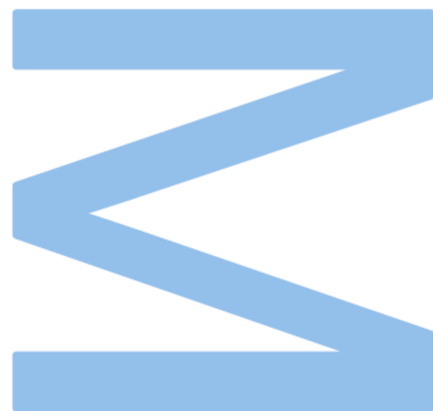
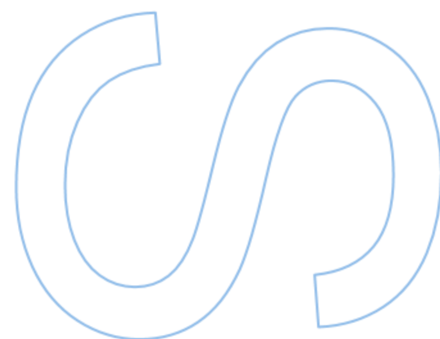
How secure are blockchains?

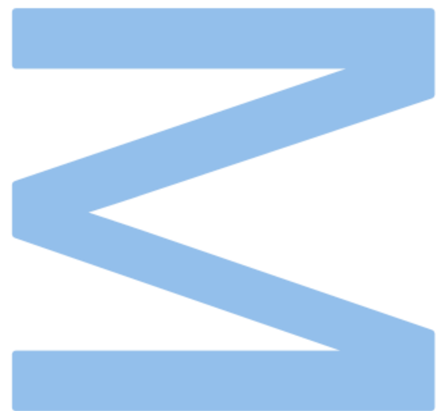
João Miguel de Sousa Regateiro
Mestrado em Segurança Informática
Departamento de Ciências e Computadores
2023

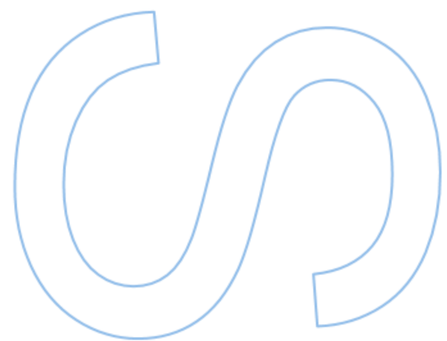
Orientador
João Paulo da Conceição Soares, Doutorado Equiparado a Investigador Principal, Faculdade de Ciências da Universidade do Porto

Coorientador
Rolando da Silva Martins, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO





Abstract

The blockchain is an evolving technology that was first introduced in 2009 with the launch of Bitcoin. Blockchains are decentralized networks maintained by peers participating in the network, this means that there is no central authority controlling the entire network and participants must communicate with each other to retrieve data.

There are different types of blockchains, with the most common being public. Public blockchains allow users to freely join and leave the network without any specific validation, which means that everyone is allowed to participate in it. This raises some concerns because malicious actors can join the network at any time and try to compromise its correct behavior or even interfere with other nodes.

This work conducted an analysis of Bitcoin with the primary objective of gaining a profound understanding of its underlying mechanisms and operations, specifically from a security perspective. The study delved into the process of a node joining the network and discovering new peers from which he would download the blockchain data. Bitcoin is currently one of the most well-known blockchain projects, serving essentially to execute transactions without the need for a trusted intermediary.

Recent projects such as Ethereum also brought new attack vectors with the introduction of new concepts as smart contracts for decentralized applications. This paper analyses some potential attack vectors to understand how blockchains are programmed to prevent them, giving special importance to eclipse attacks on the Ethereum network. By exploring previous works done in this area, the main goal is to perceive whether security measures are being applied to no longer be vulnerable to the same attacks and how the countermeasures are effective in preventing them.

The use of simulators was important to see how peers behave under certain conditions, however, some simulators lack user interactivity and others maintenance, which makes the research of some attack vectors quite difficult.

Keywords: Blockchain, Security, Ethereum, Bitcoin, Node, Proof-of-Work, Proof-of-Stake, Vulnerabilities.

Resumo

A blockchain é uma tecnologia introduzida pela primeira vez em 2009 com o lançamento da Bitcoin. Blockchains são redes descentralizadas mantidas por nós que participam na rede, isto significa que não existe nenhuma autoridade central a controlar toda a rede e que os nós participantes na mesma devem comunicar entre eles para obterem informação.

Existem diferentes tipos de blockchains, sendo as públicas mais comuns. Blockchains públicas permitem que utilizadores entrem ou saiam da rede livremente e sem serem submetidos a validações específicas, o que significa que qualquer um pode participar na mesma. Isto levanta preocupações devido à possibilidade de atores maliciosos poderem-se juntar à rede a qualquer altura e tentarem comprometer o correto funcionamento da mesma ou interferir com outros nós.

Este trabalho realizou uma análise à Bitcoin com o principal objetivo de obter conhecimento profundo sobre os mecanismos e operações subjacentes, mais especificamente numa perspetiva de segurança. O estudo aprofundou o processo de um nó se juntar à rede e descobrir novos nós através dos quais ele irá obter informação sobre a blockchain. A Bitcoin é atualmente uma das mais conhecidas blockchains, servindo essencialmente o propósito de realizar transações sem necessitar de um intermediário confiável.

Projetos mais recentes como a Ethereum trouxeram novos vetores de ataque com a introdução de novos conceitos como os contratos inteligentes usados para o desenvolvimento de aplicações descentralizadas. Esta dissertação analisa potenciais vetores de ataque e de que forma as blockchains estão protegidas contra os mesmos, focando-se especialmente em ataques de eclipse na rede Ethereum. Explorando trabalhos já realizados nesta área, o objetivo principal é perceber se estão a ser aplicadas medidas de segurança que impeçam estes mesmos ataques e se estas medidas de segurança são eficientes em preveni-los.

O uso de simuladores foi importante para perceber como os nós se comportam em determinadas circunstâncias. Contudo, alguns simuladores não permitem interatividade com o utilizador e por vezes falta de suporte, o que limita o trabalho a ser realizado.

Palavras-chave: Blockchain, Security, Ethereum, Bitcoin, Node, Proof-of-Work, Proof-of-Stake, Vulnerabilities.

Contents

Abstract **i**

Resumo **ii**

Contents **v**

List of Tables **vi**

List of Figures **vii**

Listings **viii**

Acronyms **ix**

1 Introduction **1**

1.1 Demystifying blockchain 1

1.2 Types of blockchains 2

1.3 Blockchain Properties 2

1.4 Types of nodes 3

1.4.1 Full nodes 4

1.4.2 Lightweight nodes 5

2 Background **6**

2.1 Bitcoin 6

2.1.1 Proof of Work 6

2.1.2 Joining the network . . . 7
2.1.3 Network storage . . . 7
2.1.4 Maintaining Connections . . . 8
2.1.5 Disseminating Information . . . 9
2.2 Ethereum . . . 10
2.2.1 Proof of Stake . . . 11
2.2.2 Ethereum accounts . . . 12
2.2.3 Ethereum Virtual Machine . . . 13
2.2.4 Ethereum State Transition Function . . . 14
2.2.5 Messages and Transactions . . . 15
2.2.6 Transactions . . . 15
2.2.7 Peer Discovery/Networking Layer . . . 17
2.2.8 Ethereum's network storage . . . 19
2.2.9 Ethereum Wire Protocol . . . 21
2.2.10 Gas . . . 24

3 Vulnerabilities **25**
3.1 Vulnerabilities in the Bitcoin network layer . . . 25
3.1.1 DNS Seeds . . . 25
3.2 The 51% attack . . . 25
3.3 Vulnerabilities Ethereum in the network layer . . . 26
3.3.1 Unlimited nodes creation . . . 26
3.3.2 Public Peer Selection . . . 26
3.3.3 Sole block synchronization . . . 26
3.4 Eclipse attacks . . . 27

4 Related Work **28**
4.1 Eclipse attacks by monopolizing connections . . . 28
4.2 Eclipse by table poisoning . . . 29

4.3 False friends eclipse attack 30
4.4 lookup function 32
4.5 Logdist Function 33

5 Practical Work **34**

5.1 Planning 34
5.2 Simulators 34
5.2.1 Ethereum-shadow 35
5.2.2 Blockchain Simulator 37
5.3 Ethereum Official Testnets 40
5.4 Running a node 40
5.4.1 Execution and consensus client 40
5.4.2 Syncing the network 41
5.4.3 Initializing the clients 42

6 Conclusion **44**

6.1 Limitations 45
6.2 Future Work 45

Bibliography **47**

List of Tables

2.1 Ethereum Layers. 11

List of Figures

1.1 Public Blockchain 2
1.2 Private Blockchain 2
1.3 Consortium Blockchain 3
1.4 Types of Nodes 4

2.1 How peers are selected 9
2.2 Block propagation [27] 10
2.3 Ethereum accounts 13
2.4 Ethereum Mapping Account 14
2.5 Ethereum world-state 15
2.6 Transaction State Change 15

3.1 Example of eclipse attack 27

5.1 Network topology of test scenario 1 38
5.2 Network statistics of test scenario 1 38
5.3 Network statistics of test scenario 2 39

Listings

Acronyms

ASIC Application-Specific Integrated Circuits

SPV Simple Payment Verification

EOA Externally Owned Account

EVM Ethereum Virtual Machine

ECDSA Elliptic Curve Digital Signature Algorithm

DHT Distributed Hash Table

RLP Recursive Length Prefix

TD Total Difficulty

P2P Peer to Peer

PoS Proof of Stake

PoW Proof of Work

BLS Boneh-Lyn-Shacham

SCA Smart Contract Account

DoS Denial-of-Service

ENR Ethereum Node Record

VM Virtual Machine

ENR Ethereum Node Record

ISP Internet Service Provider

NAT Network Address Translation

DNS Domain Name System

Chapter 1

Introduction

1.1 Demystifying blockchain

Blockchain is an emergent technology that gained a lot of attention when introduced in 2009 [42] by an unknown person who goes by the name of Satoshi Nakamoto. This paper describes the Bitcoin cryptocurrency and the technology behind it to work as the first decentralized digital payment system/currency, which allows individuals to make payments without the need to rely on a third party. Since then, multiple projects have been introduced – a significant amount of them aim to improve efficiency on some components, address possible deficiencies, or bring out new concepts and functionalities like the Ethereum [21] project which allows the development of decentralized applications on top of it and smart contracts, a piece of immutable code that runs on the EVM (Ethereum Virtual Machine) when certain conditions are met.

Blockchains are non-stop growing ledgers [43] that group a set of transactions into a block. Every block is hashed and stores the previous block's hash value since the genesis block, thus forming a chain of blocks, also known as the blockchain. After adding a block to the blockchain, it is no longer possible to delete it, meaning that adversaries are unable to quash transactions published on the blockchain. This makes the blockchain censorship resistant [55].

An important characteristic of this technology is the decentralization it aims to provide. This means that the blockchain data is not stored centrally. Peers in the network store small or full copies of the ledger and maintain it. To counter the inexistence of a central authority that validates blocks and transactions, the network implements a consensus protocol, which allows every peer to agree on the current state of the blockchain.

Blockchains are known to be overlay Peer to Peer (P2P) networks and so nodes need to communicate information to each other. Nodes will receive and send information to the nodes they are able to establish a connection. This can raise concerns if nodes are only surrounded by malicious actors that deliberately send incorrect information to take advantage of a victim and harm them in some way.

1.2 Types of blockchains

The most well-known blockchain systems, which are Bitcoin and Ethereum, are both public blockchains, however, this is not a unique approach. Blockchains can be also private or consortium. [49] [22]

- Public Blockchains: There is no central authority in the network. The system does not have a set of constraints that need to be fulfilled when joining the network and there is not a fixed number of consensus members. Any node can join or leave the network freely.

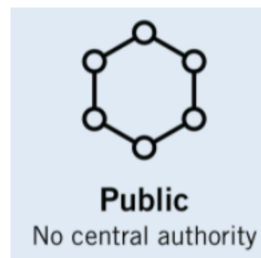


Figure 1.1: Public Blockchain

- Private Blockchains: In private blockchains, there is a central authority that controls the network and imposes restrictions on nodes that aspire to join the network meaning not everyone is free to enter. The consensus members are usually pre-defined and operate with the assumption that everybody knows each other in the network.

Figure 1.2: Private Blockchain

- Consortium Blockchains: They work similarly to private blockchains, however, they are more decentralized due to being controlled by a group or organizations instead of a single central authority.

1.3 Blockchain Properties

The blockchain implements some computer technologies that allows it to work smoothly and stand out from other systems. Some of those are the distributed data storage since nodes in the network need to keep and make available a copy of the ledger, networking (managing nodes on

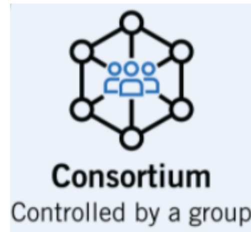


Figure 1.3: Consortium Blockchain

the network and their connections), consensus algorithms, which are extremely important to reach an agreement on the current state of the blockchain, and encryption algorithms to facilitate ownership management.

It also attracts attention for having the following properties [25]:

- **Decentralization** - The core property of blockchain technology is its decentralization. Being a decentralized technology allows for multiple nodes to be accountable on the blockchain and to agree on a consensus that ensures persistence across the network.
- **Tamper proof** - To successfully deviate the blockchain from its main chain, an attacker must control 51% of the whole network. This reveals to be extremely resource expensive and the overall return might not be as rewarding as maintaining an honest posture on the network.
- **Traceability** - All blocks are stored in the blockchain and hashed with the previous block hash which performs the blockchain data structure. It is possible to trace all blocks and transactions made on the network from the last to the very first.
- **Autonomy** - The consensus protocol implemented by the blockchain allows multiple nodes to record and update data in a trustworthy environment.

1.4 Types of nodes

A node is simply a participant in the blockchain technology that when connected to many other participants forms a network. These nodes can assume diverse types, each assigned with different roles and functions in the network. [33]

The two main types of nodes are full nodes and lightweight nodes. Full nodes have a server-like behaviour on the network and more storage capabilities, being able to provide others with blockchain data. On the other hand, lightweight nodes have fewer storage capabilities and are required to connect to full nodes for the purpose of downloading blockchain data. Figure 1.4 depicts a visual representation of the different node types within the blockchain network, organized in a tree structure.

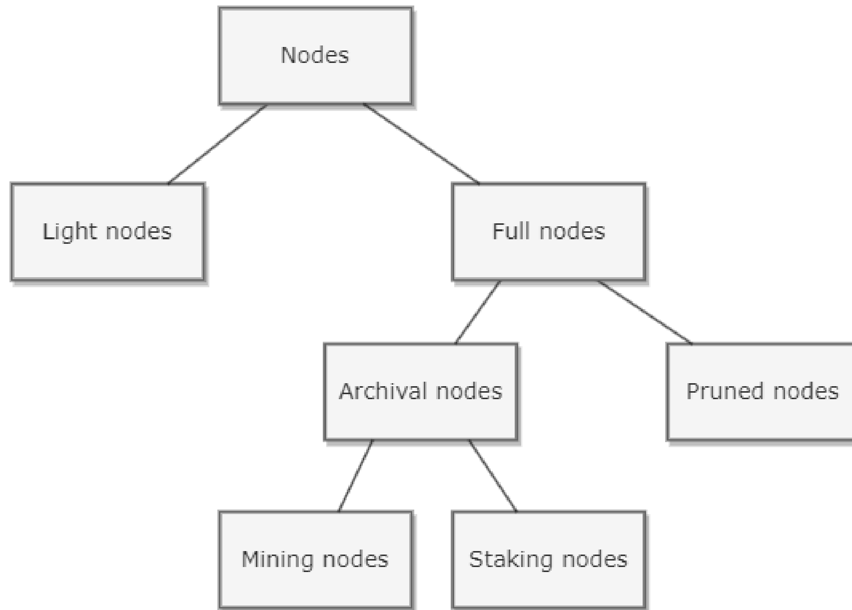


Figure 1.4: Types of Nodes

1.4.1 Full nodes

Full nodes have a very important role in maintaining the blockchain and achieving consensus over the network. They store a copy of the blockchain, validate transactions, and vote on proposals to change the network. They are also the ones from which other nodes, including themselves, download the blockchain data. Full nodes are subdivided into different categories and subcategories [33] [32], as described in figure 1.4.

- **Pruned full nodes:** The pruned full nodes are functional working full nodes with storage constraints. Due to the limited capacity of storage, these nodes download and validate every block until their memory is full. Once this happens, the oldest blocks are dropped and only the headers are maintained. This process happens repeatedly as pruned full nodes drop the oldest blocks to validate new ones as needed. Since they are considered full nodes, they are still capable of validating transactions and being involved in the consensus.
- **Archival full nodes:** These nodes are similar to the pruned ones in the sense that they also store a full copy of the blockchain. However, as storage constraints do not affect archival nodes, these nodes are able to maintain a copy of every block added to the blockchain without having to drop the oldest blocks. Within the realm of archival nodes, it is possible to distinguish two more types: Mining nodes and staking nodes.
  - **Mining nodes:** The mining nodes are on the network aiming to solve a cryptographic puzzle, so they can append a block to the blockchain. Solving the cryptographic puzzle proves that the node has put in the effort required to create a block. The goal is to be the first one to solve this problem taking advantage of high-performance hardware components such as Application-Specific Integrated Circuits (ASIC), which

is mainly used for bitcoin mining. Once the puzzle is solved, the new block will be disseminated through the network and validated by other full nodes. If consensus is achieved, the mining node, also known as "miner", will be rewarded for the effort put into solving the problem with a fee defined by the network.

– **Staking nodes:** Staking nodes are participants of networks following a different consensus algorithm, the Proof of Stake (PoS). Instead of leveraging computational power to create a new block and append it to the blockchain, nodes must stake a certain amount of their owned coins to be able to create and validate blocks. The node to be chosen to append the block will depend on several factors such as coin age (coin holding time), coins owned, number of total stakers, and a random factor.

1.4.2 Lightweight nodes

The lightweight nodes, also known as light nodes, are nodes with very few storage capabilities. They are dependent on full nodes to provide them with the necessary blockchain data. They only keep records of the most recent block headers and are still able to execute transactions despite their constraints regarding storage and computational power.

Chapter 2

Background

2.1 Bitcoin

Bitcoin is a distributed ledger technology [41] that was introduced to the world in 2009. The main purpose of bitcoin is to allow users to execute transactions over a P2P network in a decentralized manner, without the need to rely on third parties. It stores blocks of transactions in a public ledger maintained by active peers on it. As it runs on a permissionless environment, nodes are able to freely join the network and leave it at any time. The papers [27] and [47] provide detailed explanation on the processes a node go through when joining the bitcoin network.

2.1.1 Proof of Work

Bitcoin adopts in its blockchain the Proof of Work (PoW) consensus algorithm. The PoW concept was inherited by the one introduced in [17] as a Denial-of-Service (DoS) countermeasure, specifically thought to prevent email spamming. Users would have to expend computational power over a period of time to solve a cryptographic problem.

In Bitcoin, nodes known as "miners" must generate a proof of work to be able to append a block to the blockchain, a process known as "mining". These nodes must solve a difficult hash problem that consists of finding a hash with a certain number of leading zeros. The amount of leading zeros is considered the network difficulty or the PoW difficulty and is adjusted every 2016 blocks [54]. Nodes compete among themselves to be the first to solve the hashing problem, which gives them the opportunity to append a block. Calculating this hash involves numerous calculations, which translates into computational power, thus giving an advantage to the ones who have more computing resources.

Once a node "mines" a block, it must propagate it through the network and receive a reward in the form of bitcoins (BTC) for the computational work performed.

2.1.2 Joining the network

When a node joins the network for the first time, it must synchronize its local chain with the current blockchain view held by other peers in the network, however, at the moment of joining the network there are no known peers [31]. To do so, the new node can query some DNS seeds which are held by the bitcoin community. These seeds are DNS servers hardcoded in the bitcoin client that hold a list of peers that are accepting connections [5].

During the communication with the DNS servers, nodes might find some advertisement messages not to be reachable, this is because there is no specific way to leave the network. To decrease the probability of receiving DNS responses with addresses of inactive bootstrapping nodes, Bitcoin integrates dynamic DNS seeds, which are more likely to reach active peers. These Dynamic DNS seeds can automatically trace the network to get active peers while static DNS seeds need to be updated manually, making them more susceptible to responding with IP addresses of inactive nodes. There are two scenarios in which nodes might recur to DNS seeds, the first is when they join the network for the first time and the second is when they reconnect and fail to establish more than two connections within 11 seconds [34].

Nodes do not need to solely rely on DNS seeds to learn about new peers. Bitcoin Core, the most common Bitcoin client, keeps a persistent database with a set of peers to which a new node can establish a connection. Another method to find nodes' addresses is by listening to advertisement messages that are at times broadcasted to the network [47] or asking neighboring nodes.

Peers' network information is shared through ADDR messages. These can contain up to 1000 addresses and nodes that do not respect this boundary get blacklisted. ADDR messages are usually unsolicited, however, when an outgoing connection is established, the node can solicit up to three ADDR messages. To advertise itself to the network, a node shares its own information by sending ADDR messages to its connected peers, which will be forwarded to their connected peers.

2.1.3 Network storage

Nodes in the network store the public IP addresses of other nodes. Those can be stored in two persistent databases: the *tried* and *new* tables [34]. The persistence of these databases allows nodes to still acknowledge previously discovered peers if they reboot. The *new* table has 1024 buckets and holds IP addresses of peers to which the node has not yet established a successful connection. The *tried* table has 256 buckets and stores the IP addresses of peers to which the node has already established a connection with success [31]. Both tables are comprised of 64 entries for each bucket, which store IP addresses, ports, timestamp of the last connection, and the last time the respective node was seen on the network.

When a new node is discovered, it is first added to the *new* table through the GetNewBucket

function. This function determines the bucket where a node must land. If the bucket is full, an eviction process is run and evicts all the nodes that are older than one week, have failed to connect 7 times in a row or have their clock 10 minutes ahead of the current time.

Whenever a successful connection is established, the client runs a GetTriedBucket function to determine the appropriate bucket for that address in the *tried* table. If that address is already present in the bucket, its timestamp is updated. In the case that it is not but the bucket is full, four random addresses will be selected and the oldest from the four gets evicted and moved to the *new* table [28]. In an attempt to increase addresses on the *tried* table, the node periodically tries to connect with peers in *new*.

2.1.4 Maintaining Connections

Every node in the network has the capacity to establish outgoing or incoming connections with other peers over unencrypted TCP. The identity of a peer in the bitcoin network is its public IP address. Every node with a public IP address can initiate up to a default of 8 outgoing connections and receive up to 117 incoming connections, while nodes with private IP addresses can only establish 8 outgoing connections [34]. Nodes that are accepting unsolicited connections express their availability by broadcasting an ADDR message through the network saying they are accepting connections [27]. A node can choose to drop inbound connections but not outbound, unless the peers are blacklisted. Every time a node has less than 8 outgoing connections, it triggers a method for establishing the missing connections, which can happen upon a restart or if an outgoing connection is dropped [28].

Connected peers send regular messages to check on aliveness and if no response is received within 90 minutes, the peer is assumed to be disconnected from the network.

2.1.4.1 Selecting peers

Outgoing connections are established every time a client reboots or when a connection gets dropped by the network. Bitcoin nodes do not usually drop outbound connections unless they receive something that deviates from the normality (e.g., receiving addr messages that exceed the total addresses limit) [34].

The first step in choosing a node to establish an outgoing connection is deciding whether to choose an address from *new* or *tried* table. Bitcoin prioritizes selection from *tried* in case the node has few outgoing connections established, or the table has a big amount of nodes. After deciding on the table, a random node n is selected and accepted with probability $\min(1, \frac{2^{rejected}}{1+(Timestamp[n]-currentTime)})$. In this process, nodes with more recent timestamps are favoured and rejected is set to 0. If the node is accepted, a connection attempt is started. Otherwise, a new node is selected and rejected is increased. If the connection is attempted but fails, the process returns to step one [28].

Figure 2.1 helps to visualize the process detailed above.

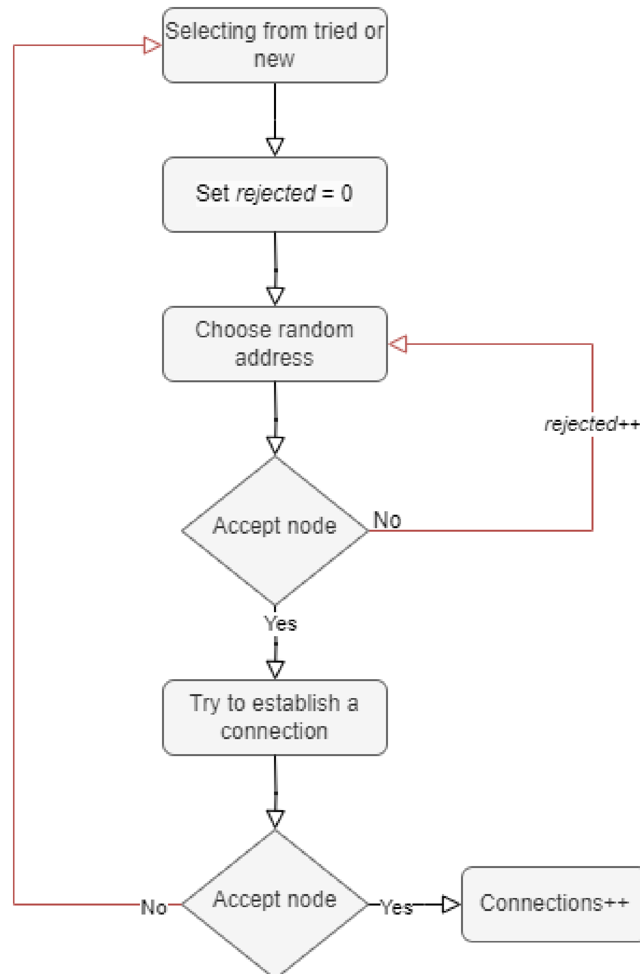


Figure 2.1: How peers are selected

2.1.5 Disseminating Information

Nodes in Bitcoin disseminate information through the gossip protocol [37]. Nodes can announce transactions and block hashes on their local chains with an inv (inventory) message. This message alerts the neighbour nodes that blocks and transactions have been validated and are ready to be transmitted. Another purpose of using this method is to avoid sending a block or transaction to a node that already knows it [27]. Nodes that are missing the block or transactions might respond with a getdata message requesting the full block or with a getheaders message requesting some block headers on top of the chain and follow with a getdata message as illustrated in 2.2. This is called the Standard Block Relay method.

When a miner discovers a new block, he does not need to send the inv message because if the block was discovered by him that means that no one else knows about it. In this case the block is shared with an unsolicited block push. The block's discover sends a direct block message to its neighbours because he knows for sure that they do not have the block on their local chain.

The last method for sharing a block is via a Direct Headers Announcement. Some nodes might prefer to receive a message with the block header instead of the inv message so the sender just sends a headers message instead. The receiver verifies the header and decides whether to send a getdata message or not. The preference for headers instead of inv message can be shared at the moment of establishing a connection [5].

Propagating a message involves previously validating the information that is going to be propagated, and this introduces in the network a propagation delay, which is the time a node takes to validate a block or a transaction, added to the time it takes to propagate and reach the target.

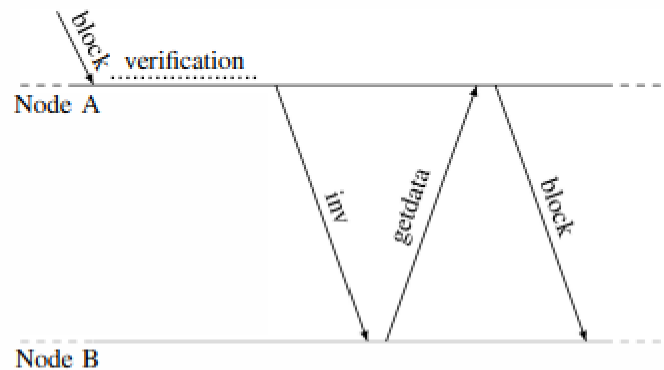


Figure 2.2: Block propagation [27]

2.2 Ethereum

Ethereum is one of the most impactful blockchain technologies operating today. It brings people together from all over the world through the power of an innovative concept. Its release in 2015 stood out when introducing new functionalities and pushing its way to being more than just a cryptocurrency.

Described in [21] by Vitalyk Buterim, Ethereum is a project designed to facilitate the creation of revolutionary decentralized applications, also known as dApps. To provide this service, the project introduced Smart Contracts, one of the core characteristics of this network that makes it excel from the others. Developers can build decentralized applications on top of the Ethereum blockchain through the development of pieces of code that are able to execute themselves when determined conditions are met – these are known as Smart Contracts. These programmable pieces of code run in a virtualized environment called the EVM (Ethereum Virtual Machine), explained in more detail in section 2.2.3

Looking at the Ethereum architecture, it is possible to denote five different layers, as illustrated in table 2.1 [30].

The focus of this paper is on the Network Layer, however, the following chapters make a brief

1	Data Layer	Data block, chain structure, hash function, asymmetric encryption, timestamp, merkle trees, etc. All of these important components ensure the reliability of the network.
2	Network Layer	Responsible for handling peer-to-peer connections. Specifies how a node joins the network and get to know their neighbours and is responsible for establishing communication channels, so nodes can send and receive blockchain data information.
3	Consensus Layer	Ensures the consistent state of the blockchain across all peers. Everyone in the network must agree on what is being seen.
4	Contract Layer	Scripts, Algorithms, and Smart Contracts to allow instructions to be run automatically and determinately. This layer is executed in the Ethereum Virtual Machine.
5	Application Layer	Applications built on top of the Ethereum blockchain are deployed in the application layer.

Table 2.1: Ethereum Layers.

introduction and explanation of some concepts such as Ethereum Accounts and the EVM.

Regarding the Network Layer, the present research explores the process of a new node joining the network as well as how nodes can establish connections with each other and receive and transmit blockchain information. The purpose is to understand if the network is vulnerable to eclipse attacks, or how can a malicious actor can compromise or pollute an honest node's connection. Section 2.2.7 describes more in-depth the process of peer discovery, which is the main point of attack when trying to eclipse a victim.

2.2.1 Proof of Stake

With the introduction of Proof of Stake to replace the Proof of Work consensus algorithm, Ethereum changed the way blocks are proposed to the network and the actors that participate in that process. Miners are no longer part of the equation since that nodes are not competing with each other to be the first to solve some difficult cryptographic puzzle leveraging their computational power. The PoS introduces a core actor for the achieving consensus, known as validator. Validators are responsible for validating blocks and transactions an occasionally append a new block to the blockchain.

Validators are randomly chosen to be head of consensus and their time on it is measured on slots and epochs. A slot is a timeline of 12 seconds and 32 slots form an epoch. Every slot represents an opportunity to add a new block to the blockchain, a pseudo-random algorithm runs to choose a validator to be the proposer of a block. The chosen validator collects a group of transactions, executes them and wraps them into a block. The block is then propagated to a specific group of validators that must execute all transactions and validate the block giving their attestation. The group of validators responsible for validating blocks is changed at every slot for

managing the network load.

To be a validator, one has to stake its own ETH, a total amount of 32ETH. This helps preventing malicious behavior from validators like continuously appending their favored blocks because since validators are staked on the network they risk to have their ETH destroyed in case of bad behavior.

2.2.2 Ethereum accounts

The Ethereum network comprises two types of accounts: the Externally Owned Account (EOA), which can be controlled by owning its private keys, and the contract accounts. Unlike EOA, these are related to smart contracts deployed on the network and thus are controlled by the code written in those. Both accounts can interact with smart contracts deployed to the network and are able to exchange Ethereum tokens.

The four following fields, as depicted in figure 2.2.2, compose the Ethereum Account [50]:

- A **nonce** that starts at 0 for each EOA and is incremented each time that account performs a transaction. By doing so, it prevents the same transaction from being processed twice. For contract accounts, it indicates the total of contracts created by the account.
- The **balance** - the amount of ether an account has expressed in WEI, a denomination of ETH.
- The **codeHash** - the hash of an account's code on the Ethereum Virtual Machine (EVM). It is a code fragment programmed to execute a set of operations. This code is immutable and thus can never be changed, it gets executed once the account receives a message call. In the case of an EOA, since there is no contract deployed on the network, the codeHash value is the hash of an empty string.
- The **storageRoot**, which is a 256-bit hash representing the entire storage of a single account (root hash of the Account Storage Trie). All the contents of a smart contract and the results of its execution are stored in this persistent storage. This field is usually empty by default and remains empty for externally owned accounts.

To be able to own an EOA, a user must generate a cryptographic key pair. This process involves generating two cryptographic keys: a private key and a public key. The first is used for signing transactions whilst the second is mainly used for verifying transactions. The public key is derived from a 64 hexadecimal character private key recurring to the Elliptic Curve Digital Signature Algorithm (ECDSA). The public address of an account is determined by adding 0x to the last 20 bytes of the Keccak-256 hash of the public key [10]. This address is public to the blockchain and serves as a unique identifier.

The contract accounts also a hexadecimal address of 42 characters instead of 64. The contract address is given by an association between the contract owner's address and its respective nonce.

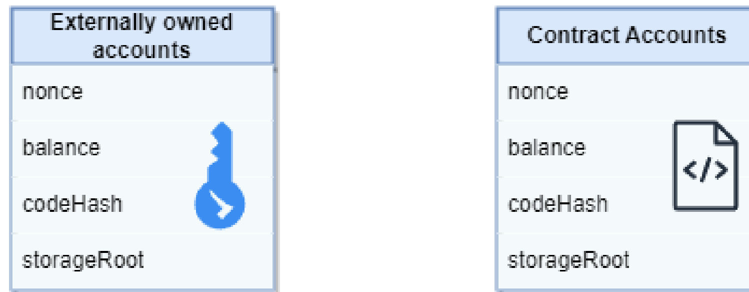


Figure 2.3: Ethereum accounts

The main difference between EOA and contract accounts is that EOA are free of charge, while contract accounts will have a cost since they are consuming network storage. Besides following a cryptographic private/public key logic, EOAs are able to initiate transactions on their own. Contract accounts are unable to initiate transactions, they can only respond to transactions if they receive one. Furthermore, they follow the logic of the smart contract code instead of being associated with private keys.

With the switch from proof-of-work to proof-of-stake, a new type of keys were introduced to the network. These are called BLS keys due to being generated with the Boneh-Lyn-Shacham (BLS) signature algorithm [16].

2.2.3 Ethereum Virtual Machine

The EVM is a key component that permits the Ethereum Ecosystem to be considered a 2.0 Blockchain model. While Blockchain 1.0 is related to payment systems, the 2.0 Blockchain introduces smart contracts, which allow Decentralized Applications to run on top of blockchains [24]. These smart contracts are executed on a computation engine not very dissimilar from virtual machines, known as EVM. This machine plays a pivotal role in transitioning from one state to another with each new block. It belongs to the execution model that defines how the blockchain state is altered based on bytecode instructions and a small set of environmental data [21]. The EVM follows a stack-based architecture with a word size of 256-bit, specifically chosen to streamline operations related to the keccak256 hash scheme and elliptic-curve signature algorithms, and is composed by the following:

- Stack: 32 bytes fields with a maximum size of 1024.
- Memory: An infinite expanding byte array stored separately in a virtual ROM environment, expanding this byte array implies paying more gas.
- Storage: Persistent memory for contract storage.
- Environment variables: These variables are stored, so the virtual machine can access the block number, mining difficulty, previous block hash, and others.

- Logs: Event registers.
- Sub-calling: Opcodes for calling other contracts.

It is considered to be a quasi-Turing-complete state machine, the term "quasi" is used because Ethereum uses the amount of gas available to solve the halting problem. The amount of computation done is always limited by the amount of gas available [50]. The EVM defines the rules for computing a new valid state for the blockchain from block to block [11] and offers a runtime environment for smart contracts to run [23].

Smart contracts are programmed in high-level languages such as solidity and compiled into byte-code so they can run on the EVM [44]. Once a developer writes a smart contract, he deploys it to the network, then the contract's code is compiled to byte code and distributed through all the machines, which means that every node on the network running an EVM will have a copy of the contract. A user participating in the network can run its EVM on geth, which is the most popular software client among the Ethereum network nodes.

2.2.4 Ethereum State Transition Function

Ethereum is described in [21] as a transaction-based state machine. This is because the Ethereum network processes transactions in order to change the current state of the blockchain, which implies that a transaction represents a valid arc between two states [50]. A state transition function described as APPLY(S,Tx) -> S' is applied to change the state. Given a block of transactions, a transaction T, and the state S, a new state S' or an error will be generated. The Ethereum world state is defined as a mapping between addresses and account state [48]. Figure 2.4 illustrates the account state, which maps an address to account fields. As mentioned in section 2.2.2 an Ethereum account state has a nonce, a balance, a storageRoot, and a codeHash, which is the EVM code.

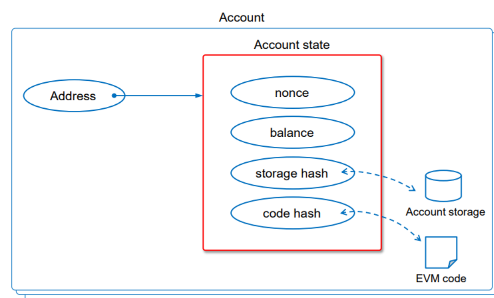


Figure 2.4: Ethereum Mapping Account

In figure 2.5 is illustrated the concept of world-state, which maps addresses to account states to reach the current state of the Ethereum Blockchain.

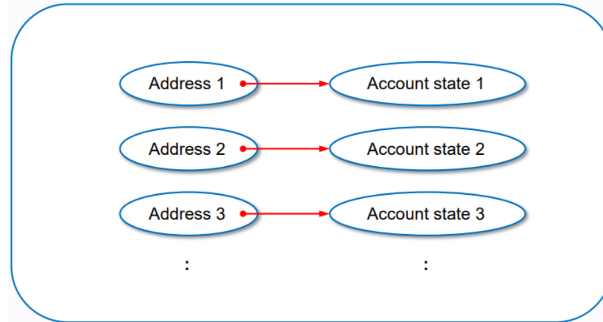


Figure 2.5: Ethereum world-state

2.2.5 Messages and Transactions

A transaction is a signed interaction between accounts [3] whose execution implies changing the state of the blockchain(2.6). Transferring ether from one account to another is seen as the simplest way of executing a transaction. Messages are similar to transactions, however, they are associated with the contracts' ability to send messages to other contracts (e.g. opcodes calls).

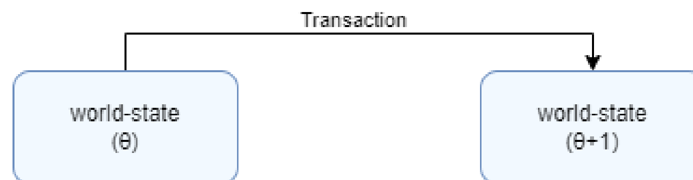


Figure 2.6: Transaction State Change

2.2.6 Transactions

Transactions are sent from an EOA [45] to other EOA or contract accounts. A transaction contains the following fields [50]:

- nonce: corresponds to the sender and is incremented each time he initiates a transaction.
- to: Account of destination, which can be an EOA or a contract account.
- value: In case of an ether transfer, this field specifies the amount Wei correspondent to the amount of ether to be transferred.
- data: An optional data field that is used to specify contract message calls.
- gasLimit: The maximum gas the sender is willing to spend.
- gasPrice: The price to be paid in Wei for each computational step needed to execute the transaction.
- (v, r, s): An ECDSA signature identifying the sender.

Each valid transaction execution involves updating the state of the corresponding accounts and thus updating the blockchain state.

The following steps describe the process that an Ethereum transaction goes through [29]:

(i) The sender constructs and digitally signs a transaction.

(ii) To submit the signed transaction to the Ethereum client, the sender must execute a JSON-RPC call.

(iii) The Ethereum client receives the transaction and, after verifying it, broadcasts it through the Ethereum P2P network.

(iv) Clients that are also miners receive the transaction and add it to their mining pool.

(v) The miner selects and execute transactions from his transaction pool, creates a block and updated the blockchain state. Transactions can be divided into three types. The simplest of transactions is a money transfer, where the specified value is transferred from an EOA to another or to a contact account. For contract creation transactions, the piece of bytecode specified in the transaction input creates a new contract account and is associated to it. For transactions that call a contract, where the recipient is the called contract and the input field specifies a callee contract function, the bytecode associated with the callee contract gets loaded into the EVM.

(vi) The miner tries to solve a PoW that consist in finding a random nonce so that the hash value of the block's metadata is smaller than a certain value, which translates into the difficulty of mining a block. While Bitcoin implements a computation-intense PoW, Ethereum adopts a memory-intense puzzle called "Ethash".

(vii) Upon creating the block, the miner broadcasts it through the P2P network in order for other clients to verify and validate it.

(viii) After verifying and validating the block, the client appends it to the blockchain.

With the recent change in Ethereum from PoW to Proof of Stake (PoS), transactions are now validated by the validators, the actors introduced in the network with the change to PoS.

2.2.6.1 Messages

Messages, unlike transactions, are specific to the execution environment. They are initiated by smart contracts when the call opcode is triggered and are used to interact with other contracts. A message is composed by:

- The message sender.

- The message recipient.
- The amount of ether to be transferred.
- A field for optional data.
- The maximum to be spent on the message.

2.2.7 Peer Discovery/Networking Layer

The networking layer of Ethereum can be separated into two stacks: the discovery stack and the DevP2P stack. The discovery stack manages a set of protocols layered on top of UDP for discovering new nodes in the network, which are implemented on version 4 of the Peer Discovery Protocol. In contrast, the DevP2P stack sits on top of TCP and includes protocols responsible for establishing and managing sessions, such as RLPx and Ethereum Wire Protocol (eth), which plays a crucial role with regard to exchanging information about the blockchain itself [2].

The process of discovering nodes in the Ethereum network is defined by version 4 of the Node Discovery Protocol [6]. Even though it is soon to be changed to version 5, this is the current protocol followed by Ethereum nodes to find other nodes on the network.

Whenever a node enters the network, it has no information about other peers. To be able to connect and acknowledge other nodes in the network, a set of bootnodes is available to be reached. These are made available by a slightly modified version of Kademlia, a Distributed Hash Table (DHT) used to store nodes' information and assist with efficiency on the lookup of identifiers and routing to the corresponding nodes. The bootstrap nodes are trusted to provide new nodes with correct information about a set of honest peers to connect to on the first time the client is initiated, however, for security concerns the boostrap nodes available on the DHT are regularly refreshed. These nodes function is solely to provide newly joined nodes with a set of peers, they do not perform other client tasks such as chain synchronization [2].

As previously stated, Ethereum nodes use a modified version of Kademlia, which is a well-known data structure with aim to optimize the storage and retrieve of data. Its purpose is to help the process of storing and retrieving nodes instead of storing and retrieving blockchain data [20]. Each node has its own DHT that contains the information required to connect to its closest peers, organized in buckets. There are 256-buckets in every DHT with 16 entries each. However, because it is very hard to find nodes closer than a certain bucket, only the first 17 buckets are filled. Nodes are mapped to buckets according to a XOR metric that measures how distant two nodes are from each other. Furthermore, the Ethereum version of kademlia applies a keccaak256 function to the nodeID and this value is represented as the ID in the table, unlike in the kademlia official version where IDs are 160-bit.

The node discovery protocol implements two messages that allow a new node to bond with the bootstrap nodes when it joins the network. By the use of the PING-PONG scheme, a node can send to a boostrap node a ping message containing the following information comprised on a

hash: information of itself, the node to connect to, and an expiry timestamp. After sending a ping message, the node awaits for a pong response containing the ping hash. To be valid, the pong message should match the ping hash of the most recent ping message. After trading these two messages, nodes are bonded and the new node can send FindNode messages to learn about more nodes in the network to bond with and add them to its DHT.

To do so, nodes perform a recursive lookup, which locates the k-closest nodes to a certain target [6]. To initiate a recursive lookup, a node chooses a random target t and sends a findNode message to the 16 nodes closest to the target. Each of the 16 nodes will respond with the closest nodes to the target they know about in a neighbors packet. This process runs iteratively until the lookup initiator encounters the closest nodes to the target from all the nodes received in the response packets. Nodes are mapped to the respective buckets according to a logdist function [20]. Every time a new node x is discovered, it is mapped to the corresponding bucket and added to its tail if not full. In case the bucket is full, the least recently seen node y is pinged to check if it is still alive. In the event of y does not respond, it is evicted from the DHT and node x is added. However, if this node responds, node x is not added to the bucket but can be added to the bucket-specific replacement list, which can hold up to 10 nodes.

During the discovery process, nodes establish RLPx sessions to exchange information and share connectivity information [9]. This transport protocol is based on TCP and employed for Ethereum nodes to communicate with each other [7]. It initiates, authenticates, and maintains connections between nodes. Moreover, RLPx encodes messages using the space-efficient Recursive Length Prefix (RLP) to encode messages.

An RLPx connection starts with an Initial Handshake. This process involves creating a TCP connection and exchanging ECIES keys [39] for nodes to change information securely and privately. The RLPx Initial Handshake is done between the initiator and recipient, which are the node that starts the connection and the one that receives it, respectively. The initiator sends an auth message to the receiver, which verifies it, and responds with an auth-ack message in case the verification succeeds. Afterward, both nodes send their first encrypted frame containing the following information in a Hello message:

- Protocol version
- Client ID
- Port
- Node ID
- list of supported sub-protocols

The main purpose of the hello message is to share the capabilities of the two nodes, which is a list of supported protocols and their versions. In concurrence with the hello message, a node might also send a disconnect message to indicate that the connection will be closed.

2.2.8 Ethereum's network storage

Ethereum has available two data structures for its clients to store information about other nodes [40]. The first one is a long-term database known as db that stores the nodes physically [20] and is persistent across the client reboot. The other database is short-term and non-persistent, thus is empty every time the client reboots. It is called table and is the kademlia-like DHT employed by the Ethereum network for facilitating the process of storing and retrieving nodes.

- **db** - This is a long-term database that stores every node the client has even connected with. It is persistent, which means that records remain after a reboot. It has no limit on size so it can store all the nodes that the client bonded with. The process of bonding is described in section 2.2.8.1 but can be as simple as sending a ping message and receiving a valid pong response. Once this is completed, the client stores in its db the following information on the node: nodeID, TCP port, UDP port, time of last ping sent, time of last pong received, and the number of times that node has failed to respond to a findnodes message. To keep the database updated and exclude nodes that are no longer alive on the network, the client runs an eviction process to discard nodes that are older than one day. The node age refers to the time that has passed since the last valid pong message.

- **table** – A short-term database with a kademlia-like structure, it is the modified DHT implemented by the Ethereum network. It is not persistent and therefore will be empty every time the client reboots [40].

  This database is composed of 256 buckets, each with k=16 entries. Every entry corresponds to a slot that may or may not be available to store a node. The client uses the *table* to keep records of nodeID, IP address, TCP port, and UDP port of every peer in it. Nodes are mapped to buckets according to their distance from the client, this is calculated by the implementation of a logdist function similar to the kademlia XOR metric [53]. This function hashes two nodeIDs to a 256-bit value and calculates the level of proximity r between them. It then maps to the correspondent bucket, which can be described as 256-r.

  Buckets are structured in a most recently seen architecture, where the nodes that responded to ping messages last are moved to the top. The ones on the tail of the bucket are the oldest ones and thus the more likely to be evicted from the *table*.

  While *db* is more directed to store network information, the *table* is designed with the purpose of selecting peers. It runs its own eviction process and discard nodes if they fail to respond to a findnode message more than four times or if they fail to respond to ping messages.

2.2.8.1 How to populate Data Structures

An Ethereum client has two data structures to keep a record of other nodes on the network, the long-term database *db* and the short-term *table*. Although *db* is persistent and does not lose

data across reboots, both databases are empty when the client joins the network for the first time. Despite *db* having six bootstrap nodes hardcoded, it is still considered to be empty as a result of them being only there to provide the new participant with a set of peers to connect to. By acknowledging these peers, the client can initiate communication with them and possibly integrate them into its databases. When the client receives information on new peers from the bootstrap nodes, it needs to bond with them so they can be added to the *db*.

To bond with a node can be as simple as exchanging valid ping and pong messages. After bonding is complete, the client will check if the node is already in its *db*, if it has a record of zero failed responses to findnode requests and if it records a valid pong message received in the last 24 hours. If the node complies with these requirements, the client tries to add it to the *table* [40]. When encountering a new node, as is the case with the set of peers received from the bootstrap nodes, the client endeavours to add them to *table* after a successful bonding. As the client sends ping messages, it can also receive unsolicited ping messages that can turn into a successful bond and further integration in the *table*. To actively search for new peers to connect to, nodes can run the lookup function and acknowledge new nodes which they may or may not establish a connection with.

To populate the buckets in *table*, the Ethereum network relies on logdist function (See 4.5) to map nodes to the according buckets. However, not all buckets are filled, due to the low probability of finding nodes that can be mapped to the lower buckets, Ethereum restricted the bucket usage to the first 17 buckets only [35].

Nodes can be added to buckets through two different functions: addSeenNode() and addVerifiedNode(). The first one is to add a seen node regardless of being live or not to the end of a bucket. The second adds a verified node, i.e., a node confirmed to be live to the front of a bucket. In both functions the node is immediately added to the bucket if it is not full, otherwise, it is added to the replacement list, which can hold up to 10 nodes. Even in cases where a bucket is not full, nodes can be left out if adding them implies not complying with IP restrictions.

For security purposes, verified nodes cannot be added if *table* is still initializing. This implementation prevents attackers from filling a table by sending repeatedly ping messages.

2.2.8.2 Establishing connections

Ethereum's nodes can establish two different types of connections, depending on whether they are exchanging information about the peer-to-peer network or sharing information about the blockchain. While UDP connections are used to fill the *db* and *table* databases, TCP connections are established to perform actions such as chain synchronization, block propagation, or other information about the node itself.

Every single UDP message is timestamped and authenticated with the sender's ECDSA key. To help mitigate replay attacks, the client must drop UDP messages that are 20 seconds older

than the client's local time. Peers in the network can exchange the following UDP messages.

- Ping packet – Message sent to see if the node is online.
- Pong packet – It is a hash of the corresponding ping message.
- FindNode packet – Message used to find nodes close to a selected target.
- Neighbors packet – Response message to the FindNode packet containing the closest nodes to the pre-determined packet, if any. Peers ca sometimes respond with an empty neighbors packet.

TCP connections are also encrypted and authenticated, however, there is a limit on the maximum of TCP connections a node can maintain at any given time. To establish a connection with another peer the client can either receive an unsolicited ping request – an incoming connection – or initiate the connection himself – an outgoing connection. Prior to geth v1.8.0, the only limit imposed on incoming connections was maxpeers, that means that a client can have all of its TCP connection slots filled with incoming connections. This represented an attack vector for eclipse attacks, since the attacker could repeatedly ping the victim and fill all the slots with relative ease. Aiming to mitigate that, incoming connections are now limited to 1/3 of maxpeers, and the default value of maxpeers was incremented from 25 to 50.

Clients establish their outgoing connections in two possible methods: one can make a request to the discovery table or fill its outgoing connections with nodes from the lookup_buffer. If the outbound slots are not all filled, the client will occupy half of the slots with nodes from the lookup_buffer and the other half with nodes from the discovery table selected through the function ReadRandomNodes. In case there is only one slot available, the lookup_buffer and the ReadRandomNodes function are favoured one at a time [35].

2.2.8.3 Seeding Process

Ethereum has implemented a function to check whether buckets are full. It runs upon client reboots and verifies if the table is non-empty. The seeding process can be described in three stages. If the table is empty, the client queries 30 nodes in the db with a maximum age of 5 days, also known as seeding nodes. This is the first stage. It then starts a lookup on itself to find neighbor nodes - this is the second stage. In the last stage, the client performs three lookups on random targets.

2.2.9 Ethereum Wire Protocol

The Ethereum Wire Protocol, also known as 'eth', is a protocol that runs on top of RPLx and has the main purpose of providing seamless interaction between peers. It defines rules for

data exchange mechanisms so nodes in the network can remain up to date with their blockchain data.

This protocol includes multiple messages such as NewBlockHashes, GetBlockHeaders, and NewBlockHeards, all describe in more detail in the following subsections. These messages are exchanged during the execution of crucial methods, for example, Chain Synchronization, Transactions, and Block Propagation.

2.2.9.1 Chain Synchronization

Nodes participating in the 'eth' protocol are expected to know the full chain of blocks. This knowledge can be obtained by downloading the full copy of the blockchain from other peers in the network.

When two peers establish a connection, they must both send a Status message containing information the blockchain and protocols supported. Among this information, nodes also share the total difficulty (TD) and the hash of the latest block added to their blockchain copy.

If the two peers do not agree on the blockchain data sent in the Status message, the peer with the smallest TD value starts to download the missing block headers via a GetBlockHeaders message. Upon receiving the block headers, the proof-of-work associated is verified. Once validates, the block bodies are requested via a GetBlockBodies message. After receiving the block bodies, these are executed through the Ethereum Virtual Machine and the state tree and transaction receipts are recreated.

2.2.9.2 Transaction Exchange

Miners have a pivotal role in the blockchain network since they are responsible for picking transactions and wrapping them onto a block for further mining. To achieve this purpose, every node on the network needs to share exchanged pending transactions with miners. These transactions are located in the client's transaction pool and may contain up to a thousand transactions.

When two peers establish a connection, they need to synchronize their transaction pools. To do so, both peers exchange a NewPooledTransactionHashes announcement including all hashes in their local pools. Upon receiving this message, each peer conducts a filtering process to request only the transactions not already present in their local pool via a GetPooledTransactions message.

If a client learns new transactions, it should propagate them, resorting to two types of messages: "Transactions" and "NewPooledTransactionHashes". A Transactions message sends the complete transaction object to a small fraction of connected peers chosen at random. The remaining peers will learn about the transactions via a NewPooledTransactionHashes, which

announces them.

To avoid flooding the network with unnecessary messages, nodes should keep a record of recently relayed transaction hashes to avoid sending back a transaction to a peer that was already aware of it [8].

2.2.9.3 Block Propagation

In order to form a block, nodes should aggregate transactions in it and try to mine it. If a node is able to successfully mine a block, it must propagate it through the network so all the nodes learn about the block. When a node receives a NewBlock announcement message, it needs to confirm that the received block's predecessor is the highest block on its local chain. Otherwise, it means that the client's local chain is not up-to-date and needs to be synchronized.

The process of synchronization involves requesting the missing blocks from other peers to update the local chain to the longest valid chain in the network. The following steps enlight how the block propagation works:

1. When a node receives a NewBlock message announcing a block, it must check if the block's predecessor is the highest block on its current chain.
2. If not, that means that the node's local chain is outdated and needs to be synchronized with the current latest state of the network.
3. The node can synchronize its local chain by requesting the missing blocks to its connected peers.
4. After synchronizing its chain, the node validates the proof-of-work value.
5. If successfully validated, the node sends the block in a NewBlock message to some of its connected peers, usually the square root of the total connected peers.
6. Following this validation, the client attaches the block to his local chain and executes all transactions included. Computing all these transactions results in a block's post-state, which must match the block's state root value.
7. Upon complete processing and validating the block, the node sends a NewBlockHashes message to the known peers not previously notified. Peers that were not notified of the block in a NewBlock message by other nodes in the network might request the full block.

Nodes must keep track of block announcers to prevent announcing a block to a peer that has previously announced the same block.

This process is utilized in Ethereum PoW and PoA networks. After the Merge, 'eth' protocol does not handle block propagation anymore and will be excluded from the protocol in future versions [8].

2.2.10 Gas

As stated in 2.2.3, Ethereum's scripting language is considered quasi-Turing-Complete because it can almost perform the computations of a Turing machine. Unlike the Turing complete machine, Ethereum's scripting language imposes a limit of execution through the use of "gas", which is a measure of computational work, thus the use of the term "quasi".

Turing-Complete machines face a challenging problem, which is the halting problem[1]. To address this issue, Ethereum imposes a maximum "gas" value on every transaction to limit the amount of computational done during the execution of that transaction, and consequently a smart contract.

The adoption of the gas mechanism serves as a countermeasure against unintentional errors or malicious purposed actors. This is because not only an attacker could program a smart contract to run forever but an honest programmer could mistakenly indulge in the same error [38]. Also, the fact that participants in the network have to pay a gas fee for executing a transaction helps prevent network spamming and Denial of Service attacks [1].

[1]The halting problem states that it is generally impossible to determine whether a program, or in this case, a smart contract, will run forever.

Chapter 3

Vulnerabilities

3.1 Vulnerabilities in the Bitcoin network layer

3.1.1 DNS Seeds

When a node joins the network it has at its disposal some DNS seeds that it can query to learn about full peers on the network from whom it can download blocks and transactions. However, the DNS seed responses are not authenticated and thus peers should not solely rely on that. Some seeds might be controlled by a malicious seed operator or results from seeds can be intercepted by a middle-man on the network and return IP addresses of malicious nodes instead [46]. Relying exclusively on DNS seeds makes the nodes vulnerable to being isolated on the attacker's network.

3.2 The 51% attack

The Bitcoin network adopts a Proof of Work (PoW) consensus algorithm, previously described in 2.1.1. The characteristics of PoW make it susceptible to 51% attacks. This attack consists of a miner or a group of colluding miners having more hash power than all the other miners in the network, thus having more chances to find a new block [36]. Moreover, controlling 51% of the network hashing power also allows the attackers to modify the blockchain by rewriting transactions and blocks and empowers attacks such as denial-of-service, eclipse, or double-spending [26].

It is extremely difficult for a single node to control 51% of the network hashing power, so it is more probable that this kind of attack happen in a colluding environment, as are mining pools. Mining pools are composed of multiple miners that share computing resources to successfully mine a block. Mining pools are not necessarily evil, but they still pose a threat to decentralization because a pool of honest miners can still obtain 51% of the hash power.

As a countermeasure to this attack, a two-phase PoW consensus is proposed in [19] attempting

to prevent the formation of large mining pools. One other proposal to mitigate 51% attacks is a PoW based on mining groups and random selection [18]. The authors propose to divide miners into groups and select a specific group to mine the subsequent block. Once the block is mined and propagated through the network, all the nodes that validate the block and transactions must also validate if the creator of the block belonged to the correct group.

3.3 Vulnerabilities Ethereum in the network layer

3.3.1 Unlimited nodes creation

This vulnerability arises from the fact that Ethereum nodes are identified by a nodeID and Geth client does not correlate IP addresses and nodeIDs thus an attacker the ability to generate as many nodeIDs as needed to monopolize the victim's incoming and outgoing connections. Furthermore, the attacker can simply keep generating nodeIDs and find the most suitable ones in a short time and with standard computer resources. Usually, the first step to take when executing an eclipse attack is to generate a huge amount of nodeIDs on a single machine. To prevent an attacker from exploiting this vulnerability, the Geth developers could include IP addresses in the process of generating a new NodeID [24].

3.3.2 Public Peer Selection

Each node participating in the Ethereum network possesses a Kademlia-like Distributed Hash Table (DHT) to store nodes and serve as a routing table. Each DHT has 256 buckets, where nodes are mapped according to their distance, following a logdist function described in Section 4.5. This process of mapping nodes to buckets is public [24], which makes it trivial for an attacker to predict what bucket a crafted node will land in. This can compromise the *table* [31], especially because, as seen in Section 3.3.1, an attacker can generate as many nodeIDs as necessary and use the most convenient ones to exploit this vulnerability.

3.3.3 Sole block synchronization

The process of synchronizing the blockchain also raises some concerns. When a node propagates a block to another node, it sends a block header containing the Total Difficulty (TD) of the block along with the cumulative TD of all blocks, reflected in totalDifficulty. This last one represents the total difficulty of the blockchain.

If client A receives a block from client B and realizes that its totalDifficulty is smaller than the totalDifficulty of the block received, client A perceives that it does not have an up-to-date copy of the blockchain and initiates the synchronization process with client B. To address network load concerns, each client is only allowed to synchronize with one client at a time. This restriction

permits client B to deliberately delay the process of synchronization causing client A to reject subsequent blocks, making it more vulnerable to double-spending and DoS attacks.

This attack is described in [52] and could be mitigated if clients are allowed to synchronize with multiple nodes at the same time [31].

3.4 Eclipse attacks

The P2P network inherent to a blockchain faces susceptibility to eclipse attacks. These attacks occur when an adversary can successfully isolate a node or group of nodes participating in the blockchain by monopolizing all their incoming and outgoing connections [20] with malicious nodes, usually crafted with the sole purpose of performing this attack.

By controlling all the victim's connections, an attacker has control over what information the victim receives. Regarding this, the attacker can purposely feed false information and trick the victim into believing he's receiving honest blockchain data.

Establishing an incoming connection with a target requires transmitting a SYN-pakcet to the desired node. However, outgoing connections are not that easy to manipulate since they are established by the victim, which means that the attacker must find a way to trick the victim's into initiating a connection with the nodes he controls. Once all the connections are compromised, the honest node becomes trapped within the malicious environment. It is then flooded with false information regarding blocks and transactions and is also prevented from synchronizing its local chain with an honest node [25].

To successfully exploit an eclipse attack, all connections must be compromised. If the victim is able to maintain at least one honest connection, they could still get information on the real state of the blockchain. Moreover, other honest nodes in the network that establish a connection with the malicious nodes might also become vulnerable to the eclipse attack [46].

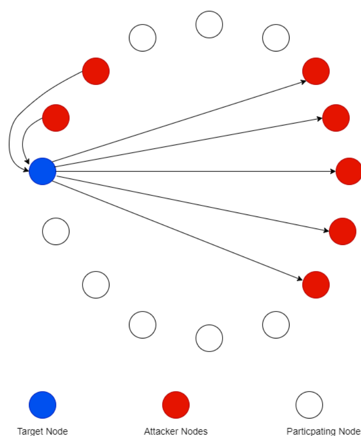


Figure 3.1: Example of eclipse attack

Chapter 4

Related Work

4.1 Eclipse attacks by monopolizing connections

This attack was seen in previous versions of Geth[1] and encompasses three key factors. To begin with, Geth client sets a default number for the maximum TCP connections any participant can have (maxpeers). Secondly, every time a client reboots they automatically drop all the previously established incoming and outgoing connections. This means that upon reboot, the client will have an empty *table*. Furthermore, an attacker can patiently wait for the client to reboot or send a packet of death. The third important factor to consider in this attack is that when a client restarts, they are open to receiving incoming connections before being able to establish outgoing connections [40].

Considering this, a malicious actor must generate a considerable number of nodeIDs, sufficient to surpass the value of maxpeers, and wait for the client to reboot. As soon as the client reboots, the attacker can use the crafted nodes closest to the victim and successively ping them [53]. The victim accepts these messages and establishes incoming connections with the crafted nodes. To be successful in eclipsing the target, the attacker must be able to flood all of their connection slots.

Previous versions of Geth allowed this attack to happen because a client could have all of their connection slots filled with incoming connections. To mitigate this attack, the current Geth version assures that every client has a certain number of outgoing connections, defined by 1/3 of the maxpeers.

[1] Older versions of geth allowed a node to have only established incoming connections, which eased the difficulty for an attacker to flood all the victim's connections and successfully eclipse them. Since all these connections are unsolicited, the attacker just needed to be persistent.

4.2 Eclipse by table poisoning

To successfully eclipse a victim, an attacker must control all incoming and outgoing connections. While incoming connections may be established via unsolicited ping messages, the same does not happen with outgoing connections. These must be established by the target with resource to the *table* and the lookup_buffer.

The attack to be described aims to control all the nodes in the victim's *table* and in the lookup_buffer so that when the target establishes outgoing connections they are established with the attacker's crafted nodes.

The first step is to craft as many nodes as needed to fill all the free slots in each bucket. To exploit this, the attacker leverages the fact that mapping node IDs to buckets is a public process, which means that a crafted node can be mapped to bucket 256 − r with a probability of 1. Honest nodes, otherwise, map to a bucket r with a probability of $1/2^{(r+1)}$. The attacker crafts n × 16 nodes, where n is the number of buckets he aspires to compromise. Nodes are continuously being crafted until all slots available are filled in bucket 256 − r. Once this is achieved, r is incremented and the process is repeated. Note that as r increments, the time to find a suitable node also increments, which creates some difficulty in finding nodes to fit in the lower buckets. For this reason, the Geth client only uses 17 kademlia buckets and thus the attacker only needs to generate and control 17 × 16 node IDs, which was done in this experiment in about 15 minutes [40].

The second step of this attack is to insert all the crafted nodes in the *db* database. To do so, the attacker only needs to send a ping message to the target from each of the crafted nodes. The target responds with a pong message, resulting in a successful bond and consequent insertion in *db*. To ensure that the attacker nodes are not evicted from *db* they ping the victim once every 24 hours and respond to every findnode message with an empty neighbors message.

The third step, just like the previous attack, takes advantage of the client reboot. Once the client reboots, its *table* is empty and the attacker's goal is to fill it with the crafted nodes. This is important because it is no longer possible for a client to establish only incoming connections, so the victim has to establish some outgoing connections with nodes that are stored in the table. By controlling all the nodes in the victim's *table*, the attacker forces the target to establish all of its k connections to the nodes he controls.

After the client reboots, the incoming connections (*maxpeers* - k) are rapidly established. All the crafted nodes repeatedly ping the victim to quickly bond and get added to the *table*. The previous step of this attack plays an important role here: due to the fact that the crafted nodes were already in the victim's *db*, they can quickly bond with the target and get added to the *table*. If the crafted nodes comply with these three requirements: 1) being in the victim's table; 2) have responded to every findnode message received; 3) has responded to a ping message in the last 24 hours; they are added to *table* before being pinged. Otherwise, the target would need to ping the attacker's nodes first before adding them, which would cost crucial time to execute the eclipse

attack.

Since the Ethereum client starts their UDP listeners before starting the seeding process (see section 2.2.8.3), the attacker leverages this to manipulate the seeding process and prevent the victim from performing the seeding process. The attacker is able to quickly insert nodes into the target's *table* and so when the seeding process initiates, it is no longer empty[2]. Impeding the seeding process also impeded the target from picking nodes from its *db* and therefore honest nodes are not inserted in its *table*. As outgoing connections are made by the victim to nodes on its table (compromised with the attacker's crafted nodes), the attacker only needs to worry about honest nodes pinging the victim.

This experiment was successful in eclipsing the victim using two machines, one to poison the table for outgoing connections and another one to ping the victim right after rebooting and establishing incoming connections with them [40].

4.3 False friends eclipse attack

After Ethereum adopted some countermeasures suggested in [40] it is now harder to execute an eclipse attack by poisoning the victim's table i.e., fill the victim's table with malicious nodes only. While before it was possible to craft multiple nodes on a single machine and poison the victim's *table*, now the Ethereum client has some restrictions on IP addresses. Every bucket is only allowed to store two node IDs with IP addresses in the same /24 subnet and can only be a total of 10 node IDs in the same /24 subnet on the whole *table*, which increases the difficulty and resources needed to eclipse a node in the network. The authors of the False Friends attack described in [35] how to successfully eclipse a node using only two IP addresses from different /24 subnets.

In this paper, the attackers established the incoming connections with the victim by instantiating multiple Geth instances on different ports and pinging the victim. This was possible because the restrictions were only imposed on *table* aiming to increase the difficulty and effort an attacker needed to have to force the victim to initiate a connection with one of its crafted nodes. To try to make the victim establish an outgoing connection with a crafted node, the attackers exploit the lookup_buffer and the ReadRandomNodes function. The attack is following a restart, however, despite a reboot might speed up the eclipsing process it was not needed since the Ethereum network connection proved to be short-lived and therefore the attack could be successfully executed in a couple of days.

At the time of this attack, Geth client had its maxpeers set to 25, which meant that four outgoing connections would be made recurring to the lookup_buffer and the other four to the ReadRandomNodes function. This function returns the nodes at the top of randomly chosen buckets. It follows this logic because the nodes at the top of the buckets are the more active

[2]The seeding process only starts if the *table* is empty

ones and therefore most likely to connect with. This can be easily exploited by being extremely active and sending ping messages to the victim, so the attacker nodes are seen as more active and stand on the bucket's head. Given this, the attacker only needs to populate each bucket with one malicious node. The adversary computes multiple nodeIDs by generating multiple ECDSA key pairs and checks if the nodeID is mapped to the bucket of interest. Due to the restrictions of the current Geth version, it is not allowed to have more than 10 nodes from the same /24 subnet in the discovery *table*. Since Geth clients maintain 17 buckets, this means that two different IP addresses are enough to compromise ReadRandomNodes.

The second method for establishing outgoing connections is the lookup_buffer, which will establish the second half of the remaining slots. This buffer is populated by the iterative kademlia-lookup to a random target t and thus will hold the 16 closest nodes to the target found through the kademlia lookup function, described in section 4.4. When establishing a connection to the lookup_buffer, the client prioritizes the nodes that are closer to the target. To exploit this buffer, an attacker needs to: 1) Make sure that an adversarial node is queried during the lookup process; 2) Ensure that nodes returned by the adversarial nodes are closer than all the other nodes. The first step is guaranteed because if the attacker has one node in each bucket it does not matter what the target is, there will be always a findnode request sent to the adversarial node in the same bucket as the random target. The second step involves computing as many nodeIDs as possible and find the 16 closest to the target. The fact that computing nodeIDs is uniformly distributed over the ID-space induces that the more node IDs the attacker generates, the more likely to craft node IDs closer to the target than honest ones [35].

If the ReadRandomNodes function and the lookup-buffer are successfully compromised, it is possible to eclipse the victim.

How to sneak into a bucket

This subsection is to detail the process of entering a bucket and how to calculate the probability of a specific node landing in the bucket of interest.

Given that SHA256 is a cryptographic hash function, it is possible to infer that nodeIDs are uniformly distributed, meaning that the probability of each bit being 1 or 0 is always $\frac{1}{2}$. Considering an existing nodeID K and a nodeID M that is yet to be generated, the probability of computing M and its first bit being the same as K is 1/2, as is the probability of them being different. Since the probability of each bit being 1 or 0 is 1/2, it is correct to assume that the probability of M landing in bucket 255 of K is 1/2.

When considering computing M_2 to land in bucket 254 of K, the attacker needs to have his two first bits equal to the first two bits of K. This results in a probability of $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$. The probability of a nodeID landing in a bucket i is dictated by: 2^{i-256}. This is easily demonstrated by calculating the probability of M_2 to land in bucket 254 of K and getting the same result: $2^{i-256} = 2^{254-256} = 2^{-2} = \frac{1}{4}$.

The authors leverage this to calculate the number of nodes they need to generate to eventually obtain a node that falls into the intended bucket i. It is given by: 2^{256-i}. The example described in [35] states that an attacker would need to generate 262 142 key pairs to land a node in each of the 17 buckets used by Geth. However, as shown in annex_to_be_created, this might be possible with just 100 000 key generations in a favourable scenario.

Receiving a findnode Request

The lookup function is started at a random node, the so-called target t. After the client randomly chooses the target, he will send find requests to the 16 closest nodes to that target t. Since 16 is also the number of nodes in each bucket, that means that the closest 16 nodes to any random target t are the ones in the same bucket as the target. This results in always receiving a findnode request every time the lookup function is executed.

Countermeasures

Following the authors' suggestions, connections have now increased from a default value of 25 to 50. ReadRandomNodes function has also been updated and considers every node in the *table* instead of only considering the head nodes in each bucket. The lookup_buffer, however, was not modified prior to this paper. Regarding incoming connections, it is now not possible to aggressively ping a target since connection attempts from the same IP must be within 30s of each other [35].

4.4 lookup function

The lookup function is an Ethereum function similar to the kademlia lookup utilized to unearth new nodes. It is an iterative process where nodes send a findnode message to their closest nodes and expect a neighbors message in response. It follows a notion of closeness to a target t where t is a 256-bit string. The distance between nodes a and b is measured by dA = SHA3(a) XOR t and dB = SHA3(b) XOR t where the smallest value is the one closest to t. This function can be triggered when the lookup_buffer is empty [35], with aim to populate it and later fill the buckets in *table*.

To start the process, the lookup initiator selects a random target t from one of its buckets. It then sends a findnode message to the 16 closest nodes to t, which will be the t itself and all the other nodes in the bucket. Every node that received the findnode message queries its own Distributed Hash Table (DHT) and returns a neighbor message with its set of nodes closest to t. Neighbor messages are limited to 12 nodes so the client will end up with 16x12=192 nodes. From these 192 nodes, the client selects the closest to t and repeats the process until the 16 closest nodes to t remain unchanged [40].

After uncovering the 16 closest nodes to t, the client adds them to the lookup_buffer. If during this process a node fails to respond to a ping message for 5 times, he is evicted from *table*.

4.5 Logdist Function

The logdist function is similar to the kademlia XOR metric. Its main purpose is to map nodes to the according buckets based on the distance between two nodes. It can be described as log2 ($\bar{N}1 \oplus \bar{N}2$) or the equivalent (255 – length of common prefix) [35].

When a client encounters a new node it uses the logdist function to determine which bucket will it go to. The process starts by applying the cryptographic hash function SHA3 to the nodeIDs to get 256-bit values. It then applies the function that will result in the output r, which represents the common most significant bits between the two nodes. This means that if two hashes have r most significant bits in common but $r + 1$ is different, then their logdist value is r [40]. Therefore, the node is mapped to the bucket 256 - r. All the process of mapping nodeIDs to buckets is public.

Chapter 5

Practical Work

5.1 Planning

This chapter focuses on putting into practice all the knowledge acquired during the research phase. Regarding all the information gathered from case studies and other research in this field, the main goal is to try to compromise a node in the Ethereum network.

The initial phase was to search for simulators that incorporate some of the Ethereum network processes and characteristics and use them to learn how nodes behave. The intention was to leverage the simulators for creating a scenario where the process of discovering nodes and establishing connections is compromised, so the node is eclipsed. The analysis of an eclipse attack is crucial to understanding what is to be improved to secure the network, so the research also covered trying to find simulators that run older versions of Ethereum that would allow replicating an eclipse attack. Outside the Ethereum scope, the objective was to find a simulator that allowed running a network where a node would have 51% of its hashing power.

The subsequent step was to dive into a more realistic scenario by running two full nodes on Ethereum official testnet, Sepolia: one acting as a "malicious" node and the other as the "target" node. In this case, it is possible to interact with the network so the "malicious" node is going to try to interact with the "target". Building on this, the goal is to escalate and compromise this node's connections upon reboot and get a better understanding of how nodes establish and manage their connections.

5.2 Simulators

The first approach to interacting with the Ethereum network was through the use of simulators. The core purpose of using simulators was to overcome the difficulties of complying with the storage and computational power requirements.

5.2.1 Ethereum-shadow

The first simulator of interest found was ethereum-shadow (https://github.com/ppopth/ethereum-shadow). It is a discrete-event network simulator. It is built with shadow [4], a framework used to test real-life applications on a simulated distributed network. It uses lighthouse as the consensus client and geth as the execution client and has already implemented discv5, the improved version of discv4 that is supposed to be introduced with Proof of Stake (PoS) and libp2p, which replaces devp2p on the consensus layer.

This simulator allows running multiple nodes and validators, simulating the interaction between them and feeding the blockchain. Although validators' behaviour is out of scope, it is possible to analyse their behaviour on the network. Understand how often they start discovery processes, and how they establish connections between them. The numbers of nodes and validators can differ, although they are set to 4 and 40 by default. The approach here is to simulate networks of different sizes in peers and try to identify any differences in them. The main scenario to be tested is to validate if the simulator employs bucket subnet restrictions and delve deeper into this subject. All the data produced from the simulation is provided on a log file that, despite being interesting for doing some analysis, does not let user interactivity and limits the research on how to eclipse a node.

1st run: The first test using Ethereum-Shadow simulator was a default test, mainly to see how the network works and have a model of nodes' behavior for future comparison. The first simulation run used the default configs, with only 4 nodes forming a network and a set of 40 validators. It simulated 10 minutes of network time, which took around the same 10 minutes in real time with all the logs being written to the specific node folder. Log files are used to analyse nodes' behavior and get a perception of what have they done and what have they been through during the network simulation.

Starting the simulation implies generating nodes' key pairs, initializing Ethereum Node Record (ENR) as this simulator uses discv5, and writing some specifications to the disk. The node then proceeds to add the bootnode to the routing table, however, first he starts a discovery service and opens the listening ports before dialling the bootnode. Nodes are pre-defined to establish a total of 80 connections, so they start peer discovery queries with a target for 80 nodes. Discovery requests have a target for 16 nodes, but since there are only 4 nodes present in the network it only returns 3 peers. Upon finding the three nodes, it attempts to establish connections with them. When successfully establishing a connection, the node is considered to be a listener or dialer, depending on whether the connection is inbound or outbound. Once the connection is established the node sends his peers a Status request for blockchain information to which they should reply. Peers then keep on exchanging blockchain information and trying to establish more connections by repeatedly sending discovery requests. Since there are only 4 peers on the network, there are no more peers to be found.

Every peer established at least one outbound connection except one. This is a violation of a previously implemented countermeasure and makes this node susceptible to being easily eclipsed

since he has no control over whom he is establishing a connection. This might be seen here due to the network being very limited in size, however, it should not be ignored.

2nd run:

The second simulation test run involved a network composed of 30 nodes and 50 validators. It took 30 minutes this time to simulate 10 minutes of network running. The initial steps for each node are the same, however it takes more time to initiate due to the existence of more nodes and validators.

Nodes join the network and quickly start sending discovery requests before even opening the listening ports or connecting with the bootnode. The first 16 nodes are rapidly found and another discovery request is sent before even setting up a connection. As connections are being setup, nodes continue their exhaustive search for new peers so they can fill the target value of 80 connections. In this test, since there are more nodes on the network, it is possible to observe things happening concurrently. While a node is dialing with a discovered peer, he is already exchanging metadata and Status messages with others with whom he has successfully connected. The protocol exchange messages occur later than expected, this message is seen after nodes being connected and communicating with each other. There are at times problems connecting peers, which might lead to disconnections. The two cases observed in this simulation are a peer discovered and successfully connected that incurs a dialing error which leads to a disconnection. However, later in time this peer is able to successfully connect and exchange information. The other case is from a peer that is discovered and an error on the transport protocol occurs while being dialed. The peer is marked as disconnected on the Distributed Hash Table (DHT) and is only added again after being re-discovered on a discovery request message.

Nodes keep sending discovery requests even though there are no more nodes to find. The rest of the simulation is basically nodes exchanging data and receiving blocks through a gossip pub/sub scheme. Each time a node receives a block, he processes and validates that same block before transmitting it to the next node.

Even though this simulation had far more nodes than the previous one, nodes 5 and 14 still only were able to establish 1 outbound connection, while inbound connections were set at 29.

3rd run:

The third test simulates a network with 80 nodes, which is also the maximum number of connections a node can have. This network also was composed of 120 validators, so it took 1h30 to simulate 10 minutes of network.

Having a total of 80 nodes with each capable of establishing 80 connections would mean that all nodes would have 79 established connections, based on the data from the previous tests run. This was confirmed by some nodes, however, not by all. Node 7 was only capable of establishing 75 connections, 3 outgoing and 72 incoming. The node did not establish any more outgoing connections and was not allowed to establish more incoming due to a limit set to 72. There were

also another 10 nodes that only established a total of 65 connections all outbound ones. They kept looking for more nodes to connect to, but never sent nor received any connection message. It was possible to observe that even though a discovery request handles up to 16 peers, nodes can initiate a discovery request with a smaller target in case they do not need 16 peers.

Once again, there is a node with only 3 outbound connections in a total of 75. This scenario facilitates the attacker to eclipse a node since the most difficult connections to forge are outbound.

Conclusion:

With the three tests that were run, it was possible to confirm that in this simulator, the node indeed starts listening for connections before trying to establish any outgoing connection. It was also helpful to see the flow of communication between the peers, how they manage failed connections, and how they transfer blockchain data between them. However, some key points of this research were not achievable, there is a lack of control on the network and the simulator does not seem prepared to handle eclipse attack testing. It was also not perceivable how nodes are mapped to buckets and where are they being mapped on the DHT. There were also some log files that mentioned times of last ping and other information that could be related to the Ethereum data structures, but unfortunately these files were not readable.

5.2.2 Blockchain Simulator

This simulator is a Proof of Work (PoW) blockchain network simulator [51]. It runs a PoW consensus algorithm and allows crafting specific network topologies to analyze the behavior of nodes under certain attacks or threats. It also provides a log of the network properties for analysis and lets the user manipulate node-specific attributes according to what the user is testing. The scenario testing can either be static or dynamic.

To perform a static analysis, one should create the network topology desired, define the attributes of the nodes such as mining power, latency, and connections and run the network to analyze how it behaves for a specific period of time. The dynamic analysis involves everything above, however, it also includes changing nodes' properties while the network is running. The idea with this simulator is to leverage its visual component to set up the network topology of interest and the logging functions to analyze the behaviour of the nodes in the network.

The first scenario to be tested is when a node or a mining pool has 51% of the network hashing power. The goal is to demonstrate that even if the node is an honest one or the mining pool is also composed of honest nodes, having 51% of the hashing power still threatens the decentralization property that characterizes most blockchains, especially Bitcoin.

1st test:

The network topology selected is illustrated in Figure 5.1. The network is composed of twelve connected nodes that are active miners with different computational resources and thus different hash rate capacities. Every node has the same latency and from 1 to 4 bidirectional connections.

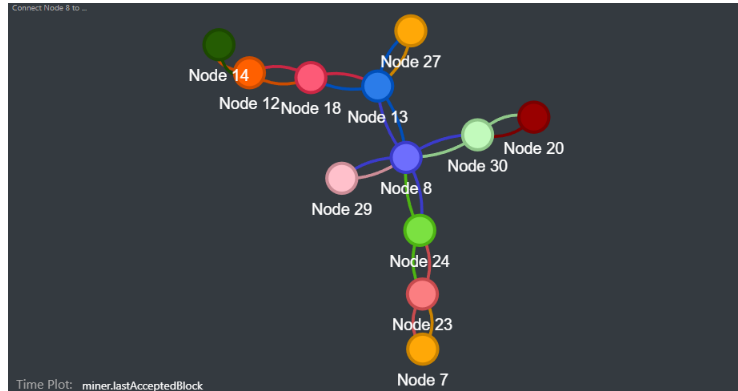


Figure 5.1: Network topology of test scenario 1

The total hash power score of the network is 56, meaning that 56 hashes are generated per second, however, node 8 alone has a score of 29 which represents 51.7% of the total network hashing power. This is not to be considered a malicious node, it is simply an honest node with more resources than any other. The difficulty of the network is at the default 0.01, which is not a very difficult value, but the number of hashes generated per second by all the nodes is not very high either. Two and a half minutes running the network was enough to understand how someone controlling 51% of the hashing power poses a threat to decentralization.

Looking at Figure 5.2 it is possible to see that 51% of the blocks were found by Node 8, the one with 51% of the network hashing power. The other nodes were also able to mine some blocks, however, they got a lot more stale blocks in comparison to Node 8.

Refresh	Account balance	%	Power (H/s)	%	Blocks in flight	Latency (ms)	Downlink (MBps)	Uplink (MBps)	In peers	Out peers	Block height	Total blocks	Stale blocks	Max fork length
Node 7	7	10	4	7.142	0	100	Infinity	Infinity	1	1	70	9	2	3
Node 8	39	55.714	29	51.785	0	100	Infinity	Infinity	4	4	70	40	1	2
Node 12	0	0	1	1.785	0	100	Infinity	Infinity	2	2	70	0	0	2
Node 13	6	8.571	3	5.357	0	100	Infinity	Infinity	3	3	70	6	0	2
Node 14	2	2.857	2	3.571	0	100	Infinity	Infinity	1	1	70	2	0	2
Node 18	2	2.857	3	5.357	0	100	Infinity	Infinity	2	2	70	3	1	2
Node 20	2	2.857	2	3.571	0	100	Infinity	Infinity	1	1	70	2	0	2
Node 23	2	2.857	4	7.142	0	100	Infinity	Infinity	2	2	70	5	3	3
Node 24	5	7.142	2	3.571	0	100	Infinity	Infinity	2	2	70	5	0	3
Node 27	2	2.857	3	5.357	0	100	Infinity	Infinity	1	1	70	2	0	2
Node 29	0	0	1	1.785	0	100	Infinity	Infinity	1	1	70	0	0	2
Node 30	3	4.285	2	3.571	0	100	Infinity	Infinity	2	2	70	3	0	2
Sum	70		56		0	1200	Infinity	Infinity	22	22	840	77	7	27
Average	5.833		4.666		0	100	Infinity	Infinity	1.833	1.833	70	6.416	0.583	2.25

Figure 5.2: Network statistics of test scenario 1

2nd test:

The second test followed the same network topology illustrated in Figure 5.1. This time, all the nodes have a lot more hashing capabilities: 50 hashes per second for every node except Node 8 who has the capacity of generating 600 hashes per second and once again has more than 51% of the network hashing power. The network ran for 3 minutes with all nodes together producing

1356 blocks. By looking at 5.3 it is easily understandable that Node 8 did not produce 50% of the total blocks, however, not all blocks found are appended to the blockchain. From the 1356 blocks found, 605 turned out to be stale blocks, which means that those did not integrate the longest chain. The highest block height seen is 756, which is shared by Node 8 and the nodes surrounding it so it is solid to assume that the longest chain has 756 blocks. From those, 51% were appended by node 8.

During the experiment, it was possible to observe that nodes farther from Node 8 were sometimes 3 blocks behind, thus spending their computational resources trying to find blocks that would eventually become stale. The nodes with the most stale block's percentage are the nodes with the most hops to Node 8, with an average of 83% block staling. The rest of the miners, despite not having as many blocks getting stale as Nodes 7, 12, and 14 still have many more stale blocks than Node 8. Since Node 8 is capable of generating 12 times more hashes per second than all the other nodes, it is usually ahead on the chain, which led to some forks that were eventually dropped to follow the leading chain.

Refresh	Account balance	%	Power (H/s)	%	Blocks in flight	Latency (ms)	Downlink (MBps)	Uplink (MBps)	In peers	Out peers	Block height	Total blocks	Stale blocks	Max fork length
Node 7	23	3.038	50	4.347	0	50	Infinity	Infinity	1	1	755	102	79	5
Node 8	389	51.387	600	52.173	0	50	Infinity	Infinity	4	4	756	418	30	6
Node 12	10	1.321	50	4.347	0	50	Infinity	Infinity	2	2	756	76	66	6
Node 13	54	7.133	50	4.347	0	50	Infinity	Infinity	3	3	756	90	38	6
Node 14	11	1.453	50	4.347	0	50	Infinity	Infinity	1	1	755	85	74	6
Node 18	36	4.755	50	4.347	0	50	Infinity	Infinity	2	2	755	88	52	6
Node 20	24	3.17	50	4.347	0	50	Infinity	Infinity	1	1	755	80	56	6
Node 23	37	4.887	50	4.347	0	50	Infinity	Infinity	2	2	755	84	47	4
Node 24	45	5.944	50	4.347	0	50	Infinity	Infinity	2	2	756	76	31	4
Node 27	24	3.17	50	4.347	0	50	Infinity	Infinity	1	1	755	76	53	6
Node 29	60	7.926	50	4.347	0	50	Infinity	Infinity	1	1	756	98	39	6
Node 30	44	5.812	50	4.347	0	50	Infinity	Infinity	2	2	756	83	40	6
Sum	757		1150		0	600	Infinity	Infinity	22	22	9066	1356	605	67
Average	63.083		95.833		0	50	Infinity	Infinity	1.833	1.833	755.5	113	50.416	5.583

Figure 5.3: Network statistics of test scenario 2

The results of these two experiments showed that even if a node is playing an honest role, having more than 51% of the network hashing power still poses a threat to the decentralization property of blockchains. The differences in blocks appended to the blockchain and stale blocks between Node 8 and the rest of the network are significantly high. Dishonest nodes with this much hash power might start redoing some blocks and eventually get the leading block height to fork the blockchain to their own local chain. Since PoW consensus follows the longest chain, all nodes would eventually follow the attacker's chain if all the precedent blocks have been properly redone.

The experiment only considers a single node, but it is actually more likely for a mining pool to get 51% of the network hashing power. The consequences are the same, the main difference is instead of a single node acting maliciously, there is a group of dishonest miners colluding to get control of the network.

5.3 Ethereum Official Testnets

The Ethereum project maintains two primary testnets: goerli and sepolia. Sepolia is the preferred choice for testing applications and deploying contracts, while goerli is usually where users test network upgrades and run validator for testing purposes [13]. In this work, we have selected Sepolia as our preferred testnet. It is faster to sync due to being more recent than goerli. Moreover, goerli is being deprecated in 2023 and replaced by Holesovice, which is scheduled for release by the end of September 2023.

5.4 Running a node

To interact with the network as a node, one needs to run a client. Geth [12], also known as "Go Ethereum", is an Ethereum client implementation written in Go and it's the most widely used client among all nodes. Moreover, most of the research papers cited in this work are from studies made in the geth client, where it was possible to analyse the process nodes go through when entering the network. This led to the conclusion that geth was the most suitable option given that it gives a continuity of previous research.

5.4.1 Execution and consensus client

As previously mentioned, Ethereum started as a PoW network. This meant that consensus was achieved by expending computational power in order to prove ownership over a block. Every "miner" in the network worked hard on solving a difficult mathematical problem, and the one that solved it first got the privilege of wrapping up transactions and adding them to the blockchain.

After Ethereum merge, one of the biggest changes in the network until current days, the consensus algorithm changed from PoW to PoS. This change eliminated miners from the network and introduced the validators, who are now responsible for voting and sometimes appending blocks to the blockchain. Furthermore, nodes do not need to expend computational power to be able to append a new block to the blockchain.

Besides changes at the consensus level, Ethereum also made some changes on the client level. At the current time, it is no longer sufficient to run an execution client solely. Ethereum now requires an execution client and a consensus client to run together, smoothly communicating with each other to stay updated on the recent blocks and facilitate interaction with the network.

- **Execution client** - The client used to listen to transactions and execute them on the EVM. Also maintains the most up-to-date state version and all recent Ethereum data.
- **Consensus client** - Also known as the Beacon Node, used to implement PoS. Allows the network to achieve agreement on the data validated by the execution client.

In addition, it is possible to run the validator-dedicated software in case a user aspires to participate in the attestation process or desires to append new blocks to the blockchain.

In this work, the chosen execution client was geth. For consensus client the selected option was prysm, also written in the Go programming language.

5.4.2 Syncing the network

If nodes want to actively participate in the blockchain, they must be synchronized with the current network state. This is achieved by querying other peers in the network the desired blockchain data and downloading it to the local database to create a local blockchain. There are different synchronization methods that essentially depend on the role a node is playing on the network.

At the execution layer, there are three possible ways in which a node can synchronize:

- **Full archive sync** - Downloads every block from the genesis until the most recently added to the blockchain. While downloading blocks, all the respective transactions are executed.
- **Full snap sync** - Download and verification is also made block by block, however, the starting point is not the genesis block but from a trusted checkpoint on the blockchain. Nodes delete older data to save space but keep these considered trusted checkpoints for faster synchronization. This method is currently the default on Ethereum mainnet [14].
- **Light sync** - Used in clients with less computational and storage capabilities, this method consists of downloading all block headers but only verifying some at random. It only saves the current state and can be rapidly set up.

With Ethereum changing to PoS a new client was introduced to the network: the consensus client. This client is responsible for handling the propagation of blocks and consensus logic. Consensus clients and execution clients work together to synchronize the chain. The execution client validates the blocks downloaded by consensus clients. Furthermore, geth can no longer be synchronized if not connected to a consensus client. It will require a header from the consensus clients, which geth uses as a syncing flag. After retrieving the header, geth downloads all the headers between that flag point and the top header of the local chain to understand if the block sequence is correct. After performing this validation, geth starts downloading the blocks' data [15].

Geth can retrieve the header through two different methods:

- **Optimistic sync** - Consists of downloading the blocks before the execution client validates them. Instead of only downloading the headers first, nodes assume that the information they are receiving is legit so they download the blocks' data and verify them from front

to back. While every block is not correctly validated, nodes cannot participate in the attestation process.

- **Checkpoint sync** - Consists of downloading a header from a trusted source and trusting that the information is correct. From that time onwards, the node will act as a full node and perform verifications block by block. Though this is a fast way to synchronize the chain, it is important to take precautions with whom to trust.

To synchronize with the network, the selected methods were the default snap sync for the execution client and a checkpoint sync via https://checkpoint-sync.sepolia.ethpandaops.io/.

5.4.3 Initializing the clients

As previously mentioned, the plan was to run two nodes where one would be malicious, and the other would be the target. That creates a need for two environments to run geth, so two Virtual Machines (VMs) were created. Both VMs had 350 GB of disk and 8 GB of RAM and SWAP memory with Ubuntu 22.10 installed.

To join the network, it is essential to establish an account. This is achieved by using Clef, which is considered a best practice for creating new Ethereum accounts. Subsequently, the execution client is started using geth and passing the chain ID of sepolia testnet. Since Ethereum is now a PoS network, it is mandatory to start a consensus client as well, which was done through prysm. Even though the consensus client is performing a snap sync, the process of synchronization is still time consuming and can take up to 24 hours to synchronize, depending on the RAM available of the system, downloading capabilities and whether both VMs are sharing resources. However, the consensus client can be synced more rapidly depending on the I/O capacity of the disk in use.

Once both nodes are synced with the network, the following task in to establish a connection between them. To see if the network is fully synced one can run *eth.syncinc* and if it returns true that means that the network is still being synchronized. There are various namespaces to retrieve information from the node or the network, such as: devp2p, net, admin, and eth. The eth command allows the retrieval of information about the blockchain, signing and submitting transactions. The net command permits to get the version of the network, which in this case would be 11155111, the version number of sepolia network. It also retrieves the total connections a node has and whether it is listening for connections or not. The admin command is used to retrieve information about the node itself, it can also be used to add new peers or get information about the connected peers (e.g., ENR, enode id, local and remote addresses). The devp2p is an official go-ethereum tool but is external to geth so it must be installed on the local machine. It can be used to print node records, perform operations on node keys, discv4 or discv5 operations, Domain Name System (DNS) commands, and others.

The devp2p tool was installed on both machines and used to perform ENR dumps, discv4,

and discv5 operations. The first operation done was an ENR dump on the target machine, this retrieved the nodeID, the enode url, the ip address and some other information like udp and tcp ports. However, node is not reachable either with discv4 or discv5 ping. This tool also provides a function to crawl the DHT and retrieve all nodes stored in it, but that as well outputted an error.

The admin RPC was used in an attempt to connect both nodes. It has two functions that permit adding nodes, the first is addPeer() and the second is addTrustedPeer(). To add a peer one should execute one of these functions and pass the enode url as the network is still working on discv4. Both functions were executed on the malicious node side to add the target and even though it returned *true*, the node was never seen on the list of peers. To eliminate any possible issue with these functions, other peers were tried and successfully added to the list of peers. The unsuccessful results on trying to add the peer might be related to being behind a Network Address Translation (NAT) on a VM. After changing the network configuration on the VMs and setting them as bridged adapter, it was possible to establish a successful connection between the two nodes. When querying the malicious node peers, it was possible to see an outbound connection to the target node. On the other side, the target node had in its list of peers an inbound connection established with the malicious node. During all the tests and even after letting the nodes run for a couple of days, the only inbound connection ever seen was the one manually established during the test.

Geth allows the generation of a configuration file based on the current network configurations. This network file contains network configurations that are executed when geth is initiated and the bootnodes to which the node must communicate when joining the network for the first time. This file can be modified to include static and trusted peers so the target enode as trusted peer. Geth client was initiated according to the modified configuration file and even though the connection was not immediately established, the peer was considered a trusted peer by default and could be added just by using the addPeer() function.

Chapter 6

Conclusion

This dissertation started by delving into the concept of blockchains and their properties, as it is important to perceive the capabilities of this technology and how can it be integrated into real-world applications. The main projects known are Bitcoin and Ethereum which implement the most well-known consensus algorithms: Proof of Work (PoW) and Proof of Stake (PoS). Even though at the beginning of this work Ethereum was a PoW blockchain, in September 2022 this network began what is known to be the biggest network change. A transition from PoW to PoW introduced a new actor on the network and a different method to propose blocks.

This study dived more deeply into these two blockchains with the intention of comprehending how nodes integrate into the network and the different roles played by each. Analysing two different consensus algorithms permitted the expansion of knowledge in this technology, especially at the consensus layer.

After conducting preliminary research on vulnerabilities in the different layers of blockchains, this work delved deeper into the investigation of vulnerabilities in the network layer. Entering the network can sometimes be a tricky process, nodes have no information on the blockchain, so they need to connect to other nodes to get it. The paper [31] [34] [47] allowed an understanding of how nodes joined the Bitcoin network and papers [40] [52] [35] provided valuable insights into the process of nodes joining the Ethereum network and how it has evolved in implementing countermeasures to prevent nodes from being trapped by malicious actors.

The purpose of this work is to use simulators and Ethereum official testnet to test whether blockchains are still vulnerable to eclipse and 51% attacks. [Blockchain simulator] was used to run two different scenarios in which a node has 51% of the network to demonstrate that event honest miners threaten the decentralization of a blockchain if they have more power than anyone else. The Ethereum-shadow simulator was an important tool to analyse the first steps node take on joining the Ethereum network. The idea was to leverage this simulator to enhance the knowledge of the network behavior and even though it did not allow interaction, it provided log files with every node behavior. Using the Ethereum main testnet introduces a more realistic scenario since it provides real-world communication. Being behind a Network Address Translation (NAT) did

not facilitate the communication between nodes.

Even though it was not possible to eclipse a node in the Ethereum, this network still presents some vulnerabilities that could be exploited by a malicious actor with more computational resources, a crucial aspect to execute this sort of attack. It was possible to create and run more than one node on the same machine with the same public IP address. By using only one Virtual Machine (VM), it is feasible to manually deploy multiple nodes to the network or develop an automation script that is able to create the nodes and initiate geth.

6.1 Limitations

During the practical execution of this work, some limitations were encountered, specifically on hardware requirements. In regard to simulators, some simulation scenarios would not run simply because there was not enough memory. The disk space is also not ideal since it has to be split between the VMs running the nodes on Ethereum official network and the ones running or testing the simulators. Moreover, the range of simulators available is also limited and from the ones available some are somewhat old and lack maintenance. They were also not really appropriate to test attack scenarios, but more to perform an analysis of some specific metrics.

6.2 Future Work

The continuity of this dissertation should follow an approach with more computing capabilities. One optimal approach would be to deploy as many nodes as there are slots available on the target's Distributed Hash Table (DHT) to compromise all node connections without crossing any network restrictions. Compromising the whole DHT ensures that the target would only establish connections to the attacker's crafted nodes and lookups would also be performed against the same nodes. This would give the attacker full control of the lookup buffer, and since readRandomNodes function selects nodes from the DHT, there is a great chance of being able to eclipse a node.

Bibliography

[1] Gas and Fees. https://ethereum.org/en/developers/docs/gas/, .

[2] Networking Layer. https://ethereum.org/en/developers/docs/networking-layer/, .

[3] Transactions. https://ethereum.org/en/developers/docs/transactions/, .

[4] Shadow. https://shadow.github.io/.

[5] Bitcoin Developer. https://developer.bitcoin.org/devguide/index.html, 2014.

[6] Node Discovery Protocol. https://github.com/ethereum/devp2p/blob/master/discv4.md, 2018.

[7] The RLPx Transport Protocol. https://github.com/ethereum/devp2p/blob/master/rlpx.md, 2020.

[8] Ethereum Wire Protocol (ETH). https://github.com/ethereum/devp2p/blob/master/caps/eth.md, 2020.

[9] Go-Ethereum, [Online] Available:. https://github.com/ethereum/go-ethereum/wiki/geth, 2020.

[10] Ethereum Accounts. https://ethereum.org/pt/developers/docs/accounts/, 2022.

[11] Ethereum Virtual Machine (EVM). https://ethereum.org/en/developers/docs/evm/, 2022.

[12] go-ethereum. https://geth.ethereum.org/, 2022.

[13] Networks. https://ethereum.org/en/developers/docs/networks/, 2022.

[14] Nodes-and-clients. https://ethereum.org/en/developers/docs/nodes-and-clients/#checkpoint-sync, 2022.

[15] Sync modes. https://geth.ethereum.org/docs/fundamentals/sync-modes, 2022.

[16] keys in proof-of-stake Ethereum. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/keys/, 2023.

[17] Adam Back. Hashcash - a denial of service counter-measure. 09 2002.

[18] Jaewon Bae and Hyuk Lim.

[19] Martijn Bastiaan. Preventing the 51%-attack: a stochastic analysis of two phase proof of work in bitcoin. 2015.

[20] Dhanasak Bhumichai and Ryan Benton. Feature extraction of network traffic in ethereum blockchain network layer for eclipse attack detection. In *SoutheastCon 2023*, pages 869–876, April 2023. doi:10.1109/SoutheastCon51012.2023.10115126.

[21] Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/, 2014.

[22] Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild, 2017. doi:10.48550/ARXIV.1707.01873.

[23] Wren Chan and Aspen Olmsted. Ethereum transaction graph analysis. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 498–500, 2017. doi:10.23919/ICITST.2017.8356459.

[24] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks and defenses. *CoRR*, abs/1908.04507, 2019.

[25] Yourong Chen, Hao Chen, Yang Zhang, Meng Han, Madhuri Siddula, and Zhipeng Cai. A survey on blockchain systems: Attacks, defenses, and privacy preservation. *High-Confidence Computing*, 2(2):100048, 2022. ISSN: 2667-2952. doi:https://doi.org/10.1016/j.hcc.2021.100048.

[26] Mauro Conti, E. Sandeep Kumar, Chhagan Lal, and Sushmita Ruj. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys Tutorials*, 20(4):3416–3452, 2018. doi:10.1109/COMST.2018.2842460.

[27] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10, 2013. doi:10.1109/P2P.2013.6688704.

[28] Varun Deshpande, Hakim Badis, and Laurent George. Btcmap: Mapping bitcoin peer-to-peer network topology. In *2018 IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*, pages 1–6, 2018. doi:10.23919/PEMWN.2018.8548904.

[29] Li Duan, Yangyang Sun, Kejia Zhang, and Yong Ding. Multiple-layer security threats on the ethereum blockchain and their countermeasures. *Secur. Commun. Netw.*, 2022:1–11, February 2022.

[30] Li Duan, Yangyang Sun, Kejia Zhang, Yong Ding, and Yuling Chen. Multiple-layer security threats on the ethereum blockchain and their countermeasures. *Sec. and Commun. Netw.*, 2022, jan 2022. ISSN: 1939-0114. doi:10.1155/2022/5307697.

[31] Jean-Philippe Eisenbarth, Thibault Cholez, and Olivier Perrin. A comprehensive study of the bitcoin p2p network. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 105–112, 2021. doi:10.1109/BRAINS52497.2021.9569782.

[32] Christian Trummer Eric Demuth, Paul Klanschek. What is a Bitcoin Node? https://www.bitpanda.com/academy/en/lessons/what-is-a-bitcoin-node/#what-is-defined-as-a-full-node, 2014.

[33] John Evans. Blockchain Nodes: An In-Depth Guide. https://nodes.com/, 2018.

[34] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on Bitcoin's Peer-to-Peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, Washington, D.C., August 2015. USENIX Association. ISBN: 978-1-939133-11-3.

[35] Sebastian Henningsen, Daniel Teunis, Martin Florian, and Björn Scheuermann. Eclipsing ethereum peers with false friends, 2019. doi:10.48550/ARXIV.1908.10141.

[36] Tam T. Huynh, Thuc D. Nguyen, and Hanh Tan. A survey on security and privacy issues of blockchain technology. In *2019 International Conference on System Science and Engineering (ICSSE)*, pages 362–367, 2019. doi:10.1109/ICSSE.2019.8823094.

[37] Meng-Jang Lin and Keith Marzullo. Directional gossip: Gossip in a wide area network. In Jan Hlavička, Erik Maehle, and András Pataricza, editors, *Dependable Computing — EDCC-3*, pages 364–379, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN: 978-3-540-48254-3.

[38] Fangxiao Liu, Xingya Wang, Zixin Li, Jiehui Xu, and Yubin Gao. Effective gasprice prediction for carrying out economical ethereum transaction. In *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, pages 329–334, 2020. doi:10.1109/DSA.2019.00050.

[39] Soo Hoon Maeng, Meryam Essaid, and Hong Taek Ju. Analysis of ethereum network properties and behavior of influential nodes. In *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 203–207, 2020. doi:10.23919/APNOMS50412.2020.9236965.

[40] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. *IACR Cryptol. ePrint Arch.*, 2018:236, 2018.

[41] Joanna Moubarak, Eric Filiol, and Maroun Chamoun. On blockchain security and relevant attacks. In *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*, pages 1–6, 2018. doi:10.1109/MENACOMM.2018.8371010.

[42] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[43] Sunny Pahlajani, Avinash Kshirsagar, and Vinod Pachghare. Survey on private blockchain consensus algorithms. pages 1–6, 04 2019. doi:10.1109/ICIICT1.2019.8741353.

[44] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. Security analysis methods on ethereum smart contract vulnerabilities: A survey, 2019. doi:10.48550/ARXIV.1908.08605.

[45] Sara Rouhani and Ralph Deters. Performance analysis of ethereum transactions in private blockchain. In *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 70–74, 2017. doi:10.1109/ICSESS.2017.8342866.

[46] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and David Mohaisen. Exploring the attack surface of blockchain: A comprehensive survey. *IEEE Communications Surveys Tutorials*, 22(3):1977–2008, 2020. doi:10.1109/COMST.2020.2975999.

[47] Yahya Shahsavari, Kaiwen Zhang, and Chamseddine Talhi. Performance modeling and analysis of the bitcoin inventory protocol. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 79–88, 2019. doi:10.1109/DAPPCON.2019.00019.

[48] Takenobu T. Ethereum EVM illustrated. https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf, 2018.

[49] Kathleen E. Wegrzyn Eugenia Wang. Types of Blockchain: Public, Private, or Something in Between. https://www.foley.com/en/insights/publications/2021/08/types-of-blockchain-public-private-between, 2021. [Online; accessed 19-August-2021].

[50] Daniel Davis Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[51] Simeon Wuthier and Sang-Yoon Chang. Demo: Proof-of-work network simulator for blockchain and cryptocurrency research. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 1098–1101, 2021. doi:10.1109/ICDCS51616.2021.00110.

[52] Karl Wüst and Arthur Gervais. *Ethereum Eclipse Attacks*. 2016. doi:10.3929/ETHZ-A-010724205.

[53] Guangquan Xu, Bingjiang Guo, Chunhua Su, Xi Zheng, Kaitai Liang, Duncan S. Wong, and Hao Wang. Am i eclipsed? a smart detector of eclipse attacks for ethereum. *Computers & Security*, 88:101604, 2020. ISSN: 0167-4048. doi:https://doi.org/10.1016/j.cose.2019.101604.

[54] Shi Yan. Analysis on blockchain consensus mechanism based on proof of work and proof of stake. In *2022 International Conference on Data Analytics, Computing and Artificial Intelligence (ICDACAI)*, pages 464–467, 2022. doi:10.1109/ICDACAI57211.2022.00098.

[55] Haofan Zheng, Tuan Tran, and Owen Arden. Total eclipse of the enclave: Detecting eclipse attacks from inside tees. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5, May 2021. doi:10.1109/ICBC51069.2021.9461081.