Balance Attack on a Forkable Blockchain

MELTING AVALANCHE

Master Thesis

Marcel Matti Zauder

University Bern

July 21, 2023

Abstract

This master thesis investigates the Balance Attack, an attack technique aimed at exploiting the forkable nature of blockchains, with a specific focus on the Avalanche consensus protocol. The research explores the theoretical aspects of the attack, highlighting former groundwork and the original paper on the Balance Attack, analyzing its potential implications for network integrity and security. Additionally, it provides an in-depth exploration of the Avalanche consensus protocol, a robust and decentralized consensus algorithm designed for blockchain networks. The protocol's key concepts, mechanisms, and algorithms are introduced and explained, shedding light on its inner workings. Furthermore, the Avalanche blockchain, which is built upon the Avalanche consensus protocol, is examined, highlighting its unique features and characteristics. While attempts were made to implement the attack in a practical setting, challenges related to blocking network traffic hindered the full realization of the implementation. Nonetheless, the studied methods are presented and thoroughly examined in this thesis.
Through a comprehensive analysis of the attack's technical aspects and limitations, this research contributes to the understanding and advancement of secure blockchain systems. The findings highlight the importance of robust security measures and offer insights into future research directions for defending against such attacks.

"Through this research, we seek to inspire
Implementing measures that will never tire.
A more secure, resilient blockchain we shall see,
Fostering trust and a thriving crypto decree."

Keywords: Avalanche, Balance Attack, Blockchain, Bitcoin, Ethereum

Prof. Dr. Christian Cachin, Cryptology and Data Security (CRYPTO), Institute of Computer Science, University Bern, Supervisor
Ignacio Amores-Sesar, Cryptology and Data Security (CRYPTO), Institute of Computer Science, University Bern, Assistant

Table of Contents

List of Abbreviations

approx. ..... Approximately

AVM ..... Avalanche Virtual Machine

DAG ..... Directed Acyclic Graph

DApp ..... Decentralized Application

DLT ..... Distributed Ledger Technology

EVM ..... Ethereum Virtual Machine

GHOST ..... Greedy Heaviest Observer Sub-Tree

NFT ..... Non-Fungible Token

PoA ..... Proof-of-Authority

PoS ..... Proof-of-Stake

PoW ..... Proof-of-Work

UFW ..... Uncomplicated Firewall

VF ..... Virtuous Frontier

VM ..... Virtual Machine

Chapter 1

Introduction

Blockchains, cryptocurrencies, decentralized finances: These are subjects that captured significant attention in recent years. The concept of distributed ledger technology was first introduced in 2008 with the advent of Bitcoin by Satoshi Nakamoto [Nak08]. This groundbreaking innovation paved the way for the development of numerous blockchain and consensus protocols, each offering distinct approaches to consensus mechanisms and transaction verification. As the blockchain ecosystem continues to evolve, it is crucial to explore and understand the diverse landscape of blockchain technologies, their underlying principles, and their impact on various industries. The hype surrounding new blockchain technology has driven over 1 trillion US dollars of investment into cryptocurrencies, profoundly transforming the landscape of performing transactions.[1] As the DLT is a rather young field of research, there are still many (un)explored vulnerabilities that have already led to attacks and scams with millions of dollars in damage.

In 2017, Natoli and Gramoli [NG17] introduced a novel attack scheme known as the Balance Attack. This attack leverages the inherent forkable nature of blockchains to either facilitate double-spending attacks or indefinitely fork the entire network. The core principle behind this attack is to partition the network into two distinct subgroups and prevent communication between them. The original paper demonstrates the execution of the Balance Attack on the Ethereum network, albeit requiring a communication blockade of at least 4 minutes to achieve success. Consequently, the attack was deemed highly improbable in real-world scenarios, given the significant challenge of sustaining such a prolonged communication lockdown.

Another Top-20 blockchain in terms of market capitalization is Avalanche which advertises with a "very fast transaction verification and commitment time". With an average time to finality until a transaction can be considered committed of under one second[2] it is many times faster than, for example, Ethereum (4 minutes) or Bitcoin (around 1 hour). Following the security analysis by Amores-Sesar, Cachin, and Tedeschi [ACT22], the hypothesis arose about whether the Balance Attack was feasible on the Avalanche network.

Based on the research papers *"The Balance Attack or Why Forkable Blockchains are Ill-Suited for Consortium"* and *"When is Spring coming? A Security Analysis of Avalanche Consensus"* this master thesis report will handle the following **RESEARCH QUESTION:**

Is the Balance Attack, as described by Natoli and Gramoli, feasible and viable on the Avalanche network?

[1] `https://coinmarketcap.com/` - July 21, 2023.
[2] `https://avascan.info` - July 21, 2023.

Chapter 2

Strategy

Initially, this thesis addresses the fundamental questions regarding the underlying principles and methodologies. Additionally, upcoming limitations and mitigations are discussed, as well as potential opportunities that arise during the setup and execution phase.

2.1 Motivation

As previously discussed in Chapter 1, the cryptocurrency market and its related technologies have attracted significant investments. Consequently, the potential financial gains from a successful attack on blockchain systems can be substantial, with losses reaching millions of dollars. This situation is particularly concerning due to the relative novelty of blockchain technology, which means that vulnerabilities and potential "backdoors" are still being discovered and addressed.
The primary motivation behind this master thesis is to provide a comprehensive theoretical analysis of the Balance Attack within the Avalanche network. Although the successful execution of the Balance Attack may not be feasible in practice, exploring its theoretical aspects is valuable in understanding potential vulnerabilities and improving the overall security of blockchain systems. By highlighting the importance of implementing robust security measures and precautions, this research aims to foster a more secure and resilient blockchain ecosystem.

2.2 Methodology

The initial step involves a thorough review of published papers and scientific works on the Balance Attack, allowing for a deep understanding of its requirements and implications. Additionally, the implementation of the Avalanche consensus protocol is studied to gain insights into the inner workings of this blockchain system.
To evaluate the potential effects of the Balance Attack on the Avalanche network, a local instance is set up. The rationale behind the chosen setup is provided, along with a description of important commands to initiate the network. Through this local instance running on a virtual machine, various performance studies and investigations are conducted to explore the hypothetical impact of the Balance Attack.

The analysis focuses on assessing the possible consequences and the potential vulnerabilities that could arise in the Avalanche network. The evaluation considers factors such as the number of malicious nodes in the network and the presence of faulty transactions. By gathering and interpreting the results of these analyses, the study aims to provide insights into the overall security and resilience of the Avalanche network in the face of potential attacks.
Despite the inability to execute the Balance Attack, this master thesis aims to contribute to the understanding of its theoretical implications on the Avalanche network. It provides a comprehensive evaluation of the potential risks and vulnerabilities, thereby identifying opportunities for enhancing the security measures and strengthening the resilience of the Avalanche protocol.

2.3 Limitations, Mitigations, and Opportunities

The Limitations, Mitigations, and Opportunities Section provides an analysis of the constraints, potential remedies, and avenues for future research within the context of the conducted study. It delves into the encountered limitations, proposes potential mitigations to address these limitations, and highlights the opportunities that emerge from the findings. By exploring the limitations, considering strategies to mitigate them, and identifying areas for further exploration, a comprehensive assessment of the research scope and potential implications is presented.

2.3.1 Limitations

Whereas having full control over the Avalanche network, including the source code, which would provide extensive customization capabilities, it may not be feasible within the scope of this study due to resource and expertise limitations. Furthermore, the goal is to create an emulation environment that closely resembles the actual Avalanche network, achieving complete accuracy can be challenging. Variations in network conditions, node behaviors, and external factors may introduce deviations from the real-world network. To maintain a realistic representation, the source code should remain as close as possible to the original implementation. This limitation restricts the level of modifications that can be made, except for implementing adversary and malicious nodes.
The Kollaps tool faced permission problems that rendered it unusable in the specific circumstances of this research. This limitation prevented the utilization of Kollaps for network emulation and required alternative approaches to be explored.

2.3.2 Mitigations

Despite the limitations, other tools and techniques can be explored to compensate for the challenges faced. Alternative network emulation tools or custom implementations can be considered to achieve the desired level of control and accuracy over

the Avalanche network.
To mitigate the limitations related to source code modifications, careful design of test scenarios and configurations can be employed. By focusing on creating realistic conditions within the constraints of the original code, meaningful insights can still be obtained regarding the behavior and security of the Avalanche network.

2.3.3 Opportunities

The limitations encountered provide opportunities for future research and development. Exploring and improving network emulation tools, like Kollaps, to address permission problems and enhance customizability can enable more comprehensive experiments and evaluations of blockchain network behaviors.
Despite the limitations, the study conducted using the available resources and tools still provides valuable insights into the security and resilience of the Avalanche network. The findings contribute to the understanding of potential vulnerabilities and the development of robust countermeasures in real-world blockchain deployments.

Chapter 3

Preliminaries

A thorough exploration of the fundamental concepts that form the backbone of blockchains and consensus protocols is presented. It commences with an overview of blockchains, emphasizing their decentralized nature and robust security measures.
The focus then shifts to the *Avalanche* blockchain, which boasts a distinctive architecture and innovative consensus protocols inspired by gossip algorithms. The chapter delves into the intricacies of the Avalanche transaction validation framework, which employs a partially ordered model to enhance efficiency and scalability.
Furthermore, this chapter delves into the exploration of the *Balance Attack*, a cunning strategy that takes advantage of the forkable characteristics present in widely recognized blockchains like Bitcoin and Ethereum. Through an in-depth analysis of the Proof-of-Concept project conducted by Natoli and Gramoli, the chapter sheds light on the vulnerabilities that the Balance Attack exposes.
By examining these critical subjects, readers shall gain valuable insights into the intricate dynamics among blockchains, consensus mechanisms, and potential security risks.

3.1 Blockchain - Distributed Ledger Technology

The blockchain, a type of Distributed Ledger Technology (DLT), enables multiple users to collectively maintain a synchronized and shared database. A defining feature of this technology is its decentralized nature, eliminating the need for a central authority to verify transactions and other data recorded on the blockchain. Instead, participants in the network collaboratively validate and maintain the integrity of the system.
Overall, blockchain technology provides a secure and transparent platform for recording and verifying transactions. Its decentralized and immutable nature makes it particularly suitable for applications requiring trust, transparency, and resilience to censorship or unauthorized changes.

3.1.1 Key Components

Subsequently, an overview of the structural components of a blockchain is presented along with brief explanations of their significance. These components form the foundation of blockchain technology and are essential to its decentralized and secure nature.

Distributed Ledger
The distributed ledger is the fundamental database of a blockchain. Each network node maintains a complete and synchronized copy of the ledger, ensuring its accuracy and consistency. With its transparency and immutability, the ledger enables public visibility and provides a guarantee of data integrity.

Blocks
Transaction and data within a blockchain are organized into blocks. Each block is assigned a unique identifier called a hash, which helps maintain data integrity and security. What makes the structure intriguing is the linkage between blocks. By containing a reference to the previous block, a tree-like structure is effectively formed, allowing for a chronological order of transactions, thereby ensuring total order, and guaranteeing the consistency and immutability of the blockchain.
In most blockchains, like BitCoin or Ethereum, nodes agree upon a main chain within the tree, however, this must not be the case for all blockchains, as it was the case for Avalanche (Section 3.2).

Consensus Mechanism (Section 3.1.2)
In an untrustworthy environment, a consensus mechanism is employed to achieve agreement among all nodes. Consensus algorithms such as Proof-of-Work or Proof-of-Stake are utilized to validate the next block in the blockchain. These algorithms ensure that nodes agree on the state of the blockchain and maintain its integrity.

Cryptography
Cryptographic algorithms like hash functions, digital signatures, and encryption techniques are employed to ensure integrity, authenticity, and tamper resistance in blockchain technology.

Peer-to-Peer Network
Blockchains are commonly established on a peer-to-peer network, where all nodes possess equal rights and responsibilities.

3.1.2 Consensus and Consensus Mechanism [CGR11]

In an environment fraught with uncertainty and distrust, the deployment of a robust consensus mechanism stands as a critical foundation for agreeing upon a common value among the distributed nodes. Embracing sophisticated consensus algorithms like Proof-of-Work or Proof-of-Stake, the validation of subsequent blocks within the blockchain becomes a well-defined process. These meticulously designed algorithms promote agreement among nodes, ensuring a consensus on the blockchain's state and safeguarding its integrity.

Proof-of-Work Consensus Protocol

The most common and widely adopted consensus protocols in blockchain networks are Proof-of-Work-based (*PoW*) ones. During the validation process participants, also known as miners, compete to solve complex mathematical puzzles, hash functions, in order to validate the next block - thus adding new blocks to the blockchain. The miners invest resources in solving these mathematical complex puzzles, and the first to find the solution is rewarded with newly minted cryptocurrency tokens. PoW is renowned for its security and resilience against malicious attacks, as altering a block's content would require an immense amount of computational power. However, this consensus protocol requires a significant amount of computational power, hence sparking concerns regarding its environmental impact due to its energy-intensive nature.[1] As blockchain technology continues to evolve, alternative consensus mechanisms, like Proof-of-Stake, are being explored to address these challenges.

Proof-of-Stake Consensus Protocol

Proof-of-Stake-based (*PoS*-based) consensus protocols are utilized in blockchain networks as an alternative to Proof-of-Work. Participants of the blockchain network can "stake" a number of cryptocurrency tokens they hold as collateral in order to become a block validator. These can then create new blocks and validate transactions based on their stake, with higher stakes resulting in a greater chance of being chosen. Therefore, PoS eliminates the need for resource-intensive mining activities and instead relies on a more efficient and environmentally friendly approach. Nevertheless, concerns persist regarding centralization and the potential for malicious actors with substantial stakes to launch attacks, making them crucial factors to consider during the design and implementation of PoS-based blockchain systems.
The aforementioned Avalanche blockchain deploys the Snow consensus protocol. These types of consensus protocols implement a kind of PoS approach, as the owners of validator nodes need to stake their tokens. However, they are not built upon the traditional PoS approach as they rely on gossip algorithms to agree upon a common state.

3.1.3 Security and Trust

The remarkable potential of blockchain in transforming data integrity and trust has sparked significant attention toward its robust security features.
Immutability represents a prominent aspect of blockchain, ensuring that once data is recorded on the blockchain, it becomes unalterable. Each block in the chain contains a unique identifier, generated by a cryptographic hash function. This identifier relies on the data within the block as well as the hash of the preceding block. Therefore,

[1] see https://www.coindesk.com/policy/2022/04/21/sweden-eu-discussed-bitcoin-proof-of-work-ban-report/.

any tampering with the data would lead to a mismatch in the hash, immediately alerting the network to the manipulation attempt.
Moreover, blockchains use further advanced cryptographic mechanisms to ensure the confidentiality, integrity, and authenticity of data. For instance, the utilization of pulbic-key cryptography provides security for transactions and ensures the verification of participants' identities.
Additionally, the decentralized nature of blockchain, with its distributed network of nodes, enhances security by eliminating single points of failure and reducing the susceptibility to attacks.
Collectively, these security features render the blockchain an inherently robust and tamper-resistant technology. These attributes instill confidence and trust in a wide array of applications, spanning from financial transactions to supply chain management and beyond.

3.1.4 Scalability

Scalability and performance pose significant challenges for blockchain systems as the user base and transaction volumes are steadily increasing.
The primary obstacle is the inherent trade-off between decentralization, security, and scalability - the so-called "Blockchain-Trilemma".[2] In traditional blockchains like Bitcoin, the obligation for each node to validate and store every transaction leads to limited throughput capacity, causing congestion, increased transaction processing times, and diminished system performance.
To overcome these challenges, a multitude of scaling solutions have emerged:

- Layer 2 protocols, such as the "Lightning Network" [PD16] facilitate off-chain transactions, alleviating the load on the primary blockchain. However, protocols of this nature compromise decentralization and security to enhance speed and users may be required to use multiple accounts therefore needing to stake more money in order to be able to take part in the validation process.
- Sharding [WSNH19] divides the blockchain into smaller partitions, or shards, to process transactions in parallel. This approach introduces complexity to the system, necessitating a meticulous implementation of an appropriate sharded database architecture. Otherwise, flawed implementations can readily result in performance degradation and data loss.
- Innovative consensus algorithms, such as Directed Acyclic Graphs (utilized in Avalanche) and Byzantine Fault Tolerance (BFT), bolster scalability while upholding stringent security measures.

As blockchain technology continues to evolve, tackling scalability, security, and performance concerns remains a vital area of focus in order to fully unlock the potential of blockchain across diverse sectors, including finance, supply chain management, and decentralized applications.

[2] first introduced in 2014 by Ethereum co-founder Vitalik Buterin - `https://coinmarketcap.com/alexandria/glossary/blockchain-trilemma`.

3.2 Avalanche[3]

Avalanche distinguishes itself from other blockchain protocols, such as Bitcoin or Ethereum, by offering exceptional scalability and a favorable ratio of throughput to energy consumption. As of July 21, 2023, its AVAX token is ranked 17th in terms of market capitalization worldwide, with a value of over 4.5 billion US dollars.[4]
The underlying consensus protocol of Avalanche belongs to the Snow family, which achieves consensus through iterative random sampling of staked validator nodes within the network. This unique approach enables a remarkably short block creation time of less than 1 second and a potential throughput of up to 4′500 transactions per second. As of July 21, 2023, the current block creation time is approx. 2 seconds with a daily transaction throughput of 2 million transactions.[5] Furthermore, Avalanche adopts a directed acyclic graph (DAG)[6] structure instead of a traditional chain, enabling parallel processing and significantly boosting the potential throughput. This innovative approach addresses key challenges in traditional blockchains, such as scalability and throughput, while maintaining robust security guarantees and decentralized principles.
To establish consensus, Avalanche adopts a Proof-of-Stake (PoS) mechanism, where individuals can become validators by staking a minimum of 2′000 AVAX tokens (equivalent to approximately 28′000 US dollars). The *Primary Network* currently boasts over 1′200 active validators, as of the latest available data.[7]

3.2.1 Architecture

Avalanche is built up of three separate blockchains: **Exchange** Chain (**X**-Chain), **Platform** Chain (**P**-Chain), and **Contract** Chain (**C**-Chain). These are validated and secured by the *Primary Network*, which is the overlaying network containing all members.
The C-Chain handles the creation and execution of EVM Contracts and other Smart Contracts and also functions as an enabler for using NFTs, DApps, and the use of other tokens, like the ERC20 Ethereum token, in the Avalanche network. As this blockchain is an instance of the EVM it additionally enables the use of the Solidity programming language, native to Ethereum.
The P-Chain is used to coordinate validators and controls the creation, deletion, and maintenance of platform primitives, like subnets, which are separate and distinct blockchain networks operating within the larger Avalanche platform containing their own set of validators, consensus rules, and state.
Both the Contract and Platform Chain are implementing the *Snowman* consensus protocol which is different from the Avalanche consensus protocol as it provides total order. The process of the *Snowman* consensus protocol is not explained further

[3] retrieved from `https://docs.avax.network/`.
[4] `https://coinmarketcap.com/`.
[5] `https://avascan.info/` - July 21, 2023.
[6] This specific choice has been deprecated in the Avalanche version v1.10.0 on April 4, 2023.
[7] see Footnote 5.

in this master thesis.
The X-Chain administers the creation and trade of digital assets and governs all transactions according to a set of predefined rules. It is an instance of the Avalanche Virtual Machine (AVM) hence implementing the Avalanche consensus protocol (Section 3.2.2).

3.2.2 Consensus Protocol [TYS+19]

In order to achieve transaction validation and consensus on a common state of the blockchain, Avalanche employs the *Snow consensus protocol*, which is inspired by gossip algorithms. Unlike traditional blockchains that often provide a total order of transactions, this consensus protocol in Avalanche only partially orders transactions. However, the causality of transactions is still maintained. When a node, such as node u in this example, observes a new transaction T, it sends queries to a set of k nodes asking if they also know about transaction T (Red arrows in Figure 3.1). Each node responds with its *"Virtuous Frontier"* (VF(i)), which is a randomly selected set of non-conflicting leaf nodes that the node is aware of. Upon receiving the response, node U checks if transaction T is included in the VF. For simplicity reasons, $T \in \text{VF}(i) \vee \exists d \in desc(T)$[8] $: d \in \text{VF}(i)$ is considered to be a value of **1** and $T \notin \text{VF}(i) \wedge \nexists d \in desc(T) : d \in \text{VF}(i)$ as **0**. If the number of successful responses exceeds a predefined threshold value α, the counter corresponding to that transaction is set to 1. Additionally, the counters of its ancestors are incremented by 1. For example, when ctr_A is increased, ctr_B is also increased. If the check is unsuccessful, meaning $\sum_i (T \in \text{VF}(i)) < \alpha$, the counter is reset to 0. Once the counter exceeds the threshold value β_1, and T is the only transaction in its conflict set (high-

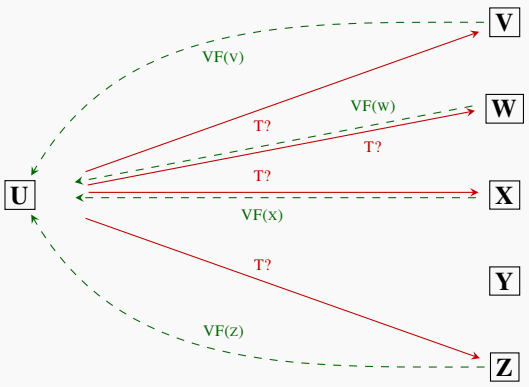

Figure 3.1
Transaction Verification in Avalanche

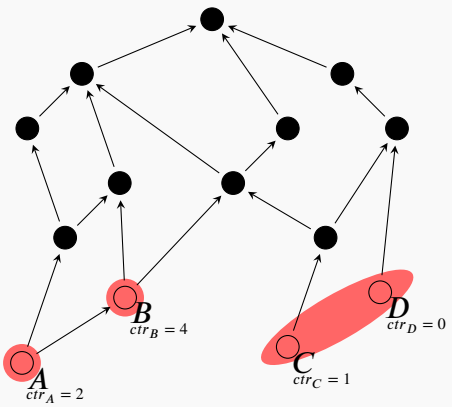

Figure 3.2
DAG in an Avalanche Network

[8] desc(T) contains each descendant of T

lighted in light red in Figure 3.2), T is committed. This applies to transactions A and B in the diagram. Otherwise, if T is part of any conflict set (e.g., nodes C or D), it is confirmed once its counter exceeds the threshold value of β_2. It is important to note that only one transaction can be committed from a conflict set. Therefore, if transaction C is committed, transaction D can never be. Conflict sets can also overlap, while the rule of allowing only one committed transaction per conflict set still applies. Transactions that are not part of a conflict set with a previously committed transaction can be re-polled at any time, provided that no new transaction or no-op transaction is present. The process of re-polling is triggered when the primary voting process fails to achieve a strong consensus on a particular transaction set. It allows the network to continuously try to reach a definitive consensus on conflicting transactions by gathering new votes regarding these specific transactions that were in conflict.
A detailed pseudo-code implementation of the algorithm can be found in Appendix A.

3.3 The Balance Attack [NG17]

Due to miners working concurrently on creating the next block, different nodes can contain varying states of the blockchain. A fork occurs when nodes fail to reach a consensus on the common state of the blockchain. In order to resolve such divergences and establish a unified blockchain state, several algorithms have been introduced. This chapter introduces two of these algorithms (Appendix B).

1. *Nakamoto's approach (Bitcoin)* (Appendix B, Algorithm B.1)
   In Bitcoin, the selection of the longest branch as the continuation of the blockchain results in the discarding of numerous blocks that do not belong to that specific branch. This approach, however, leads to significant wastage of blocks and consequently results in noteworthy energy inefficiency.
2. GHOST *protocol (Ethereum)* (Appendix B, Algorithm B.2)
   Nodes in pre-PoS Ethereum (before September 2022) utilized a modified version of the GHOST protocol to consistently choose the root of the heaviest subtree, ensuring the construction of a unified branch. This approach mitigates wastefulness by increasing the likelihood of selecting the common ancestor when sibling blocks are present. In contrast to the algorithm employed in the Bitcoin network, Ethereum's approach is less wasteful.

In Figure 3.4 the difference in choice is depicted. Nakamoto's consensus protocol would decide upon the grey branch, discarding the heavier left tree, whereas the GHOST protocol chooses the blue branch to be the continuation of the main chain.

In 2017, Natoli and Gramoli introduced the Balance Attack concept, utilizing the Ethereum network and its variant of the GHOST algorithm as an example to exploit the vulnerabilities of a forkable blockchain. The underlying idea involves manipulating and delaying the communication between two (or more) subgroups of nodes. The subgroups are chosen to have similar mining power so that the block generation time is approx. equal

between these. Despite blocking communication between the subgroups, the adversary retains the ability to interact with all nodes across the subgroups. Consequently, the adversary can broadcast one transaction, i.e. sending money to himself (T_1), to one subgroup and another transaction, i.e. buying something (T_2) with the same funds, to the other subgroup, therefore performing a double spend. Depending on which consensus protocol (Bitcoin or GHOST) is used the Balance Attack is performed differently:

3.3.1 Exploiting Forkable Blockchain with Nakamoto Consensus

Given the block generation time of Bitcoin of approx. 10 minutes, the occurrence of forks is rather infrequent, hence in Figure 3.3 possible forks within the different subchains can be disregarded. After disabling communication and broadcasting both transactions to their respective subgroups, the adversary is helping one group that received the preferred transaction, in this example subgroup SG_1, in mining. For a block to be considered committed, it must be at least six blocks deep within its chain.

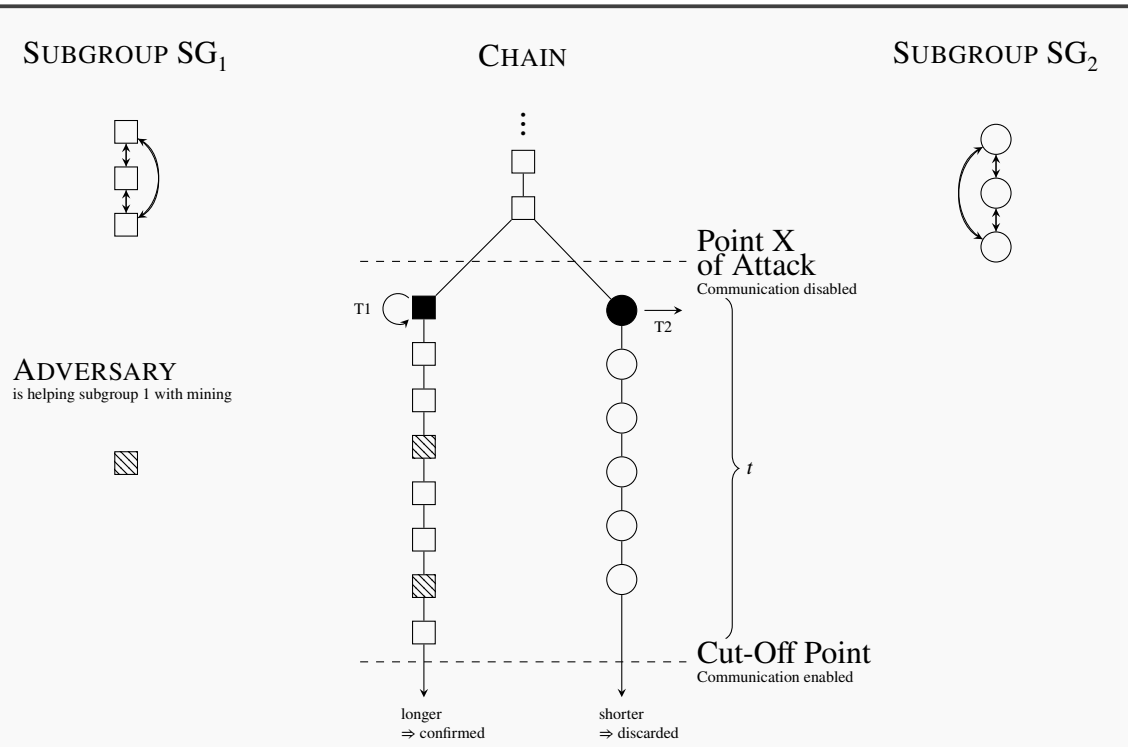

Figure 3.3: Balance Attack Scheme (BITCOIN)

The adversary blocks the communication between 2 subgroups of equal hash power. By introducing conflicting transactions to the subgroups, a fork is created intentionally. The adversary influences the decision-making of the fork-resolving algorithm by assisting one subgroup in mining its respective branch. Once both conflicting transactions can be considered committed the communication is restored and the longer branch, which benefited from the adversary's mining assistance, is recognized as the continuation of the chain.

Therefore, the communication between both subgroups needs to be delayed for a minimum of two hours to ensure the commitment of both blocks containing the respective transactions (T_1 and T_2).
Having more hash power, as the adversary supports SG_1, that subchain will eventually become longer than the chain of SG_2. Once transactions T_1 and T_2 are considered committed within their respective chains, the adversary can inform the third party involved in transaction T_2 that the transaction has been successfully validated and executed. However, upon reenabling communication between both subgroups, the consensus protocol will select the longer chain of SG_1 as the continuation of the main chain. As a result, the chain of SG_2, along with transaction T_2, is discarded. Therefore, the adversary gains the benefit of the transaction without having to pay for it in the end.

3.3.2 Exploiting Forkable Blockchain with GHOST Consensus

In Ethereum, the block generation time is significantly shorter compared to Bitcoin, typically ranging from 10 to 20 seconds. As a result, the occurrence of forks is much more common. As previously mentioned the Bitcoin approach to resolving forks becomes energy-inefficient when multiple forks occur. Therefore, Ethereum implemented a variant of the GHOST protocol [SZ15] that considers the number of descendants as a criterion for selecting the next block in the chain. This means that newly mined blocks influence the probability of their ancestors being chosen.
Similar to the Balance Attack in a Bitcoin network, the adversary assists one subgroup with mining to make the particular subtree "heavier", thereby influencing the consensus protocol to choose the preferred block to be the next one in the chain. However, another more intricate and impactful approach to performing the Balance Attack exists, resulting in the fork being unable to resolve [BKT+19]:
After broadcasting the transactions and disabling communication, the adversary acquires a copy of both subtrees and starts mining locally on two separate chains: one for subgroup SG_1 and one for subgroup SG_2. However, these newly mined blocks are not released yet. With both subgroups having an equal amount of hash power, each subtree grows at an equal pace, resulting in both roots of the respective subtrees being approximately equally weighted. Shortly before enabling communication, the adversary makes sure that both subtrees have the same number of blocks. If there is a difference in weight, the adversary releases additional blocks from its own "inventory" to balance the subtrees. For instance, in Figure 3.4, the block containing T_2 has one fewer descendant than the block with T_1. In this case, the adversary would release a pre-mined descendant of the block with T_2 (Red areas) to equalize the weights of the subtrees before enabling communication. With communication enabled, the consensus protocol attempts to select one branch of the fork. However, since both subtrees have equal weight, the protocol cannot reach a decision, resulting in the persistence of the fork. The adversary ensures that the weight of both subtrees remains equal by releasing additional blocks whenever there is a discrepancy. This prevents

the fork from being resolved, as well as successfully performing a double-spending attack as both T_1 and T_2 are eventually committed.

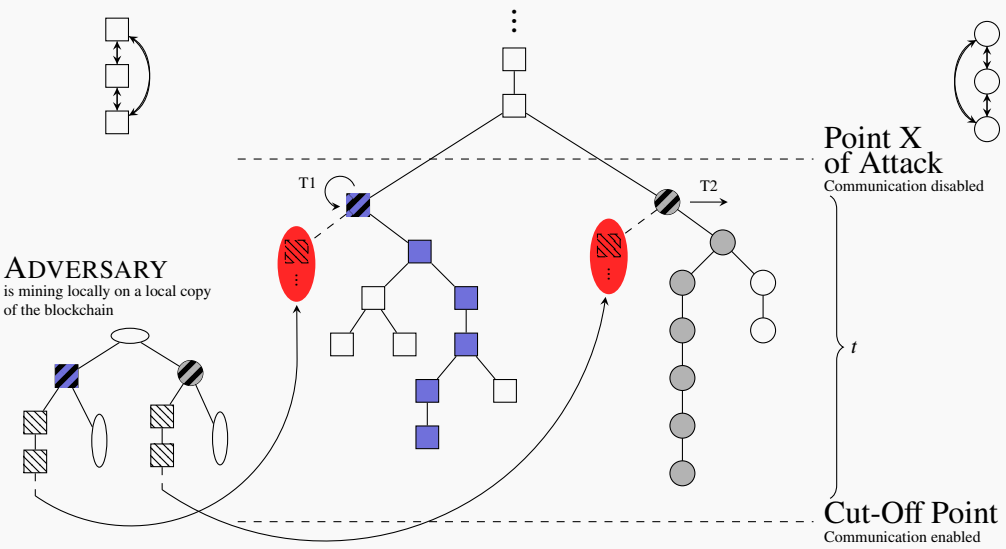

Figure 3.4: Balance Attack Scheme (ETHEREUM/GHOST)

Similar to the previous example, the creation of a fork is deliberately forced. However, in this case, the adversary obtains copies of both forks and continues local mining without releasing blocks. Before reenabling communication, the adversary ensures that both forks are equally weighted. If the weights are unequal, pre-mined blocks are released to equalize the subtrees. By strategically releasing blocks, the fork can persist indefinitely. The highlighted sections in blue and gray demonstrate the difference in chain-continuation selection. Nakamoto's algorithm favors the gray chain as the continuation of the main chain, while the GHOST protocol selects the blue chain.

Additionally, in contrast to Bitcoin, there is no need for prolonged communication blocking. In Ethereum, for a transaction to be deemed committed, the block containing it only requires a minimum of 11 descendants. With the shorter block generation time in Ethereum, a transaction can typically be considered committed after approximately 4 minutes.

Chapter 4

Setup of the Avalanche Network

To execute the Balance Attack on the Avalanche network, a local instance is established on a university server. The developers offer two primary methods for creating such an instance. However, it is important to note the limitations associated with each approach. In the subsequent sections, these limitations will be highlighted, and a step-by-step demonstration of creating a local instance will be provided.

4.1 Setup Options

Performing a successful Balance Attack on a blockchain network necessitates a degree of control over the network. Specifically, the adversary must be able to manipulate the communication flow among all nodes to partition the network into distinct subgroups. However, it is crucial that the implementation closely resembles the real network to ensure that the findings and conclusions apply to practical scenarios.

4.1.1 Avalanche-CLI[1]

Avalanche offers a command line tool specifically designed for facilitating the setup of subnets in order to support testing and development activities. This tool provides developers with a convenient way to configure and deploy subnets, enabling them to experiment, evaluate, and refine their applications in the Avalanche ecosystem. However, it is important to note that this approach is limited to simulating an honest version of the Avalanche network. The configuration options available in the provided config file allow for small setup changes to the nodes, but they do not enable the creation of malicious nodes, which is a key aspect of this work. Additionally, the number of nodes in a local network is capped at 5, which restricts the scalability and diversity of the network. To increase the number of nodes and incorporate validators, the subnet must be deployed on the Mainnet or the Fuji network, providing a more extensive and realistic environment for testing and experimentation. As a result, this technique was deemed unsuitable for implementing the Balance Attack. However, it proved to be a valuable resource for gaining insights into the workings of the Avalanche network and understanding the interconnectedness of its components.

[1] `https://github.com/ava-labs/avalanche-cli`.

More detailed information on Avalanche-CLI can be accessed at `https://docs.avax.network/subnets/create-a-local-subnet` and in Appendix C.

4.1.2 AvalancheGo[2]

For setting up a local Avalanche test network the source code is provided on GitHub. The version used in testing is v1.7.4 as problems occurred with later versions in that referenced Git repositories were not (publicly) available.[3]
Nevertheless, AvalancheGo provides a high scale of customizability, as well as the ability to alter the behavior of nodes and introduction of malicious nodes into the network. Furthermore, all information about the network, like transaction status, is publicly accessible and can hence be requested by any adversary.

4.2 Setup Commands - AvalancheGo

The coordination of the network in the local Avalanche instance involves the execution of lengthy *curl*-commands for initializing nodes, users, and transactions. To streamline this process, Python scripts were employed. These scripts, originally developed by Enrico Tedeschi, during the work on *"When is Spring coming? A Security Analysis of Avalanche Consensus"* [ACT22], were further modified and adapted for the purposes of this master thesis. The upcoming paragraphs provide a detailed explanation of how these scripts can be utilized to establish a local Avalanche network instance.

First some changes in the .gitconfig must be made in order to run the Go source code:

```
[url "git@github.com:"]
        insteadOf = git://github.com/
        insteadOf = https://github.com/
```

or a Git environment variable must be set:

```
>> export GIT_TERMINAL_PROMPT=1
```

OR

```
>> export GOPRIVATE=github.com/{organization}/*
```

Initiating Nodes

```
>> python3 main.py -r [n]
```

Creates `n` nodes - default 5 - on 127.0.0.1 starting with port 9650. It is possible to add nodes after initialization. Due to limitations in the number of ports it is not possible to create a network with more than 50 nodes.

[2] `https://github.com/ava-labs/avalanchego`.
[3] in particular *avalanchego-operator* for `github.com/ava-labs/avalanche-network-runner@v1.0.6`.

Add Nodes

```
>> python3 main.py -a [n]
```

Initializes `n` (default 1) nodes and adds them to the topology.

Add Users to Nodes

```
>> python3 main.py -c usr c k [n]
```

When creating users a particular node can be specified on which the users should be added. Otherwise, the algorithm cycles over all available nodes and creates `k` users.

Create X-Chain Address for each User

```
>> python3 main.py -c adr u
```

Must be executed for each user in order to be able to execute transactions (not yet automated).

Add Validator to Node / Make User a Validator

```
>> python3 main.py -p v [u, n]
```

Makes user `u`, or the algorithm chooses a random user - usually the first one added - of node `n`, a validator with a corresponding P-address. This validator is also initiated with a predefined number of tokens. If no argument is given the algorithm will add a validator for each active node.
!*user6 must be a validator as it is used to airdrop funds to all users in the network***!**

"Airdrop" Funds

```
>> python3 main.py -p r [x]
```

OR

```
>> python3 main.py -c add u [a]
```

The first command airdrops `x` tokens to each X-Chain address. The second adds `a` tokens to user `u`. For both commands the default amount of tokens airdropped is 100'000'000.

Delete Users

```
>> python3 main.py -d [n] [u]
```

OR

```
>> python3 main.py -c usr d k [n]
```

Users can be deleted individually by specifying the node (`n`) and the user (`u`). If the operation *-d* is called without arguments duplicated users are deleted. Otherwise, a similar call as creating users can be made and `k` users are deleted either from one specified node `n` or cycling over all users.

Visualization of the network's state

Network Overview

In order to get a printout overview of the current state of the network there are two commands:

```
>> python3 main.py -c lst
```

OR

```
>> python3 main.py -c info
```

The first one prints out all active nodes and their corresponding users. The second operation gives a more detailed view of the network, displaying the total number of nodes and users, how many tokens are in circulation, and which user has the most and the fewest amount of tokens (Figure 4.1).

```
+------------+-----------+-----------+
| Num. Nodes | Num. Users | tot. AVAX |
+============+============+===========+
| 5          | 7          | 7.780e+08 |
+------------+-----------+-----------+

+------------+-------+-----------+
|            | User  |  Balance  |
+============+=======+===========+
| More Funds | user2 | 1.494e+08 |
+------------+-------+-----------+
| Less Funds | user1 | 41065727  |
+------------+-------+-----------+

+-------+-------+
| Nodes | Users |
+=======+=======+
|  9652 |  2, 7 |
+-------+-------+
|  9654 |     3 |
+-------+-------+
|  9656 |     4 |
+-------+-------+
|  9658 |     5 |
+-------+-------+
|  9650 |  1, 6 |
+-------+-------+
```

Figure 4.1: Output of "-c info"

User Overview

```
>> python3 main.py -v wlt
```

This command displays the individual information of each user, like X-Chain address and token balances. A user with a P-Chain address is a validator of their node.

```
Node 9652:
        user2
                X-local1fdsp2k7ugkd6pvt3nxqks9s6y5y0wfvm586k4z
                asset: AVAX
                balance: 149401461
                P-local18jma8ppw3nhx5r4ap8clazz0dps7rv5u00z96u
                balance: 31196332732752149
                 |-unlocked: 31196332732752149
                 |-locked stakeable: 0
                 |-locked not stakeable: 0
        user7
                X-local1l3fqule6pjzmuzantpus22999aq0668j3722wa
                asset: AVAX
                balance: 106000000
```

Figure 4.2: Output of "-v wlt"

Transaction Overview

In order to get the list of all known transactions and their statuses the following command can be used:

```
>> python3 main.py -v t [v]
```

By changing the optional argument `[v]` to `1` or `2` additional information for example transaction id or time of execution can be displayed as well.
The status of each transaction can be either ACCEPTED, PENDING, or UNKNOWN.

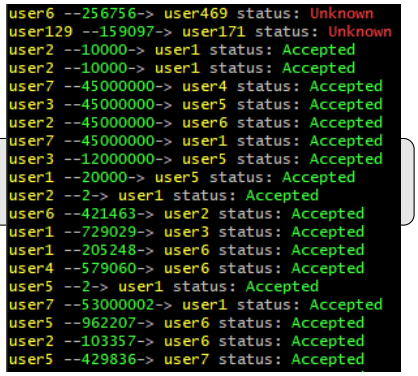

Figure 4.3: Output of "-v t 0"

Handling of Transactions

Single Transaction

```
>> python3 main.py -s a r s [m]
```

By specifying the amount (`a`), a receiver (`r`), a sender (`s`), and a possible message (`m`) a simple transaction can be specified and is performed by the avalanche network. A fee of 1'000'000 AVAX is deducted from the sender's token balance. If the sender does not have enough AVAX tokens in its wallet the transaction is rejected and the algorithm will try again to carry out the transaction at a later point in time.

Automatic Generation of Transactions

It is also possible to define an interval in which several transactions are carried out automatically by choosing two random users and performing a transaction between these parties. As the balance of the sender is checked before the amount sent is specified it is ensured that enough tokens are present.

```
>> python3 main.py -i [s] [n] [c]
```

The argument `s` ($\in \mathbb{Z}$) defines the length of each time interval in seconds, `n` the number of transactions created within each time interval, and `c` can be defined to be TRUE such that these transactions can be conflicting with each other. If nothing is specified, the algorithm will create one non-conflicting transaction every 60 seconds.

Chapter 5

Proof-of-Concept

Next, an in-depth analysis of the technical aspects of the Balance Attack on Avalanche is provided. Additionally, various methods of blocking or delaying communication will be explored, along with an examination of their respective advantages and disadvantages.

5.1 Strategy to Perform the Balance Attack on Avalanche

In order to break the Avalanche consensus protocol, the idea by Natoli and Gramoli [NG17] is taken up and continued. As in the original strategy the communication between two subgroups is blocked/delayed, however, as Avalanche is using a PoS consensus scheme, these groups are not divided based on equal hashing power but the sheer number of validators. Furthermore, malicious validators are used to impact the decision-making of the network.
Similar to the Balance Attack description in Section 3.3, the adversary blocks the communication between two subgroups (SG_1 and SG_2) that have an equal number of validator nodes. Options to block communication between nodes are discussed in Section 5.2. However, the adversary or a group of malicious validators that are associated with the attacker can communicate across both subgroups without hindrance. After disabling communication, each subgroup is sent a conflicting transaction, T_1 and T_2, generated by the adversary. For reasons of consistency, in this example, T_1 is broadcasted to subgroup SG_1 and T_2 to subgroup SG_2, respectively. During the sampling process, nodes of both subgroups can also sample the adversary. In order to not leak any information of the other transaction a malicious node responds according to the following pseudocode (Algorithm 5.1). This algorithm overwrites the honest response behavior of a node, described in lines 51-54 in Appendix A. Furthermore, the following assumptions are made:

1. A malicious validator $\mathcal{F}$ can determine the subgroup a request comes from
2. All malicious validators know both transactions T_1 and T_2
3. The malicious validators can behave honestly until the attack starts/the conflicting transactions have been released
4. During the attack, malicious nodes do not initiate samplings of the network

Algorithm 5.1 Respond Generation of malicious validator $\mathcal{F}$

1: **upon receiving** message [QUERY, T] from party v **do**:
2: **if** $v \in \mathrm{SG}_1$ **then**: // $\mathcal{F}$ can determine the subgroup v belongs to
3: $\mathcal{VF}^* \leftarrow \mathcal{VF} \cup T_1$
4: **else**:
5: $\mathcal{VF}^* \leftarrow \mathcal{VF} \cup T_2$
6: **send** message [VOTE, $\mathcal{F}, T, \mathcal{VF}^*$] to party v

In short, depending on which subgroup is requesting the VF of a faulty validator the respective transaction - T_1 for SG_1 and T_2 for SG_2 - will be part of it to support the decision-making. Notably, the "*Virtuous Frontier*" of malicious nodes is not updated during the attack to simplify the implementation.
Eventually, both transactions are committed in their respective subgroups hence breaking the consensus protocol as T_1 and T_2 are accepted although they are part of the same conflict set and a successful double-spending attack is performed.

5.2 Blocking/Delaying Communication between Nodes

A vital requirement for executing a successful Balance Attack is the immensely significant topic of blocking or delaying communications between nodes. Introduced and explored are three distinct possible methods to block or delay communication between Avalanche nodes. By delving into these techniques, valuable insights are gained into the intricacies of impeding network communication effectively. Furthermore, this section evaluates the practicality and limitations of each method, providing valuable insights into their viability and implications.

5.2.1 Kollaps[1] - Decentralized Container Based Network Emulator [GNS+20]

The innovative tool Kollaps enables the emulation and simulation of complex network environments by leveraging containerization technology. Hence, it becomes possible to create and manage decentralized network topologies, facilitating the evaluation and analysis of network behavior under predefined configurations. By providing a decentralized and scalable network emulation solution, it can be useful to examine the Balance Attack under different circumstances.
In the following paragraphs, the inner workings of Kollaps are explored, including detailed setup steps and a comprehensive assessment of its potential use cases within the context of this master thesis.

[1] `https://github.com/miguelammatos/Kollaps`.

Architecture

The architecture of Kollaps is designed to provide a decentralized and container-based network emulation environment. It consists of multiple components working together to simulate network conditions and behaviors. At its core, Kollaps makes use of containerization technology, such as Docker or Kubernetes, to encapsulate network nodes within isolated environments. These nodes can represent different entities, such as hosts, routers, or switches, and their interactions are orchestrated through a distributed control plane. The control plane enables the management and coordination of network behaviors, such as latency, packet loss, and bandwidth limitations. By distributing the emulation across multiple containers, Kollaps achieves scalability and flexibility, allowing users to create complex network topologies and replicate real-world scenarios. Furthermore, dynamic events like node joining and leaving can be incorporated into the emulation, enabling realistic network simulations and expanded research possibilities. However, these modifications must be defined during the setup process as they are precomputed offline. Additionally, Kollaps provides a convenient built-in dashboard feature that allows users to monitor and observe the status of services, track ongoing traffic, and stay informed about dynamic events occurring within the network emulation environment.

Setup and Deployment Steps

Kollaps can be easily installed by cloning the respective project from GitHub[2] by for example using the following command:

```
>> git clone --branch master --depth 1 --recurse-submodules
       https://github.com/miguelammatos/Kollaps.git
```

The execution of this command guarantees the cloning of all necessary submodules. Also included in the project, an example topology is provided in order to ensure that the installation of Kollaps was successful and allows users to verify the proper functioning of the tool.
By defining node configurations and customizing the network behavior a topology can be designed to accommodate the deployment of AvalancheGO nodes. This enables the simulation of realistic network scenarios, including the implementation of the Balance Attack by selectively blocking communication between distinct subgroups of nodes through dynamic events.
Further details for setting up and deploying Kollaps can be found on the project's GitHub page (`https://github.com/miguelammatos/Kollaps`).

[2] `https://github.com/miguelammatos/Kollaps`.

Conclusion

In conclusion, the exploration of the Kollaps - Decentralized Container Based Network Emulator has revealed its suitability as a tool for network emulation in cases requiring customizability and scalability. The ability to simulate dynamic events and observe network behavior through the built-in dashboard offers valuable insights for evaluating complex network scenarios.
However, despite the potential benefits, the deployment of Kollaps in combination with AvalancheGO was hindered by permission issues, limiting the feasibility of utilizing this specific combination for the intended purposes of this master thesis. Nonetheless, the evaluation of Kollaps underscores its potential as a powerful tool for network emulation in various other research and development contexts.

5.2.2 Uncomplicated Firewall[3]

The subsequent introduction of the Uncomplicated Firewall (UFW) and its integration with IP Tables aims to equip readers with a foundational understanding of UFW and IP Tables by introducing some of the most commonly used command-line operations to configure firewall rules.

UFW and IP Tables

UFW, a user-friendly command-line tool, offers simplified management of firewall rules in Linux systems. By leveraging the power of IP Tables, UFW enables users to control incoming and outgoing network traffic effectively.
IP Tables, on the other hand, is a powerful command-line utility in Linux systems that allows for fine-grained control over network packet filtering and network address translation.

Commands to Manage IP Tables

Essential for the use in VMs:

```
>> sudo ufw allow ssh
```

OR

```
>> sudo ufw allow 22
```

If the tool is used on a Virtual Machine - as was done in this master thesis - the firewall must allow the ssh connection otherwise the communication is blocked and the connection disabled.

[3] https://www.linode.com/docs/guides/configure-firewall-with-ufw/.

Enable/Disable Firewall:

```
>> sudo ufw enable|disable
```

Set Default Rules for IP Tables:

```
>> sudo ufw default allow|deny|reject incoming|outgoing
```

Sets the default response for incoming and outgoing connections is a critical step in firewall configuration. It is important to note that using the *"deny"* or *"reject"* rules without proper consideration can potentially lock the user out of the associated Linode. Therefore, it is crucial to ensure that essential services such as SSH, TCP, and UDP connections have been appropriately configured to be allowed before enabling the firewall. Taking these precautions helps prevent unintended access restrictions and ensures the continued accessibility of necessary services.

Add Entry to IP Tables:

```
>> sudo ufw allow from 127.0.0.1 to any port 9650
```

In addition to permitting or denying communication to specific ports, UFW offers support for more advanced configurations involving pairs of IP addresses or subnets, and a IP address, subnet, or port. In the provided example, any message originating from the IP address 127.0.0.1 and received at port 9650 is allowed.

Display All Rules:

```
>> sudo ufw status [verbose/numbered]
```

This instruction shows a list of all rules established in UFW, regardless of whether it is active or not.

Delete Rule:

```
>> sudo ufw delete [rule-id]
```

Executing this instruction allows the targeted removal of a specific rule from the UFW configurations. The rule can be either identified with its rule ID, obtainable from the UFW status block, or its respective service name, such as *"allow 80"* for all HTTP traffic, as an example.

Logging:

```
>> sudo ufw logging on low|medium|high
```

Advantages and Disadvantages

UFW and IP Tables offer several advantages for firewall management. These include robust firewall capabilities that provide a strong defense against unauthorized access and security threats. The customizable rule configuration allows for precise control over network traffic based on various criteria. With a user-friendly interface, UFW simplifies the configuration and management of firewall rules, making it accessible to users with limited networking expertise. Being integrated into Linux systems ensures compatibility and seamless integration with existing infrastructure. Additionally, UFW and IP Tables provide logging and monitoring capabilities for effective network traffic analysis and troubleshooting.
An inherent limitation, particularly relevant to this master thesis, is the fact that nodes deployed by AvalancheGO are assigned different ports on the same IP address. UFW, however, does not provide the flexibility to fine-tune communication between two specific ports. Compounding the issue, the UFW documentation explicitly states that any IP Table rule specified does not apply to Docker containers. Consequently, employing UFW to block communication between Avalanche nodes becomes impractical, as it lacks the capability to block communication between two specific ports within a Docker container.

Digression: Avalanche Network across Multiple Virtual Machines

Given that UFW is unable to block Port-to-Port communication, an alternative approach was pursued, involving the deployment of the Avalanche network across multiple VMs. Each subgroup would be represented by a separate VM, allowing for fine-grained control over the communication between them. Tools like UFW can then be used to adjust and modify the communication between the VMs, as it falls within the scope of such tools. This approach offers greater flexibility in managing and blocking specific communication paths between subgroups.
However, during the setup process, it was discovered that AvalancheGO lacks the inherent capability to be set up across multiple virtual machines without significant code adjustments. This means that without further modifications, it is not feasible to establish separate subgroups represented by different VMs within AvalancheGO. Such adjustments would require substantial code changes that fall outside the scope of this thesis. Therefore, the focus remains on exploring alternative methods and techniques within the existing framework of AvalancheGO to achieve the desired network behavior for the Balance Attack simulation.

5.2.3 Altering the Avalanche Source-Code[4]

Shifting the focus onto potential code adjustments that can be made to manipulate the communication between Avalanche nodes, aiming to introduce delays or blockages. By meticulously exploring these modifications, valuable insights into network communication intricacies within the Avalanche protocol are unveiled.

Benchlist-Manager

The benchlist manager in the Avalanche blockchain plays a crucial role in maintaining network connectivity among validator nodes. The implementation details of this manager can be found in the AvalancheGO source code, specifically in the *"snow/networking/benchlist"* directory. Each node in the Avalanche network is equipped with a benchlist manager, responsible for managing a dynamic list of peers. This benchlist serves as a penalty for non-responsive nodes, optimizing network performance and upholding the integrity of the consensus protocol.
When a node fails to respond in time to ten consecutive network sampling requests generated by another node, it is added to the benchlist of the requesting node. For the next 15 minutes, the nodes on the benchlist are excluded from the sampling process. After this period, the node is removed from the benchlist and can participate in subsequent network samplings.

This mechanism can also serve as a passive aid for the Balance Attack. As unresponsive nodes are added to the benchlist, a situation arises where a majority of nodes belonging to the opposing subgroup are placed on this list. Since the sampling process excludes the "blacklisted" nodes, it significantly speeds up the overall process by drastically reducing the probability of sampling a node from the other subgroup. Consequently, the presence of the benchlist enhances the efficiency of the Balance Attack.

The benchlist feature serves as a powerful tool to generate message-blocking behavior within the network. By listing neighboring nodes on a node's benchlist, their communication can be effectively blocked during any sampling process. This allows for the simulation of a network split, which is essential for executing the Balance Attack. To achieve this, static variables can be used to represent the two required subgroups. By adding the nodes from the static variables to their respective node's benchlists, nodes from one subgroup are disregarded from the sampling process of the other one. Once the attack is successfully performed, all benchlists can be cleared, restoring the initial state of the network.

[4] `https://github.com/ava-labs/avalanchego`.

Docker Generation[5]

Docker generation is a powerful tool that allows for the creation and deployment of lightweight and isolated containers. These containers encapsulate applications, libraries, and dependencies, providing a consistent and reproducible environment. One significant advantage of Docker is its ability to alter network behavior within the container. By configuring network settings and implementing specific network policies, developers can control and manipulate communication patterns between containers and the outside world. This flexibility enables fine-grained control over network behavior, facilitating various testing scenarios and enabling the exploration of different network configurations.
AvalancheGo uses Docker to deploy and manage a local Avalanche network. Within the AvalancheGO ecosystem, Docker is utilized to package the necessary components and dependencies of the Avalanche node into a containerized environment. This containerization allows for easy deployment, scalability, and reproducibility of the Avalanche nodes across different environments.

Inspired by the concept introduced in Section 5.2.2, the Docker container's inherent modifiability of network communication behavior can be leveraged to partition the Avalanche network into two distinct subgroups. However, given the available level of expertise, the set time frame, and the inherent complexity of the AvalancheGo Docker generation, it was not feasible to modify Docker's behavior in a manner that would meet the requirements for altering the communication behavior as outlined in this master's thesis.

[5] `https://docs.docker.com/`.

Chapter 6

Related Work

The related work chapter of this thesis explores various research papers and studies that are closely related to the topic of blockchain security analysis, with a specific focus on the seminal paper by Natoli and Gramoli titled "The Balance Attack or Why Forkable Blockchains are Ill-Suited for Consortium" [NG17]. This influential paper investigates the vulnerabilities and limitations of forkable blockchains in the context of consortium-based settings. Additionally, the chapter discusses the research conducted by Bagaria et al. in their paper "Deconstructing the Blockchain to Approach Physical Limits" [BKT+19], which offers a comprehensive analysis of the physical limitations and challenges faced by blockchain technology.
By examining these and other relevant works, this chapter aims to provide a comprehensive overview of the existing research landscape and its implications for the security and resilience of blockchain systems.

The Balance Attack. In 2016/2017, Christopher Natoli and Vincent Gramoli, researchers from the University of Sydney, introduced and executed a novel attack called the Balance Attack. This attack exploits the inherent forkable nature of blockchains, aiming to disrupt the consensus protocol and compromise the integrity of the blockchain.
During their research, Natoli and Gramoli conducted an in-depth analysis of the Balance Attack on an Ethereum testnet of the R3 consortium. The adversary in the Balance Attack can strategically split the network into two subgroups, each with equal hash power. This allows the adversary to execute different tactics, such as issuing conflicting transactions to each subgroup or selectively issuing a specific transaction to only one subgroup. The attacker in the Balance Attack can manipulate the fork-resolving algorithms implemented in blockchains by actively mining on their preferred subtree. By doing so, they aim to increase the likelihood that one subtree, which does not contain the eventually committed dummy transaction, outweighs another subtree.
Therefore an attacker can issue a transaction in one subgroup and utilize their mining power to support the creation of new blocks in another subgroup. This strategic manipulation allows them to exploit the consensus mechanism and potentially execute double spending, compromising the integrity and security of the blockchain.
Natoli and Gramoli's research findings revealed that in order to achieve a high success rate for performing a Balance Attack on Ethereum, an individual machine would need to block communication for approximately 20 minutes. However, in the case of a consortium possessing one-third of the total mining power, the required communication blockage time reduces to around 4 minutes, while still maintaining a success rate of 94%.

Indefinite Fork. In their work on introducing Prism, a new blockchain, Bagaria et al. [BKT+19] made a significant contribution by formulating a precise method for sustaining a fork indefinitely during a Balance Attack. When initiating a fork in the network, such as by issuing two conflicting transactions, the adversary strategically divides its hashing power to mine on both subtree branches simultaneously. To ensure an equal weight distribution between the subtrees, before communication is enabled, the adversary selectively releases blocks from its local storage if an imbalance is detected. Once communication is re-established, the adversary employs a strategy to maintain the forked state of the blockchain by addressing any imbalances present in the subtrees.

Chapter 7

Conclusion and Future Work

This study sheds light on the Balance Attack on the Avalanche network and presents avenues for future work in enhancing the security and resilience of blockchain systems.

7.1 Conclusion

This master thesis focused on the execution of the Balance Attack on the Avalanche network. The study extensively explores the theoretical aspects of the attack, offering valuable insights into potential vulnerabilities and implications for blockchain systems. These findings contribute to a deeper understanding of security risks and vulnerabilities in blockchain systems, with a particular focus on the Avalanche network. However, it was determined that the actual execution of the Balance Attack was not feasible within the constraints of the research, despite the comprehensive analysis.

One of the key challenges encountered was the difficulty in setting up a reliable and effective system for blocking communication within the Avalanche network. Such a system is crucial for simulating the required network split and executing the Balance Attack. Complications and limitations arose in implementing a robust communication-blocking mechanism, hindering the practical execution of the attack.

Nonetheless, the identification of these limitations and challenges surrounding communication-blocking is a significant contribution to the field. By recognizing and highlighting these obstacles, further research and development can be directed toward finding suitable solutions to overcome them.

In conclusion, while the practical execution of the Balance Attack was not feasible due to complications with setting up a communication blocking system, the theoretical understanding gained and the identification of limitations contribute to the advancement of knowledge in blockchain security. The findings of this research provide a stepping stone for future studies aiming to enhance the security and resilience of blockchain networks, including the Avalanche network.

7.2 Recommendations for Further Research

Although the execution of the Balance Attack was not achieved, this master thesis serves as a comprehensive exploration of the attack's theoretical aspects. It lays the groundwork for further research and encourages the development of more robust security measures

in blockchain networks. Additionally, the identification of limitations to communication blocking provides a foundation for future investigations to address these challenges and devise effective countermeasures.

Practical Balance Attack on the Avalanche Blockchain

Exploring additional, unexplored ideas for blocking communication or finding workarounds to make the existing approaches feasible would be beneficial in achieving the goal of splitting the network into subgroups and executing the Balance Attack. By pursuing these avenues, the following questions can be addressed:

- What is the minimum duration of communication blocking needed to achieve a reasonable chance of successfully executing a Balance Attack?
- What are the potential implications and consequences if the adversary does not respond to any network sampling requests during the execution of a Balance Attack?
- How does the presence of multiple adversaries affect the success rate and speed of executing a Balance Attack?

Enhancing Communication Blocking Mechanisms

In the realm of research, it is valuable to investigate alternative approaches for establishing a robust communication-blocking system. This exploration is not limited to distributed systems like blockchains but extends to other domains as well. By developing such systems, researchers can gain insights that facilitate the analysis of security-related topics. Furthermore, it is important to explore techniques that can delay or disrupt communication between nodes. Simulating network splits and assessing the resulting impact on consensus protocols and potential vulnerabilities can enhance our understanding of distributed systems and contribute to their security analysis.

Expanding the Adversarial Model

Further enhance the existing adversary model by incorporating more sophisticated Byzantine behaviors, such as strategic node selection or adaptive attack strategies. This extension will enable a comprehensive investigation into the impact of different types of Byzantine nodes on the resilience and security of the Avalanche consensus protocol. By exploring the behavior and characteristics of these advanced adversaries, a deeper understanding can be gained regarding the vulnerabilities and potential countermeasures necessary to strengthen the overall security of the network. The analysis of these complex Byzantine behaviors will provide valuable insights into the design and implementation of more robust and resilient consensus protocols for distributed systems like Avalanche.

LITERATURE

[ACT22] AMORES-SESAR, Ignacio ; CACHIN, Christian ; TEDESCHI, Enrico: When Is Spring Coming? A Security Analysis of Avalanche Consensus. In: *OPODIS* Bd. 253, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022 (LIPIcs), S. 10:1–10:22

[BKT+19] BAGARIA, Vivek K. ; KANNAN, Sreeram ; TSE, David ; FANTI, Giulia ; VISWANATH, Pramod: Prism: Deconstructing the Blockchain to Approach Physical Limits. In: *CCS*, ACM, 2019, S. 585–602

[CGR11] CACHIN, Christian ; GUERRAOUI, Rachid ; RODRIGUES, Luís E. T.: *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011

[GNS+20] GOUVEIA, Paulo ; NEVES, João ; SEGARRA, Carlos ; LIECHTI, Luca ; ISSA, Shady ; SCHIAVONI, Valerio ; MATOS, Miguel: Kollaps: decentralized and dynamic topology emulation. In: *EuroSys*, ACM, 2020, S. 23:1–23:16

[Nak08] NAKAMOTO, Satoshi: Bitcoin: A Peer-to-Peer Electronic Cash System. (2008). `https://bitcoin.org/bitcoin.pdf`

[NG17] NATOLI, Christopher ; GRAMOLI, Vincent: The Balance Attack or Why Forkable Blockchains are Ill-Suited for Consortium. In: *DSN*, IEEE Computer Society, 2017, S. 579–590

[PD16] POON, Joseph ; DRYJA, Thaddeus: The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. (2016). `https://lightning.network/lightning-network-paper.pdf`

[SZ15] SOMPOLINSKY, Yonatan ; ZOHAR, Aviv: Secure High-Rate Transaction Processing in Bitcoin. In: *Financial Cryptography* Bd. 8975, Springer, 2015 (Lecture Notes in Computer Science), S. 507–527

[TYS+19] TEAM ROCKET ; YIN, Maofan ; SEKNIQI, Kevin ; RENESSE, Robbert van ; SIRER, Emin G.: Scalable and Probabilistic Leaderless BFT Consensus through Metastability. In: *CoRR* abs/1906.08936 (2019)

[WSNH19] WANG, Gang ; SHI, Zhijie J. ; NIXON, Mark ; HAN, Song: SoK: Sharding on Blockchain. In: *AFT*, ACM, 2019, S. 41–61

Appendix A

Avalanche Consensus Protocol

Based on the algorithms found in the paper *"When is Spring Coming?"* [ACT22].

Algorithm A.1 Avalanche Main Loop (party u), state

Global parameters and state

1: $\mathcal{N}$ // set of parties
2: $maxPoll \in \mathbb{N}$ // maximum number of concurrent polls, *default value 4*
3: $k \in \mathbb{N}$ // number of parties queried in each poll, *default value 20*
4: $\alpha \in \{\lceil \frac{k+1}{2} \rceil, \dots, k\}$ // majority threshold for queries, *default value 15*
5: $\beta_1 \in \mathbb{N}$ // threshold for early acceptance, *default value 15*
6: $\beta_2 \in \mathbb{N},\ \beta_2 > \beta_1$ // threshold for acceptance, *default value 150*
7: $\mathcal{T} \leftarrow \emptyset$ // set of known transactions
8: $\mathcal{Q} \subset \mathcal{T} \leftarrow \emptyset$ // set of queried transactions
9: $\mathcal{R} \subset \mathcal{Q} \leftarrow \emptyset$ // set of repollable transactions
10: $\mathcal{D} \subset \mathcal{T} \leftarrow \emptyset$ // set of no-op transaction to be queried
11: $\mathcal{VF} \subset \mathcal{Q} \leftarrow \emptyset$ // set of transactions in the virtuous frontier
12: $conPoll \in \mathbb{N} \leftarrow 0$ // number of concurrent polls performed
13: $conflictSet : HashMap[\mathcal{T} \rightarrow 2^{\mathcal{T}}]$ // conflict set
14: $\mathcal{S} : HashMap[\mathcal{T} \rightarrow \mathcal{N}]$ // set of sampled parties to be queried with a transaction
15: $votes : HashMap[\mathcal{T} \times \mathcal{N} \rightarrow \{\text{FALSE, TRUE}\}]$ // variable to store the replies of queries
16: $d : HashMap[\mathcal{T} \rightarrow \mathbb{N}]$ // confidence value of a transaction
17: $pref : HashMap[2^{\mathcal{T}} \rightarrow \mathcal{T}]$ // preferred transaction in the conflict set
18: $last : HashMap[2^{\mathcal{T}} \rightarrow \mathcal{T}]$ // preferred transaction in the last query
19: $cnt : HashMap[2^{\mathcal{T}} \rightarrow \mathbb{N}]$ // counter for acceptance of the conflict set
20: $accepted : HashMap[\mathcal{T} \rightarrow \{\text{FALSE, TRUE}\}]$ // indicator that a transaction is accepted
21: $timer : HashMap[\mathcal{T} \rightarrow \{timers\}]$ // timer for the query of transaction
22: $ack : HashMap[\mathcal{T} \times \mathcal{T} \rightarrow \mathbb{N}]$ // number of votes on T reporting that T' is not preferred
23: $\mathcal{G} : HashMap[\mathcal{T} \times \mathcal{N} \rightarrow 2^{\mathcal{T}}]$ // ancestors of positively reported transactions

Algorithm A.2 Avalanche Main Loop (party u), part 1

24: **upon** *broadcast*(tx) **do**:
25: **if** $V(tx)$ **then**:
26: $T \leftarrow (tx, \mathcal{VF})$ // up to a maximum number of parents
27: $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
28: $accepted[T] \leftarrow$ FALSE
29: updateDAG(T)
30: gossip message [BROADCAST, T]

31: **upon** hearing message [BROADCAST, T] **do**:
32: **if** $T \notin \mathcal{T}$ **then**:
33: $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$
34: $accepted[T] \leftarrow$ FALSE

35: **upon** $conPoll < maxPoll$ **do**:
36: $conPoll \leftarrow conPoll + 1$
37: **if** $\mathcal{D} \neq \emptyset$ **then**: // prefer no-op transaction
38: $T \leftarrow$ least recent transaction in $\mathcal{D}$
39: **else if** $\mathcal{T} \backslash \mathcal{Q} \neq \emptyset$ **then**: // take any not yet queried transaction
40: $T \xleftarrow{R} \mathcal{T} \backslash \mathcal{Q}$
41: $d[T] \leftarrow 0$
42: **else**: // all transaction queried already, take one of them
43: *updateRepollable*()
44: $T \xleftarrow{R} \mathcal{R}$
45: $S[T] \leftarrow sample(\mathcal{N} \backslash \{u\}, k)$ // sample k parties randomly according to state
46: send message [QUERY, T] to all parties $v \in S[T]$
47: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\perp, \mathcal{VF} \backslash \{T\})\}$ // create a no-op transaction
48: **start** $timer[T]$ // duration Δ_{query}
49: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{T\}$
50: updateDAG(T)

51: **upon receiving** message [QUERY, T] from party v **do**:
52: **if** $T \notin \mathcal{T}$ **then**: // party u sees T for the first time
53: updateDAG(T)
54: **send** message [VOTE, $u, T, \mathcal{VF}$] to party v

55: **upon receiving** message [VOTE, $v, T, \mathcal{VF}'$] from party $v \in S[T]$ **do**:
56: $votes[T, v] \leftarrow \mathcal{VF}'$ // $\mathcal{VF}'$ is the vote
57: **for** $T' \in votes[t, v]$ **do**: // build the ancestors of the reported transaction
58: $\mathcal{G}[T, v] \leftarrow \mathcal{G}[T, v] \cup ancestors(T')$
59: **for** $T' \in \mathcal{G}[T, v]$ **do**: // count the number of parties that reported T'
60: **if** $ack[T, T'] = \perp$ **do**:
61: $ack[T, T'] \leftarrow 1$
62: **else**:
63: $ack[T, T'] \leftarrow ack[T, T'] + 1$

Algorithm A.3 Avalanche Main Loop (party u), part 2

64: **upon** $\exists T \in \mathcal{T}$ such that $|\{votes[T, v]\}| = k$ **do**: // every queried party has replied
65: **stop** $timer[T]$
66: $votes[T, *] = \perp$ // remove all entries in *votes* for T
67: **for** $T' \in \mathcal{Q}$ **do**:
68: **if** $ack[T, T'] \geq \alpha$ **then**:
69: $d[T'] \leftarrow d[T'] + 1$
70: **if** $d[T'] > d[pref[conflictSet[T']]]$ **then**:
71: $pref[conflictSet[T']] \leftarrow T'$
72: **if** $T' \neq last[conflictSet[T']]$ **then**:
73: $last[conflictSet[T']] \leftarrow T'$
74: $cnt[conflictSet[T']] \leftarrow 1$
75: **else**:
76: $cnt[conflictSet[T']] \leftarrow cnt[conflictSet[T']] + 1$
77: **else**:
78: $cnt[conflictSet[T']] \leftarrow 0$
79: $ack[T, *] = \perp$ // remove all entries in *ack* for T
80: $\mathcal{G}[T, *] \leftarrow \perp$ // remove all entries in $\mathcal{G}$ for T

81: **upon** $\exists T \in \mathcal{T}$ such that $isAccepted(T) \wedge \neg accepted[T]$ **do**:
82: $(tx, parents \leftarrow T)$
83: **if** $V(tx)$ **then**:
84: $accepted[T] \leftarrow$ TRUE
85: *deliver* tx

86: **upon** timeout from $timer[T]$ **do**: // not enough votes on T received
87: $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{T\}$
88: $votes[T, *] \leftarrow \perp$ // remove all entries in *votes* of T
89: $\mathcal{S}[T] \leftarrow []$ // u will not consider more votes from this query

Algorithm A.4 Avalanche , auxiliary functions

90: **function** updateDAG(T):
91: $\mathcal{VF} \leftarrow$ set of non-conflicting leaves in the DAG
92: $conflictSet[T] \leftarrow \emptyset$
93: **for** $T' \in \mathcal{T}$ such that $T' \neq T$ and T' has common input with T **do**:
94: $conflictSet[T] \leftarrow conflictSet[T] \cup \{T'\}$
95: $conflictSet[T'] \leftarrow conflictSet[T'] \cup \{T\}$
96: **if** $conflictSet[T] = \emptyset$ **then**: // T is non-conflicting
97: $pref[conflictSet[T]] \leftarrow T$
98: $last[conflictSet[T]] \leftarrow T$
99: $cnt[conflictSet[T]] \leftarrow 0$
100: $conflictSet[T] \leftarrow conflictSet[T] \cup \{T\}$

101: **function** *getParents*(T):
102: $(tx, parents \leftarrow T)$
103: **return** *parents* // set of parents stored in T

104: **function** *preferred*(T):
105: **return** $(T \stackrel{?}{=} pref[conflictSet[T]])$

106: **function** *stronglyPreferred*(T):
107: **return** $(\bigwedge_{T' \in ancestors(T)} preferred(T'))$

108: **function** *acceptable*(T):
109: **return** $(|\ conflictSet[T]\ | = 1\ \wedge\ cnt[conflictSet[T]] \geq \beta_1)$
$\wedge\ \bigwedge_{T' \in parents(T)} acceptable(T')\ \vee\ cnt[conflictSet[T]] \geq \beta_2$

110: **function** *isRejected*(T):
111: **return** $(\exists T' \in \mathcal{T}$ such that $\forall T' \in conflictSet[T] \backslash \{T\}\ :\ acceptable(T'))$

112: **function** *updateRepollable*():
113: $\mathcal{R} \leftarrow \emptyset$
114: **for** $T \in \mathcal{T}$ **do**:
115: **if** $acceptable(T) \vee \bigwedge_{T' \in parents(T)} (stronglyPreferred(T')$
$\wedge \neg isRejected(T'))$ **then**:
116: $\mathcal{R} \leftarrow \mathcal{R} \cup \{T\}$

Appendix B

Consensus Protocols for Resolving Forks

Based on the algorithms **1**, **2** and **3** from the paper *"The Balance Attack or Why Forkable Blockchains are Ill-Suited for Consortium"* [NG17].

Algorithm B.1 Nakamoto's consensus protocol at node p_i

$l_i = \langle B_i, P_i \rangle$ DAG representation of the blockchain at node p_i with blocks B_i and pointers P_i

```
1:  get-main-branch()_i:                          // select the longest branch in forked blockchain
2:      b ← genesis-block(B_i)                    // start search from root of blockchain
3:      while b.hasChildren do:                   // search while block has children
4:          block ← argmax_{c∈children(b)}{depth(c)}   // choose block with deepest branch
5:          B ← B ∪ {block}                       // update vertices of main branch
6:          P ← P ∪ {⟨block, b⟩}                  // update edges of main branch
7:          b ← block                             // update to next block
8:      return ⟨B, P⟩                             // returning the updated main branch

9:  depth(b):                                     // depth of longest branch rooted in b
10:     if b.hasNoChildren then return 1          // stop at leaves
11:     else return 1 + max_{c∈children(b)} depth(c)   // recursive algorithm for all children
```

Algorithm B.2 GHOST consensus protocol at node p_i

$l_i = \langle B_i, P_i \rangle$ DAG representation of the blockchain at node p_i with blocks B_i and pointers P_i

```
1:  get-main-branch()_i:                          // select the longest branch in forked blockchain
2:      b ← genesis-block(B_i)                    // start search from root of blockchain
3:      while b.hasChildren do:                   // search while block has children
4:          block ← argmax_{c∈children(b)}{num-desc(c)}   // choose block with heaviest tree
5:          B ← B ∪ {block}                       // update vertices of main branch
6:          P ← P ∪ {⟨block, b⟩}                  // update edges of main branch
7:          b ← block                             // update to next block
8:      return ⟨B, P⟩                             // returning the updated main branch

9:  num-desc(b):                                  // number of descendants of b
10:     if b.hasNoChildren then return 1          // stop at leaves
11:     else return 1 + ∑_{c∈children(b)} num-desc(c)   // recursive algorithm for all children
```

Appendix C

Avalanche-CLI Setup Commands

C.1 Introduction

The following appendix provides a reference guide for essential setup commands that can be utilized with the Avalanche-CLI (see Section 4.1.1) - a powerful command line interface tool designed for interacting with Avalanche). It should serves as a handy resource, presenting the most important commands to facilitate efficient utilization of the Avalanche-CLI tool.
For further information and a comprehensive exploration of Avalanche-CLI the author refers to the documentation[1].

C.2 Commands

The following section provides an overview of essential commands available in the Avalanche-CLI. This collection of commands covers various functionalities, including network management, account operations, contract deployment, and more. Each command is accompanied by a description, syntax, and usage examples, empowering users to navigate and harness the capabilities of Avalanche-CLI efficiently.

Command: Subnet Create

Function:
A new *genesis file* is created, and an interactive wizard is launched to configure and set up a new Avalanche subnet. Additionally, the utilization of predefined VM binaries, like Ethereum Virtual Machine Subnets, is supported.

Syntax:

```
avalanche subnet create [subnetName] [flags]
```

[1] https://docs.avax.network/subnets/reference-cli-commands.

Flags[2]:

```
    --custom              use a custom VM template
    --evm                 use the SubnetEVM as a base template
-f, --force               overwrite the existing configuration
    --genesis string      file path of genesis to use
-h, --help                help for create
```

Usage Example:

```
avalanche subnet create testnet1 --evm -f
```

Command: Subnet Delete

Function:
Deletes an existing subnet configuration.

Syntax:

```
avalanche subnet delete [subnetName]
```

Command: Subnet Deploy

Function:
This command deploys the subnet configuration locally, on the Fuji Testnet, or the Mainnet. Note that a subnet can only be deployed once per network, and redeployment requires calling `avalanche network clean`.

Syntax:

```
avalanche subnet deploy [subnetName] [flags]
```

Flags[3]:

```
-f, --fuji testnet              deploy to fuji (alias to testnet)
-h, --help                      help for deploy
-l, --local                     deploy to local network
-m, --mainnet                   deploy to mainnet
```

Usage Example:

```
avalanche subnet deploy testnet1 --local
```

[2]The flags listed are options for this command and do not cover all possibilities.
[3]see Footnote 2.

Command: Subnet AddValidator

Function:
Whitelists a primary network validator that can validate the associated subnet. It should be noted that this command is applicable exclusively when the subnet is deployed on either the Fuji Testnet or the Mainnet.

Syntax:

```
avalanche subnet addValidator [subnetName] [flags]
```

Flags[4]:

```
    --fuji fuji                  join on fuji (alias for 'testnet')
-h, --help                       help for addValidator
    --mainnet mainnet            join on mainnet
    --staking-period duration    how long this validator will stake
    --start-time string          UTC start time when this validator
                                    starts validating
```

Usage Example:

```
avalanche subnet addValidator testnet1 --staking-period 100
```

Command: Subnet Configure

Function:
Enables the modification of configuration files for both the subnet itself and for each individual chain within the subnet.

Syntax:

```
avalanche subnet configure [subnetName] [flags]
```

Flags[5]:

```
    --chain-config string            path to chain configuration
-h, --help                           help for configure
    --subnet-config string           path to the subnet configuration
```

Usage Example:

```
avalanche subnet configure testnet1 --subnet-config ./testnet1.config
```

[4]The flags listed are options for this command and do not cover all possibilities.
[5]see Footnote 4.

Command: Network Clean

Function:
This command is designed to gracefully shut down all subnets and delete their respective states. It is recommended to utilize this command whenever there is a need to redeploy a new subnet with a revised configuration.

Syntax:

```
avalanche network clean [flags]
```

Flags:

```
    --hard                        Also clean downloaded avalanchego
                                     and plugin binaries
-h, --help                        help for clean
```

Usage Example:

```
avalanche network clean --hard
```

Command: Network Start

Function:
This command initiates a local, multi-node Avalanche network on the current machine. However, it will fail if the local network subnet is already set up and running.

Syntax:

```
avalanche network start [flags]
```

Flags:

```
    --avalanchego-version string  use this version of avalanchego
                                     (default "latest")
-h, --help                        help for start
    --snapshot-name string        name of snapshot to use to start the
                                     network from
```

Usage Example:

```
avalanche network start --avalanchego-version v1.17.12
```

Command: Network Status

Function:
This command displays the status of a local Avalanche network, including whether it is currently running, along with essential statistics about the network.

Syntax:

```
avalanche network status
```

Usage Example:

```
avalanche network status
```

Command: Network Stop

Function:
This command allows for the graceful termination of all deployed subnets, ensuring that their states are safely preserved. If the snapshot flag is provided, the state is saved in the specified file. Alternatively, if the snapshot flag is not provided, the state is saved to the default snapshot.

Syntax:

```
avalanche network stop [flags]
```

Flags:

```
-h, --help                          help for stop
    --snapshot-name string          name of snapshot to use to start the
                                       network from
```

Usage Example:

```
avalanche network stop --snapshot-name "snapshot-12345"
```

Declaration of Consent

on the basis of Article 30 of the RSL Phil.-nat. 18

Name/First Name:	Zauder, Marcel Matti
Registration Number:	16-124-836
Study Program:	Computer Science
	Bachelor ☐ Master ☑ Dissertation ☐
Title of the Thesis:	Balance Attack on a Forkable Blockchain Melting Avalanche
Supervisor:	Prof. Dr. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of 5 September, 1996 is authorized to revoke the title awarded on the basis of this thesis.
For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with future theses submitted by others.

Place/Date

Ostermundigen, July 21, 2023

Signature