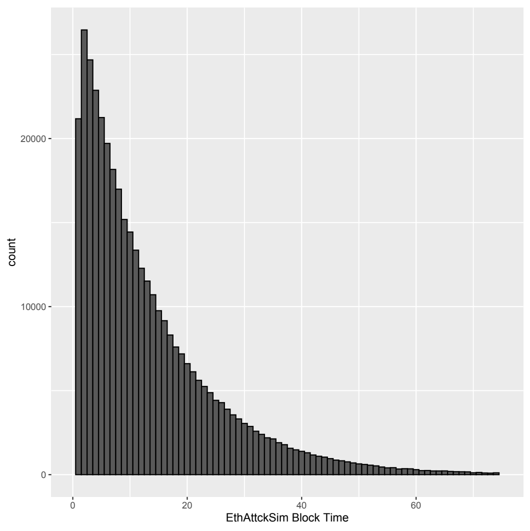
Figure 5.6: Comparison of ETH & EthAttackSim block time distribution

... seconds) in the block header a little longer to gain an advantage when it is to be decided if the own block becomes the new head or an uncle. In concurrent work, Yaish et al. outline the possibility of such adversarial timestamp manipulation, which they refer to as the "Uncle Maker attack" and empirically show that some miners are currently engaging in such behavior [YSZ22]. Hence, the observed discrepancy between our simulator results and the empirical data can not only be adequately explained but also highlights how simulation could contribute towards uncovering abnormal or adversarial behavior. A one-second block naturally occurs less than a two-second block in both datasets because a new block has to be broadcasted to peers and verified by them before mining on top of it. Looking at the on-chain data, 1-second blocks seem a little overrepresented compared with the simulator dataset. This might again be caused by the same reason of flexible block timestamps, but it could also be caused by mining pools sending new mining jobs only about every two seconds, which leads to the earliest data points being shifted slightly towards the one-second bucket.

The last comparison between on-chain and simulator data for block times is done by a KS-Test. The null hypothesis H_0 indicates that the samples are drawn from a common distribution, H_1 denotes the opposite. For the computation of this test, 15 single KS-Tests are executed for every two-day simulator dataset by selecting a random range of precisely the same amount of blocks from the on-chain data. These 15 single tests are then averaged to create a single p-value for every two-day dataset. Suppose the test shows statistical significance ($p < \alpha = 0.05$), H_0 has to be dismissed, if not, H_0 can not be dismissed



(a) ETH on-chain block time



(b) EthAttackSim block time

Figure 5.7: ETH block time comparison

and therefore, it is assumed that both datasets are drawn from a common distribution. After executing the tests for our sample datasets, H_0 has been dismissed for 8 of 30 tests because of statistically significant differences in the two sample distributions, which seems like a worrying result at first glance. However, when digging into the issue, we found out that the main reason for the dismissal of H_0 was again the over-representation of blocks in the first (1s) bucket. Cleaning those blocks gives a perfect match of all samples to a common distribution.

It was already shown in the past that miners set the block timestamp to their advantage [Fou16]. Although fundamentally different, research investigating Bitcoin block arrivals also states that it is a non-trivial problem to obtain real data regarding blocks and their timestamps related to the consensus mechanism [BKKT18]. The results of our simulation may inform future research about the discrepancy in the distribution of block times in the 1-second bucket to better understand whether this represents an artifact of the technical implementation of mining and mining pools or whether miners intentionally deviate from prescribed protocol rules in order to increase their potential profit.

Uncle Rate

The uncle rate describes the amount of uncle blocks occurring on the Ethereum blockchain during a day and shows if the simulator's consensus mechanism for block selection, the block time, as well as the communication delays, are implemented correctly and suitable configuration parameters have been selected. It is worth mentioning that the on-chain data collected (see [Mai22]) is more granular than the EthAttackSim's data points. On-chain uncle rate was collected for every 1000 blocks (roughly $\frac{1}{6}$ of a day) while the simulator calculates the uncle rate per run, in this case for two days. This will lead to a spread in the on-chain data distribution. And this is exactly what we see. Table 5.6 shows the numerical distribution summary while Figure 5.8 demonstrates the same data via a boxplot. The minimum and maximum uncle rate of Ethereum data differs by a lot, while the 1. and 3. quantiles, as well as the median and mean are located close together.

	Min.	1st Quantile	Median	Mean	3rd Quantile	Max.
ETH	227.5	331.5	364	365.1	396.5	520
EthAttackSim	311	354.8	372	374.8	399.5	454

Table 5.6: Distribution summary of uncles per day

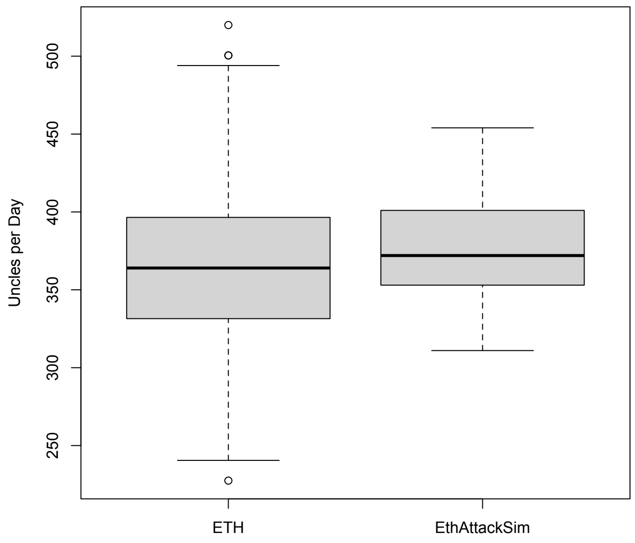


Figure 5.8: Comparison of ETH & EthAttackSim uncles per day distribution

Performing a KS-Test on the two collected sample sets results in a p-value of 0.1417, so we fail to reject H_0 (samples are drawn from a common distribution) because no statistically significant differences are found in the two distributions (for $\alpha = 0.05$).

Blocks Found per Node

The blocks found per node during a run, which are directly proportional to the amount of Ether mined, have to strongly correlate with the hash rate of a node to be valid for our simulation purposes. Figure 5.9 shows an overview of the top 50 mining nodes from the simulation runs sorted by hash rate descending, represented by black dots, together with an overlay of their percentage of mined blocks, represented by red crosses. It can be seen in the chart that for nearly all nodes, the mark is hit perfectly, only for node 1 it slightly differs. To understand why that is, it is worth to mention that the lower the hash rate of a node is, the longer it takes for the expected mined blocks and the actually mined blocks to even out, which conversely leads to the nodes with the highest hash rates to be overrepresented until it all evens out in the long run.

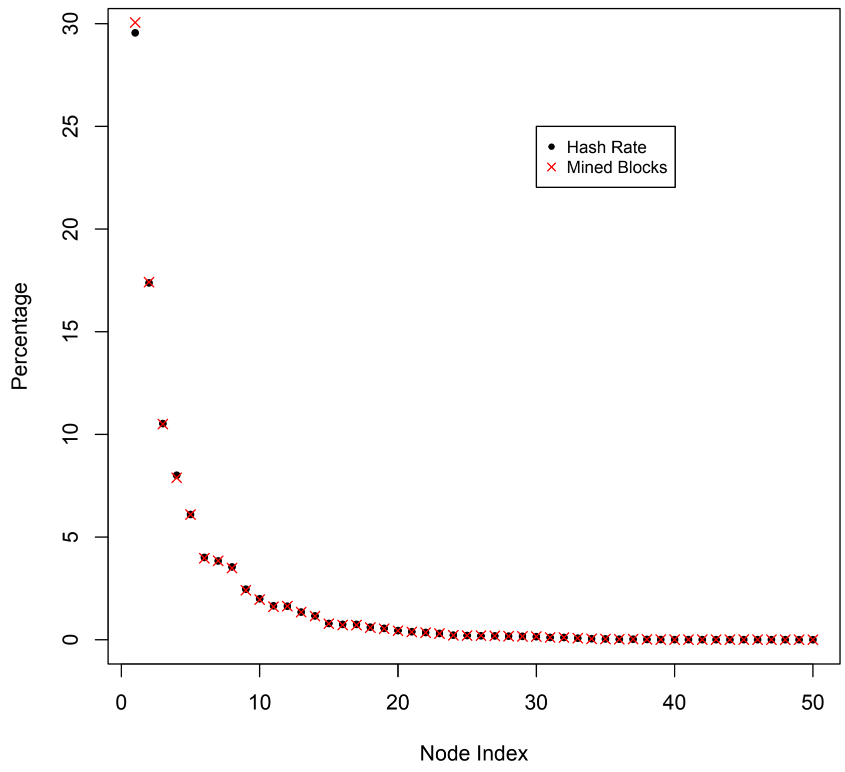


Figure 5.9: EthAttackSim found blocks comparison

5.7 Summary

Sim.	ETH [1]	Consensus [2]	Network [3]	Attacks [4]	Ext. [5]	Resources [6]
Blocksim (F)	+	−−	~	−−	+	++
Blocksim (A)	+	−−	−	−−	+	++
SimBlock	−	−−	+	−−	+	++
LUNES	−	+	+	−−	+	−−
Shadow-Bitcoin	−	++	++	−−	−	−−
EthAttackSim	+	++	++	++	+	~

[1] Suitability for simulating Ethereum
[2] Authenticity of consensus model
[3] Network modelling
[4] Simulation of attack scenarios
[5] Extensibility of the simulator
[6] Resource intensiveness

Table 5.7: Comparison of existing simulator frameworks, including EthAttackSim

In extension to Table 3.1, Table 5.7 now shows the properties of EthAttackSim compared to the simulator frameworks discussed in Section 3.4 *Existing Simulators*. It can be seen that EthAttackSim offers all the previously required parts for simulating attacks against the Ethereum protocol not offered by other simulation frameworks, namely a sufficiently advanced consensus and network model, together with high extensibility for attack simulation. The sophisticated network and consensus models, however, lead to a higher resource intensiveness in terms of memory and CPU consumption, as well as to higher disk space requirements related to intensive logging. Ensuring repeatability of the results on similar runs based on the introduced seed for the pseudo-random number generator – although important for reproducibility – was hard to achieve due to some Go-specific peculiarities, e.g., randomization when iterating over maps, sometimes leading to different results even though the same seed and parameters were used.

In summary, it can be stated that the highly configurable and extensible simulator EthAttackSim models all the critical parts of the Ethereum protocol necessary for immersive attack simulation, in particular for simulating selfish mining and verifier's dilemma attacks. It has been shown that EthAttackSim is able to simulate realistic Ethereum network conditions by passing validation tests, so all requirements for further attack simulation are met, although future research may engage in further evaluation of the statistical divergence in block time, for example, the overrepresentation of one-second blocks.

To reproduce the results highlighted in this chapter and for possible further evaluation of the simulator in future research, the code of EthAttackSim, together with data used for its configuration and validation, is made available on github.com and open source [Mai22].

CHAPTER 6

Attack Simulation

This chapter focuses on the simulation and analysis of attack scenarios based on EthAttackSim and demonstrates how the simulator is of value to spot possibly feasible attacks quickly as well as eliminate unfeasible ones. To be more specific, the verifier's dilemma [LTKS15, PTS19] and the selfish mining attack [NF19, SSZ17, NKMS16] will be the attack scenarios that are considered within this thesis.

This chapter will follow the steps 5-8, as well as parts of the steps 1 and 3 of a simulation study according to Banks et al. and Maria as described in Section 3.3 *Simulation Study* [BCNN10, Mar97]. The steps are executed for every attack that is evaluated. The first steps 1-4 have already been conducted in Chapter 5 *Ethereum Attack Simulator*. Because the simulation of different attack scenarios leads to a new problem specification and an extension of the simulator's model, steps 1 and 3 have to be executed again. This is done in sections 6.1.1 and 6.2.1 for the problem specification, as well as in sections 6.1.3 and 6.2.3 for the model extension. Step 5 of a simulation study, the experiment design, is done in sections 6.1.2 and 6.2.2, where the hypotheses deducted from literature review are stated. A simulation study's steps 6 and 7 are executed in sections 6.1.4 and 6.2.4, where the execution of the experiments is described and the measurements are pointed out. The interpretation and discussion of results, corresponding to step 8 of a simulation study, are elaborated in sections 6.1.5 and 6.2.5, respectively.

The results of an attack simulation and the effectiveness of an attack itself will be measured by the monetary advantage an attacker gains by executing such an attack compared to the monetary value of simply sticking to the Ethereum consensus protocol. An additional measure of an attack's success that is evaluated in this chapter is the impact on consensus security through an attacker's advantage of including more blocks into the main chain when executing the specific attack. If significant, the monetary advantage will be examined for pre- and post-EIP1559 implementation [BCD22], determining the impact the difference in transaction fee pricing and burning has on attack feasibility.

6.1 Selfish Mining

As already described in Section 2.2.1 *Selfish Mining*, the selfish mining attack is a form of withholding newly discovered blocks by a selfish miner so that honest miners waste computational power by mining on a potentially stale system state. This section investigates the possible significance of this attack on the Ethereum network.

6.1.1 Execution

Figure 2.1 shows an illustration of the selfish mining strategy, with M_h being the honest miner and M_s being the selfish miner. The selfish mining attack starts when M_s finds block b_{n+1}. Without releasing it, M_s starts mining on block b_{n+2}, so M_h still tries to find b_{n+1}. This approach is common within all different strategies [ES13, NF19, SSN$^+$19, LRDJ18]. This thesis will now go on with the strategy of Eyal and Sirer [ES13], who first described the selfish mining attack, and mention other strategies that deviate from specific points. M_s mines on the private branch for as long as he is at least one block ahead of the public chain. If M_h now finds b_{n+1} before M_s finds b_{n+2}, M_s immediately releases b_{n+1} for a block race that could go either way. In case M_s has already found b_{n+2} and only then M_h finds b_{n+1}, M_s releases all of his private blocks and wins the rewards of both blocks. If M_s is more than two blocks ahead, and M_h finds a new block, M_s only publishes the first unpublished block in the private branch. If the private and public chain is equal in length, M_s still mines on the private branch because finding the next block could guarantee M_s the rewards of prior blocks that would otherwise be stale blocks.

6.1.2 Hypotheses

Eyal and Sirer showed in their initial publication on the selfish mining attack on Bitcoin that for an attacker A with a network capacity $\gamma = 0.5$, the threshold for the fraction of the hash power for becoming profitable is $\alpha = 0.25$ [ES13]. The parameter γ denotes the amount of miners mining on top of the block of an attacker when it comes to block races, with $\gamma = 0.5$, meaning that 50% of miners choose the attacker's block. Because EthAttackSim implements the Geth client's behaviour as much as possible, the network capacity is assumed to be 50% through random selection of the block to mine on by default.

This leads to the formulation of our first null hypothesis

H_{0_1} = An attacker with a hash power of $\alpha \geq 0.25$ has a financial advantage over honest miners all the time

with an alternative hypothesis H_{1_1} indicating that there is an attacker with $\alpha \geq 0.25$ that has no financial advantage over honest miners.

The second null hypothesis is derived from Niu and Feng's paper on the selfish mining attack on Ethereum [NF19]. They show that with Ethereum's specific protocol characteristics, with uncle block rewards leading the way, an attacker A is profitable with a hash power threshold $\alpha = 0.163$, a network capacity $\gamma = 0.5$ and $K_u = 4/8$, where K_u denotes the average reward received for an uncle block, which is 4/8 of the block reward in this case. In the Ethereum consensus protocol, K_u has a range of 2/8 to 7/8 of the block rewards, decreasing with increasing difference in block height from uncle to nephew block.

This leads to the formulation of our second null hypothesis, this time in a negative formulation

H_{0_2} = An attacker with a hash power of $\alpha < 0.163$ has no monetary advantage over honest miners

with an alternative hypothesis H_{1_2} indicating that there is an attacker with $\alpha < 0.163$ that has a monetary advantage over honest miners.

6.1.3 Setup

Figure 5.1 shows the overview of the EthAttackSim architecture. All that has to be done is to change the consensus layer to match the strategy of a selfish mining attacker described in Section 6.1.1 *Execution* and add the new implementation to our factory to instantiate the attacker at the beginning of the simulation.

To implement the selfish mining consensus algorithm, it is necessary to hook into events indicating the reception of a new block or parts of it from other miners, as well as to change the behaviour of the event that an attacker mines a block. If an attacker's private chain is ahead of the public chain – that means a selfish mining attack is ongoing – it listens to received events from other nodes that indicate a new block to decide how to proceed with the private blocks according to the strategy defined previously. Additionally, the attacker's block announcement was designed to broadcast a block entirely to all peers and not only to a part of the peers as designed by the Ethereum consensus protocol, which, although it does not increase the network capacity γ, makes the blocks arrive faster at some peers. Besides that, the configuration parameters used are the same as described in Section 5.6 *Validation*.

It is worth noting that in theoretical research related to the selfish mining attack, transaction fees have not been considered, while this work considers transaction fees for simulations pre-EIP1559. To have comparable results, the post-EIP1559 simulation results should be favoured because of the transaction fee burning that was introduced with EIP1559. The post-EIP1559 simulation results do not include transaction fees as well.

6.1.4 Simulation Measurements

Similar to the data collection for the simulator validation described in Section 5.6.1 *Collecting Data*, the simulations are executed 30 times for two days of simulation time, each with an amount of 200 mining nodes. The measurement points have been chosen by selecting the top 4 mining pools at the time of writing according to miningpoolstats.stream

[Min22], one time using the top two mining pools grouped together as one attacker to simulate an exorbitant amount of hash power near the 50% threshold. Table 6.1 shows the most important measurements according to the previously defined hypotheses.

Hash Rate % [1]	Mined Blocks % [2]	Rewards %	Reward Increase %
46.94 [3]	63.21	55.73	43.24
29.55	35.62	31.94	25.16
17.38	17.62	17.28	9.16
10.53	9.76	10.12	1.96
8.01	7.10	7.53	-1.55

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Biggest two mining pools combined

Table 6.1: Measurement summary of the selfish mining attack using real-world hash rate of the 4 largest mining pools

It is assumed that the Ethereum fee market change that happened with EIP1559 changes the dynamics of attacks on the Ethereum network, so the simulation measurements have been collected for this new system of transaction pricing as well [eth22]. As EthAttackSim does not monitor the account balances, the two transaction pricing systems are distinguished in the postprocessing of the simulation, allowing to change certain parts of it without conducting the whole simulation again. Table 6.2 shows the measurements after EIP1559 was applied.

Hash Rate % [1]	Mined Blocks % [2]	Rewards %	Reward Increase %
46.94 [3]	63.21	53.46	46.52
29.55	35.62	30.77	26.96
17.38	17.62	17.17	12.18
10.53	9.76	10.24	5.56
8.01	7.10	7.69	2.27

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Biggest two mining pools combined

Table 6.2: Measurement summary of the selfish mining attack after EIP1559 using real-world hash rate of the 4 largest mining pools

6.1.5 Results

This section discusses the simulation measurements with regard to the hypotheses defined in Section 6.1.2 *Hypotheses*. It is worth mentioning that although the hypotheses based on literature review only consider possible monetary advantages of attackers, this section additionally considers an advantage of included blocks in the main chain, possibly leading to consensus security problems.

Hypothesis 1

H_{0_1} = An attacker with a hash power of $\alpha \geq 0.25$ has a financial advantage over honest miners all the time.

As Table 6.1 and Table 6.2 show, the monetary rewards for an attacker increase when $\alpha \geq 0.1053$ or even with $\alpha \geq 0.0801$ for post-EIP1559, therefore, H_{0_1} cannot be dismissed when taking into account the monetary advantage. When considering the impact a selfish miner has on the Ethereum network stability, the data shows that an attacker can propose over 63% of the network's blocks with a hash rate below 47%. With about 30% hash rate, the attacker can still propose over 35% of blocks in the main chain. Considering an attacker that aims at harming the network rather than gaining monetary advantage, H_{0_1} cannot be dismissed either.

Hypothesis 2

H_{0_2} = An attacker with a hash power of $\alpha < 0.163$ has no monetary advantage over honest miners.

Table 6.1 shows that the threshold for the hash rate α is between 0.0801 and 0.1053, with an attacker with $\alpha = 0.1053$ still gaining a monetary advantage over honest miners. This is even more the case when looking at Table 6.2 because post-EIP1559, even an attacker with $\alpha = 0.0801$ gains monetary advantage. The data shows a monetary advantage for an attacker with $\alpha < 0.163$, so H_{0_2} is dismissed for H_{1_2}. However, this is not true for the amount blocks included in the main chain, here H_{0_2} cannot be dismissed because no attacker could be found that increased its amount of included blocks with $\alpha < 0.163$.

Other Observations

One important observation is that the increase of monetary rewards for an attacker rises after EIP1559 has gone live. This may be caused by the transaction fees carrying no more weight in the equation, meaning a lost block race does not affect the monetary rewards for an attacker as much as Pre-EIP1559. This also leads to a decreasing hash rate threshold α for being profitable after EIP1559. Transaction pricing post-EIP1559 can still include tips, which were not considered in these simulations and it would be an interesting topic for future research to investigate if tips have an impact on, e.g., the monetary advantage of selfish mining.

Additionally, it is very interesting that an attack not only increases the monetary reward for the attacker but also for the honest miners. Pre-EIP1559, the increased rewards range from about 0.85-1.5% on average (for $\alpha \leq 0.163$ and $\alpha \geq 0.4694$) up to 14.9% for α in between. Post-EIP1559, the range is 4.2-12.5% on average, directly proportional to the α value of the attacker. This increase could result from the increased amount of uncle blocks associated with increased nephew rewards for including uncles, but this should definitely be investigated further in future research.

Lastly, it can be said that the higher impact a selfish mining attack has compared to the findings of Eyal and Sirer [ES13] seems to be caused mainly by Ethereum's uncle and nephew rewards, just like Niu and Feng described [NF19].

6.2 Verifier's Dilemma

The verifier's dilemma (see Section 2.2.2 *Verifier's Dilemma*) is an attack that exploits unrewarded parts of the Ethereum protocol to gain leverage. More precisely, miners must first validate any potential state updates upon receiving a new block, and this necessity for validation is not adequately reimbursed as part of the protocol rules. Furthermore, techniques like parallel validation of state changes are difficult to implement because of dependencies between transaction state updates. This enables the verifier's dilemma attack to gain an advantage by exploiting the sequential validation process and hence increasing the overall validation time of a received block. The upcoming section focuses on the implications a verifier's dilemma attack has on the stability and reward distribution in Ethereum.

6.2.1 Execution

There are two known forms of the verifier's dilemma attack, the classical or plain verifier's dilemma and sluggish mining or the forced verifier's dilemma attack. This subsection describes them and their execution further.

Verifier's Dilemma

The plain verifier's dilemma attack exploits the fact that there is no remuneration for verifying transactions in Ethereum. Admittedly, there is a penalty for including or accepting invalid transactions in a block because such a block will not be accepted by other nodes. This, however, may not affect miners at all if only valid blocks are broadcasted to the network, hence no attacks and no random errors in block creation occur. The exploit, in that case, is pretty simple: do not verify transactions or compute the state changes, at least for transactions that are known to be computationally expensive. This way, the attacker benefits from not spending the amount of time on block verification but instead starts to mine on the received block instantaneously, giving the adversary a slight time advantage.

Sluggish Mining (Forced Verifier's Dilemma)

The forced verifier's dilemma (also called sluggish mining [PTS19]) is an extension to the plain variant, where the attacker uses the fact that honest miners verify the validity of each block to achieve a time advantage when mining on a new block. This is done by deliberately including transactions into a self-mined block which increases the time consumption when verifying them, delaying their mining start time for the next block.

6.2.2 Hypotheses

Alharby et al. show in their paper about the verifier's dilemma that every attacker who skips verification of transactions (without including any transaction that increases block validation time) when the block gas limit is 12.5M gas/block should have a monetary advantage of about 2.5% over honest miners [ACLAvM20].

This leads to the formulation of our first null hypothesis

H_{0_1} = A verifier's dilemma attacker with $\alpha \geq 0.05$ gains monetary leverage of about 2.5% with a block gas limit of $BL \geq 12.5Mgas/block$

with an alternative hypothesis H_{1_1} indicating that there is an attacker with $\alpha \geq 0.05$ that has no monetary advantage over honest miners.

The second null hypothesis is derived from the works of Pontiveros et al. about sluggish mining [PTS19]. They showed that it is possible to increase the advantage when using the forced verifier's dilemma attack. According to their studies, with a block reward of 2 ETH and transaction fees of about 0.08 ETH per block (equals to about 6.4 GWEI per unit with a block limit of 12.5M gas/block), an attacker should start to be profitable with $\alpha \geq 0.35$. Their experiment setup was based on a block verification time of 1.21 seconds with a block gas limit of 8M gas/block and blocks filled with sluggish transactions at 100%. As shown in Table 5.4, this indicates a transaction state computation delay of 2280 gas/MHz/s (see also Section 5.4.1 *Transaction State Computation*).

This leads to the formulation of our second null hypothesis

H_{0_2} = With a block gas limit of $BL \geq 12.5Mgas/block$, transaction fees of 0.08 ETH per block and a transaction state computation delay of 2280 gas/MHz/s, a sluggish miner starts being profitable with $\alpha \geq 0.35$

with an alternative hypothesis H_{1_2} indicating that there is an attacker with $\alpha \geq 0.35$ that has no advantage over honest miners.

6.2.3 Setup

As there are two variants of the verifier's dilemma attack that need to be simulated, this subsection explains in detail what had to be done for each. Besides that, the configuration parameters used are the same as described in Section 5.6 *Validation*.

Verifier's Dilemma

For the plain verifier's dilemma, a new consensus layer was implemented that does not verify or compute the state of any transactions. It was assumed that there is no other adversary currently running an attack on the network.

Sluggish Mining (Forced Verifier's Dilemma)

The sluggish mining attack is trickier to implement. A new consensus layer was developed that extends the one from the plain verifier's dilemma and additionally fills a block to a configurable extent with an adversarial transaction that increases transaction verification time for honest miners. Table 5.4 shows again an overview of the different transaction state computation times that have been identified as plausible values for the transaction state computation delay. Additionally, a configurable parameter was introduced that allows an attacker to increase the gas limit of a block up to the highest allowed value by the Ethereum consensus protocol.

6.2.4 Simulation Measurements

The simulations are executed 30 times for two days of simulation time, each with an amount of 200 mining nodes. The measurement points have been selected according to the defined hypotheses. Again, because of the assumption of impacts through the fee market change introduced with EIP1559, the simulation results have been computed with EIP1559 in mind too. Table 6.3 and Table 6.4 show the measurements pre- and post-EIP1559 related to hypothesis H_{0_1}. Table 6.5 and Table 6.6 show the measurements pre- and post-EIP1559 related to hypothesis H_{0_2}. Since the verifier's dilemma is a highly dynamic attack customizable with different parameters in EthAttackSim, more simulations that may lead to interesting results have been conducted and are summarized below, as shown in Table 6.7 and Table 6.8.

Hash Rate % [1]	Mined Blocks % [2]	Rewards %	Reward Increase %
46.94 [3]	48.45	47.66	1.05
35.00 [4]	35.85	35.37	0.78
29.55	30.11	29.73	0.43
10.53	10.53	10.51	-0.20

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Biggest two mining pools combined
[4] Profitability threshold according to Pontiveros et al. [PTS19]

Table 6.3: Measurement summary of the verifier's dilemma attack with average gas price of 65.5 GWEI using real-world hash rate of selected mining pools and the profitability threshold according to Pontiveros et al. [PTS19]

Hash Rate % [1]	Mined Blocks % [2]	Rewards %	Reward Increase %
46.94 [3]	48.45	47.14	0.01
35.00 [4]	35.85	35.17	-0.21
29.55	30.11	29.58	-0.23
10.53	10.53	10.49	-0.34

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Biggest two mining pools combined
[4] Profitability threshold according to Pontiveros et al. [PTS19]

Table 6.4: Measurement summary of the verifier's dilemma attack with average gas Price of 65.5 GWEI after EIP1559 using real-world hash rate of selected mining pools and the profitability threshold according to Pontiveros et al. [PTS19]

Hash Rate % [1]	Mined Blocks % [2]	Rewards %	Reward Increase %
46.94 [3]	48.76	46.45	-2.34
35.00 [4]	35.94	34.39	-2.13
29.55	30.14	28.93	-2.19

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Biggest two mining pools combined
[4] Profitability threshold according to Pontiveros et al. [PTS19]

Table 6.5: Measurement summary of sluggish verifier's dilemma attack simulations with average gas price of 6.4 GWEI, 2280 gas/MHz/s and blocks filled with sluggish transactions at 100% using real-world hash rate of selected mining pools and the profitability threshold according to Pontiveros et al. [PTS19]

Hash Rate % [1]	Mined Blocks % [2]	Rewards %	Reward Increase %
46.94 [3]	48.76	46.36	-2.59
35.00 [4]	35.94	34.33	-2.33
29.55	30.14	28.87	-2.32

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Biggest two mining pools combined
[4] Profitability threshold according to Pontiveros et al. [PTS19]

Table 6.6: Measurement summary of sluggish verifier's dilemma attack simulations with average gas price of 6.4 GWEI, 2280 gas/MHz/s and blocks filled with sluggish transactions at 100% after EIP1559 using real-world hash rate of selected mining pools and the profitability threshold according to Pontiveros et al. [PTS19]

Hash Rate % [1]	Mined Blocks % [2]	Rew. %	Rew. Inc. %	Gas/MHz/s	Gas % [3]
35.00 [4]	79.23	71.11	115.21	47.61	50
35.00	46.93	40.64	32.59	47.61	25
35.00	36.01	28.32	-26.70	2280.00	100
35.00	35.97	34.84	-1.70	2280.00	10

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Percentage of a sluggish block's gas limit filled with sluggish transactions
[4] Profitability threshold according to Pontiveros et al. [PTS19]

Table 6.7: Measurement summary of other sluggish verifier's dilemma attack simulations using the profitability threshold according to Pontiveros et al. [PTS19]

Hash Rate % [1]	Mined Blocks % [2]	Rew. %	Rew. Inc. %	Gas/MHz/s	Gas % [3]
35.00 [4]	79.23	65.68	97.11	47.61	50
35.00	46.93	38.69	19.35	47.61	25
35.00	36.01	24.81	-38.48	2280.00	100
35.00	35.97	34.37	-3.55	2280.00	10

[1] Percentage of the attacker's network hash rate
[2] Percentage of mined blocks in the main chain
[3] Percentage of a sluggish block's gas limit filled with sluggish transactions
[4] Profitability threshold according to Pontiveros et al. [PTS19]

Table 6.8: Measurement summary of other sluggish verifier's dilemma attack simulations after EIP1559 using the profitability threshold according to Pontiveros et al. [PTS19]

6.2.5 Results

This section discusses the simulation measurements with regard to the hypotheses defined in Section 6.2.2 *Hypotheses*. It is worth mentioning that, although the hypotheses based on literature review only consider possible monetary advantages of attackers, this section additionally considers an advantage of included blocks in the main chain, possibly leading to consensus security problems.

Hypothesis 1

H_{0_1} = A verifier's dilemma attacker with $\alpha \geq 0.05$ gains monetary leverage of 2.5% with a block gas limit of $BL \geq 12.5 Mgas/block$

Table 6.3 shows that the monetary reward increases only a little compared to honest miners, or even decreases for an $\alpha = 0.1053$, therefore, H_{0_1} is dismissed with only the monetary advantage in mind in favor of H_{1_1}, indicating that there is an attacker with $\alpha \geq 0.05$ that has no financial advantage over honest miners. Regarding the mined blocks, things look a little different. One can see that there is at least no decrease in percentage, but an advantage of about 2.5% in block inclusions can not be reached with any α, so H_{0_1} can still be dismissed. After EIP1559, the outcome gets worse, having no increase in monetary rewards effectively (see Table 6.4).

Hypothesis 2

H_{0_2} = With a block gas limit of $BL \geq 12.5Mgas/block$, transaction fees of 0.08 ETH per block and a transaction state computation delay of 2280 gas/MHz/s, a sluggish miner starts being profitable with $\alpha \geq 0.35$

When looking at Table 6.5, no increase in monetary advantage over honest miners can be observed, the same is true for post-EIP1559 measurements (see Table 6.6). That leads to the hypothesis H_{0_2} being dismissed in favour of H_{1_2}, indicating that there is an attacker with $\alpha \geq 0.35$ that has no financial advantage over honest miners, because all adversaries decrease their monetary rewards regardless of their α value. Regarding the mined blocks, an adversary gains an advantage of up to 1.82% in blocks on the main chain, which can be considered a slight advantage, so H_{0_2} cannot be dismissed in this case, although an attacker would probably not risk its money for that little of an advantage. The slight advantage in included blocks occurs for all simulation runs regardless of the attacker's α value.

Other Observations

To gain insights into which parameters of the verifier's dilemma would lead to a significant destabilization or increase of monetary advantage, some additional simulations have been executed. These observations can be seen in Table 6.7 and Table 6.8. When adducing the lowest ever recorded value for the transaction state computation in gas/MHz/s – as seen in Ethereum Foundations blog post [Fou21b] – 47.61 gas/MHz/s, with only 50% of the block filled with the sluggish transaction, an adversary with $\alpha = 0.35$ can include a large amount of 79.23% of blocks into the main chain while increasing its monetary rewards by 115.21%. Post-EIP1559, the attacker can still increase its reward by 97.11%. The advantage gained by an adversary in terms of hash rate would completely undermine the security guarantees of the protocol and could have been used to attack the network. However, this bug in transaction pricing was concealed until it was fixed to prevent exploitation. Nevertheless, by using our simulator, it is now more straightforward to quantify the actual risk this transaction pricing mismatch would have had on the security of the protocol, so this is a good example of why adversarial simulation environments are needed.

A second observation was concluded in addition to the measurements for hypothesis H_{0_2}, but with a more reasonable gas price for today of about 65.5 GWEI, contrary to the price of 6.4 GWEI that was initially used. Here it can be seen that the rewards pre- and post-EIP1559 decreased significantly. The decrease of monetary reward in post-EIP1559 simulation can be traced back to the change in the fee market price of Ethereum that came with EIP1559, where currently, the base fee of a transaction is burned. This way, the attacker loses all the fees paid for the sluggish transaction. The decrease in monetary reward in pre-EIP1559 simulation can be attributed to the increase in lost transaction fees caused by the increased costs of the sluggish transaction.

6.3 Summary

The analysis of the hypotheses showed that according to the simulation outcomes, the selfish mining attack could be of great advantage, even if an attacker does not own a big part of Ethereum's hash rate. It appears that the verifier's dilemma is currently less effective than expected based on prior research, however, this depends on the concrete parametrization and that a mismatch in the real execution cost in relation to the price paid in transaction fees can result in catastrophic failures of the underlying security assumptions. In terms of EIP1559, the selfish mining attack gained an advantage after EIP1559 was implemented, while the verifier's dilemma lost some of its power.

In conclusion, it can be said that the results and findings of previous research need to be continuously re-evaluated and updated because parametrizations and designs of real-world blockchains, such as gas price changes, novel fee mechanisms, or protocol rules, can change and seriously impact the overall security.

CHAPTER 7

Conclusion

Cryptocurrencies and blockchain technologies are currently being developed in a fast-paced environment where designs and concrete parametrizations, such as active users, the value of coins and tokens, or even implementation details, can change frequently. It is a challenge to keep up to date and even more to evaluate such changes on a regular basis. Theoretical research showed that a lot of effort is needed to analyze, e.g., certain attack vectors or even to understand the behaviour of processes implemented in client programs.

To answer research question RQ_1*: "What is the current state of the art in regard to attacks against blockchain systems and how can they be categorized?"*, this thesis provides an overview of the state of the art regarding blockchain attacks and their theoretical and practical consequences. Based on literature review, a suitable categorization of attacks is elaborated and the resulting categories are explained in detail. Subsequently, various attacks available in literature are classified to give an overview of attack category assignment in Table 2.1. The chapter finishes with detailed descriptions of two selected attacks from the P2P system attack category – the most assigned category – that are relevant in the following, namely the selfish mining attack and the verifier's dilemma, which are chosen because of their comparable properties and effects on the Ethereum network.

Blockchain simulators are investigated in the ensuing chapter that focuses on research question RQ_2*: "What is the current state of the art regarding blockchain simulators?"*. Starting with a comparison of simulation and emulation, this chapter further elaborates on the different simulator types and implementation strategies and shows how a simulation study is constructed. Subsequently, various available simulator frameworks are compared regarding their suitability to simulate the Ethereum protocol with a focus on attack scenarios. Table 3.1 shows the results of this comparison. Unfortunately, no simulator that fits the requirements regarding Ethereum attack simulation is found.

To enable researchers and other interested readers to quickly get insights into how the Ethereum blockchain works, as well as to create the foundation for developing a simulation framework later on, this work provides a deep dive into the client implementation of Ethereum while concentrating on research question RQ_3*: "How does the most widespread client implementation, Geth, behave under various conditions?"*. The chapter starts with an overview of Ethereum client distribution at the time of writing and then focuses on the de-facto standard client implementation, Geth. Afterwards, Geth is examined via code review and the most important consensus protocol behaviours, as well as the message passing processes between clients, are stated in a sophisticated overview including state diagrams of specific flows.

With the knowledge gained from the literature and technological review, a simulation framework called EthAttackSim is created to enable evaluating the Ethereum blockchain regarding the impact attacks and parametrization changes can have on protocol security. The architecture and implementation of the simulator, as well as how the parameters for the stochastic model are acquired, are described in detail. The simulator is subsequently evaluated regarding block time, uncle rate and found blocks per node using real-world data collected from the live Ethereum mainnet to build a solid foundation for the simulation of realistic Ethereum network conditions. This evaluation also discovers that on-chain data does not follow the assumed distribution regarding block time, showing that miners may engage in block timestamp manipulation. Table 5.7 further illustrates that EthAttackSim fulfills previously selected requirements for simulating attacks against the Ethereum protocol while offering the user high configurability with the versatility of quickly implementing new features and attack scenarios, which solves research question RQ_4*: "How can realistic Ethereum network setups and attacks against them be simulated effectively?"*.

Finally, this work answers research question RQ_5*: "What impacts do the selfish mining attack and the verifier's dilemma have on real-world Ethereum setups?"* by simulating and evaluating both attacks, as well as a variation of the verifier's dilemma, the sluggish mining attack (or forced verifier's dilemma), under various conditions. This is done by explaining the attack execution in detail, followed by stating hypotheses derived from literature related to the specific attack and defining the setup and implementation regarding EthAttackSim. Afterwards, simulation runs are conducted and the results are evaluated with reference to the previously stated hypotheses and possible other observations. That gives excellent insights into the impact of these attacks regarding monetary rewards and possible security impacts on the Ethereum protocol before as well as after EIP1559 was implemented. The main conclusions are that the verifier's dilemma attack is a vast security concern of Ethereum if there is a mismatch between actual transaction execution costs and transaction pricing, as well as that the selfish mining attack can have a significant impact on Ethereum protocol security and attacker revenue. Additionally, it is vital to continuously re-evaluate the findings of previous research because parametrization changes, such as the implementation of EIP1559, could seriously impact overall security. Overall, it was shown that EthAttackSim can quickly provide much-needed insights into the practical impacts of attacks on Ethereum protocol security.

To reproduce the results of this thesis and for possible further evaluation of the simulator, the selfish mining attack and the verifier's dilemma, as well as other attack scenarios in future research, the code of EthAttackSim, together with data used for its configuration and validation, is made available on github.com and open source [Mai22].

7.1 Limitations

This work has a few limitations regarding the simulator implementation. For example, a simulator can only be as good as its model. As the whole Ethereum ecosystem is a very complex one, the model can never cover all needed parameters. However, it is the best approximation leaving aside some complex behaviours because they can not be easily modelled. Other parameters, like the CPU power of mining nodes, can only be estimated and may not accurately capture reality because the used hardware is not homogeneous. Also, the CPU power may not be the only parameter determining the amount of gas a node can handle per second. Furthermore, the timekeeping and event handling of the simulator could be improved because, e.g., very high block verification times lead to exceptionally long waiting times and high memory consumption.

It has to be stated that for an entirely accurate simulation of the attacks, the simulator has to also model the difficulty adjustment algorithm of the Ethereum protocol. This is the case because each of the attacks divides the network into two groups, a group of honest miners and the attacking group, and at least some time during the consensus process, the honest group is not mining on the longest chain, which leads to a decrease in the observed overall network hash rate and hence to a longer block time if the difficulty adjustment is not modelled. Unfortunately, with EthAttackSim, the difficulty adjustment is currently not part of the model and is an important topic for future research. To work around that issue, the rewards are calculated using extrapolation. By using the mean block time of a network without attackers collected earlier in this thesis, and the mean block time of the simulation runs with attackers, it is possible to extrapolate the rewards of attackers and other miners. A similar approach was taken by Eyal and Sirer in their original paper related to the selfish mining attack [ES13].

7.2 Future Work

This thesis introduces various possibilities for future research. One possibly interesting topic would be the investigation of the verifier's dilemma's importance on Ethereum after the merge (Ethereum's transition to PoS), as it seems to continue to be worrying, even though in a different form. Another issue is the discrepancy in the distribution of block times and the overrepresentation of blocks in the one-second bucket observed on the Ethereum mainnet, as it would be interesting to better understand whether this represents an artifact of the technical implementation of mining and mining pools or whether miners intentionally deviate from prescribed protocol rules in order to increase their potential profit. As transaction pricing post-EIP1559 can still include tips, which are not considered in the simulations, it would be an attractive topic for future research to investigate if tips have an impact on, e.g., the monetary advantage of selfish mining.

Regarding the simulator implementation, the fact that difficulty adjustment and network bandwidth are currently not part of the model or that very high block verification times lead to exceptionally long execution times and high memory consumption in the simulator opens up different options to improve in future work, as it may be interesting to add more dimensions to the simulation of required resources. The circumstance that the simulator has no ability to simulate the Ethereum network after the merge could lead to further important topics for future research.

The following list gives an overview of suggestions for future research, including but not limited to:

- model and evaluate other theoretically known attacks on Ethereum,
- investigate the overrepresentation of blocks in the one-second bucket,
- analyze the verifier's dilemma's importance on Ethereum after the merge,
- integrate a comprehensive model of transaction fees into the simulator,
- analyze Ethereum's behaviour by adjusting different parameters such as block reward, uncle reward, etc.,
- implement difficulty adjustments for correctly simulating the hash rate distribution if miners mine on different branches,
- investigate the impact of tips post-EIP1559,
- improve time-handling in the simulator,
- integrating the changes made to the consensus protocol by the merge,
- improve space and runtime requirements of the simulator,
- integrate network bandwidth into simulations,
- enable node discovery through a known algorithm used in, e.g., Geth.

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 31

ARTIS Advances RTI System. 25

Blocksim (A) BlockSim (Alharby & van Morsel). 24–26, 44, 63

Blocksim (F) BlockSim (Faria & Correira). 24, 26, 29, 63

DDoS Distributed Denial of Service. 11

DeFi Distributed Finance. 105

DESE Discrete Event Simulation Engine. 24–26, 44

DevOpSec Development & Operations Security. 8–11, 102–105

DNS Domain Name System. 9

EIP Ethereum Improvement Proposal. xi, xiii, 2, 65, 67–70, 72–76, 78, 80, 83, 84

EthAttackSim Ethereum Attack Simulator. 43–45, 48, 49, 57–63, 65–68, 72, 78, 79, 81, 83

EVM Ethereum Virtual Machine. 2

GAIA Generic Adaptive Interaction Architecture. 25

Geth GoEthereum. xi, xiii, 2–4, 6, 14, 24, 29–33, 43, 44, 49, 55, 66, 78, 80

Go Golang. 31, 56, 63

GoF Goodness of Fit. 58

ICO Initial Coin Offering. 8

IMA Incentive Manipulation Attack. 11

JSON Javascript Object Notation. 31

KS-Test Kolmogorov-Smirnov-Test. 56, 58, 59, 61

LUNES Large Unstructured Network Simulator. 25, 26

MDP Markov Decision Process. 1

OPSEC Operations Security. 8

P2P Peer-to-Peer. 8–12, 30, 32, 44, 57, 77, 103

PADS Parallel and Distributed Simulation. 25, 26

PDF Probability Density Function. 22

PoS Proof of Stake. 2, 32, 79

PoW Proof of Work. 2, 12, 13, 23, 24, 41

RLP Recursive Lenght Prefix. 33

RPC Remote Procedure Call. 31

RQ Research Question. 4–6, 77, 78

Bibliography

[ABL+18] Divesh Aggarwal, Gavin K. Brennen, Troy Lee, Miklos Santha, and Marco Tomamichel. Quantum attacks on Bitcoin, and how to protect against them. *ledger*, 3, October 2018. arXiv: 1710.10377. URL: `http://arxiv.org/abs/1710.10377`, `doi:10.5195/ledger.2018.127`.

[ACLAvM20] Maher Alharby, Roben Castagna Lunardi, Amjad Aldweesh, and Aad van Moorsel. Data-Driven Model-Based Analysis of the Ethereum Verifier's Dilemma. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 209–220, Valencia, Spain, June 2020. IEEE. URL: `https://ieeexplore.ieee.org/document/9153385/`, `doi:10.1109/DSN48063.2020.00038`.

[ADMM15] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. On the Malleability of Bitcoin Transactions. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, volume 8976, pages 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. Series Title: Lecture Notes in Computer Science. URL: `http://link.springer.com/10.1007/978-3-662-48051-9_1`, `doi:10.1007/978-3-662-48051-9_1`.

[Ale19] Ana Alexandre. Report: Crypto-Related Fraud and Theft Resulted in $4.4B Loss in 2019, November 2019. URL: `https://cointelegraph.com/news/report-crypto-related-fraud-and-theft-resulted-in-44b-loss-in-2019`.

[ALRV05] Andres A Aristizabal, Hugo A Lopez, Camilo Rueda, and Frank D Valencia. Formally Reasoning About Security Issues in P2P Protocols: A Case Study. page 23, January 2005.

[ANC+19] Ayman Alkhalifah, Alex Ng, Mohammad Jabed Morshed Chowdhury, A. S. M. Kayes, and Paul A. Watters. An Empirical Analysis of Blockchain Cybersecurity Incidents. In *2019 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pages 1–8, Melbourne, Australia, December 2019. IEEE. URL: `https://ieeexplore.ieee.org/document/9162381/`, `doi:10.1109/CSDE48274.2019.9162381`.

[AOK+19] Yusuke Aoki, Kai Otsuki, Takeshi Kaneko, Ryohei Banno, and Kazuyuki Shudo. SimBlock: A Blockchain Network Simulator. *arXiv:1901.09777 [cs]*, March 2019. arXiv: 1901.09777. URL: http://arxiv.org/abs/1901.09777.

[AvM19] Maher Alharby and Aad van Moorsel. BlockSim: A Simulation Framework for Blockchain Systems. *SIGMETRICS Perform. Eval. Rev.*, 46(3):135–138, January 2019. URL: https://dl.acm.org/doi/10.1145/3308897.3308956, doi:10.1145/3308897.3308956.

[AvM20] Maher Alharby and Aad van Moorsel. BlockSim: An Extensible Simulation Tool for Blockchain Systems. *Front. Blockchain*, 3:28, June 2020. arXiv: 2004.13438. URL: http://arxiv.org/abs/2004.13438, doi:10.3389/fbloc.2020.00028.

[Azi19] Master the Crypto Founder Aziz. Bitcoin Scams: Bitcoin Hacks,Theft and Exit Scams History, November 2019. Section: Bitcoin. URL: https://masterthecrypto.com/bitcoin-scams/.

[AZV17] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. *arXiv:1605.07524 [cs]*, March 2017. arXiv: 1605.07524. URL: http://arxiv.org/abs/1605.07524.

[Bas15] Martijn Bastiaan. Preventing the 51%-Attack: a Stochastic Analysis of Two Phase Proof of Work in Bitcoin. *22nd Twente Student Conference on IT*, page 10, 2015.

[BBC19] BBC. Cryptoqueen: How this woman scammed the world, then vanished. *BBC News*, November 2019. URL: https://www.bbc.com/news/stories-50435014.

[BCD22] Vitalik Buterin, Eric Conner, and Rick Dudley. Ethereum Improvement Proposals (EIPs), April 2022. original-date: 2015-10-26T13:57:23Z. URL: https://github.com/ethereum/EIPs/blob/7716002ef5d7e4092dd7bb41576f784f563a7dbc/EIPS/eip-1559.md.

[BCNN10] Jerry Banks, John Carson, Barry Nelson, and David Nicol. *Discrete-Event System Simulation*. Prentice Hall, 5 edition, January 2010.

[BDWW14] Tobias Bamert, Christian Decker, Roger Wattenhofer, and Samuel Welten. BlueWallet: The Secure Bitcoin Wallet. In Sjouke Mauw and Christian Damsgaard Jensen, editors, *Security and Trust Management*, volume 8743, pages 65–80. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-319-11851-2_5, doi:10.1007/978-3-319-11851-2_5.

[BHH+14] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic Curve Cryptography in Practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security*, volume 8437, pages 157–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. Series Title: Lecture Notes in Computer Science. URL: `http://link.springer.com/10.1007/978-3-662-45472-5_11`, `doi:10.1007/978-3-662-45472-5_11`.

[BKKT18] R. Bowden, H. P. Keeler, A. E. Krzesinski, and P. G. Taylor. Block arrivals in the Bitcoin blockchain. *arXiv:1801.07447 [cs]*, January 2018. arXiv: 1801.07447. URL: `http://arxiv.org/abs/1801.07447`.

[Bon01] Peter L. Bonate. A Brief Introduction to Monte Carlo Simulation:. *Clinical Pharmacokinetics*, 40(1):15–22, 2001. URL: `http://link.springer.com/10.2165/00003088-200140010-00002`, `doi:10.2165/00003088-200140010-00002`.

[Bon17] Joseph Bonneau. On hostile blockchain takeovers. page 4, 2017.

[But21] Vitalik Buterin. Ethereum Whitepaper, February 2021. URL: `https://ethereum.org`.

[BW10] Eduard Babulak and Ming Wang. Discrete Event Simulation: State of the Art. In Aitor Goti, editor, *Discrete Event Simulations*. Sciyo, August 2010. URL: `http://www.intechopen.com/books/discrete-event-simulations/discrete-event-simulation-state-of-the-art`, `doi:10.5772/9894`.

[Byr17] Todd Byrne. Cocaine, Women and Crime Bosses: How Onecoin Became One of the Biggest Scams in Crypto History, April 2017. URL: `https://bitsonline.com/onecoin-biggest-scam-history/`.

[CB14] Nicolas T. Courtois and Lear Bahack. On Subversive Miner Strategies and Block Withholding Attack in Bitcoin Digital Currency. *arXiv:1402.1718 [cs]*, December 2014. arXiv: 1402.1718. URL: `http://arxiv.org/abs/1402.1718`.

[CD20] Gertrude Chavez-Dreyfuss. Cryptocurrency crime losses more than double to $4.5 billion in 2019, report finds. *Reuters*, February 2020. URL: `https://www.reuters.com/article/us-crypto-currencies-crime-idINKBN2051VT`.

[CFS06] R. Chertov, S. Fahmy, and N.B. Shroff. Emulation versus simulation: a case study of TCP-targeted denial of service attacks. In *2nd International Conference on Testbeds and Research Infrastructures for the Development* *of Networks and Communities, 2006. TRIDENTCOM 2006.*, pages 10 pp.–325, Barcelona, 2006. IEEE. URL: http://ieeexplore.ieee.org/document/1649164/, doi:10.1109/TRIDNT.2006.1649164.

[CHH+19] Vincent Chia, Pieter Hartel, Qingze Hum, Sebastian Ma, Georgios Piliouras, Daniel Reijsbergen, Mark van Staalduinen, and Pawel Szalachowski. Rethinking Blockchain Security: Position Paper. *arXiv:1806.04358 [cs]*, April 2019. arXiv: 1806.04358. URL: http://arxiv.org/abs/1806.04358.

[Cip21] CipherTrace. Cryptocurrency Crime and Anti-Money Laundering Report, February 2021 - CipherTrace, 2021. URL: https://ciphertrace.com/2020-year-end-cryptocurrency-crime-and-anti-money-laundering-report/.

[Coi22a] CoinMarketCap. Bitcoin (BTC) Kurs, Grafiken, Marktkapitalisierung, August 2022. URL: https://coinmarketcap.com/de/currencies/bitcoin/.

[Coi22b] CoinMarketCap. Ethereum (ETH) Kurs, Grafiken, Marktkapitalisierung, August 2022. URL: https://coinmarketcap.com/de/currencies/ethereum/.

[Coi22c] CoinMarketCap. Globale Markttabellen für Kryptowährung, August 2022. URL: https://coinmarketcap.com/de/charts/.

[Com10] Bitcoin Community. Strange block 74638, August 2010. URL: https://bitcointalk.org/index.php?topic=822.0.

[Com11] Bitcoin Community. I just got hacked - any help is welcome! (25,000 BTC stolen), June 2011. URL: https://bitcointalk.org/index.php?topic=16457.0.

[Com20] Ethereum Community. openethereum/parity-ethereum, December 2020. original-date: 2015-11-23T11:07:32Z. URL: https://github.com/openethereum/parity-ethereum.

[Com21a] Ethereum Community. etclabscore/core-geth, February 2021. original-date: 2020-02-19T14:14:21Z. URL: https://github.com/etclabscore/core-geth.

[Com21b] Ethereum Community. ethereum/aleth, February 2021. original-date: 2013-12-26T21:26:06Z. URL: https://github.com/ethereum/aleth.

[Com21c] Ethereum Community. ethereum/ethereumj, February 2021. original-date: 2014-05-13T07:50:14Z. URL: https://github.com/ethereum/ethereumj.

[CPNX19] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses. *arXiv:1908.04507 [cs]*, August 2019. arXiv: 1908.04507. URL: http://arxiv.org/abs/1908.04507.

[CQHM19] Tarun Chitra, Monica Quaintance, Stuart Haber, and Will Martino. Agent-Based Simulations of Blockchain protocols illustrated via Kadena's Chainweb. *arXiv:1904.12924 [cs]*, April 2019. arXiv: 1904.12924. URL: http://arxiv.org/abs/1904.12924.

[Cry21] CryptoScamDB. Scams, January 2021. URL: https://cryptoscamdb.org/scams.

[DW13] Christian Decker and Roger Wattenhofert. Information Propagation in the Bitcoin Network. *13th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P*, page 10, November 2013. doi:https://doi.org/10.1109/P2P.2013.6688704.

[EGSvR16] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, page 16, 2016.

[ELMC18] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A First Look at Browser-Based Cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–66, London, April 2018. IEEE. URL: https://ieeexplore.ieee.org/document/8406561/, doi:10.1109/EuroSPW.2018.00014.

[ES13] Ittay Eyal and Emin Gun Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. page 18, November 2013. URL: https://arxiv.org/pdf/1311.0243.pdf.

[etha] ethereum.org. Ethereum Wire Protocol (ETH). URL: https://github.com/ethereum/devp2p/blob/master/caps/eth.md.

[ethb] ethereum.org. Kademilia-Peer-Selection. URL: https://eth.wiki/en/ideas/kademlia-peer-selection.

[ethc] ethereum.org. Node Discovery Protocol v4. URL: https://github.com/ethereum/devp2p/blob/master/discv4.md.

[ethd] ethereum.org. Node Discovery Protocol v5. URL: https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md.

[eth20a] ethereum.org. ethereum/go-ethereum v1.9.21, December 2020. original-date: 2013-12-26T13:05:46Z. URL: https://github.com/ethereum/go-ethereum.

[eth20b] ethernodes. Clients - ethernodes.org - The Ethereum Network & Node Explorer, December 2020. URL: https://www.ethernodes.org/.

[eth21a] ethereum.org. ethereum/pyethapp, February 2021. original-date: 2015-04-05T05:26:13Z. URL: https://github.com/ethereum/pyethapp.

[eth21b] ethereum.org. ethereum/trinity, February 2021. original-date: 2018-02-27T22:39:21Z. URL: https://github.com/ethereum/trinity.

[eth22] ethereum.org. EIP1559, August 2022. original-date: 2015-10-26T13:57:23Z. URL: https://github.com/ethereum/EIPs/blob/5ebec3ecf7606330d3e070665a00ffe01a74f0b9/EIPS/eip-1559.md.

[Eya14] Ittay Eyal. The Miner's Dilemma. *arXiv:1411.7099 [cs]*, November 2014. arXiv: 1411.7099. URL: http://arxiv.org/abs/1411.7099.

[Far] Carlos Faria. carlosfaria94/blocksim: A discrete event Blockchain Simulator. URL: https://github.com/carlosfaria94/blocksim.

[Far21] Carlos Faria. BlockSim: Blockchain Simulator, December 2021. original-date: 2018-03-12T11:34:18Z. URL: https://github.com/carlosfaria94/blocksim.

[FC19] Carlos Faria and Miguel Correia. BlockSim: Blockchain Simulator. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 439–446, Atlanta, GA, USA, July 2019. IEEE. URL: https://ieeexplore.ieee.org/document/8946201/, doi:10.1109/Blockchain.2019.00067.

[Fou] Ethereum Foundation. Ethereum WIki - Light Client Protokol. URL: https://eth.wiki/en/concepts/light-client-protocol.

[Fou16] Ethereum Foundation. The Homestead Release — Ethereum Homestead 0.1 documentation, 2016. URL: https://ethdocs.org/en/latest/introduction/the-homestead-release.html.

[Fou21a] CertiK Foundation. Blockchain Hacks: 2020 | $15 billion lost, how can we mitigate hacks in 2021?, January 2021. URL: https://medium.com/certik-foundation/blockchain-hacks-2020-15-billion-lost-how-can-we-mitigate-hacks-in-2021-f1ff8f051785.

[Fou21b] Ethereum Foundation. Dodging a bullet: Ethereum State Problems, May 2021. URL: https://blog.ethereum.org/2021/05/18/eth_state_problems/.

[Fou21c] Hyperledger Foundation. hyperledger/besu, February 2021. original-date: 2019-09-04T21:11:20Z. URL: https://github.com/hyperledger/besu.

[Fou22] Ethereum Foundation. The Merge, August 2022. URL: https://ethereum.org.

[FP94] J.W.C. Fu and J.H. Patel. Trace driven simulation using sampled traces. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 1, pages 211–220, 1994. doi:10.1109/HICSS.1994.323170.

[GCR16] Ilias Giechaskiel, Cas Cremers, and Kasper B Rasmussen. On Bitcoin Security in the Presence of Broken Crypto Primitives. In *Lecture Notes in Computer Science*, volume 9879, page 17. Springer, Cham, February 2016. doi:https://doi.org/10.1007/978-3-319-45741-3_11.

[Gog20] Jeff Gogo. $100 Million Liquidated on Defi Protocol Compound Following Oracle Exploit – News Bitcoin News, November 2020. URL: https://news.bitcoin.com/100-million-liquidated-on-defi-protocol-compound-following-oracle-exploit/.

[Gri17] Merunas Grincalaitis. The ultimate guide to audit a Smart Contract + Most dangerous attacks in Solidity, November 2017. URL: https://medium.com/ethereum-developers/how-to-audit-a-smart-contract-most-dangerous-attacks-in-solidity-ae402a7e7868.

[Har19] Harry. 2019 In Review: Major Blockchain/Crypto Security Incidents, December 2019. URL: https://medium.com/mycrypto/2019-in-review-major-blockchain-crypto-security-incidents-adb0e87e0f25.

[Har21] Harry. 2020 In Review: Major Blockchain/Crypto Security Incidents, January 2021. URL: https://medium.com/mycrypto/2020-in-review-major-blockchain-crypto-security-incidents-6c5ced8dc81e.

[Hat18] Taylor Hatmaker. Italian cryptocurrency exchange gets hacked for $170 million in Nano, February 2018. URL: https://social.techcrunch.com/2018/02/12/bitgrail-hack-nano/.

[HGL10] Robert L. Harrison, Carlos Granja, and Claude Leroy. Introduction to Monte Carlo Simulation. pages 17–21, Bratislava (Slovakia), 2010. URL: http://aip.scitation.org/doi/abs/10.1063/1.3295638, doi:10.1063/1.3295638.

[HKZG15] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. page 17, 2015.

[HN05] O Hoes and F Nelen. Continuous simulation or event-based modelling to estimate flood probabilities? *WIT Transactions on Ecology and the Environment*, 80:8, 2005.

[HNT+20] Ethan Heilman, Neha Narula, Garrett Tanzer, James Lovejoy, Michael Colavita, Madars Virza, and Tadge Dryja. Cryptanalysis of Curl-P and Other Attacks on the IOTA Cryptocurrency. *ToSC*, pages 367–391, September 2020. URL: `https://tosc.iacr.org/index.php/ToSC/article/view/8707`, `doi:10.46586/tosc.v2020.i3.367-391`.

[Inc20] Chainalysis Inc. PlusToken Scammers Didn't Just Steal $2+ Billion Worth of Cryptocurrency. They May Also Be Driving Down the Price of Bitcoin. [UPDATED 3/12/2020], March 2020. URL: `https://blog.chainalysis.com/reports/plustoken-scam-bitcoin-price`.

[Inc21] FIXR Inc. The Speed of Crypto Hacks is Picking Up: This Month Alone Thieves Stole $71.5M, January 2021. URL: `https://howmuch.net/articles/biggest-crypto-hacks-scams`.

[JH11] Rob Jansen and Nicholas Hooper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation:. Technical report, Defense Technical Information Center, Fort Belvoir, VA, September 2011. URL: `http://www.dtic.mil/docs/citations/ADA559181`, `doi:10.21236/ADA559181`.

[JSZ+19] Aljosha Judmayer, Nicholas Stifter, Alexei Zamyatin, Itay Tsabary, Ittay Eyal, Peter Gaži, Sarah Meiklejohn, and Edgar Weippl. Pay-To-Win: Cheap, Crowdfundable, Cross-chain Incentive Manipulation Attacks on Cryptocurrencies. page 26, 2019.

[Kaa20] Steve Kaaru. PlusToken scam: Top operators jailed for up to 11 years, December 2020. Section: Business. URL: `https://coingeek.com/plustoken-scam-top-operators-jailed-for-up-to-11-years/`.

[KAC12] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. page 17, 2012.

[KB12] Franziska Klügl and Ana L. C. Bazzan. Agent-Based Modeling and Simulation. *AIMag*, 33(3):29, September 2012. URL: `https://ojs.aaai.org/index.php/aimagazine/article/view/2425`, `doi:10.1609/aimag.v33i3.2425`.

[KKS+17] Yujin Kwon, Dohyun Kim, Yunmok Son, Eugene Vasserman, and Yongdae Kim. Be Selfish and Avoid Dilemmas: Fork After Withholding (FAW) Attacks on Bitcoin. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, pages 195–209, 2017. arXiv: 1708.09790. URL: `http://arxiv.org/abs/1708.09790`, `doi:10.1145/3133956.3134019`.

[Lei15] Matthias Lei. Exploiting Bitcoin's Topology for Double-spend Attacks. page 21, August 2015.

[LHXL20] Yizhong Liu, Yiming Hei, Tongge Xu, and Jianwei Liu. An Evaluation of Uncle Block Mechanism Effect on Ethereum Selfish and Stubborn Mining Combined With an Eclipse Attack. *IEEE Access*, 8:17489–17499, 2020. Conference Name: IEEE Access. `doi:10.1109/ACCESS.2020.2967861`.

[Llo15] Lloyd's. Lloyd's Emerging Risk Report - Bitcoin, 2015. URL: `https://www.lloyds.com/~/media/files/news-and-insight/risk-insight/2015/bitcoin--final.pdf`.

[LLW+19] Ziyao Liu, Nguyen Cong Luong, Wenbo Wang, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. A Survey on Blockchain: A Game Theoretical Perspective. 7:29, 2019.

[LRDJ18] Hanqing Liu, Na Ruan, Rongtian Du, and Weijia Jia. On the Strategy and Behavior of Bitcoin Mining with N-attackers. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security - ASIACCS '18*, pages 357–368, Incheon, Republic of Korea, 2018. ACM Press. URL: `http://dl.acm.org/citation.cfm?doid=3196494.3196512`, `doi:10.1145/3196494.3196512`.

[LSM05] Brian Neil Levine, Clay Shields, and N Boris Margolin. A Survey of Solutions to the Sybil Attack. *Technical Report of Univ of Massachussets Amherst*, (2006–052):11, November 2005.

[LTKS15] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying Incentives in the Consensus Computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 706–719, Denver, Colorado, USA, 2015. ACM Press. URL: `http://dl.acm.org/citation.cfm?doid=2810103.2813659`, `doi:10.1145/2810103.2813659`.

[LV17] Loi Luu and Yaron Velner. SMARTPOOL: Practical Decentralized Pooled Mining. page 19, August 2017.

[Mai22] Alexander Maitz. ethattacksim, October 2022. original-date: 2022-10-26T13:04:04Z. URL: `https://github.com/tu1426/eas`.

[Mar97] Anu Maria. Introduction to Modeling and Simulation. page 7, 1997.

[McG] Ryan McGeehan. Blockchain Graveyard. URL: `https://magoo.github.io/Blockchain-Graveyard/`.

[McG02] I. McGregor. The relationship between simulation and emulation. In *Proceedings of the Winter Simulation Conference*, volume 2, pages 1683–1688, San Diego, CA, USA, 2002. IEEE. URL: `http://ieeexplore.ieee.org/document/1166451/`, `doi:10.1109/WSC.2002.1166451`.

[MHG18] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network. page 15, 2018.

[Min22] MiningPoolStats.stream. Mining Pool Stats, October 2022. URL: `https://miningpoolstats.stream/`.

[MJ] Andrew Miller and Rob Jansen. Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications. page 8.

[Mos18] Jamal Hayat Mosakheil. Security Threats Classification in Blockchains. *Culminating Projects in Information Assurance*, page 142, 2018. URL: `https://repository.stcloudstate.edu/msia_etds/48`.

[MSH17] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. Refund Attacks on Bitcoin's Payment Protocol. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security*, volume 9603, pages 581–599. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017. Series Title: Lecture Notes in Computer Science. URL: `http://link.springer.com/10.1007/978-3-662-54970-4_34`, `doi:10.1007/978-3-662-54970-4_34`.

[Net21] Nethermind. NethermindEth/nethermind, February 2021. original-date: 2017-08-23T15:10:51Z. URL: `https://github.com/NethermindEth/nethermind`.

[NF19] Jianyu Niu and Chen Feng. Selfish Mining in Ethereum. *arXiv:1901.04620 [cs]*, April 2019. arXiv: 1901.04620. URL: `http://arxiv.org/abs/1901.04620`.

[NKMS16] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 305–320, Saarbrucken, March 2016. IEEE. URL: `http://ieeexplore.ieee.org/document/7467362/`, `doi:10.1109/EuroSP.2016.32`.

[OL04] P. Owezarski and N. Larrieu. A trace based method for realistic simulation. In *2004 IEEE International Conference on Communications (IEEE Cat. No.04CH37577)*, pages 2236–2239 Vol.4, Paris, France, 2004. IEEE. URL: `http://ieeexplore.ieee.org/document/1312915/`, `doi:10.1109/ICC.2004.1312915`.

[oSaT12] National Institute of Standards and Technology. NVD - CVE-2010-5139, 2012. URL: `https://nvd.nist.gov/vuln/detail/CVE-2010-5139`.

[Par18] Helen Partz. India: Crypto 'Scamsters' Bhardwaj Brothers Arrested For Duping Investors Out Of $300 Mln, April 2018. URL: `https://cointelegraph.com/news/india-crypto-scamsters-bhardwaj-brothers-arrested-for-duping-investors-out-of-300-mln`.

[PTG09] Rob Pike, Ken Thompson, and Robert Griesemer. The Go Programming Language, 2009. URL: `https://golang.org/`.

[PTS19] Beltran Borja Fiz Pontiveros, Christof Ferreira Torres, and Radu State. Sluggish Mining: Profiting from the Verifier's Dilemma. page 15, 2019.

[RDF19] Edoardo Rosa, Gabriele D'Angelo, and Stefano Ferretti. Agent-based Simulation of Blockchains. *arXiv:1908.11811 [cs]*, 1094:115–126, 2019. arXiv: 1908.11811. URL: `http://arxiv.org/abs/1908.11811`, `doi:10.1007/978-981-15-1078-6_10`.

[Ros11] Meni Rosenfeld. Analysis of Bitcoin Pooled Mining Reward Systems. *arXiv:1112.4980 [cs]*, December 2011. arXiv: 1112.4980. URL: `http://arxiv.org/abs/1112.4980`.

[SB20] John Slyusarev and Dyma Budorin. KuCoin September 2020 Hack: Hacken Research, October 2020. URL: `https://blog.coinmarketcap.com/2020/10/16/kucoin-september-2020-hack-hacken-research/`.

[SCN+19] Muhammad Saad, Victor Cook, Lan Nguyen, My T. Thai, and Aziz Mohaisen. Partitioning Attacks on Bitcoin: Colliding Space, Time, and Logic. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1175–1187, Dallas, TX, USA, July 2019. IEEE. URL: `https://ieeexplore.ieee.org/document/8884930/`, `doi:10.1109/ICDCS.2019.00119`.

[SGD20] Gudmundur Sigurdsson, Alberto Giaretta, and Nicola Dragoni. Vulnerabilities and Security Breaches in Cryptocurrencies. In Paolo Ciancarini, Manuel Mazzara, Angelo Messina, Alberto Sillitti, and Giancarlo Succi, editors, *Proceedings of 6th International Conference in Software Engineering for*

Defence Applications, volume 925, pages 288–299. Springer International Publishing, Cham, 2020. Series Title: Advances in Intelligent Systems and Computing. URL: `http://link.springer.com/10.1007/978-3-030-14687-0_26`, doi:`10.1007/978-3-030-14687-0_26`.

[SMGC20] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart Contract: Attacks and Protections. *IEEE Access*, 8:24416–24427, 2020. URL: `https://ieeexplore.ieee.org/document/8976179/`, doi:`10.1109/ACCESS.2020.2970495`.

[SSN+19] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and Aziz Mohaisen. Exploring the Attack Surface of Blockchain: A Systematic Overview. *arXiv:1904.03487 [cs]*, April 2019. arXiv: 1904.03487. URL: `http://arxiv.org/abs/1904.03487`.

[SSZ17] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal Selfish Mining Strategies in Bitcoin. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security*, volume 9603, pages 515–532. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017. URL: `http://link.springer.com/10.1007/978-3-662-54970-4_30`, doi:`10.1007/978-3-662-54970-4_30`.

[SZ16] Yonatan Sompolinsky and Aviv Zohar. Bitcoin's Security Model Revisited. *arXiv:1605.09193 [cs]*, May 2016. arXiv: 1605.09193. URL: `http://arxiv.org/abs/1605.09193`.

[TCM+20] Muoi Tran, Inho Choi, Gi Jun Moon, Anh V Vu, and Min Suk Kang. A Stealthier Partitioning Attack against Bitcoin Peer-to-Peer Network. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, page 16, 2020. URL: `https://erebus-attack.comp.nus.edu.sg/erebus-attack.pdf`.

[TR19] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *arXiv:1908.04756 [cs, econ]*, August 2019. arXiv: 1908.04756. URL: `http://arxiv.org/abs/1908.04756`.

[VnE18] VnExpress. Vietnamese investors fall for $650 mln cryptocurrency scam - VnExpress International, April 2018. URL: `https://e.vnexpress.net/news/business/vietnamese-investors-fall-for-650-mln-cryptocurrency-scam-3735380.html`.

[Vya14] Chinmay A Vyas. Security Concerns and Issues for Bitcoin. *International Journal of Computer Applications*, page 3, 2014.

[Whi19] Preston White. K. Preston White Jr. Technical report, August 2019. type: dataset. URL: `http://pubsonline.informs.org/do/`

10.1287/a04f47ed-9a5e-492a-b34a-1dfac3366735/abs/, doi:10.1287/a04f47ed-9a5e-492a-b34a-1dfac3366735.

[Woo14] Dr Gavin Wood. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. page 39, 2014.

[wpx18] wpx_blogspri. Parity Multi-sig wallets funds frozen (explained) - Springworks Blog, 2018. URL: https://www.springworks.in/blog/parity-multi-sig-wallets-funds-frozen-explained/.

[YSZ22] Aviv Yaish, Gilad Stern, and Aviv Zohar. Uncle Maker: (Time)Stamping Out The Competition in Ethereum. page 66, July 2022.

[YWL+19] Xiaochun Yun, Weiping Wen, Bo Lang, Hanbing Yan, Li Ding, Jia Li, and Yu Zhou, editors. *Cyber Security: 15th International Annual Conference, CNCERT 2018, Beijing, China, August 14–16, 2018, Revised Selected Papers*, volume 970 of *Communications in Computer and Information Science*. Springer Singapore, Singapore, 2019. URL: http://link.springer.com/10.1007/978-981-13-6621-5, doi:10.1007/978-981-13-6621-5.

[Ła17] Małgorzata Łatuszyńska. Web-Based Tools for System Dynamics Simulation. *Foundations of Management*, 9(1):287–298, October 2017. URL: https://www.sciendo.com/article/10.1515/fman-2017-0022, doi:10.1515/fman-2017-0022.

APPENDIX A

Appendix

A.1 Successful Attacks against Blockchain Systems

According to Alkhalifah et al. [ANC+19], the total estimated amount of monetary loss due to blockchain incidents beginning in the year 2011 up until the first half of the year 2019 was a little over US$ 3 billion. Chia et al. claim that the value of losses from 2011 until the end of 2018 is as high as US$ 3.55 billion[CHH+19]. According to different sources [CD20, Ale19, Cip21] – that all leave out the OneCoin scam worth an estimated US$ 4 billion [Har19] – the monetary loss of 2019 is about US$ 4.5 billion. For 2020 CipherTrace released a report stating a loss of US$ 1.9 billion [Cip21]. Adding the most recent incidents of 2019 and 2020 to the list, the amount of money lost up to the end of 2020 rises to about US$ 10 billion or even US$ 14 billion, including the estimation of the OneCoin scam.

However, it is tough to keep track of all the incidents that appear in the wild and many sources suggest diverging amounts for the same timespans. Although [CHH+19] claim to have created an incident database, this is not published at the time of writing this thesis. Some of the most complete databases for blockchain incidents and attacks are (i) *Blockchain Graveyard*, containing an overview of well-known attacks [McG] and (ii) *howmuch.net*, providing the same services as Blockchain Graveyard but additionally including good visualizations [Inc21]. Despite the two collections being well structured, there are attacks missing in there too, especially the most recent ones. Regarding scams, there is a website that collects all known occurrences named CryptoScamDB [Cry21].

For now, this thesis only investigated attacks that are known in the context of blockchain or, more precisely, attacks that have been described in theoretical research. In the appendix, we also want to present attacks that have been observed in a real-world scenario before.

A.1.1 The First Known Attack

On the 15th of August in 2010, the first known attack against a public blockchain was executed. The attacker exploited an integer overflow in the transaction check of Bitcoin, which allowed the attacker to create (mint) about 184 billion Bitcoins, although the maximum supply of Bitcoin is fixed at 21 million. The vulnerability was fixed within a few hours, the mining community forked the chain, so the Bitcoins never existed and nobody did come to grief [oSaT12, Com10]. This attack can be classified as a blockchain structure attack.

A.1.2 The First Known Attack With Monetary Loss

The first reported case of an incident leading to a significant loss is the case of user allinvain of bitcointalk.org, who stated that 25 thousand Bitcoins – worth around US$ 500 000 back then – had been stolen from his wallet by compromising his Windows computer [Com11]. This attack fits into the DevOpSec category.

A.1.3 Top Ten Blockchain Incidents

This section shows a list of blockchain system attacks, sorted descending by the amount of monetary loss.

#1 OneCoin - US$ 4 Billion

Most probably the biggest fraud in the history of blockchain – an estimated US$ 4 billion – was caused by OneCoin. It is only an assumption because it is not very clear to this point how many of the funds collected by OneCoin are actually lost, and therefore the scam is not yet represented in the overall statistics [Har19, Cip21, BBC19]. Launched in 2014 by Ruja Ignatova and Sebastion Greenwood, OneCoin attracted investors all over the world with their "fixed and finite" blockchain solution, but actually, there was no development of a blockchain involved at all [Byr17]. Although it was widely supposed that there was no product, it finally came out when in October 2016, a recruiting firm contacted a Norwegian developer named Bjorn Bjercke with a luring promise of a high salary for developing a blockchain solution by using OneCoin's SQL servers. This alarmed Bjercke, and after he saw the OneCoin promoting itself as "better than blockchain" a few months later, he went public with his insider knowledge, which increased the pressure on OneCoin and led to fraud investigations [Byr17, BBC19]. Ignatova disappeared in late 2017 and with her a lot of money. The case is not yet fully investigated, so it remains thrilling [BBC19]. This attack can be placed in the DevOpSec category.

#2 PlusToken - US$ 3 Billion

Based in South Korea – some sources also say China – PlusToken was founded by Chen Bo in 2018 as a cryptocurrency wallet and exchange implementation. Its assumed damage is roughly US$ 3 billion, caused by a massive Ponzi scheme that promised its users high interest rates for buying the associated token generated by "exchange profit, mining income, and referral benefits" [Har19, Kaa20, Inc20]. In June/July 2020, the Chinese government terminated the scam by arresting most of the PlusToken leaders [Kaa20]. Again, this attack belongs to the DevOpSec category.

#3 iFan & Pincoin - US$ 660 Million

Another scam and Ponzi scheme that leaked out in 2018 in Vietnam is called iFan and Pincoin. The company sold its promise of high-interest payment of up to 48% with iFan, giving fans of artists a new possibility to connect with their idols and pay for, e.g., their songs. When in late 2017 and early 2018, the investors were unable to withdraw their earnings, the scam was unveiled [VnE18, ANC+19]. The category DevOpSec is once again the most suitable.

#4 Coincheck - US$ 530 Million

The first incident in this list that is considered a hack is from January 2018, when Japanese-based exchange and wallet service Coincheck was deprived of the cryptocurrency equivalent of about US$ 530 million. According to executives, the hack was caused by an error the developers made when implementing the NEM multi-sig contract, allowing the attackers to steal the private key of the hot wallet and all the NEW coins within [ANC+19, McG]. As this hack is based on implementation and detection failure, this attack is again related to the DevOpSec category.

#5 Mt. Gox - US$ 480 Million

Mt. Gox, a Japanese cryptocurrency exchange, had to file for insolvency after its third and biggest hack occurred in February 2014. After the first two hacks happened in 2011, which caused a damage of roughly US$ 17 million, the third and last one came together with a monetary loss of US$ 480 million [ANC+19]. The attackers used a transaction malleability attack that exploited the lack of security control and management in Mt. Gox's software development. To execute this attack, the attackers had to withdraw their funds from the exchange and publish the same transaction with a different ID right after using Bitcoin's transaction malleability flaw, where a transaction's ID could be altered without invalidating the transaction. Because Mt. Gox just used the transaction ID to check if a withdrawal was successful, and the attackers modified it, for Mt. Gox, the withdrawal was never successful and the attackers could withdraw a large amount of about 744 000 Bitcoins [ANC+19, McG]. Due to its dual threat, the attack belongs to the categories DevOpSec and P2P system.

#6 GainBitcoin - US$ 300 Million

GainBitcoin, a Bitcoin mining and trading platform founded in 2013 by Amit and Vivek Bhardwaj was accused of being a scam and Ponzi scheme in 2018 [Par18, ANC$^+$19]. The estimated money that investors lost with GainBitcoin is US$ 300 million [ANC$^+$19, Inc21]. GainBitcoin promised its investors to pay an interest rate of guaranteed 10 percent for 18 months. Also, the company was blamed for manipulating the price of their related token MCAP [Par18]. This attack can be classified as a DevOpSec attack.

#7 Kucoin - US$ 281 Million

In late September 2020, the Asian cryptocurrency exchange Kucoin lost Bitcoin, Bitcoin-SV, Litecoin, Ether, USD Tether and some other tokens with a countervalue of roughly US$ 281 million [Har21, SB20]. The hot wallets of the exchange were compromised, but there is no further information available. Kucoin claimed they could recover about US$ 64 million and the rest would be covered by their insurance fund, so the clients stay damage-free [SB20, Har21]. The hack belongs to the DevOpSec category.

#8 Bitgrail - US$ 187 Million

The attack against the Italian cryptocurrency exchange Bitgrail in February 2018 was worth roughly US$ 187 million [ANC$^+$19, Hat18]. Somehow the attackers were able to gain full control of a wallet and steal 17 million NANO tokens. As Bitgrail could not afford to pay its clients, bankruptcy was sentenced by a court [McG]. Although the exact reason for the theft is unknown, the attack will fit perfectly into the DevOpSec category.

#9 Parity - US$ 152 Million

A bug in the Parity Ethereum client led to about US$ 152 million of frozen funds [ANC$^+$19, Azi19]. The problem was that the multi-sig wallets – that are actually Ethereum smart contracts – all depended on another smart contract, which acted as a library. The bug was located in the library that was not properly initialized, so a user could make himself the owner of the library and afterwards kill the contract, which makes it unusable. Since the wallets depended on that particular contract, all the funds were frozen forever [wpx18]. Because of the handling error and the smart contract nature of it, this incident fits into the attack categories DevOpSec and blockchain application.

#10 Compound - US$ 90 Million

In November 2020, the Distributed Finance (DeFi) platform Compound lost an estimated amount of US$ 90 million because of errors on the side of the price oracle provided by Coinbase [Fou21a, Gog20]. The Coinbase price oracle provided incorrect data for the Ethereum-based stablecoin DAI, which led to a large number of liquidations on the DeFi platform because of under-collateralization [Gog20]. This incident is a bit harder to categorize, but the fact that a decentralized platform uses a price oracle of a centralized company makes it fall into the DevOpSec attack category.

A.2 Simulator Configuration Parameters

Listing A.1: config.yml

```
seed: 1
outPath: "../out"
useMetrics: true
usePprof: false
printLogToConsole: true
printAuditLogToConsole: false
printMemStats: true
endTime: 172800000000000 # nanos
nodeCount: 200 # overall node count in the simulation
simulateTransactionCreation: false
checkPastTxWhenVerifyingState: false # disable for speedup if no invalid TX happen
auditLogTxMessages: true
txPerMin: 1000
noneNodeUsers: 100 # these (and all normal nodes) are the originators of txs
maxUncleDist: 7
bombDelay: 9000000 # inactive as difficulty adjustment is not implemented
overallHashPower: 863020000
miningPoolsHashPower: [255060000, 149980000, 90850000, 69130000, 52640000, 34590000,
                       33110000, 30600000, 21240000, 17230000, 14270000, 14090000,
                       11630000, 10020000, 6850000, 6460000, 6460000, 5290000,
                       4790000, 3850000, 3370000, 3050000, 2650000, 1930000,
                       1770000, 1660000, 1610000, 1510000, 1430000, 1290000, 966690,
                       947590, 652980, 419190, 342780, 227520, 216540, 132350, 80410,
                       54750, 48330, 43450, 42240, 41090, 40990, 36390, 33960, 20790,
                       14870, 14550, 9800, 9650, 5700, 1750, 1280, 1120, 825, 613,
                       443] # in MH/s
miningPoolsCpuPower: [4450, 4300, 4400, 3900, 4150, 4050, 4200, 3800, 4500, 3600, 3850,
                      4450, 4300, 4400, 3900, 4150, 4050, 4200, 3800, 4500, 3600, 3850,
                      4450, 4300, 4400, 3900, 4150, 4050, 4200, 3800, 4500, 3600, 3850,
                      4450, 4300, 4400, 3900, 4150, 4050, 4200, 3800, 4500, 3600, 3850,
                      4450, 4300, 4400, 3900, 4150, 4050, 4200, 3800, 4500, 3600, 3850,
                      4450, 4300, 4400, 3900] # in MHz
blockNephewReward: 0.0625 # eth
blockReward: 2 # eth
limits:
  initialGasLimit: 12500000
  minTxGas: 21000
sizes: # bytes
  hash: 42
  tx: 200
  getHeaders: 54
  header: 90
attackerActive: false
attacker:
  type: "verifiersDilemmaForced"
  hashPower: [255060000]
  maxPeers: [75]
  cpuPower: [4400]
  location: ["Ireland"]
  numbers:
    percentOfGasToForceVerifiersDilemma: 0.5 # 0.5 = 50%
    percentOfMaxGasLimitIncrease: 0.0 # 0.5 = 0.5 * parentGasLimit/1024
    specialTxStateComputation: 2280.0 # 10230.0 # 47.61
```

Listing A.2: delays.yml

```
locations:
  Tokio:
    Tokio:
      latency: # ms
        distribution: lognorm
        params:
          - -0.6492601754237427
          - 0.09577031701337546
      sendThroughput: # mbps
        distribution: gamma
        params:
          - 49.94349786281565
          - 0.11859680175780735
      receiveThroughput: # mbps
        distribution: gamma
        params:
          - 49.893438353289
          - 0.11949707026288697
    Ireland:
      latency:
        distribution: norm
        params:
          - 222
          - 0.000000000000005301592619562191
      sendThroughput:
        distribution: gamma
        params:
          - 74.59395106047405
          - 2.395749575793742
      receiveThroughput:
        distribution: gamma
        params:
          - 111.94889901518795
          - 3.59116760253906
    Ohio:
      latency:
        distribution: uniform
        params:
          - 155
          - 157
      sendThroughput:
        distribution: norm
        params:
          - 61.2494357971314
          - 14.78131405107846
      receiveThroughput:
        distribution: norm
        params:
          - 58.33332812489563
          - 14.770520363010117
  Ireland:
    Ireland:
      latency:
        distribution: lognorm
        params:
          - 0.013111341501152469
          - 0.05382936134631561
      sendThroughput:
        distribution: lognorm
        params:
          - 6.077781223913004
          - 0.250891532897607
      receiveThroughput:
        distribution: gamma
        params:
          - 22.156734395988906
          - 0.049076686678163256
    Tokio:
      latency:
        distribution: norm
        params:
```

```
        - 222
        - 0.000000000000005301592619562191
    sendThroughput:
      distribution: gamma
      params:
        - 74.59395106047405
        - 2.395749575793742
    receiveThroughput:
      distribution: gamma
      params:
        - 111.94889901518795
        - 3.59116760253906
  Ohio:
    latency:
      distribution: norm
      params:
        - 84.6732887358493
        - 0.08653429632523468
    sendThroughput:
      distribution: gamma
      params:
        - 9.787573446109892
        - 0.1385155169665812
    receiveThroughput:
      distribution: gamma
      params:
        - 6.565283355712892
        - 0.09
Ohio:
  Ohio:
    latency:
      distribution: lognorm
      params:
        - -0.6417829001164197
        - 0.11782830914498797
    sendThroughput:
      distribution: gamma
      params:
        - 74.77728455126282
        - 0.14955179631710092
    receiveThroughput:
      distribution: gamma
      params:
        - 74.88645660602555
        - 0.14703122690352527
  Tokio:
    latency:
      distribution: uniform
      params:
        - 155
        - 157
    sendThroughput:
      distribution: norm
      params:
        - 61.2494357971314
        - 14.78131405107846
    receiveThroughput:
      distribution: norm
      params:
        - 58.33332812489563
        - 14.770520363010117
  Ireland:
    latency:
      distribution: norm
      params:
        - 84.6732887358493
        - 0.08653429632523468
    sendThroughput:
      distribution: gamma
      params:
        - 9.787573446109892
        - 0.1385155169665812
    receiveThroughput:
        distribution: gamma
        params:
          - 6.565283355712892
          - 0.09
timeBetweenBlocks: # seconds
  distribution: exp # fixed to exp, do not change
  params:
    - 12.91017 # this is the targeted mean block time (including uncles!)
txGas: # tx gas will be a random var out of this + min tx gas (min tx gas is set in config.yml)
  distribution: norm
  params:
    - 21841
    - 32762
gasPrice: # gwei
  distribution: uniform
  params:
    - 91
    - 40
txStateComputation: 10230 # gas per Mhz per s
baseHeaderVerification: 100 # headers per Mhz per s
baseBodyVerification: 100 # bodies per Mhz per s
baseTxVerification: 1000 # TXs per Mhz per s
```