University of Trento

Department of Information Engineering and Computer Science

Master's Degree in Computer Science

Final Dissertation

Simulating large scale network attacks against Bitcoin

Advisor
Alberto Montresor

Student
Davide Pedranz

Academic Year 2017-2018

Abstract

Bitcoin and other cryptocurrencies are becoming more and more popular in the last few years. Bitcoin combines classical ideas of distributed systems with powerful cryptographic techniques to solve the Byzantine Consensus problem and achieve eventual consistency on the serialization of valid transactions. The main problem of Bitcoin is the possibility of forks, temporary inconsistencies in the blockchain which may allow an attacker to double spend its money under certain conditions. In this thesis, we analyze some security properties of the Bitcoin protocol under adverse network conditions. We implement an event driven simulator to test the low-level Bitcoin protocol used to build and maintain the network topology and propagate information about transactions and blocks. Our results show that, while Bitcoin behaves well under normal conditions, it shows serious problems with large scale network attacks.

Contents

Abstract **1**

1 Introduction **5**

2 Bitcoin **6**
2.1 Blockchain . . . 6
2.2 Blocks . . . 7
2.3 Transactions . . . 7
2.4 Addresses . . . 8
2.5 Wallets . . . 8
2.6 Forks . . . 8
2.7 Mining & Proof of Work . . . 8
2.8 Mining Pools . . . 10

3 Protocol **12**
3.1 Topology . . . 12
3.1.1 Peer discovery . . . 13
3.1.2 Connection establishment . . . 14
3.1.3 Address propagation . . . 15
3.1.4 Peer cleanup . . . 16
3.1.5 Misbehaving peers . . . 16
3.1.6 Control messages . . . 16
3.2 Core . . . 18
3.2.1 Transaction propagation . . . 18
3.2.2 Block propagation . . . 18
3.2.3 Initial block download . . . 19
3.2.4 Data messages . . . 20

4 Attacks **22**
4.1 Double Spending . . . 22
4.1.1 Race Attack . . . 22
4.1.2 Finney Attack . . . 24
4.2 Majority Attack . . . 24
4.3 Selfish Mining . . . 26
4.4 Eclipse Attack . . . 27
4.5 BGP Hijacking . . . 28
4.6 Balance Attack . . . 30

5 Simulator **33**
5.1 Simulation . . . 33
5.2 Discrete event simulation . . . 33
5.3 PeerSim . . . 34
5.3.1 Components . . . 35
5.3.2 Simulation life cycle . . . 37

5.4 Simulator design . . . 37
5.4.1 Topology . . . 37
5.4.2 Core . . . 38
5.4.3 Data structures and utilities . . . 39
5.4.4 Attack . . . 40

6 Results **42**
6.1 Settings . . . 42
6.2 Parameters . . . 42
6.3 Experiments . . . 43
6.3.1 Protocol at rest . . . 43
6.3.2 Generalized network delays . . . 44
6.3.3 Balance Attack . . . 44
6.4 Evaluation . . . 50

7 Conclusions **51**
7.1 Future work . . . 51

List of Figures

2.1 Schematic representation of a blockchain 6
2.2 Schematic representation of a Merkle tree 7
2.3 Schematic representation of a blockchain different branches 9
2.4 Estimation of the hashrate distribution amongst the largest mining pools 11

3.1 Example topology of an overlay network 13
3.2 Connection establishment in Bitcoin 14
3.3 Propagation of Addr messages in Bitcoin 15
3.4 Illustration of the trickling procedure 16
3.5 Overview of the Bitcoin control messages 17
3.6 Blocks propagation in Bitcoin 19
3.7 Block request in Bitcoin 19
3.8 Overview of the Bitcoin data messages 20

4.1 Illustration of the race attack 23
4.2 Illustration of an orphaned block 23
4.3 Illustration of the different phases of a majority attack 25
4.4 Illustration of the Selfish Mining strategy 26
4.5 Illustration of an Eclipse Attack 27
4.6 Illustration of a BPG Hijacking attack against a Bitcoin miner 29
4.7 Illustration of a network partitioning caused by a Balance Attack 31

5.1 Illustration of the main components of PeerSim 35

6.1 Blocks generation for networks of different sizes 43
6.2 Forks distribution for a network of 1000 nodes with different delays 45
6.3 Forks distribution for a network of 9000 nodes with different delays 46
6.4 Forks distribution for a network of 1000 nodes under Balance attack 47
6.5 Forks distribution for different networks under Balance attack 48
6.6 Forks distribution for a network of 1000 nodes under a Balance attack with different message drops 49
6.7 Forks distribution for a network of 1000 nodes under Balance attack with different numbers of partitions 50

Chapter 1

Introduction

Cryptocurrencies are getting more and more popular and gained the attention of the media in the last few years. They are used for multiple purposes, including online and in-shop payments, low-cost money transfer, and anonymous money spending. Among all digital currencies, Bitcoin is the first one who gained some real adoption and it is today the most used and valuable cryptocurrency available on the market, with a value of about 141 billion $ [35, 15].

Bitcoin is a complex technology that takes advantage of powerful cryptographic techniques and well-studied ideas of distributed systems to solve the Byzantine Consensus problem and achieve an eventual consistency on the order of the transactions. It organizes transactions in sequential blocks, which form the so called *blockchain*. The blockchain is distributed to all nodes in the network and form a public ledger of all transactions that can be used to deterministically compute the balances of all addresses of Bitcoin. Everybody can join the network and participate in the protocol, but a block needs to include the solution of a non-trivial computational puzzle to be considered valid. This idea is commonly known as *Proof of Work* [3] and it is one of the pillars of Bitcoin security: it guarantees that an attacker can not generate too many blocks and take control of the entire blockchain, since this would require a huge amount of computational power. Public-key digital signature guarantees that only the owner of a Bitcoin address can spend the money stored inside.

Many attacks against Bitcoin have been proposed and analyzed in the literature. One of the main problems of Bitcoin is the possibility of forks, temporary inconsistencies in the blockchain. Under certain conditions, forks allow an attacker to spend its money twice [40], for instance by including conflicting transactions in blocks belonging to different forks: the network will eventually choose one of the forks as main chain and discard all blocks in the other chain, including all the transactions stored inside. Most attacks try to take advantage of weaknesses in the protocol to increase the probability of forks and achieve double-spending.

In this thesis, we analyze some of the attacks that focus on the network. We implement an event-driven simulator to test the low-level Bitcoin protocol used to build and maintain the network topology and propagate information about transactions and blocks. We use the simulator to evaluate the performances of the Bitcoin protocol in different situations, both at rest and under attack. We focus on the Balance attack [57, 56], a recent proposal where the attacker takes control of a significant portion of the network and tries to create two groups of nodes with about the same computational power: it delays messages between nodes in different partitions to create forks that can be exploited to spend the money twice. Our results show that, while Bitcoin behaves well under normal conditions, it can have serious problems with large scale network attacks.

The rest of this thesis is organized as follows: Chapter 2 gives an overview of Bitcoin and explains its main concepts and ideas; Chapter 3 describes the low-level protocol used by Bitcoin to construct an overlay network and propagate information on top of it; Chapter 4 illustrated some of the most common attacks proposed against Bitcoin; Chapter 5 explains the architecture and the implementation of our simulator; Chapter 6 describes the most interesting experiments with the simulator and the obtained results; finally, Chapter 7 summarizes the most relevant results and gives the conclusions.

Chapter 2

Bitcoin

Bitcoin (BTC) is a decentralized digital currency without a central bank or single administrator. The original paper [50] and the first implementation of Bitcoin were published respectively in November 2008 and January 2009 by an unknown person or group of people using the name of Satoshi Nakamoto [8]. Since its release, Bitcoin has gained popularity and attention of the media, especially between the end of 2017 and the beginning of 2018 [11, 75, 70]. Nowadays, Bitcoin is the most used and valuable cryptocurrency available on the market, with a price of over 8000 $/BTC and a market cap of about 141 billion $ [35, 15, 16, 20, 14, 49]. Bitcoin can be used for both online and in-shop payments, fast and low-cost money transfer, and pseudonymous money spending.

Bitcoin is a complex technology that takes advantage of modern cryptography and distributed algorithms to solve the complex Byzantine Consensus problem [42]. This section covers the main building blocks and concepts and gives an overview of the overall working of Bitcoin.

2.1 Blockchain

The blockchain is a decentralized, distributed, public digital ledger that records Bitcoin transactions. It is implemented as a chain of blocks, connected to each other using linked timestamping [55, 79]: each block stores the cryptographic hash of the previous one (Figure 2.1). This technique guarantees that the transactions stored in the ledger can not be changed easily, since a single modification would break the hashes of each following block in the chain. The first block is called *genesis* and it is hard-coded in the software implementation.

Nodes that participate in the Bitcoin protocol run a consensus algorithm to agree on the order of blocks in the ledger: in particular, it is enough to agree on the last block in the chain, thanks to the guarantees given by the linked timestamps. The blockchain is stored in each computer that participates in the consensus: the geographical distribution of nodes around the world and the decentralization of the protocol make attacks that try to change the history of transactions stored in the ledger very difficult or nearly impossible to achieve.

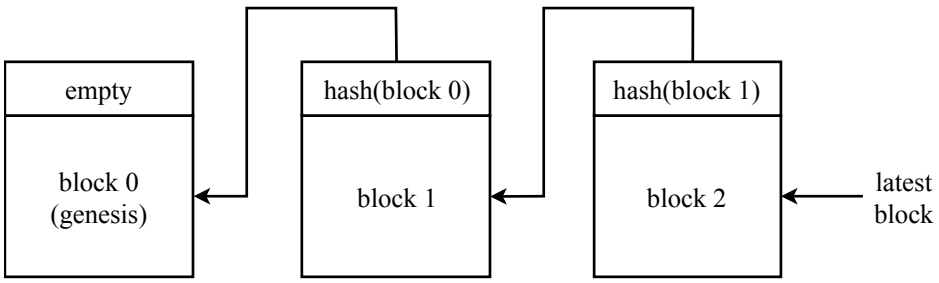


Figure 2.1: Schematic representation of a blockchain. A blockchain is a list of blocks, connected to each other with an hash pointer. Each block contains a set of transactions.

2.2 Blocks

Each block contains about 3000 transactions. By design, a new block is generated and appended to the blockchain every 10 minutes on average [50]. Bitcoin blocks are distributed using a peer-to-peer protocol, which is explained in detail in Chapter 3.

The transactions inside each block are organized as a Merkle tree [46], a special binary tree with hash pointers (Figure 2.2). The items in the tree are grouped in pairs and the hash of each of them is stored in the parent node. The parent nodes are then grouped in other pairs and their hashes are stored in their parents: this construction is repeated recursively until a single root node is created. In the specific case of Bitcoin, each item in the tree represents a single transaction.

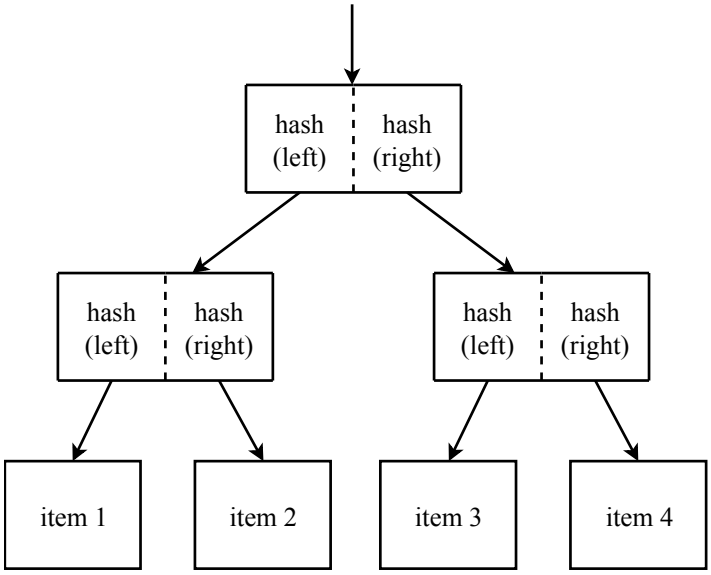


Figure 2.2: Schematic representation of a Merkle tree with 4 items.

The main advantage of using a Merkle tree is that the hash of the root uniquely identifies a specific set of transactions. In fact, a change of a single item would modify the hashes of all ancestor nodes in the tree, thanks to the properties of the cryptographic hash functions. This allows to divide a Bitcoin block in two parts - a header and a body - and distribute them independently of each other [10]. The header contains only the hash of the Merkle tree root, the hash of the previous block in the blockchain, the address of the miner that created the block (see Section 2.4 and Section 2.7) and a couple of additional information such as protocol version and timestamp of creation. The body stores the set the transactions as a serialized representation of the Merkle tree.

The second advantage of a Merkle tree is that transaction lookups in a block take $\mathcal{O}(\log n)$ time, where n is the number of transactions stored in the block. This property is very useful to efficiently validate new transactions.

2.3 Transactions

Transactions are used to spend bitcoins, i.e. move bitcoins between different addresses (see Section 2.4). Transactions in Bitcoin are not a simple tuple of `⟨sender, amount, receiver⟩`. They are expressed as small scripts, written in a custom, stateless and non-Turing-complete language. Transactions can have many inputs and many outputs, offer a transaction fee as incentive to be processed faster (see Section 2.7) or even define a kind of contract between parties (for example, some money "spent" in the transaction might be redeemable only on certain conditions).

For the sake of this thesis, we do not need to go into all details of Bitcoin transactions, which can be found in the Bitcoin Developer Guide [9]. It is only important to outline that:

- the history of all transactions is stored in the blockchain and allows to deterministically compute balances and validate new transactions (new transactions can only spend bitcoins that are available in an address, i.e. not already spent);
- transactions are valid only if signed with the private key of the owner of the address, so that nobody can steal bitcoins from third parties addresses without permission.

2.4 Addresses

A Bitcoin address corresponds to a pair of public and private keys: the hash of the public key is the actual Bitcoin address, while the private key is needed to make payments using the bitcoins stored in the address. Each address can store some bitcoins, which are the result of transactions stored in the blockchain. The owner of the private key associated with the address is able to create valid transactions to move the stored bitcoins to other addresses.

A Bitcoin address can be compared to the concept of a bank account: it stores an amount of money (bitcoins in this case) and its identifier is the only information required to make a payment to someone; in addition, only the owner is able to spend the money contained in the account. In contrast to traditional bank accounts, Bitcoin developers suggest to use each address only for a single transaction to preserve the privacy of the owner [9]: since all transactions are publicly stored in the blockchain, it is trivial to trace all transactions involving a specific address.

2.5 Wallets

A wallet is a convenient way to manage Bitcoin addresses. Bitcoin wallets are able to create public keys (i.e. addresses) where to receive bitcoins and use the corresponding private keys to spend the bitcoins in the addresses. They can simultaneously manage multiple addresses and provide a nice user interface to facilitate different tasks, such as buying something in a shop or making an online payment. A wallet also solves the problem of rotating addresses at each transaction in order to make the user more difficult to track, without need of human intervention.

Wallets can be either a software or a physical device. Software wallets are usually easier to use, since they can simply use the available internet connection to interact with the peer-to-peer network to get information from the blockchain and broadcast new transactions. On the other hand, software wallets are more vulnerable to attacks, since an attacker only needs to compromise the user's device running it to steal the private keys stored in the wallet and take control of the bitcoins stored in the corresponding addresses. Hardware wallets are physical devices able to manage Bitcoin addresses and store the private keys securely in a dedicated hardware module.

2.6 Forks

A fork is a bifurcation of the blockchain: it happens when two or more different blocks have the same parent block, as shown in Figure 2.3. Blocks in different branches of the blockchain may contain different or even contrasting transactions and create inconsistencies in the global state, since different nodes may decide to follow different forks. To solve this problem, Bitcoin introduces a resolution rule for forks: Bitcoin nodes should follow the longest chain, i.e. the chain whose last block has the higher number of preceding blocks.

Forks are probably the biggest problem in Bitcoin, since they allow an attacker to try to spend the money twice by creating two contrasting transactions that get stored in blocks published on different branches of the blockchain. We will cover the *double-spend* attack in greater detail in Section 4.1.

2.7 Mining & Proof of Work

The usage of modern cryptography in Bitcoin addresses and transactions guarantees that only the owner of some money can actually spend it (at least until the private key associated with the address

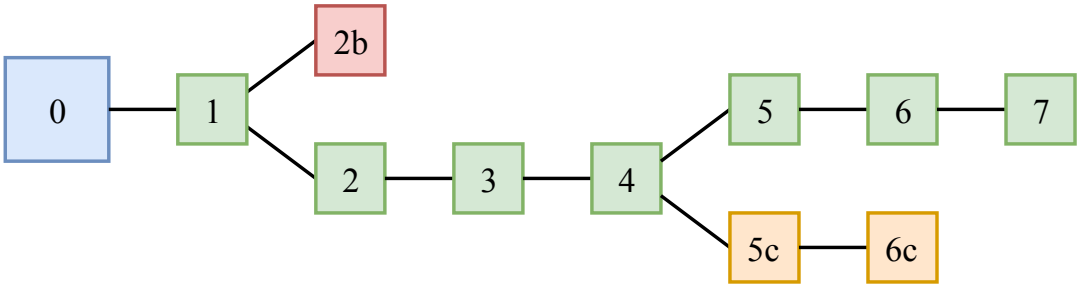


Figure 2.3: Schematic representation of a blockchain with 3 branches. The red block `2b` and the yellow blocks `5c` and `6c` are forks of the main chain that originate respectively at blocks `1` and `4`. Green blocks are on the longest chain, since block `7` is the one with the highest number of preceding blocks. The blue block `0` represents represents the genesis.

is carefully protected). Also, since all transactions are recorded and stored in the blockchain, the blockchain uses cryptographic hashes to detect tampering and it is replicated around many computer and devices around the world, it is very difficult or even impossible to remove old transactions from the past under normal conditions (i.e. revert payments). However, there is so far no mechanism that prevents an attacker to create a big number of forks by simply forging new blocks.

Bitcoin uses Proof-of-Work (PoW) [3] to reduce the feasibility of such an attack. It requires to solve a cryptographic puzzle and publish its solution to create a valid block. The difficulty of the puzzle is agreed by the network and depends on the total computational power available. The puzzle consists in finding a nonce to include in the block such that the cryptographic hash of the serialized block is less than a target number, i.e. it starts with a certain number of zeros. The maximum value of a block's hash that successfully solves the challenge is commonly known as *target* [74]. Thanks to the properties of cryptographic hash functions, the only way to solve the puzzle is a brute-force approach, which requires time and resources (in terms of both computation power and energy). This mechanism prevents an attacker to forge too many blocks, since it would require too many resources and time.

Miners that manage to create a new block get a reward in bitcoins when the block is stored in the main chain; also, they get the fees of all transactions stored in the block. The address of the miner is stored in the header of the block, so the other nodes that run the Bitcoin protocol know who to acknowledge for both rewards. It is important to underline that a miner is able to spend the reward obtained only if the corresponding block is stored in the main chain: blocks in different branches of the blockchain are not recognized, since all correct nodes will only consider the longest (main) chain as valid. Bitcoin rules try to encourage miners to behave correctly: if a miner has a block stored in the blockchain, it is its interest to keep mining on the longest chain, in order to make more difficult or impossible for an attacker to create a branch which gets longer than the main chain, since this might cancel the miner's rewards.

The Bitcoin mining process can be seen as a competition between miners. As soon as a new block is created, it is published and appended to the longest branch in the blockchain. Then, miners move the next block: they choose the transactions to include from the ones not yet processed, create the block by including the current timestamp, their address and the other information required and start to look for a nonce value that solves the new cryptographic puzzle. When a miner finishes a block, all the others stop and start again from the longest branch: it would make no sense to complete the current block [1], since it would be in conflict with the current longest chain and would probably be ignored by the other nodes in the network.

The idea behind this strategy - which is the one implemented in official Bitcoin implementation originally created by Satoshi Nakamoto [51] - is to waste as little computational power as possible:

[1] At least for honest miners (those that does not deviate from the Bitcoin protocol); malicious miners might have an advantage to choose a different strategy, as explained in Section 4.3.

when a block is attached to the longest chain, it would be a waste to complete another block that would go to the same place as the first one.

2.8 Mining Pools

It is important to notice that the cryptographic puzzle used by Bitcoin is highly parallelizable, both on a single computer with a multi-core central processing unit (CPU) and as a distributed algorithm in a cluster of machines. The only known way to solve the puzzle is a brute-force approach, which means trying all possible values for the nonce until a good one is found. Also, since the nonce has a binary representation on a finite number of bits, it is trivial to partition the set of all possible values in ranges. Each range can be assigned to a different core of the CPU or distributed to a different machine in a cluster; each core or machine reports the good nonces to some central coordinator, which it responsible to publish the block to the network and distribute new tasks to perform.

Because of the properties of cryptographic hash functions, the expected number of trials required to find a good nonce is 2^{b-1}, where b is the number of bits in the binary representation of the nonce in a block header. To check if a nonce is good for a block, a miner only needs to put the nonce in the correct location in the block header and compute the hash function on the complete block. The expected time to complete a block is thus proportional to the speed of the miner in computing an hash function. This value is usually called *hashrate* in the Bitcoin vocabulary [12].

Miners are in competition with each other: after a new block is published, honest miners start to work on the next block. The probability of a miner to be the first one to complete a block is proportional to its hashrate. Miners with small hashrates (e.g. those that use a general-purpose computer) are statistically very unlikely to ever find a block, and thus to earn a reward. For this reason, miners cooperate with each other in groups and create the so-called *mining pools*.

A mining pool is a group of miners that share their computational power in the mining process. Each pool has a coordinator that participates in the normal Bitcoin protocol: it collects pending transactions, distribute blocks etc. In addition, the coordinator creates ranges of nonces values and distributes them to the pool; when a node finds a good nonce, it reports it to the coordinator, which completes and distributes the block to the rest of the Bitcoin network. The same process is repeated for the next blocks. The coordinator is also responsible for sharing the rewards among the nodes in the pool: all blocks are created using the address of the coordinator, which collects all rewards, and periodically pays each node based on its contribution to the mining process.

Mining pools are a way to minimize the risk. If the pool has a sufficient hashrate, it is likely to complete blocks before other miners and get the reward; even small miners can contribute to a pool and get a small reward. Because of this, most mining power is concentrated in just a few mining pools. Table 2.1 shows an estimate of the hashrate distribution among the biggest pools: the top 5 pools control about 65% of the total hashrate; the top 10 pools count for about 90% and the remaining 10% is shared among small pools and alone miners. For convenience, the same information is shown in Figure 2.4 as a pie chart.

Mining pools use Stratum [72], a protocol introduces in 2012 to coordinate the mining process. The protocol is based on plain HTTP [71]: miners authenticate against the pools, but there is no verification of the pool's identity and the network traffic is in clear. The pool coordinator and the miners communicate using JSON-RPC: some commands are used to establish the initial connection between a miner and a pool (authenticate, reconnect, show message to the user, etc.), other commands are available to distribute mining tasks and collect the results.

#	Pool	Hashrate Share
1	BTC.com	20.2
2	AntPool	13.8
3	ViaBTC	11.9
4	BTC.TOP	11.6
5	SlushPool	11.6
6	F2Pool	8.0
7	Poolin	4.8
8	DPOOL	2.6
9	BitClub	2.5
10	Bixin	2.3
11	BWPool	1.9
12	BitFury	1.7
13	WAYI.CN	1.6
14	BTCC	1.5
15	Bitcoin.com	1.0

Table 2.1: Hashrates of the 15 largest mining pools estimated from the number of blocks published in the blockchain during the months of June, July and August 2018 [62].

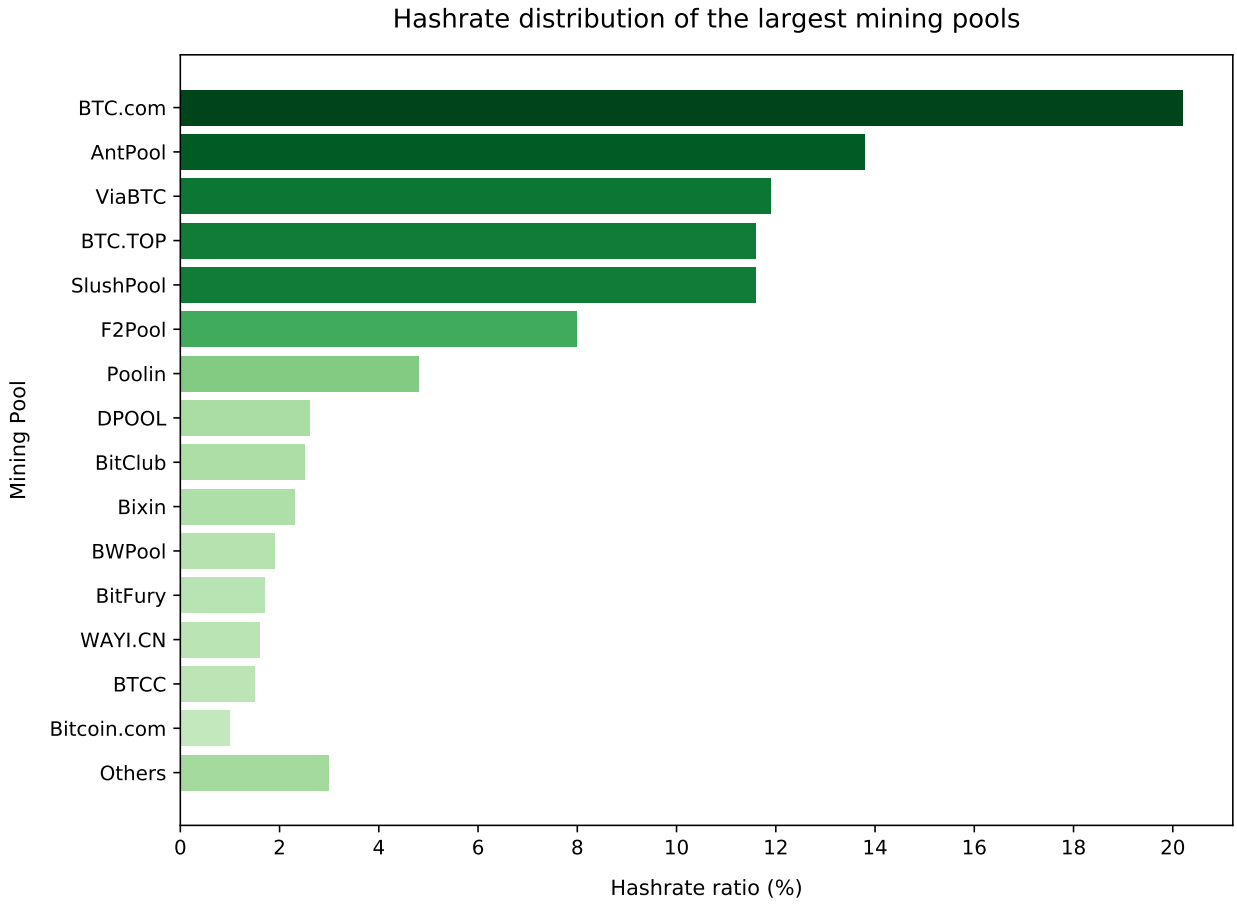


Figure 2.4: Estimation of the hashrate distribution amongst the largest mining pools in the months of June, July and August 2018 [62].

Chapter 3

Protocol

This chapter describes the low-level peer-to-peer protocol that runs Bitcoin. Unfortunately, the Bitcoin protocol has never been documented properly: the original paper [50] does not describe the details of the protocol and no official and complete description is available. Some unofficial documentation does exist, but it is often incomplete and outdated. The single point of truth available is the open-source original Bitcoin client, `bitcoind` [51]: at the time of writing, around 95% of nodes in the network run some version of this client [13]. Unfortunately, the source code is quite hard to understand and contains almost no comment. Thus, the content of this chapter is primarily based on academic papers [33, 5], on some online references [10, 9] and partially on the source code itself. Please note that this chapter refers to `bitcoind`: other clients may choose different settings or strategies and behave slightly differently in some cases.

Peers in the Bitcoin network are identified by their IP address. Each node can initiate up to 8 outgoing connections with other nodes, and accept up to 117 incoming connections, for a total of 125 [2]. All connections use unencrypted TCP channels. Nodes propagate and store only public IP addresses. At the moment of writing, there are about 9000 reachable server, while the number of clients is estimated to be between 100,000 and 200,000. Nodes can also connect to the network via Tor [7].

We divide the description of the protocol in two layers: topology and core. Both layers use an epidemic (or gossip) paradigm [23] to achieve a fast information propagation and be tolerant to failures.

3.1 Topology

The topology layer is responsible to create and maintain the peer-to-peer overlay network between Bitcoin nodes. Its main functions are:

- discover new peers in the Bitcoin network;
- propagate the IP addresses of nodes in the network;
- detect and remove peers that left the network or are offline for a long period of time;
- form an overlay network that can be used to efficiently broadcast blocks and transactions to all peers in the network (see Section 3.2);
- detect misbehaving peers and ban them if necessary.

Figure 3.1 shows an example of overlay network with 5 nodes. We will use this small network to illustrate different message exchanges between nodes in the rest of the chapter.

[2] With the default settings (it is possible to configure custom values if necessary). Measures show that the majority of nodes have at most 8 outgoing connections and never reach the maximum number of incoming ones [47].

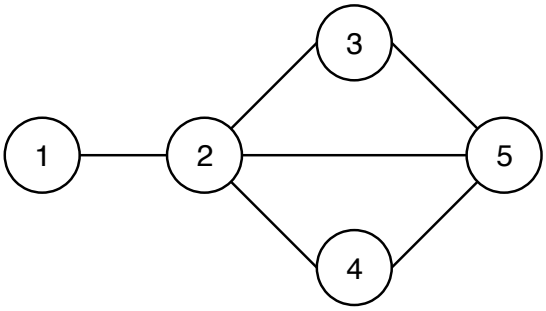


Figure 3.1: Example topology of an overlay network. The topology is represented as a graph: nodes indicate peers, edges indicate connections between peers.

3.1.1 Peer discovery

The peer discovery phase is crucial for the bootstrap of the network: at the first startup, a node does not know anything about the other peers. Bitcoin nodes use different strategies to discover the first peers to connect to: this allows to achieve a higher failure resistance and security, since an attacker might be able to interfere with some of the strategies, but it is less unlikely that it might interfere with all of them [66]. Once a node connects to the first peer, it can start to run the Bitcoin protocol to gather information about other connected peers.

IP discovery

The first step in peer discovery is to get its own IP address. After the startup, a Bitcoin node issues a HTTP GET request to two hard-coded websites, which reply with the IP address assigned by the Internet Service Provider (ISP): the client reads the HTTP response and parses the IP address [66]. It is possible that the node discovers more than one IP address: in that case, public IP addresses are preferred over private ones [6]. When a client establishes an outgoing connection to a remote peer, it first sends a `Version` message to advertize the chosen address (see Section 3.1.6).

Callback address

When a node receives an incoming connection, the remote node sends a `Version` message containing its IP address. After sending its own address, it sends a `GetAddr` request message to the remote node to learn about more addresses.

DNS seeders

A DNS seeder is a server that responds to Domain Name System (DNS) queries from Bitcoin nodes with a list of IP addresses of other peers. The seeder obtains these addresses by periodically crawling the network, looking for active nodes. At the time of writing, the Bitcoin network has 7 seeder and their hosts are hard-coded in the client source code [52]: each server is maintained by members of the Bitcoin community. The number of addresses in the query response in limited by the DNS constraints: a typical DNS packet over UDP contains up to 25 addresses [37]; if used over TCP, a DNS answer can contain up to 4000 addresses [32]. Please note that the list of addresses is not cryptographically-authenticated and could be easily compromised by an attacker with control of the victim's network (for example, mounting a man-in-the-middle attack) [9]. The DNS seeders are queried only in the following two cases:

1. a new node joins the network for the first time;
2. an existing node restarts and fails to reconnect to the old peers; the seeders are queried only if the node has less than two outgoing connections and after 11 seconds since the initial connection attempt.

Addr messagges

`Addr` messages are used to obtains network information from peers. They can be sent either unsolicited or as a response to a particular event. Each message contains up to 1000 IP addresses each; a single `Addr` message can theoretically contain any number of addresses, however it is rejected by peers running recent versions of `bitcoind` is it contains more than 1000 addresses. Addresses propagation using `Addr` messages is discussed in details in Section 3.1.3.

Database of known addresses

All peers discovered with the various strategies are stored in a database of known IP addresses locally to the node. When the node restarts, it can use the database to randomly select some nodes to connect to. The idea is behind this strategy is that Bitcoin nodes can change IP address over time, but it is unlikely that all of them change address at the same moment: after a restart, the node is likely to be able to connect to some of the old peers.

Hard-coded seed addresses

The client contains hard coded IP addresses that represent Bitcoin nodes: at the moment, the list contains about 1250 IP addresses [53]. The list is regularly updated by the Bitcoin contributors, so that clients running the latest versions of `bitcoind` have good chances to find some Bitcoin nodes willing to accept incoming connections. These addresses are only used as a last resort, if no other method has produced any address at all.

Configuration

Finally, a user can specify peer addresses in the software configuration, either with command line options or providing a text file of addresses. These addresses are not advertized in response of a `GetAddr` message [66]. The user can also specify the first peer to connect to.

3.1.2 Connection establishment

Nodes connect to each other over TCP. Once the connection is established, the nodes start to exchange messages. The first one is a `Version`: it provides information about the Bitcoin client and protocol version running on the node. The receiver replies with a `VerAck` message for acknowledgement. At this point, the nodes can start to exchange any type of message. Figure 3.2 illustrate a typical exchange of messages after the establishment of a new connection.

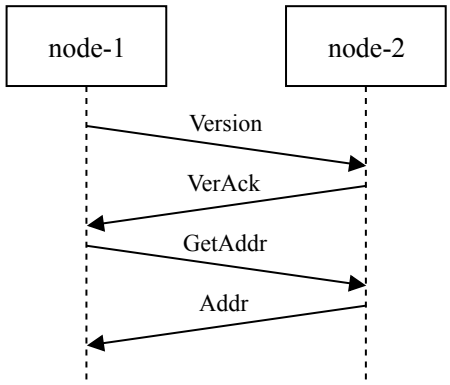


Figure 3.2: Connection establishment in Bitcoin. `node-1` connects to `node-2`: it sends a `Version` and gets as a reply a `VerAck` message; then, it asks for a list of known peers with a `GetData` message, and `node-2` sends back the list inside an `Addr` one.

3.1.3 Address propagation

Bitcoin uses `Addr` messages to propagate information about peers in the network and construct the topology. Each node maintains a list of addresses of other peers in the network: each address is given a timestamp which determines it freshness. A node can store up to 20,480 addresses, stored in different tables: the `tried` table contains addresses of peers to whom the node has successfully established a connection and is limited to 4096 addresses; the `new` table contains addresses for peers to whom the node has not yet initiated a successful connection is limited to 16,384 addresses.

`Addr` messages can be solicited or unsolicited. Nodes can request list of known addresses from each other using a `GetAddr` message: this is usually done only when a new outgoing connection is established (the protocol does not prevent a client from issuing a `GetAddr` message at any time) [33]. When a node receives a `GetAddr` message, it sends back 23% of the number of addresses stored in the database (chosen randomly), but no more than 2500 in total [6]. If the number of addresses exceeds the limit of 2500, multiple `Addr` messages are sent.

Nodes periodically push `Addr` messages to its peers: each day, a node sends its own IP address in an `Addr` message to each peer it is connected to [33]. Whenever a node receives an `Addr` message, it decides whether to forward it to its neighbors; the decision is taken individually for each address in the messages. An address is forwarded only if:

- the total number of addresses in the corresponding `Addr` message does not exceed 10;
- the timestamp attacked to the address is not older than 10 minutes.

If both checks pass, the node schedules a new `Addr` message containing the address to two randomly chosen nodes. If the address is non-reachable for the node (e.g. the node supports only IPv4 and the address is IPv6), it is forwarded to only one peer. Figure 3.3 illustrates an example of propagation of `Addr` messages.

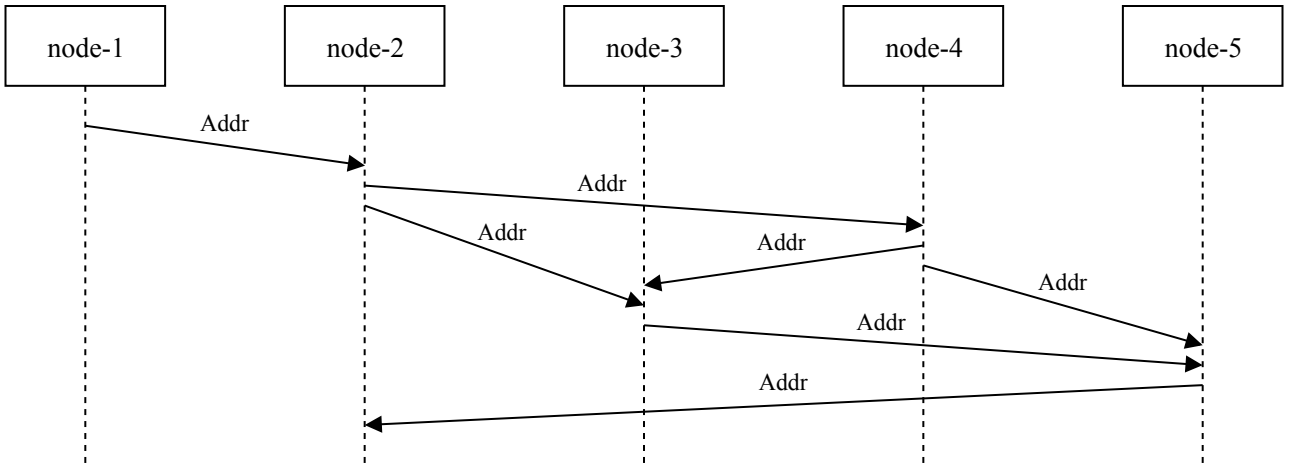
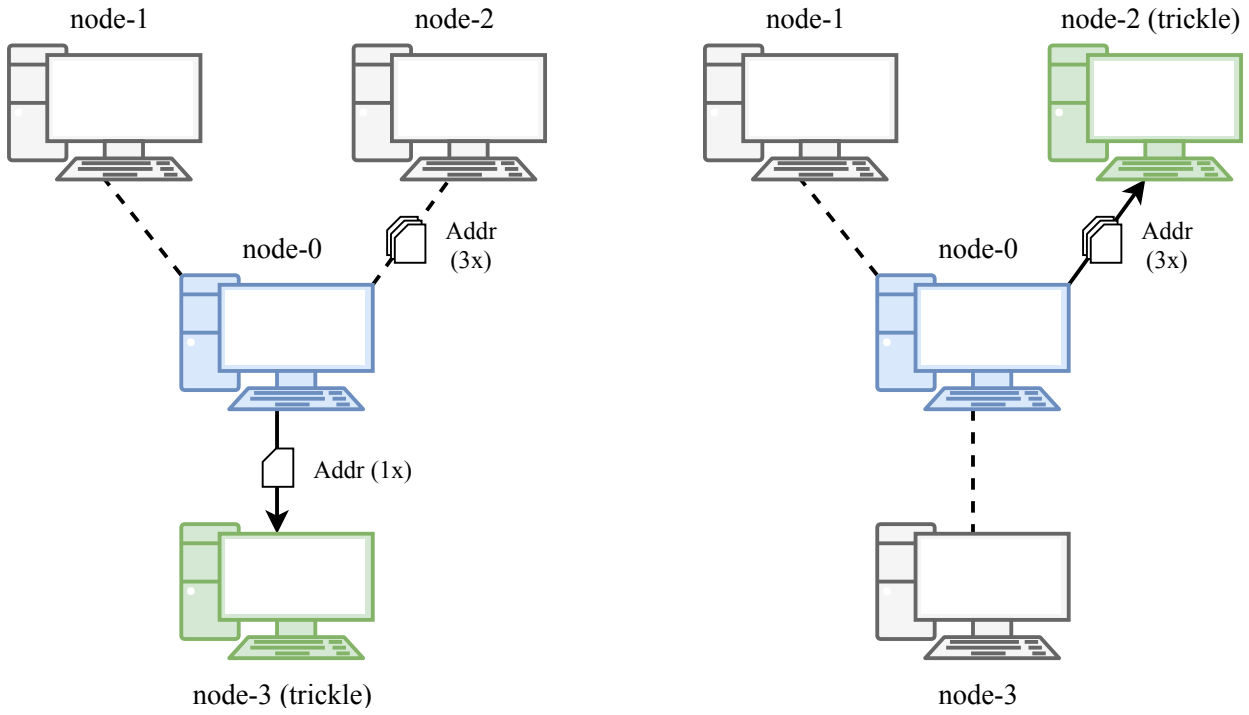


Figure 3.3: Propagation of `Addr` messages in Bitcoin. `node-1` sends an `Addr` message with its IP address unsolicited to its peer `node-2`. `node-2` randomly chooses `node-3` and `node-4` among its neighbors and sends them a new `Addr` message containing the IP address of `node-1`. `node-4` broadcasts the message to both its peers `node-3` and `node-5`. `node-3` receives the `Addr` message from both `node-4` and `node-2`, so it forwards the message to the only missing peer `node-5`. `node-5` sends the message to `node-2`, since it has no that `node-3` and `node-4` originally received the message from `node-2`. `node-2` does not forward its message anymore, since all its peer have already received it.

The actual transmission of the scheduled `Addr` messages does not happen immediately. The messages are divided in queues for different peers. Every 100 milliseconds, the node randomly selects one of the queues and flushes all the present messages. The node selected in the current round is the *trickle*

node [6]; the whole procedure is called *trickling* and is illustrated in Figure 3.4. To prevent stale `Addr` messages from endlessly propagating, each Bitcoin node remembers the messages forwarded to each connected peer: before forwarding an address, the node checks if the address was already sent over the same connection and avoids sending it again [33]. The lists of messages are flushed daily.



(a) Round 1: `node-3` is selected as trickle.

(b) Round 2: `node-2` is selected as trickle.

Figure 3.4: Illustration of the trickling procedure of `Addr` messages. `node-0` (in blue) gets an `Addr` message and select `node-3` and `node-2` for forwarding. An `Addr` message is added to the queues of both nodes, that count respectively 1 and 3 messages. `node-3` is chosen at round 1 and its queue is flushed: the scheduled `Addr` message is sent. `node-2` is chosen at round 2: all 3 messages in queue are sent together to the destination.

3.1.4 Peer cleanup

Bitcoin nodes exchange control messages to verify that remote peers are still connected and working correctly. Every 2 minutes, each node sends a `Ping` message to all connected peers [54]: upon receiving a `Ping` message, the node immediately replies with a `Pong`. If a node does not receive any `Pong` message from a peer for 20 minutes, it assumes the peer to be not working and drops the connection.

3.1.5 Misbehaving peers

Bitcoin implements a reputation-based protocol. Each connection is associated with a penalty score: whenever the node receives a malformed messages, it increases the penalty score for the sender; if the score reaches a threshold, the IP address of the corresponding peer is banned for 24 hours [6]. This mechanism helps Bitcoin to detect and attenuate denial-of-service attacks.

3.1.6 Control messages

Bitcoin nodes exchange information in messages, sent over the persistent TCP connections. Messages have a header and a body: the header always contains a fixed start string that delimits the message start, the name of the command, the size of the body and a checksum; the body changes depending on

the specific message and can be empty in some cases. None of the control messages are authenticated in any way: they could potentially contain incorrect or intentionally harmful information.

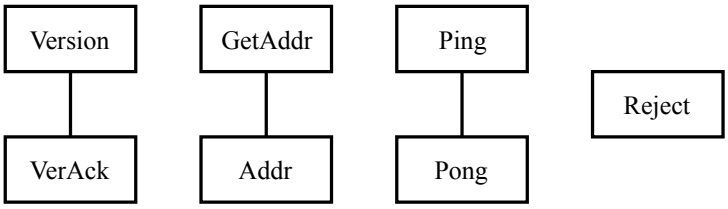


Figure 3.5: Overview of the Bitcoin control messages: lines between boxes indicate that the corresponding messages are correlated.

Some messages are related to each other [10], as illustrated in Figure 3.5: one is usually the reply to the other (for example, `Version` and `VerAck`); in some cases, one message can be both the reply of another one and be used alone (for example, `Addr`). Many of the control messages have been touched already in the description of the protocol: this section gives an overview of the main ones.

Version

The `Version` message provides information to the receiving node about the node that initialized an outgoing connection. If the incoming connection is accepted, the receiving node should reply with a `VerAck` message before sending any other message through that channel. Until both peers have exchanged the `Version` and `VerAck` messages, no other message will be accepted. The `Version` message also contains information about the client version, user-agent, configuration, sender and receiver IP addresses.

VerAck

The `VerAck` message acknowledges a previously received `Version` message and accepts the incoming connections: it informs the connecting node that it can begin to send other messages. The `VerAck` message has no payload and is sent only in reply to a `Version` message.

GetAddr

The `GetAddr` message requests an `Addr` message containing IP addresses of other nodes in the network from the receiving node. The transmitting node can use the message to quickly update its list of available nodes, rather than waiting for unsolicited `Addr` messages. The `GetAddr` is usually sent on new outgoing connections after receiving the `VerAck` from the other peer. The `GetAddr` message has no payload.

Addr

The `Addr` messages relays information about other known peers in the network (IP address and TCP port). The message can contain up to 1000 items: if a node wants to transmit information about more than 1000 peers, it needs to send multiple `Addr` messages. An `Addr` can be sent in reply to a `GetAddr` message or unsolicited (daily to broadcast the node's self IP address, or to forward a newly received address): details about the situations in which an `Addr` message is sent are discussed in Section 3.1.3.

Ping

The `Ping` message helps the transmitting node to verify that the receiving peer is still connected. If a networking error at either the TCP or IP level is encountered when sending the `Ping` message (e.g. a connection timeout), the transmitting node can assume that the receiving node is disconnected. The response to a `Ping` message is the `Pong` message. The `Ping` message contains a nonce in the payload that needs to be included in the reply `Pong` message: this allows nodes to keep track of the latency.

Pong

The `Pong` message is sent as a reply to a previously received `Ping` message: it confirms that the current node is alive and correctly connected to the node that sent the `Ping` message. The `Pong` message include the nonce sent in the original `Ping` message.

Reject

The `Reject` message informs the receiving node that one of its previous messages has been rejected. The message contains information about the rejected message and the reason for that. Examples of rejection reasons are a malformed block or a peer running an obsolete version of the protocol.

3.2 Core

The core layer uses the underlying overlay network created by the topology layer to propagate blocks and transactions to all Bitcoin nodes. Its main functions are:

- quickly propagate transactions to a big portion of the network, so they can be included in some block in a short time;
- propagate blocks as fast as possible to minimize the work wasted on orphan blocks.

3.2.1 Transaction propagation

Transactions are propagated using `Tx` messages: when a client generates a new transaction, it packages it in a `Tx` message and relays it to some Bitcoin node [10]. Similarly to `Addr` messages, `Tx` messages are not always immediately forwarded to all peers [6]: there are immediately sent with a probability of 0.25; otherwise, they are put in the queue of the corresponding peer and flushed with the peer becomes the trickle node (see Section 3.1.3).

The transaction is then propagated to the entire network. Forwarding a transaction from one peer to another involves several steps:

- the sender node transmits an `Inv` message of type "transaction" containing the hash of the new transactions;
- the receiver checks if it knows already the hashes; if it is the case, no further step is performed;
- if the hashes are new, the receiver sends a `GetData` message to request the transmission of the full transactions;
- the sender transmits one `Tx` message for each requested transaction;
- when the receiver gets the transactions, it advertizes them to its peers with an `Inv` message.

As with `Addr` messages, each Bitcoin node maintains a list of forwarded transactions for each peer and avoids to forward the same transaction to the same peer twice. Each node keeps the list of all received transactions in its memory pool. If it receives a transaction with the same hash as one in the pool or in a block in the main chain, the received transaction is rejected. Each node also keeps track of the transactions already included in the main chain and avoids to include them in new blocks. On the other hand, if a block is orphaned, the transactions which are not already in another block on the main chain are again available for the next blocks.

3.2.2 Block propagation

Blocks are propagated in a similar way as transactions. When a new block is generated, the miner sends a `Block` message to all its peers. When a node receives a block, it checks its validity and then forwards it to all connected peers. To avoid broadcasting the block too many times, nodes maintain

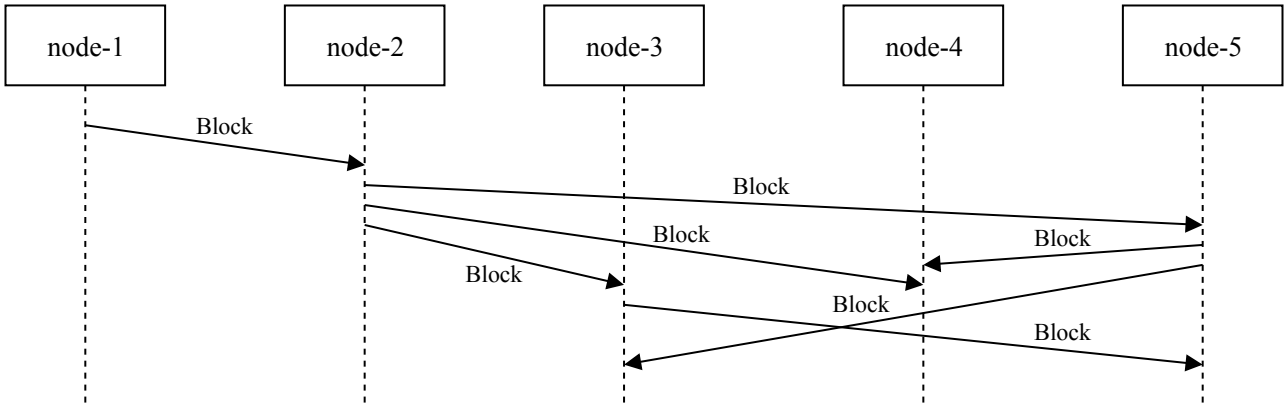


Figure 3.6: Blocks propagation in Bitcoin. `node-1` mines a new block and sends it inside a `Block` message to its peer `node-2`. `node-2` forwards the block to all its neighbors `node-3`, `node-4` and `node-5`. `node-5` receives the message before the others, so it sends it to both `node-3` and `node-4`. `node-5` gets the block from `node-2` and sends it to `node-5`, since it does not know yet about the incoming message from the peer.

a list of already sent blocks: each time they receive a block, they first check if that block was already forwarded; only if the answer is negative, the node sends it to its peers.

Blocks can also be requested an need with a `GetBlocks` message: this mechanism is used when a node receives a block with an unknown parent (in case two subsequent blocks are mined, it is possible for some nodes in the network to receives the second before the first one), or for the bootstrap of a new node, as explained in the next section.

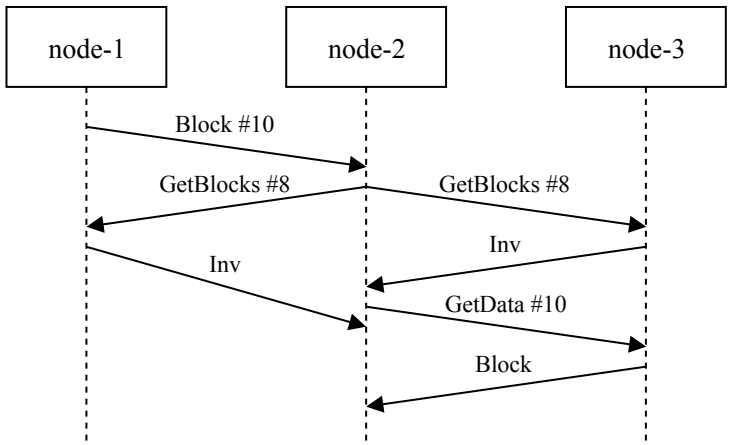


Figure 3.7: Block request in Bitcoin. `node-1` mines a new block and sends it inside a `Block` message to its peer `node-2`. `node-2` forwards the block to all its neighbors `node-3`, `node-4` and `node-5`. `node-5` receives the message before the others, so it sends it to both `node-3` and `node-4`. `node-5` gets the block from `node-2` and sends it to `node-5`, since it does not know yet about the incoming message from the peer.

3.2.3 Initial block download

New miners do not have any information about the current status of the blockchain. Once they successfully connect to some peers in the network, they need to download the entire blockchain to

be able to verify the new transactions and mine new blocks. This procedure is called *initial block download* and works as follows:

- the new miner selects one of the connected peers, called the *sync node*, and sends it a `GetBlocks` message, requesting all blocks after the genesis (the only known block, since it is hard-coded in the software);
- upon receipt of the `GetBlocks` message, the sync node searches its local longest chain, starting from the genesis; it then replies with an `Inv` message containing the hashes for the first 500 blocks after the genesis (the maximum allowed for a single response);
- the miner replies with a `GetData` message and requires the transmission of the complete missing blocks;
- upon receipt of the `GetData` message, the sync node replies with a `Block` message for each requested block;
- the miner validated each received blocks; the miner then sends a new `GetBlocks` message to request the next blocks;
- the process is repeated until all blocks in the longest chain has been transmitted.

The same procedure can be used if a miner goes offline for some period of time and wants to fast get up-to-date with the current state of the blockchain.

3.2.4 Data messages

Similarly to the control messages, data messages have header and body, and are sometimes correlated to each other. Serialized messages and are not authenticated in any way, but both transactions and blocks can be easily verified: a transaction is valid only if correctly signed with the private key of the sender address, while blocks are valid only if their hash is less or equal to the current target.

Figure 3.8 illustrates the main data messages used by the core layer of the Bitcoin protocol; each message in explained in detail in a dedicated section.

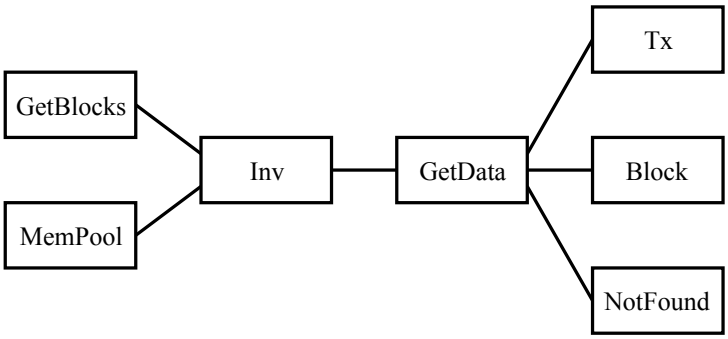


Figure 3.8: Overview of the Bitcoin data messages: lines between boxes indicate correlated messages.

GetBlocks

The `GetBlocks` message requests an `Inv` message that provides the hashes of blocks starting from a particular point in the blockchain. It allows a peer which has just joined the network for the first time or has been disconnected or started for a while to get information about the current state of the blockchain and later request unseen blocks. Note that the receiving peer may respond with an `Inv` message containing hashed of orphaned blocks: it is responsibility of the requesting node to poll all its peers to find the longest chain.

MemPool

The `MemPool` message requests the hashes of valid transactions that has not yet been stored in a block in the longest chain. That is, transactions which are in the receiving node's memory pool. The response to the `MemPool` message are one or more `Inv` messages containing the hashes of all transactions in the memory pool. The message is particularly useful for new miners that just joined the network to get quickly up-to-date with the list of transactions available for the next block. It is also used by non-miner clients that quickly want to check the state of a new transaction just submitted to the network. The `MemPool` message has no body.

Inv

The `Inv` message transmits one or more inventories (hashes, used as unique identifiers) of objects (blocks or transactions) known to the transmitting peer. It can be sent unsolicited to announce new transactions or blocks, or it can be the reply to a `GetBlocks` message or `MemPool` message. The receiving peer can compare the inventories contained in an `Inv` message to determine the unseen object and require them using a `GetData` message. An `Inv` message must contain all objects of the same type.

GetData

The `GetData` message requests one or more objects from another node. The objects are requested by their hashes (inventories), which the requesting node usually discovers from `Inv` messages. The response to a `GetData` message depends on the requested object and can be a `Tx` message, `Block` message, a `NotFound` message, or other special messages not described in this text.

Tx

The `Tx` message transmits a single transaction. It can be sent either unsolicited, for new transactions generated by the node, or in response to a `GetData` message. The body of a `Tx` message contains a single serialized transaction.

Block

The `Block` message transmits a single block. It can be sent for two different reasons:

1. solicited, in response to a `GetData` message that requests block objects;
2. unsolicited, broadcasting a newly-mined blocks to all of nodes in the network.

NotFound

The `NotFound` message is a reply to a `GetData` message which requested an object unknown or not available to relay (for example, Bitcoin nodes does not need to relay historic transactions that are already been included in the blockchain). The body includes the hashes of object that were not found on the receiving node.

Chapter 4

Attacks

Various attacks has been performed against Bitcoin in the past. Attacks can work at different levels with very different approaches and can have a wide range of effects on Bitcoin users. This chapter starts from an overview of the most relevant ones. At the end, it describes in details the Balance attack, whose analysis is the main object of the simulations.

4.1 Double Spending

Double-spending is the result of successfully spending some digital currency more than once. The attack is the oldest known one against Bitcoin and it is even partially discussed in the original Bitcoin paper [50]. The attack has been analyzed many times in the literature [39, 40, 41, 64] and has been reported to be quite easy to mount against fast payments [39]. Real episodes of the attack has been reported multiple times in the past [27, 31]. Double-spending breaks the most important guarantee that Bitcoin tries to give: transactions are irreversible and bitcoins can be spent only once.

The attack works by creating two conflicting transactions (that spend all the bitcoins in the same address) and submitting them to different parties, for example `tx-1` to a merchant and `tx-2` to the Bitcoin network: if the merchant accepts `tx-1` but the Bitcoin network receives `tx-2` before `tx-1`, it is likely that `tx-2` will be stored in the blockchain, while `tx-1` will be discarded, since not compatible with `tx-2`. The attacker only pretends to spend some bitcoins to pay the merchant, but it actually does not spend anything: it gets the purchase for free, while the merchant never receives the due bitcoins. The double-spending attack can be performed directly in some cases, or it can be the result of more complex attacks. There are many variants of double-spending, we analyze the most common ones here.

4.1.1 Race Attack

The race attack involves merchants that immediately accepts payments, without waiting for the unconfirmed transaction to be securely store in some block in the blockchain [38]. The race attack has a high degree of success for an attacker [40]. The attack is illustrated in Figure 4.1 and works as follows:

- the attacker sends a transaction `tx-1` to the merchants;
- at the same time, the attacker sends a conflicting transaction `tx-2` to some miner; the conflicting transaction spend the bitcoins stored in the same address used in `tx-1` and sends them to new address in control of the attacker;
- the merchants sees the unconfirmed transaction and accepts the payment; this scenario is more likely for in-shop purchases, where, in many cases, the merchant can not wait for the transaction to be stored in the blockchain;
- `tx-2` is received before `tx-1` from the miners, so it is more likely to be stored in some block before `tx-1`;
- `tx-2` is stored in a block, while `tx-1` is rejected since in conflict with `tx-1`.

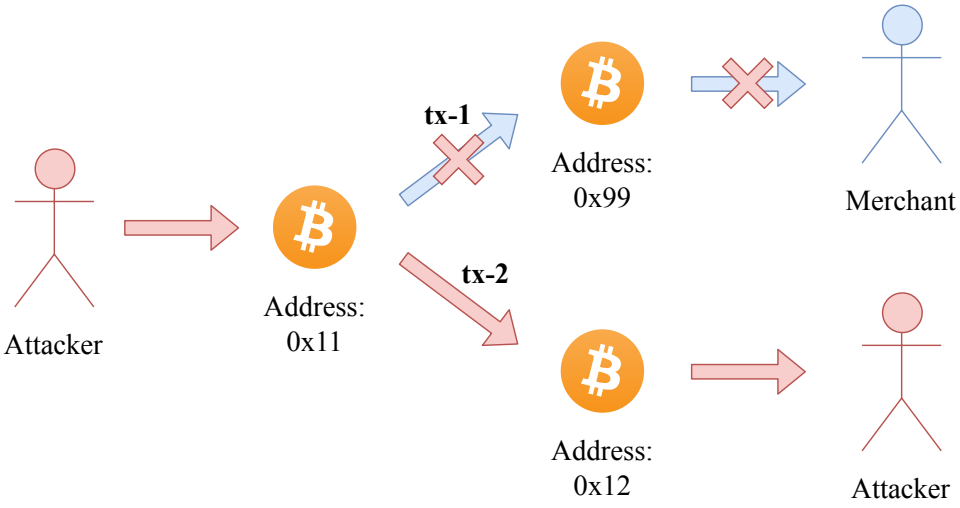


Figure 4.1: Illustration of the race attack, a variant of double-spending. The attacker (in red) owns some bitcoins in the address `0x11`. It submits a transaction `tx-1` to move the money from its address `0x11` to the merchant's one `0x99` (in blue). At the same time, it submit to the Bitcoin network a conflicting transaction `tx-2` to move the the money from `0x11` to a new address `0x12` in its control. If the network includes `tx-2` in some block before `tx-1`, `tx-1` is rejected, and the money stay control of the attacker. The red arrows indicate the real flow of bitcoins, after that `tx-2` is stored in the blockchain and `tx-1` is rejected; the blue arrows with red crosses indicate what the correct flow of money would if the attack fails.

A trivial defense for the merchant is to wait for the transaction `tx-1` to be stored in the blockchain: if it is rejected because of any conflicts, the merchant can simply refuse the payment and do not sell the good. Unfortunately, Bitcoin requires on average 10 minutes to confirm a transaction, so this technique can not be always applied. Bitcoin's developers recommendation is to wait for 6 confirmations, i.e. to wait for the block that stored the transaction to be in the longest chain and to have 6 following blocks [18]. Some blocks can be conflicting with each other (if they have the same parent): only one of them can belong to the longest chain, while the others are said to be "orphaned" [60] and simply ignored, as illustrated in Figure 4.2. Thus, a transaction can be rejected even if it manages to get part of a block, if that block is orphaned. Waiting for 6 confirmations gives a good tradeoff between the time to wait (on average 1 hour) and the statistical guarantee that the transaction is unlikely to be rejected later [50].

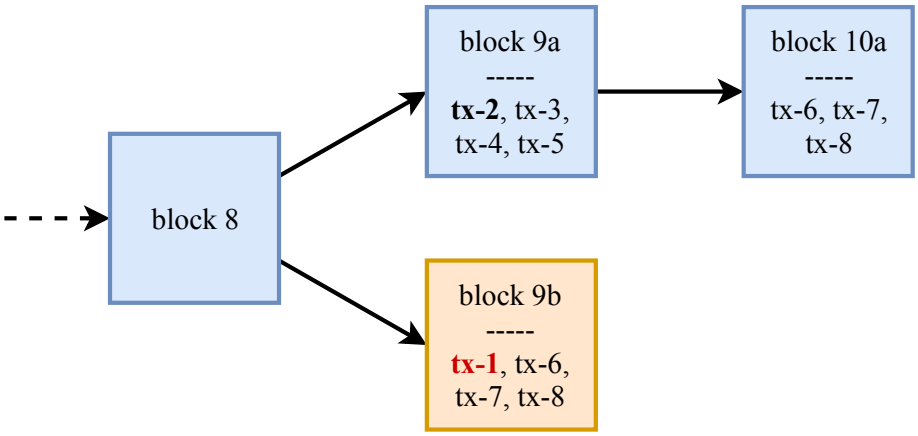


Figure 4.2: Illustration of an orphaned block. Blocks `9a` and `9b` have the same parent. Since `9a` has a child `10a`, it is on the longest chain. Transactions `tx-6`, `tx-7` and `tx-8` from the orphaned block `9b` are stored in `10a`; `tx-1` (in red) is in conflict with `tx-2` and is thus rejected.

4.1.2 Finney Attack

The Finney attack is another variant of double-spending targeting merchants that accepts unconfirmed payments. It can be performed by an attacker that is able to mine some blocks. The attack works as follows:

- the attacker controls addresses `addr-a` and `addr-b`;
- the attacker mines a new block on the longest chain; the block includes a transactions `tx-2` that moves all bitcoins from address `addr-a` to `addr-b`;
- before broadcasting the block to the Bitcoin network, the attacker sends a transaction `tx-1` to a merchant, spending again the bitcoins in `addr-a`;
- as soon as the merchants accepts the payment, the attacker broadcasts its block to the network;
- `tx-2` is stored in the blockchain before `tx-1`, so `tx-1` is rejected because in conflict with `tx-2`;
- the attacker receives its bitcoins back, while the merchant does not get anything.

Similarly to the Race attack, the Finney attack relies on conflicting transactions and the significant time required to confirm a transaction with a high confidence in Bitcoin. The defense for the merchant is the same discussed in Section 4.1.1.

4.2 Majority Attack

The majority attack, also known as 51% attack, refers to a situation in which one miner (or one group of miners) control more that 50% of the entire network's mining hash rate [1, 45]. An attacker with such a computational power can generate blocks faster than the rest of the network: it can potentially take control of the entire blockchain and each block in the longest chain. The attacker strategy is to mine its private chain, ignoring the current longest chain and each block honest miner published. This strategy gives statically guarantees to eventually generate the longest chain, no matter the advantage of honest miners (in terms of the difference in the number of blocks on the current longest chain and the number of blocks in the attacker's chain). Such an attacker could, potentially, start its own chain from the genesis block and revert the entire history of Bitcoin, as illustrated in Figure 4.3.

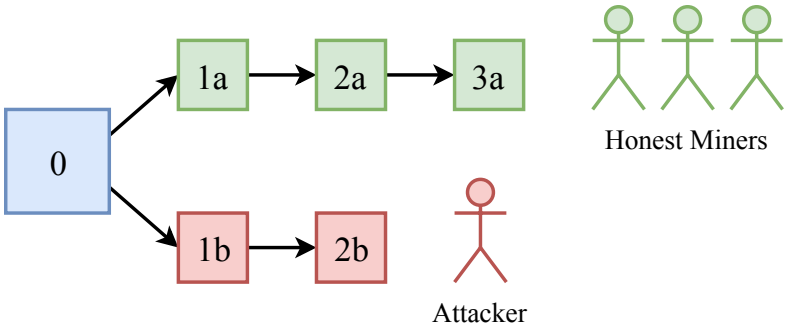
There is no defense against this attack: Bitcoin's security relies on an honest majority of nodes that collectively control more computational power than any attacker (or group of attackers) [50]. To be more precise, the system is not vulnerable to the majority attack as long as no single coalition of miners controlling more than 50% of the total hash rate [38].

The majority attack allows an attacker to perform the following actions [76]:

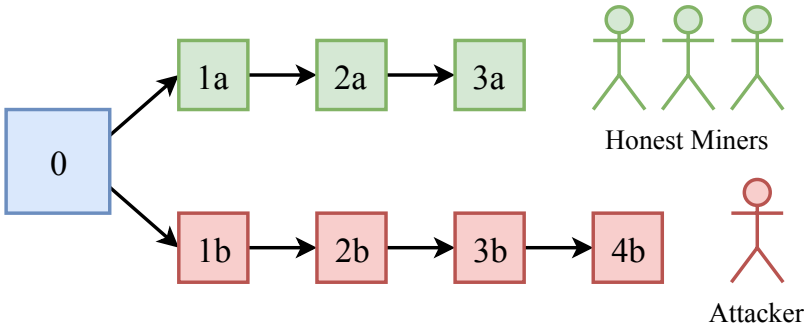
- rewrite the history of the blockchain;
- prevent some or all transactions from becoming part of the longest chain;
- prevent some or all other miners from attaching new blocks to the longest chain (and thus getting any reward);
- reverse transactions it has made in the past, effectively double-spending its bitcoins;
- gain the revenue of all new blocks.

A majority attack can completely destroy the working of Bitcoin and violate some of its main guarantees (e.g. spending the same bitcoins only once). Such an attack is thought to be very unlikely [28, 63], because of its huge cost in terms of computing power and the risk for an attacker not to gain anything from it: even if it is theoretically possible to revert all blocks following the genesis, it is incredibly expensive. The attack would immediately destroy the credibility of Bitcoin and lead to the following two possible effects:

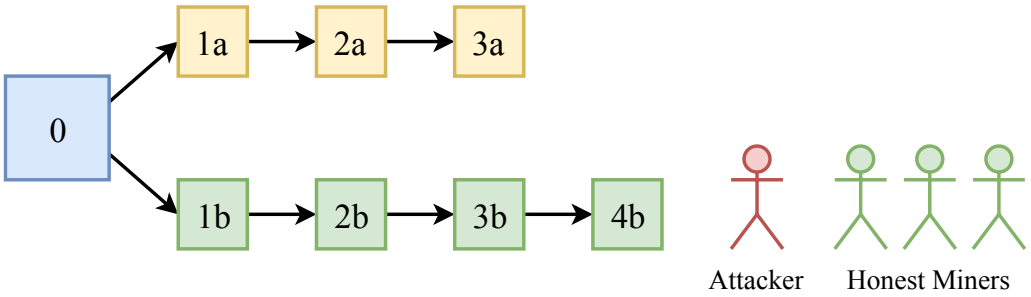
1. the value of bitcoins drops completely, and the attacker does not earn any real reward;
2. the rules of the Bitcoin software are changed by the community to revert the attack [76].



(a) The attacker starts to mine block (in red) from the genesis (number `0`, in blue) instead of following the longest chain (in green). Honest miners keep following the green chain that finishes with block `3a`.



(b) Eventually, the attacker's chain gets longer than the previous longest chain. The attacker broadcasts all blocks in his chain to the Bitcoin network.



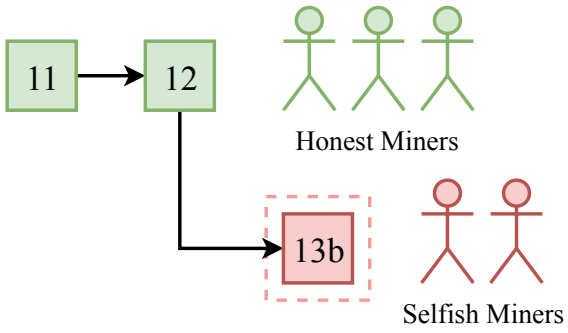
(c) Honest miners start following the attacker's chain (now in green), since it is not longer than the chain terminating with block `3a`. The oldest chain (now in yellow) is orphaned: all transactions stored in blocks `1a`, `2a` and `3a` are not valid anymore. Potentially, the attacker may include transactions conflict with the canceled once in any of the new blocks `1b`, `2b`, `3b` and `4b`, effectively double-spending some or all of its bitcoins.

Figure 4.3: Illustration of the different phases of a majority attack, where an attacker takes control of the entire blockchain, starting from the genesis block. The attacker is able to revert the history of all transactions stored in the blockchain.

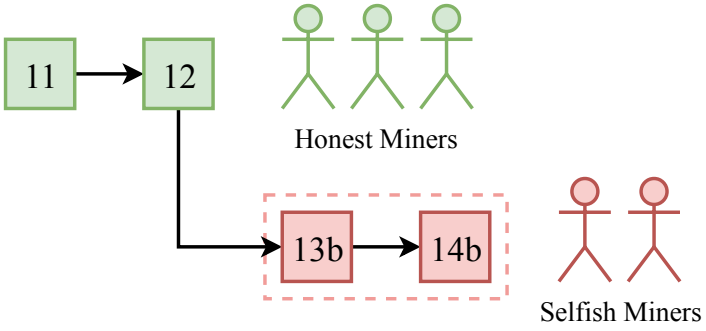
In January 2014, the mining pool `GHash.IO` reached 42% of the total Bitcoin hash rate [30, 43], and later in July 2014 it exceeded the threshold of 51% [81, 1, 38]. Even though there is no evidence of a majority attack to ever been mounted against Bitcoin, the episode was quite controversial in 2014: a number of miners voluntarily dropped out of the pool and `GHash.IO` implemented a mechanism to prevent the situation to ever happen again in the future [36, 34, 26].

4.3 Selfish Mining

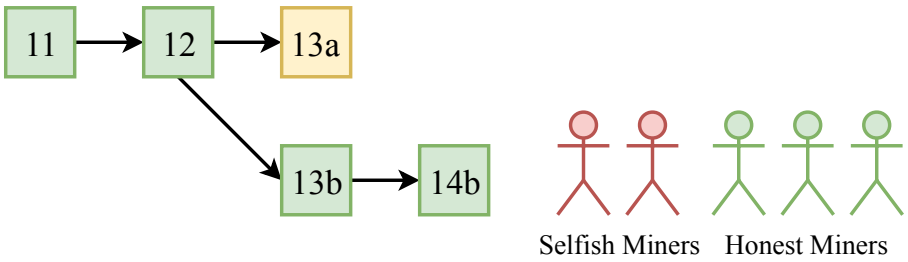
Selfish Mining [25] is a strategy where a group of miners strategically chooses when to submit the new blocks to the public chain, rather than submitting them immediately upon discovery. The selfish miners keep the discovered blocks private, intentionally forking the blockchain and mining on their own private branch. The honest nodes do not know about the blocks discovered by the selfish miners, so they continue to mine on the public chain. If the public longest chain approaches the selfish miners' private branch length, they reveal some blocks to surpass the public chain again. The selfish miners causes the honest miners to waste a lot of computational power on obsolete chains, since the longest one is not yet public. The idea is illustrated in Figure 4.4.



(a) The selfish miners secretly mine block `13b` from the current public longest chain (in green). The honest miners do not know about block `13b`, so they keep working from block `12`.



(b) The selfish miners manage to secretly mine another block `14b` and keep it secret (secret blocks are indicated by the red dashed box).



(c) Honest miners manage to mine block `13a` and publish it. Selfish miners reveal the secret chain, which is longer than the public longest chain. Honest miners discard block `13a` (in yellow) and start to follow the new longest chain (block `14b`). The computational power used to mine block `13a` gets wasted and the honest miners do not get any revenue.

Figure 4.4: Illustration of the Selfish Mining strategy.

Selfish Mining has been proposed in 2013 by Ittay Eyal and Emin Gün Sirer [24]. In their paper, they show that both the selfish and the honest miners waste some resources, but the honest miners waste proportionally more: overall, the rewards share for the selfish miners is higher than the proportion of computational power they own. In practice, this gives the selfish miners an advantage over the others and encourages rational miners to join the group of selfish miners. According to the paper, relatively small group of miners can adopt the selfish strategy and attract many other miners, with the risk of gaining the majority of the mining power (see Section 4.2).

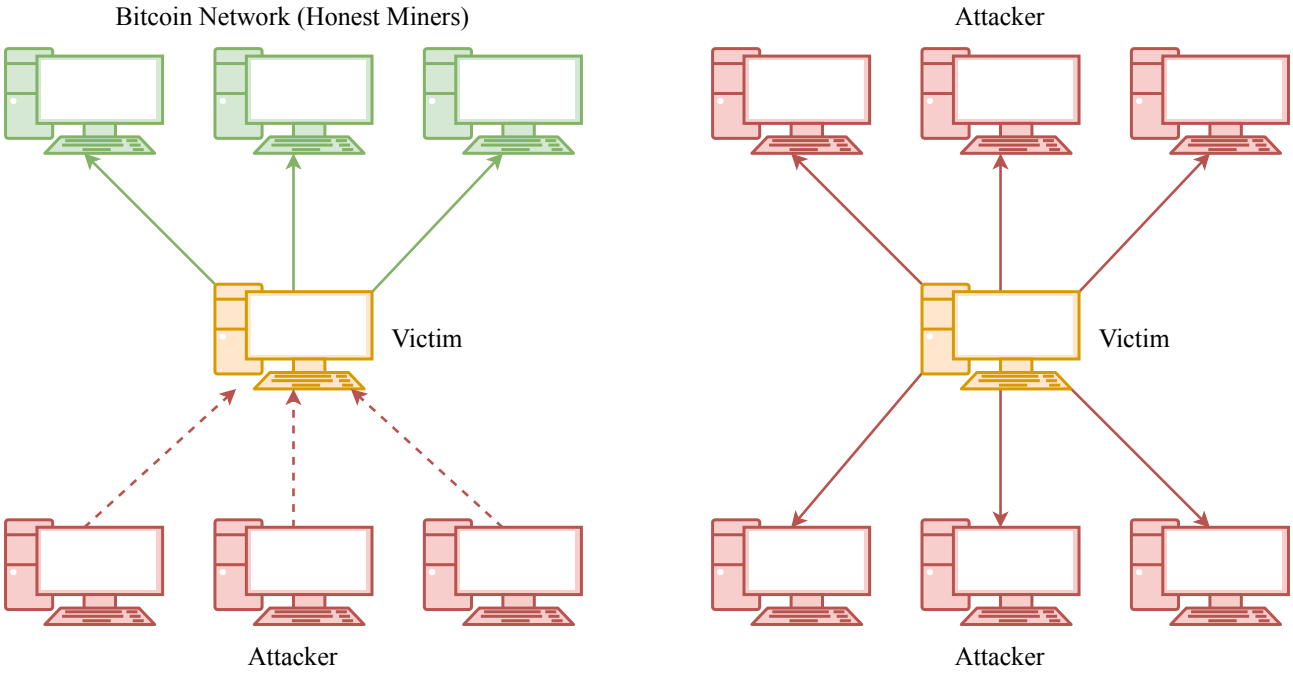
Some work has been done in academics to investigate alternative mining strategy [19, 68].

Nayak Kartik et at. have proposed Stubborn Mining [58], a strategy heavily inspired by Selfish Mining, but that accepts some more risk (e.g. keep mining on chains shorter than the longest one) in return to a higher expected revenue. They claim that stubborn mining strategies can beat Selfish Mining by up to 25%.

Ayelet Sapirshtein, Yonatan Sompolinsky and Aviv Zohar have investigated the space of possible selfish mining strategies, looking for the optimal strategy for selfish miners [65]. They also show that selfish mining strategies can be combined with double-spending attempts to make them even more profitable for selfish miners.

4.4 Eclipse Attack

The general idea of the Eclipse Attack [69] is to force a victim to connect only nodes under the control of the attacker: if the attacker succeeds in doing that, it is able to control all the network traffic to and from the victim. In the specific case of Bitcoin, the attacker can mount many different attacks against the victim, for example a double-spending against a merchant.



(a) At the beginning, the victim is connected to some honest miners in the Bitcoin network. The attacker opens new connections against the victim from many different malicious nodes under its control. If the attack succeeds, the victim will eventually forget the addresses of honest nodes and fill its tables with addresses of malicious nodes.

(b) When the victim reboots, it connects to known nodes. Since its tables are full of addresses of hosts under the control of the attacker, the victim will connect only to malicious nodes. The attacker can now decide which messages to forward to the victim and which not to: the attacker can effectively eclipse the victim from the peer-to-peer network.

Figure 4.5: Illustration of an Eclipse Attack.

Each Bitcoin node maintains two tables of addresses [33]: the `tried` table contains addresses of peers that were successful connected with the current node at least one in the past; the `new` table contains addresses discovered through any of the peers-discovery strategies used by Bitcoin (see Section 3.1.1) to whom the node has not yet initiated a successful connection.

A Bitcoin running the original client [51] with the default settings can have up to 8 outgoing and up to 117 unsolicited incoming connections, for a total of 125 [6, 33] (see Chapter 3 for the details). Connections are long-lived, i.e. they are interrupted only if either one of the two nodes goes down or in case of networking problems. Each Bitcoin node always tries to have exactly 8 outgoing connections, in order to maintain a solid network topology. If the number of outgoing connections gets less than 8, the node randomly chooses a new IP address from the `new` table and tries to connect to the corresponding node: if the connection is not successful, the node tries with the next address.

An Eclipse Attack against Bitcoin [33] is illustrated in Figure 4.5 and works as follows:

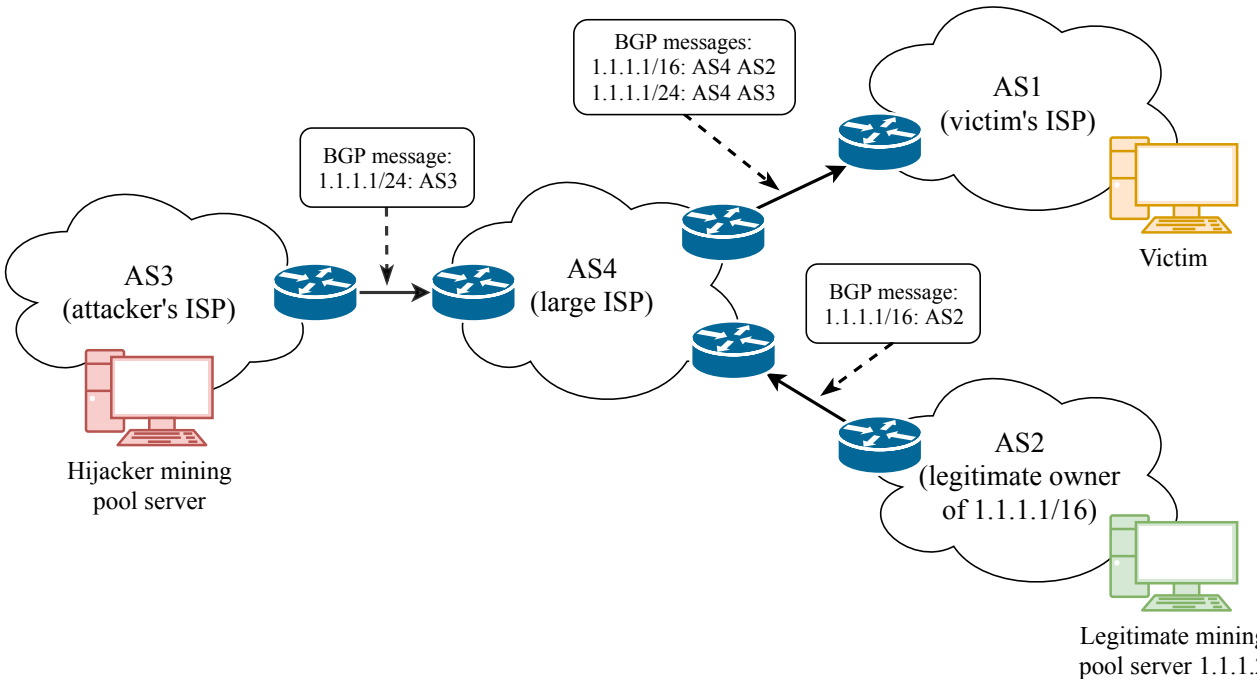
- the attacker rapidly and repeatedly opens new unsolicited connections to the victim from a set of malicious nodes under its control; when a new connection is establishes, the attacker also sends unsolicited `Addr` messages containing a list of up to 1000 "trash" addresses (IPs of host that do not run Bitcoin or addresses under the control of the attacker);
- if the attack succeeds, the victim will eventually replace all addresses stored in both the `tried` and the `new` tables with the IP addresses of the malicious nodes;
- the attacker forces the victim to restart [21, 17] or simply waits for it to restart naturally, for example after a software update or after some failure;
- when the victim restarts, it will connect to 8 malicious nodes randomly chooses from the `tried` and the `new` tables;
- the attacker effectively isolates the victim from the honest nodes in the network and controls which messages, block and transactions to relay, delay or drop.

The attacker can mount many different attacks against eclipsed nodes. If the victim is a miner (or a mining pool), the attacker can adopt strategies similar to the Selfish Mining to waste the computing power of the eclipsed nodes: the attacker can drop `Block` messages containing blocks discovered by the victim and forward blocks from the network that conflict with the ones found by the victim. If the victim is a merchant, the attacker can easily mount double-spending attacks. It can send a transaction `tx-1` to the merchant, and a conflicting one `tx-2` to the Bitcoin network: since the attacker controls all of merchant's connections, the merchant might not be able to broadcast `tx-1` to the network and the attacker might be able to double-spend its bitcoins with `tx-2`.

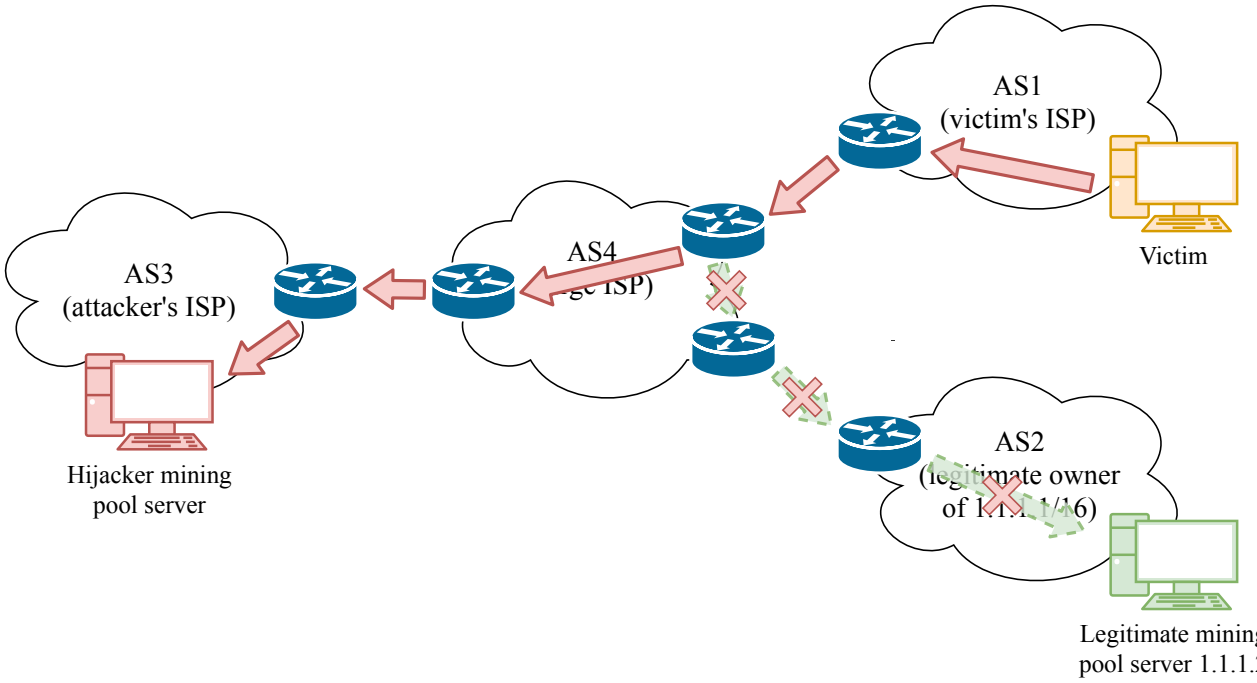
4.5 BGP Hijacking

Border Gateway Protocol (BGP) is a standardized network protocol used to exchange routing information between autonomous systems (AS) [77] of different ISP and organizations on the Internet [86, 78]. The primary function of BGP is to exchange information about network reachability with other BGP systems. The information collected and distributed by the protocol are used to construct a graph of connectivity between different AS, which can be used to make optimal routing decisions among the networks of different parties. BGP routing also depends on policies and rules configured by network administrators.

While not directly related to Bitcoin, BGP is fundamental for the working of Internet, and thus for each distributed systems build on top of that. Under certain conditions, attacks against BGP can affect Bitcoin and some of the guarantees given by the protocol. Researchers at Dell SecureWorks Counter Threat Unit discovered an attacker that repeatedly hijacked traffic destined to networks belonging to Amazon, Digital Ocean, OVH, and other large hosting companies between February and May 2014 [44]: they registered 51 compromised networks from 19 different ISP. Attacks with a similar scale were also reported between October 2015 and April 2016 [2, 4].



(a) Broadcast of the malicious route. The legitimate miner (in green) has IP address `1.1.1.2` on the network `1.1.1.1/16` of `AS2`. `AS2` broadcasts its networks to the neighboring AS. The attacker (in red) tries to hijack the victim's connection: since `AS4` is paired with `AS3`, the BGP message from the attacker is accepted. The malicious route is more specific than the legitimate one, so it is overridden in the `AS4` routing tables.



(b) The victim tries to connect to the legitimate mining pool server at IP `1.1.1.1`. The IP packets are correctly routed to `AS4`, but then `AS4` sends them to `AS3` instead of `AS2`: the victim connects to the attacker instead. The routing is indicated with red arrows; green arrows with red crosses indicate what the correct routing would be.

Figure 4.6: Illustration of a BPG Hijacking attack against a Bitcoin miner.

A BGP attack requires a malicious AS and some manual configuration. A network administrator of the malicious AS must configure BGP to announce wrong ranges of IP addresses: the other AS will learn wrong routes and override the correct routing rules. If the attack is successful, some Internet traffic destined to the hijacked networks goes through the malicious AS and can thus be intercepted, analyzed and manipulated by the attacker in any ways. Since all the Bitcoin traffic is in clear and there in no authentication between different peers, Bitcoin is vulnerable to hijacking [2]. Similarly, traffic of the Stratum protocol used in by mining pools is vulnerable to routing attacks.

Figure 4.6 shows an example of BGP hijacking used against a Bitcoin miner running the Stratum protocol. The attack works as follows:

- the miner continuously connects to a legitimate pool, asking for tasks to perform;
- the attacker publishes new BGP routes to intercept the victim's Stratum network traffic;
- when the miner attempts to connect to a legitimate pool, some BGP route directs the traffic to a pool controlled by the attacker;
- the malicious pool issues a `client.reconnect` [71] command to instruct the miner to connect to a new pool maintained by the attacker;
- the victim connects to the second malicious pool;
- the attacker ceases the attack, since BGP hijacking is not needed anymore at this stage;
- the miner performs the assigned tasks correctly, but does not get any revenue from the malicious pool, effectively wasting its computing power for free.

In the paper "Hijacking Bitcoin: Routing Attacks on Cryptocurrencies", Maria Apostolaki et Al. show that large scale networks attack against Bitcoin are possible and have been already done in the past. They claim that anyone with access to a BGP-enabled network and able to hijack less than 900 prefixes (about 0.15% of the total) can perform attacks commonly believed to be hard, such as isolating 50% of the mining power. Attackers controlling some networks can potentially:

- force miners to connect to a malicious pool which exploits their mining power for free;
- isolate part of the network by delaying or dropping messages to or from the victims (see the Eclipse Attack described in Section 4.4);
- perform double-spend attacks against merchants;
- take control of a significant amount of computing power and attempt various kind of mining attacks (see Section 4.3).

4.6 Balance Attack

The Balance attack is a new attack proposed by Christopher Natoli and Vicent Gramoli first in 2016 in the technical report [56] and later in 2017 in their paper "The Balance Attack or Why Forkable Blockchains Are Ill-Suited for Consortium" [57]. The attack targets *forkable* blockchains, i.e. a blockchain that allow multiple branches at the same time. The most famous examples of systems using a forkable blockchains are Bitcoin [50] and Ethereum [85].

Forkable blockchains implement some strategy to solve conflicts: for example, Bitcoin always follows the longest chain if multiple branches exists. The rule guarantees that the system will eventually agree on the same longest chain, and thus on the same set of transactions stored in the blockchain. Unfortunately, blocks in branches different from the longest chain are wasted, since their content is not recognized by the network. Wasted blocks are probably the biggest problem of forkable blockchains: they waste computational power of the network and facilitate double-spending attacks [64].

Except for attacks, network delays are the main cause of forks in the blockchain [22]. Honest miners immediately stop the mining process if they receive a new valid block that attaches to the longest chain and they start to mine the following block; the delay between a block discovery and its broadcast may allow other miners to complete and publish a conflicting block, effectively creating a fork in the blockchain. In practice, an attacker can increase the amount of wasted blocks in the system and slow down the growth of the longest chain by simply delaying the propagation of new blocks [57].

The Balance attack takes advantage of this idea. The attacker partitions the Bitcoin nodes by disrupting communications between different subgroups: the messages containing new blocks are delayed or possibly dropped completely. The paper suggests to select groups of similar computing power to maximize the probability of success of the attack. Figure 4.7 shows an example of Balance attack, where the attacker controls one router and is able to filter and manipulate traffic for the nodes connected to Internet through that router. Multiple results recently demonstrated how simply a motivated attacker can perform networking attacks on the Internet and thus delaying messages in blockchain networks [44, 33, 2].

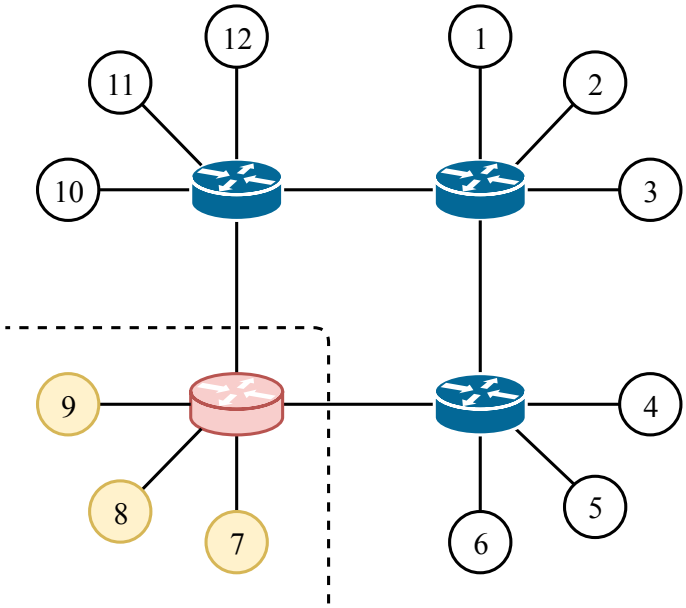


Figure 4.7: Illustration of a network partitioning caused by a Balance Attack. The attacker controls the bottom-left router (in red) and delays all communications between pair of nodes (a, b) where $a \in \{7, 8, 9\}$ and $b \notin \{7, 8, 9\}$: nodes inside the same partition can communicate with each other without any delay.

In details, the attack works as follows:

- the attacker identifies subgroups of miners with a similar mining power; different works in the literature [5, 47] describe techniques to learn the network topology and identify influential nodes;
- the attacker identifies the routes of Bitcoin messages between different subgroups over Internet; let us assume the miner finds a partitions of nodes into two groups `G1` and `G2`;
- the attacker runs some networking attack (man-in-the-middle, BGP hijacking [44]) to take control of the communications links between groups and delays the Bitcoin traffic;
- because of the delay, transactions and blocks are not immediately propagated to the entire network, so `G1` and `G2` are very likely to have different views of the blockchain and to mine conflicting blocks;
- the attacker takes advantage of these inconsistencies and performs double-spending attacks.

The authors implemented and tested the attack against a private Ethereum test network with characteristics similar to R3, a consortium of more than 70 word-wide financial institutions. The testnet consisted of 18 machines running Ethereum and mining blocks (the mining process in Ethereum is based on PoW and is similar to the one of Bitcoin, at least for the matter of the Balance attack). The authors report that a single machine needs to delay messages for 20 minutes to double spend, while an attacker controlling a third of the mining power only needs a delay of 4 minutes to achieve 94% of attack success rate.

Chapter 5

Simulator

Testing and evaluating a peer-to-peer protocol with thousands of nodes in a real environment is very difficult, expensive and time consuming, especially if the goal is to evaluate the effects of large-scale network attacks. For this reason, we decided to implement a simulator that measures the performances of the Bitcoin protocol under the different situations, at rest and under attack. This allowed us to simulate up to 10,000 Bitcoin nodes on a single general-purpose computer in a short time and evaluate the behavior of the protocol under different attack scenarios.

This chapter describes the implementation of the simulator. It covers the main concepts of discrete event simulation, the basics of PeerSim, the design of the simulator, the simplifications with respect to the complete Bitcoin protocol, and the metrics used to measure and evaluate the overall performances.

5.1 Simulation

According to Robert E. Shannon, *simulation* is "the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and / or evaluating various strategies for the operation of the system" [67]. By *model*, he means an abstract representation of an entity of group of objects, and by *system* a collection of elements that interact with each other to accomplish some objective. According to Shannon, simulation has a number of advantages [67]:

- it is often easier to understand than analytical or mathematical models;
- it is usually more credible than models, since it requires less simplifying assumptions and represents the system more accurately;
- it allows to test new designs, systems or protocols before implementing them;
- it allows to verify hypothesis by measuring their effects on the systems;
- it helps to understand better how the modeled system works and which variables are the most important with respect to the performances;
- it allows to changes the initial situation and test the system in different settings.

Simulation is used in many contexts, for example safety engineering, economics, physics and even video games [84]. There are many different approaches to simulations, each one adapted to a specific purpose. We focus on discrete event simulation, used by our simulator.

5.2 Discrete event simulation

A discrete event simulation models the system behavior as a discrete sequence of events in time. An event can represent anything, for example the arrival of a message to a node in the system or a network

timeout. Each event occurs at a particular instant of time and can cause changes in the state of the system. No event occurs between two consecutive events, so the simulation can simply jump from one event to the next. The results of the simulation can be evaluated with measurements: the metrics can be computed either online during the simulation run, or computed offline from the simulation logs.

Algorithm 1: Discrete Event Simulator

```
// initialization
state ← initializeState()
events ← initializeEvents()
queue ← new PriorityQueue(events)

// simulation loop
while ¬ queue.isEmpty() or exitConditionReached(state) do
    // process the next event
    event ← queue.deleteMin()
    newState, newEvents ← processEvent(event)

    // update the system status
    state ← newState
    queue.insertAll(newEvents)
end
```

Algorithm 1 illustrates the working of a discrete event simulation engine. The simulation has some starting state that represents the initial condition of the system. All events are stored in a priority queue sorted by event time. The queue is initialized with some events. Events in the queue are processed one at a time: the first event is removed from the queue and processed by the simulator. An event can cause other events to occur in the future and change the current state of the system. Discrete event simulators take advantage of pseudorandom number generators to emulate random variables of the system [80]: whenever an event is influenced by some random factor external to the system (e.g. latency of a TCP connection over the Internet), the simulator extracts a random variable from some distribution using the random number generators. The simulation stops when a certain condition is reached (for example, a target simulation time is reached), or when the queue of events is empty.

5.3 PeerSim

PeerSim [48] is an open source peer-to-peer systems simulator engine developed at the University of Bologna and the University of Trento. It is written in Java and its aim is to help the research and evaluation of large peer-to-peer. It has been developed with high scalability in mind, in order to support simulations with up to 1 million nodes. It is released under the GPL open source license and is available for download on SourceForce [61].

PeerSim is composed of two simulation engines, a simplified cycle-based one and an event-driven one. The cycle-based engine uses some simplifying assumptions to achieve better performances and scalability, such as ignoring the details of the transport layer in the communication protocol stack; it has been tested up to 1 million nodes [59]. The event-based engine is less efficient but more realistic and allows to easily simulate the entire network stack; it has been used for simulation of up to 250,000 nodes [59]. Both engines support many simple and extendable components, which are plugged together through a flexible configuration mechanism. Since our simulator is event-driven, this chapter focuses on the event-based engine only.

5.3.1 Components

Each component in PeerSim is created as a simple Java object that implements some interfaces defined by the engine. Figure 5.1 gives an overview of the main components of a PeerSim simulation.

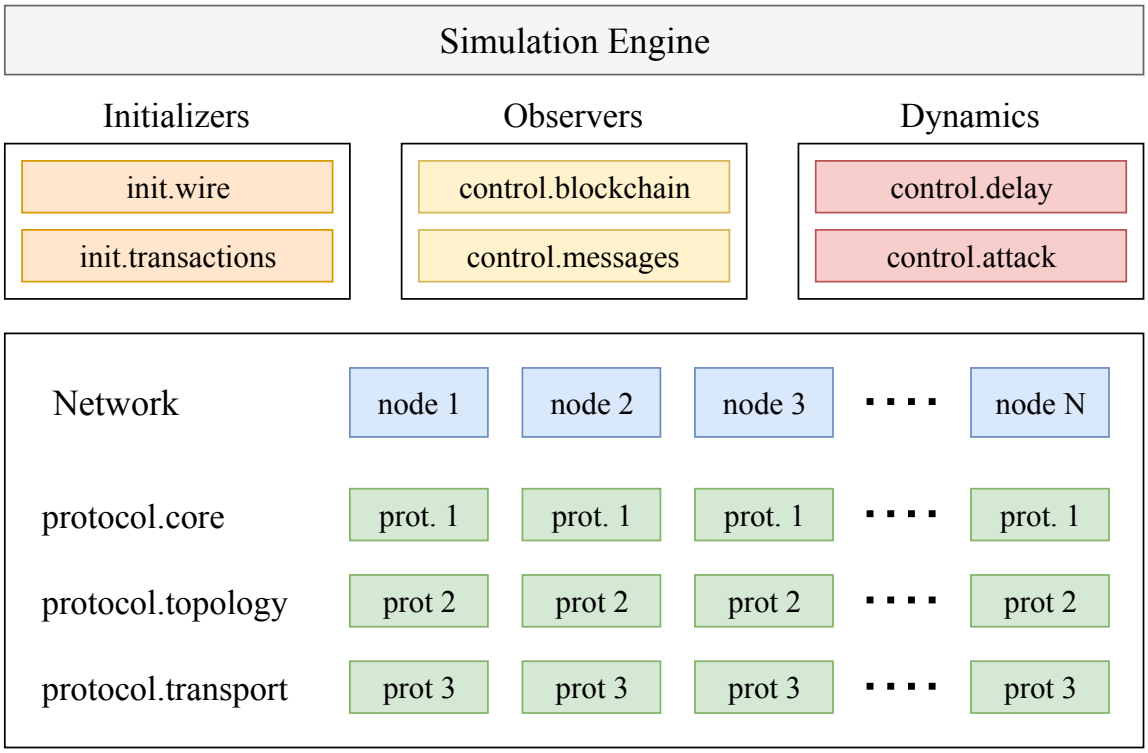


Figure 5.1: Illustration of the main components of PeerSim.

EDSimulator

The `EDSimulator` is a static singleton that implements the event-driven simulator engine. It is the entry point of each simulation and it is responsible to manage its entire life-cycle: load the configuration, bootstrap the network, run the initializers, keep the queue of events, schedule controls, and run the simulation loop. It provides a method that can be used by any component to add new events to the queue. The class is also able to run the same simulation multiple times using different seeds for the random number generators, in order to achieve a higher statistical significance.

Event

PeerSim does not have any class or interface to model an event. Events are simply represented as plain Java objects, which are passed by reference to the simulator engine and to the target components.

Network

The `Network` class is a static singleton that keeps track of all nodes in the simulation. Nodes are internally stored in an array and can be accessed by ID (their position in the array). Information about neighbors of a node is stored in a special protocol `Linkable`. It is used by all components that need to access a specific node or iterate over the available nodes.

Node

The network is composed of nodes. A node is a container of protocols (each simulation can have one or more protocols). The `Node` interface provides access to the protocols it holds, and to a fixed unique identifier of the node. The behavior of a node is defined in each of the attached protocols.

EDProtocol

The `EDProtocol` interface defines a protocol that implements an event-driven behavior. Protocols define the actions each node will perform during the simulation. The interface has only the method `processEvent`, which is invoked by the scheduler to deliver events to the protocol. Each node stores an array or protocols; a class implementing this interface will have one instance for each node in the network. Protocols can be stacked on each other to build more and more complex behaviors: for example, an implementation of `EDProtocol` might be responsible to build and maintain a topology, while a second implementation may simulate some higher level functions, such as broadcasting some objects to the entire network in a gossip style using the underlying topology. The simulation settings are specified with a configuration file; some settings can be overridden with command line parameters.

Linkable

The `Linkable` interface provides a service to other protocols to access a set of neighbor nodes. The instances of the same linkable class define an overlay network: each instance has a list neighbors nodes. Links do not need to be symmetric: in some cases it could make sense to work with a directed graph. The `Linkable` interface is usually implemented by protocols that exchanges messages over the network to build a topology.

Transport

The `Transport` interface models a transport layer in the OSI model [82]: it is used to send messages through the underlying network. Implementations of the `Transport` interface use the `EDSimulator` class to schedule the delivery of messages with some appropriate delay. They can also model packet loss of packets and other failures. Different transports can be stack on top of each other to model complex behaviors of the network.

Control

Classes implementing `Control` are used to define operations that require global network knowledge and management. They can be scheduled for execution at certain points during the simulation, depending on their function.

Initializers Initializers are executed at the beginning of the simulation and are used to bootstrap the simulation. Examples include creating the initial network topology, set the initial nodes' states and schedule the first events.

Dynamics Dynamics are executed periodically during the simulation and are used to modify the simulation in some way. Examples include adding or removing nodes, creating a failure on the network, and introducing an extra delay for messages.

Observers Similar to dynamics, observers are executed periodically during the simulation. They are used to read aggregated valued from nodes, and compute and collect statistics and metrics.

Configuration

`Configuration` is a static singleton class that provides access to the configuration for the current simulation. It proxies all accesses to the configuration file and it allows to plug different strategies to load the configuration. The configuration is stored as `<name, value>` pairs. The class provides helper methods to perform common operations, such as reading values with type checking, resolving protocols by name, parsing mathematical expressions (that can be used in the configuration file), and resolving under-specified class-names.

CommonState

`CommonState` is a singleton static class that stores the state common to all components in the simulation. Its main purposes is to simplify the simulator structure and increase the efficiency by avoiding some parameters passing. It provides access to an instance of a pseudorandom number generator, already initialized with the provided seed and ready for use: all components should use this instance to guarantee reproducibility of results. Also, it stores the current time of the simulation.

RangeSimulator

The `RangeSimulator` is a utility class able to run simulations with ranges of parameters specified in the configuration file. A range is a collection of values to be assigned to a variable. If multiple ranges are specified, the class runs a simulation for all the possible combinations of values. `RangeSimulator` invokes the standard PeerSim simulator engine to run each single experiment.

5.3.2 Simulation life cycle

Each PeerSim simulation has the following life cycle:

- load and validate the configuration; the configuration file defines which components are used and how they interact;
- run the initializers, in the order provided in the configuration file;
- schedule dynamics and observers, according to the parameters specified in the configuration;
- start the simulation loop and process one event at a time, until the maximum simulation time is reached, the queue of events is empty or some control terminates the simulation;
- run the controls scheduled for execution after the simulation;
- cleanup and schedule the next repetition of the experiment.

5.4 Simulator design

Our simulator implements the Bitcoin protocol. Since the final aim is to evaluate the effect of large scale network attacks against the protocol, we put particular emphasis on the details of the topology layer described in Section 3.1; the core layer (Section 3.2) is a bit simplified and does not cover the advanced features of Bitcoin transactions.

The code is divided into 4 main parts: topology, core, optimized data structures and utilities, and attack. Topology and core have their own Java packages, which contain events, messages, initializers, and observers specific to the PeerSim protocol.

5.4.1 Topology

The topology layer is implemented as a `EDProtocol` that implements the `Linkable` interface: it is responsible to build the overlay network that the core layer uses to propagate blocks and transactions. Its main functions are to bootstrap the network (when a new node join, it needs to discover some peer and connect to them) and to maintain the topology.

The discovery mechanism is simulated as follows:

- nodes start with empty lists of incoming and outgoing connections;
- an initializer simulates the DNS peer discovery mechanism by populating the list of known peers of each node with about 25 randomly selected nodes;
- an initializer schedules a special event `StartEvent` to each node;

- nodes start the protocol and begin to connect to some known nodes, in order to open up to 8 outgoing connections;
- when a connection is established, nodes exchange `Addr` messages; at the same time, nodes start to exchange `Ping` and `Pong` messages to maintain the newly build topology;
- nodes propagate the `Addr` messages, as described in Section 3.1.3.

The backup discovery mechanisms based on hard-coded seeds, database of known addresses and daemon configuration are not implemented. Our implementation covers the behavior of the protocol under normal conditions.

Simplifications

The implementation of the topology layer covers almost all details explained in Chapter 3. In order to make the simulation faster to run and easier to understand, we simplified some details of the protocol:

- all nodes have a single IP address and are reachable from each other node;
- all nodes follow the protocol, so there is no need to maintain a ban list or to implement the `Reject` control message;
- all nodes are miners and run exactly the same version of the protocol, so the `Version` and `VerAck` messages are simplified;
- nodes do not crash or leave the network; since we simulate large scale attacks over small periods of time, we do not expect a significant churn in nodes in the network.

Metrics

Each node in the simulator records a number of metrics, which are periodically collected and aggregated by the observers:

- number of messages sent to other peers, grouped by type;
- number of nodes removed because of timeouts;
- overall distribution of the number of connections established by each node.

The metrics are mainly to used check if the behavior of the topology layer is reasonable and thus to verify that the implementation is correct.

5.4.2 Core

The core layer is implemented as another `EDProtocol`, which uses the topology layer as a `Linkable` implementation. The main functions are to simulate discovery and propagation of blocks and gossip of transactions.

Transactions Transactions are scheduled by an initializer. Before the simulation, the initializer generates a list of transactions according to a Poisson process [83]: each transaction is scheduled to a single node, chosen following a uniform probability distribution. During the simulation, each transaction is received as an event by a single node, which then sends it to its neighbors. Eventually, all nodes in the network receive the transaction.

Blocks Each block contains a series of transactions and must include the solution to computational puzzle to be valid. We assume that all nodes in the network mine blocks independently from each other (we do not model mining pools) and have the same computational power. To solve the challenge, each node uses a brute-force approach and tries difference nonces, until a valid value is found or another node completes the current block. We observe that:

- the probability of success in a single nonce trial is negligible (Bitcoin regulates the target so that one block is published every 10 minutes on average);
- each miner solves a different problem, since its address is included in the block;
- peers compute the PoW independently; as such, the probability that one of them succeeds does not depend on the progress of the other nodes;
- the time spent on the previous block does not influence the time needed to find the next one;
- miners stop the search and start from scratch with a new block frequently.

For these reasons, we model each single miner as a Poisson process: the time expected for a single miner to complete a block follows an exponential distribution.

To simulate the mining process, each node x extracts a random variable δ from an exponential distribution and schedules the discovery of the next block at time $t+\delta$, where t is the current simulation time. If another node y publishes a block in the time interval $[t, t+\delta]$, the node x stop the process and schedules the next block from instant it receives the block from y; the block scheduled for time $t+\delta$ is cancelled. As soon as a node manages to finish a block, it broadcast it to all the neighbors and schedules the next one.

Simplifications

Since the focus of the simulation is to evaluate the performances of the protocol under different network conditions, we have a series of simplifications:

- transactions are represented as simple tuples `⟨sender, amount, receiver⟩`, instead as a sequence of instructions written in a custom scripting language;
- since all nodes join the network together, there is no need to implement the initial block download procedure;
- all nodes behave correctly, so there is no need to verify the signature of transactions or the proof-of-work of blocks;
- all nodes in the network perform some mining;
- all miners have the same computational power.

Metrics

The main metric used to measure the performances of the core layer is the number of forks in the blockchain: a fork is a branch of the chain different from the longest chain. The simulator knows the global state of the simulation: it is easy to compute the longest chain, observe forks and their lengths in any given moment. The metric considers all blocks known to the simulator, even if they have been just discovered and not yet known to all peers in the network.

5.4.3 Data structures and utilities

The simulator uses custom data structures optimized for some specific tasks. Other utilities include a class to run range simulations in parallel and the implementation of the exponential probability distribution.

CircularQueue

`CircularQueue` is an implementation of a queue based on an array. It allows to add elements to the queue, read or remove the oldest element, check if the queue is empty. The elements are stored in the array, starting from left to right; new elements are always added to the right size, old elements are removed from left. Two pointers keep track of the first and last element in the array. The queue is able to automatically resize in the underlying array is full: when a new element is added and the queue in full, an array with double capacity is allocated and all elements are moved there. The cost of the enqueue operation in this scenario is $\mathcal{O}(n)$, where n is the original capacity of the queue; since the operation is performed only once every n inserts, the amortized cost of the operation is $\mathcal{O}(1)$. All other operations have always $\mathcal{O}(1)$ cost, since elements are stored in an array.

Algorithm 2: CircularQueue

```
// constructor
CircularQueue CircularQueue(integer n)

// add an element to the queue
void enqueue(object item)

// remove the oldest element from the queue
object dequeue()

// get without removing the oldest element from the queue
object head()

// check if the queue is empty
boolean isEmpty()
```

The `CircularQueue` data structure is used in the implementation of the topology layer to handle the `Addr` messages propagation: `Addr` messages are usually not immediately sent to the target node: at each round, a random peer is selected and its queue of `Addr` messages is flushed (see the trickling procedure described in Section 3.1.3). The queue for each peer is potentially unbounded, so the implementation needs to be dynamically resized when needed. The common operations are enqueue (to schedule a new `Addr` message) and dequeue (to flush the queue): all operations have $\mathcal{O}(1)$ cost.

ParallelSimulator

The `ParallelSimulator` is a utility class able to handle ranges and generate the list of their combinations: parameters are read from the configuration file using the standard PeerSim mechanisms; the class computes their combination and prints them as a list of commands, ready to be executed from the command line. This class is particular useful when combined with the GNU Parallel, a shell tool for executing jobs in parallel using one or more computers [73]. Discrete event-driven simulations are very difficult to parallelize, since events have causal relationships which are usually not well known in advance and are influenced by random variables: in other words, they run on a single core and do not exploit modern multi-core CPUs. If the memory footprint is not the bottleneck, it is possible to reduce the total completion time of a simulation with range parameters by running them in parallel, one per available core: `ParallelSimulator` generates the list of jobs to execute, which are then run by GNU Parallel in parallel.

5.4.4 Attack

The attack simulates a Balance attack, described in Section 4.6. The balance attack was originally tested against Ethereum, but it is thought to work also against Bitcoin. The attacker tries to partition the nodes into two groups with the same computational power: nodes do not see the blockchain status

of the other group and are likely to mine conflicting blocks. In other works, the network generates many forks that the attacker can exploit to perform double-spending attacks, as detailed in Section 4.1.

The implementation of the Balance attack in PeerSim is based on a custom `Transport` class named `BalanceAttackTransport`: the class partitions the nodes in two groups based on their IDs and delays the propagation of `Block` messages between different groups. The custom transport is stacked on top of the other transport layers used in the normal simulation. There is no need to modify any other component of PeerSim: the attack can be activated or deactivated using the standard configuration mechanism. The same mechanism is used to specify the "intensity" of the attack in terms of *delay* applied to `Block` messages. Finally, it is possible to specify a *delay* drop probability. Please note that the original paper [57] does not talk about dropping messages; however, since the attacker has control of the network, it is able to selectively delay of drop any message between pair of nodes. We evaluate the effect of both factors in Chapter 6.

Chapter 6

Results

Our simulator allows to test the Bitcoin protocol under different conditions: it is possible to choose the size of the network and the delay of messages between nodes. It is also easy to simulate network attacks by simply changing the transport layer used or by adding a new layer on top of the default one.

This chapter covers our experiments with Bitcoin. First, we analyze the protocol at rest to obtain a baseline of its performances; then, we introduce delays on the communications between nodes to check if a degraded network infrastructure affects the protocol; finally we simulate different variants of the Balance attack (Section 4.6) and compare their effects on Bitcoin.

6.1 Settings

We run each simulation multiple times with different seeds for the random number generators to obtain higher statistical significance for the results. Each run simulates 3 hours of activity of the Bitcoin protocol. The parameter that regulates the blocks creation rate of the miners is tuned so that a new block is generated on average every 10 minutes. Since we do not simulate attacks that completely isolate nodes from each other or interfere with the control messages, we have disabled the ping-pong mechanism used by Bitcoin to verify that TCP connections are still alive: this allowed us to spare computational resources and run the same simulation multiple times with more different seeds.

6.2 Parameters

The simulator accepts different parameters that allow to change the settings of the simulation. All parameters can be changed independently of each other and are specified in the configuration file. In particular, the simulator accepts the following 5 values:

1. `network_size`;
2. `delay`;
3. `balance_attack_delay`;
4. `balance_attack_drop`;
5. `balance_attack_partitions`.

Network Size The `network_size` parameter controls the number of nodes simulated. It can vary from 1 to a potentially unlimited number, which is only bounded by the available computational resources. Since the Bitcoin network has about 9000 active node on average at the time of writing, we run simulations up to 10,000 nodes.

Delay The `delay` parameter controls the average delay of messages exchanged between Bitcoin nodes. The delay of each message is randomly chosen from a uniform distribution in $[-\texttt{delay}, +\texttt{delay}]$.

Balance Attack Delay The `balance_attack_delay` parameter controls the magnitude of the simulated Balance attack: it adds an additional delay to all messages of type `Block` exchanged by nodes belonging to different partitions. The delay is added to the base network delay controlled by the `delay` parameter.

Balance Attack Drop The `balance_attack_drop` parameter controls the drop rate of `Block` messages exchanged by nodes belonging to different partitions. Please note that this is an extension to the Balance attack described in [57], since the paper does not consider dropping messages between nodes.

Balance Attack Partitions The `balance_attack_partitions` parameter controls the number of partitions of equal size in which the nodes of the network are divided by the Balance attack. The setting discussed in the original paper is to partition the network in two parts only.

6.3 Experiments

Most attacks against Bitcoin aim at double spending the money. Forks on the blockchain allow an attacker to easily achieve double spending attacks. The main metric used to evaluate the performances of the protocol is thus the number of forks in the blockchain. Forks happen with a non-zero probability even at rest in Bitcoin due to some inevitable delays on the information propagation; degraded network conditions and a wide range of attacks can increase the probability of forks and open the possibility for double-spend attempts.

6.3.1 Protocol at rest

We run the simulations for different network sizes and `delay` = 50 ms, similar to the delay of a normal TCP connection over the Internet. Under these conditions, the Bitcoin nodes exchange 8 `Version`, `VerAck`, `GetAddr` and `Addr` messages each: each node connects to exactly 8 peers and each connection generates a single exchange of the above control messages.

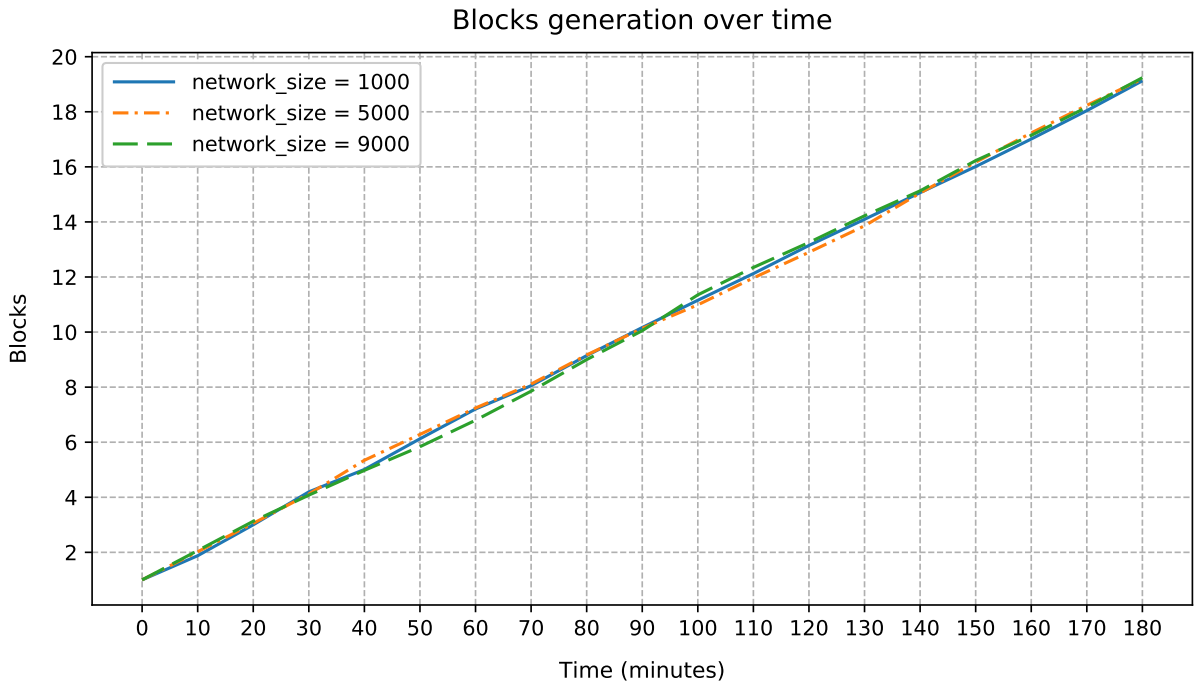


Figure 6.1: Blocks generation for networks of different sizes under normal conditions: 50 ms delay on average and no attack in progress. At time 0, only the genesis block is present. During the simulation, a new block is generated on average every 10 minutes, regardless of the network's size.

Figure 6.1 shows that the network generated on average 18 blocks during the simulation of 3 hours (1 block every 10 minutes), independently on the size of the network. The plot shows the results for networks of 1000, 5000 and 9000 nodes, but the same behavior is valid for all sizes in between. Under these conditions, the Bitcoin protocol does not produce any fork in our simulations.

6.3.2 Generalized network delays

We run experiments for different network sizes and incremental delays from 0 to 30 s with steps of 5 s in the propagation of each message. Generalized network delays on the network does not influence the number of control messages exchanged. Similar to the previous experiment, the network generates 1 block every 10 minutes on average.

Figure 6.2 shows the number of forks generated a the network of 1000 nodes over time and their distribution at the end of the simulation. The effect of network delays is a general increase in the number of forks: with a delay of 0 s, the network never generates forks; small delays of 5 or 10 s already cause the generation of 1 to 3 forks in some simulations; longer delays of 15 to 30 s produce forks in a significantly higher number of simulations.

Similar results yield for bigger networks. Figure 6.3 show the effect of delays on the forks generated by a network of 9000 nodes: the number of forks seems to be linearly correlated with the message's delay. The effect of the delays seems to be also related to the size of the network: the network with 9000 nodes produces on average 0.1 to 0.2 forks more that the one with 1000 nodes for each tested value of `delay` at the of the simulation.

6.3.3 Balance Attack

Delay

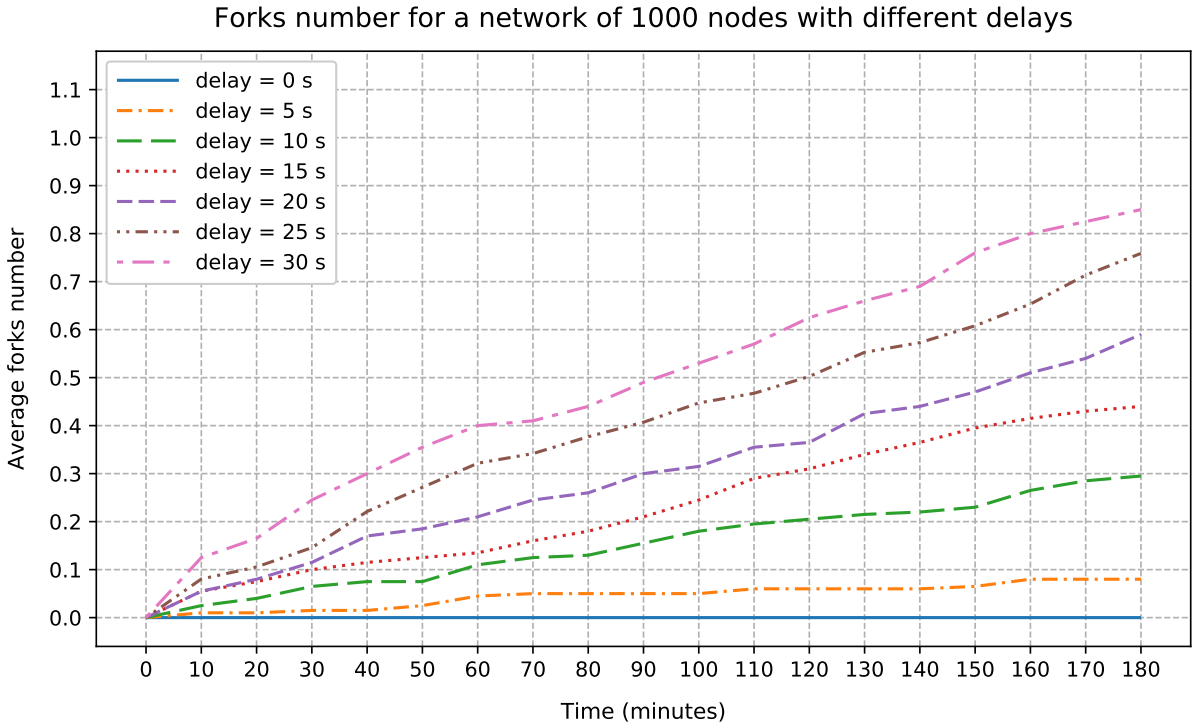
We first fix the size of the network and explore the effect of base Balance attacks with different values of `balance_attack_delay`, then we fix a delay and explore its effect on networks with different sizes. For all experiments in this section, we fix a base network delay of 50 ms.

Figure 6.4 shows the effect of a Balance attack on a network of 1000 nodes. The growth of forks over time seems to be linear for all tested delays. Even with a small delay of 10 s on the propagation of `Block` messages, the network generates on average about 0.7 forks in 3 hours of simulation; the number of forks is distributed around 1, but reaches 2 or 3 in many simulations. The distribution of the number of forks for longer delays follows a similar behavior, but it is shifted towards higher numbers: with a delay of 30 s, the network generates up to 4 - 6 forks, a very high number if compared with the 18 expected blocks.

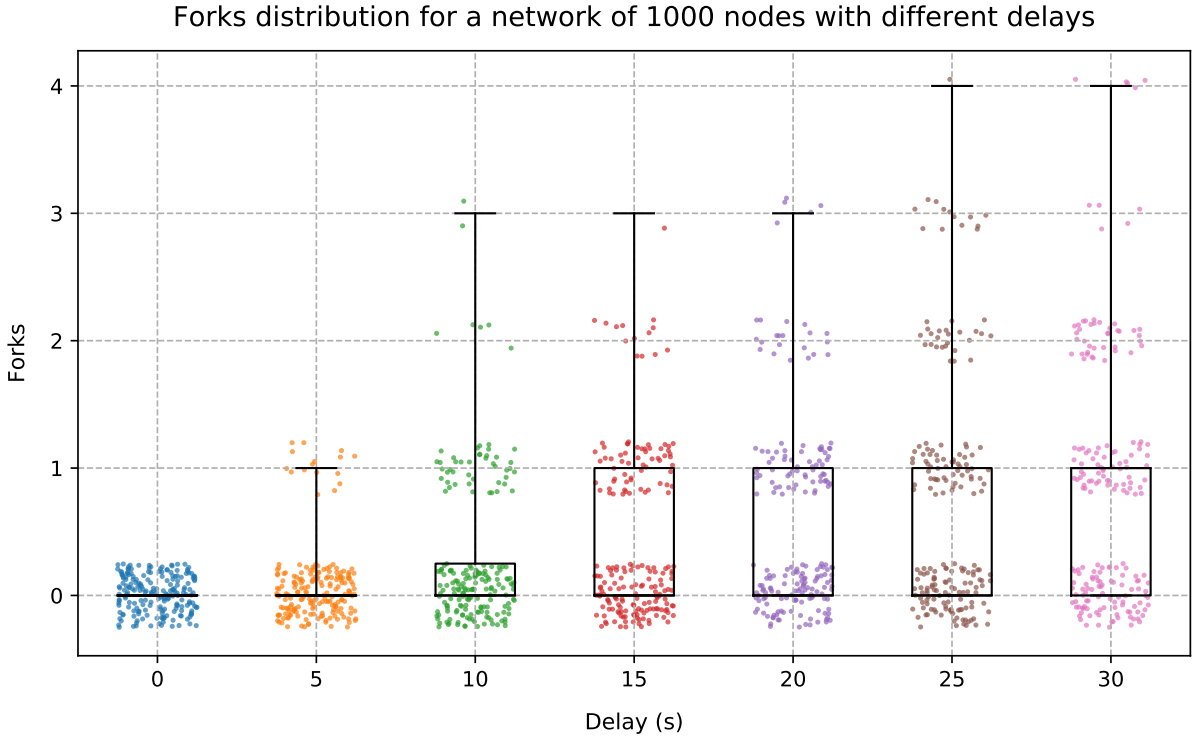
Figure 6.5 shows the effect of the Balance attack for a fixed delay of `Block` messages on network of different sizes. With both a delay of 15 and 30 s, there is no significant difference in the distribution of forks for the different networks. For all sizes, longer delays cause a higher number of forks on average.

Drop

We evaluate the effect of dropping `Block` messages instead of simply delaying them. Figure 6.6 shows the effect of a Balance attack that drops the messages with different probabilities on a network of 1000 nodes. The effect of the attack is almost absent for `balance_attack_drop` ≤ 0.7. The attack starts to work for `balance_attack_drop` $= 0.8$ and works very well for `balance_attack_drop` $\in \{0.9, 1\}$. In other words, the attack works only if most `Block` messages between nodes in different partitions are dropped. The gossip mechanism implemented in the Bitcoin protocol to tolerate loss of messages seems to work very well and makes this attack ineffective or impracticable in practice: if a `Block` message manages to reach a single node in the other partition, it is immediately gossiped to the entire partition. Delaying all messages instead of dropping some of them is a much better strategy for the attacker.

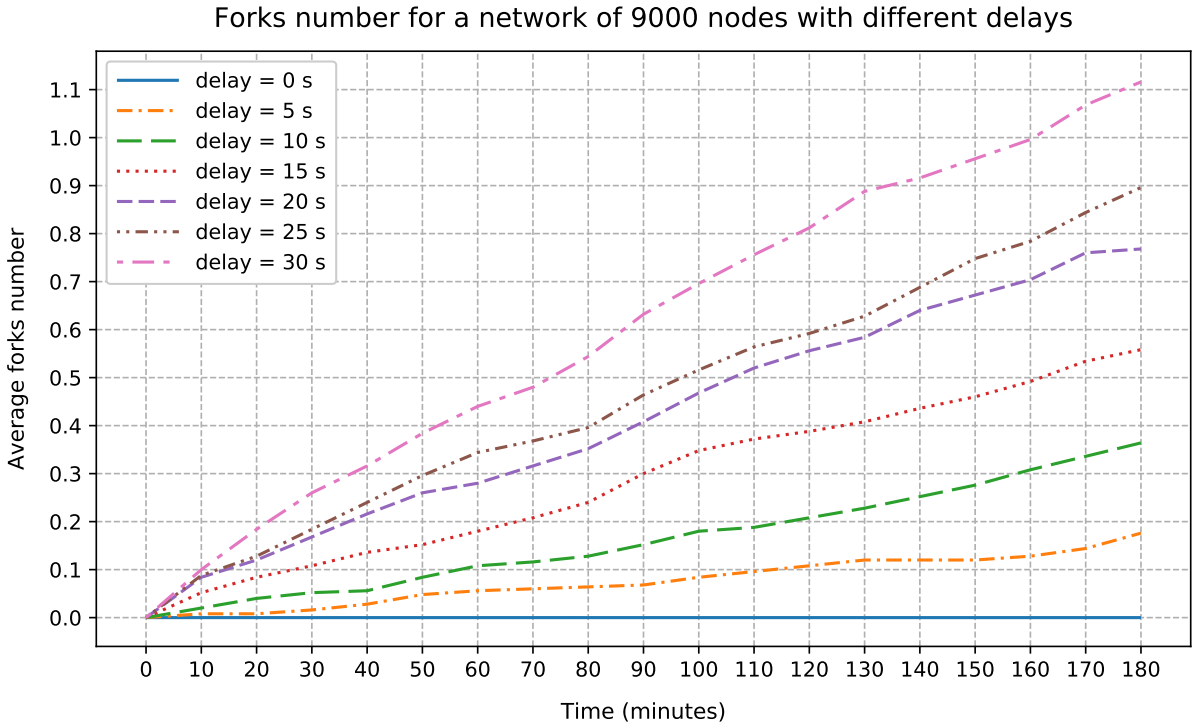


(a) Average forks number for a network of 1000 nodes with different delays. In perfect conditions of zero delay, Bitcoin nodes never generate forks. Longer delays cause a faster increasing number of forks over time.

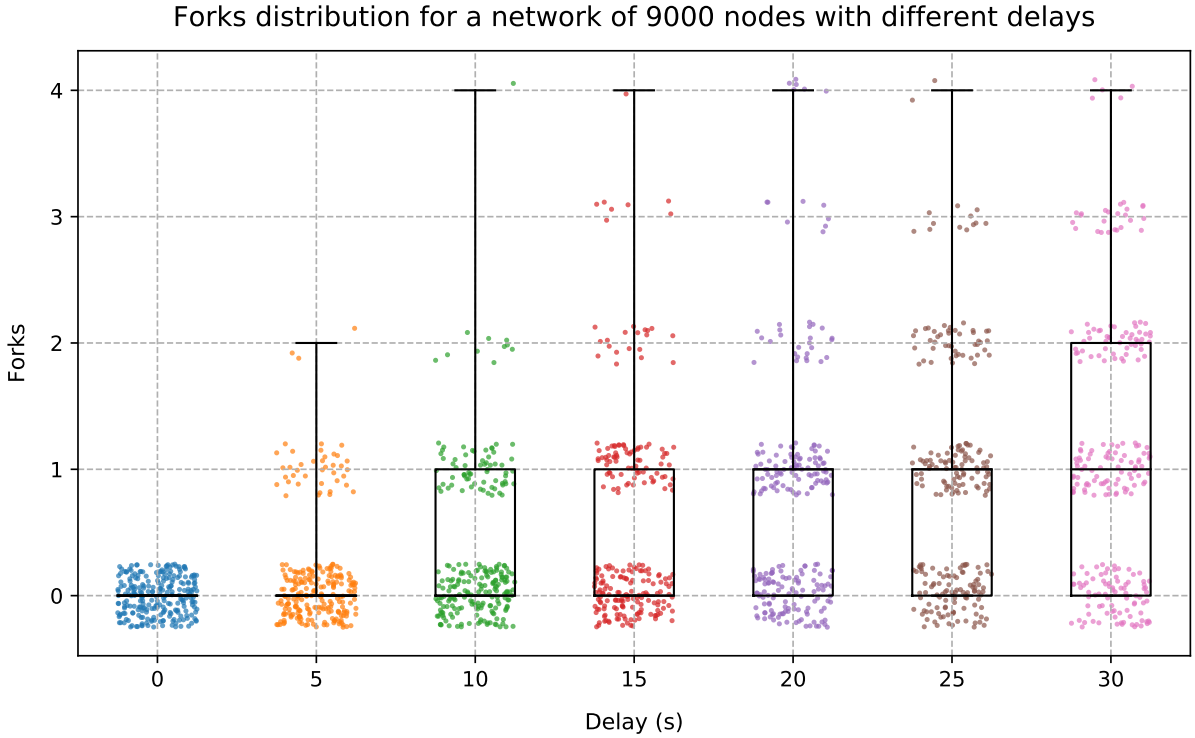


(b) Distribution of forks for a network of 1000 nodes with different delays. With zero delay, blocks are immediately distributed to all nodes and no fork is generated. Even with a small delay of 5 s, the network starts to create some forks. Longer delays produce an higher number of forks.

Figure 6.2: Forks distribution for a network of 1000 nodes with different delays.

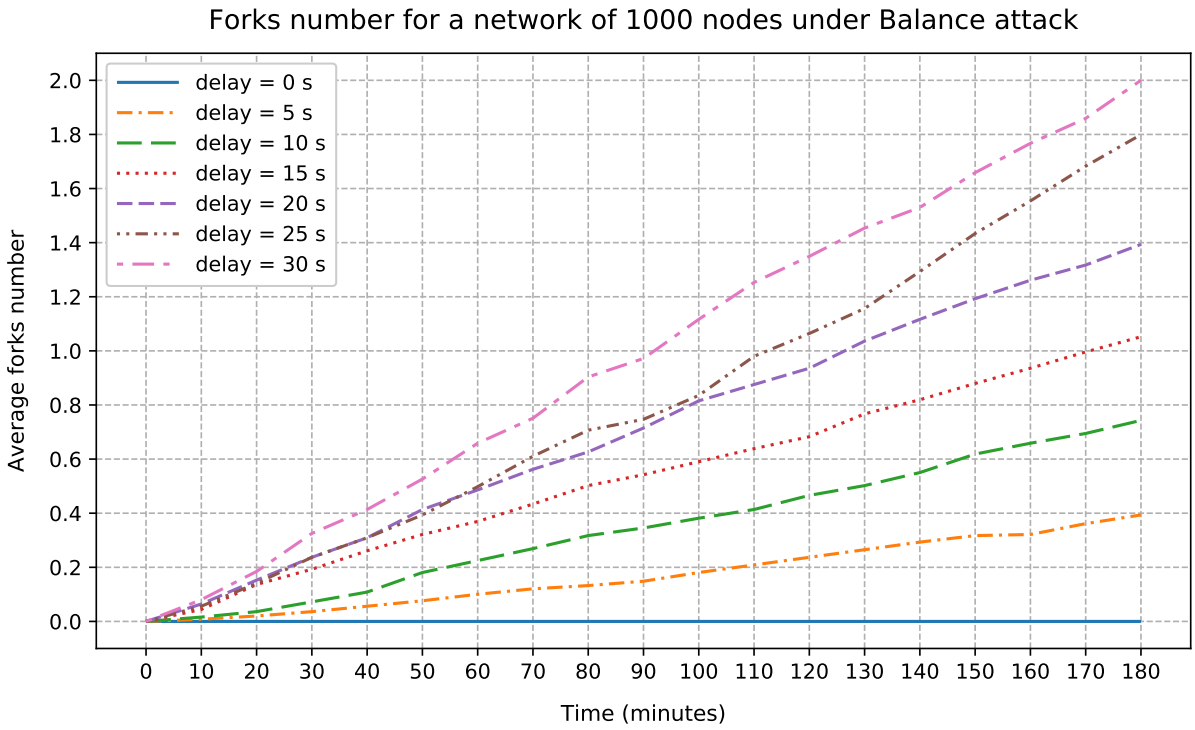


(a) Average forks number for a network of 9000 nodes with different delays. With short delays, the average number of forks is small. With longer delays, Bitcoin generates on average about 1 fork every 3 hours.

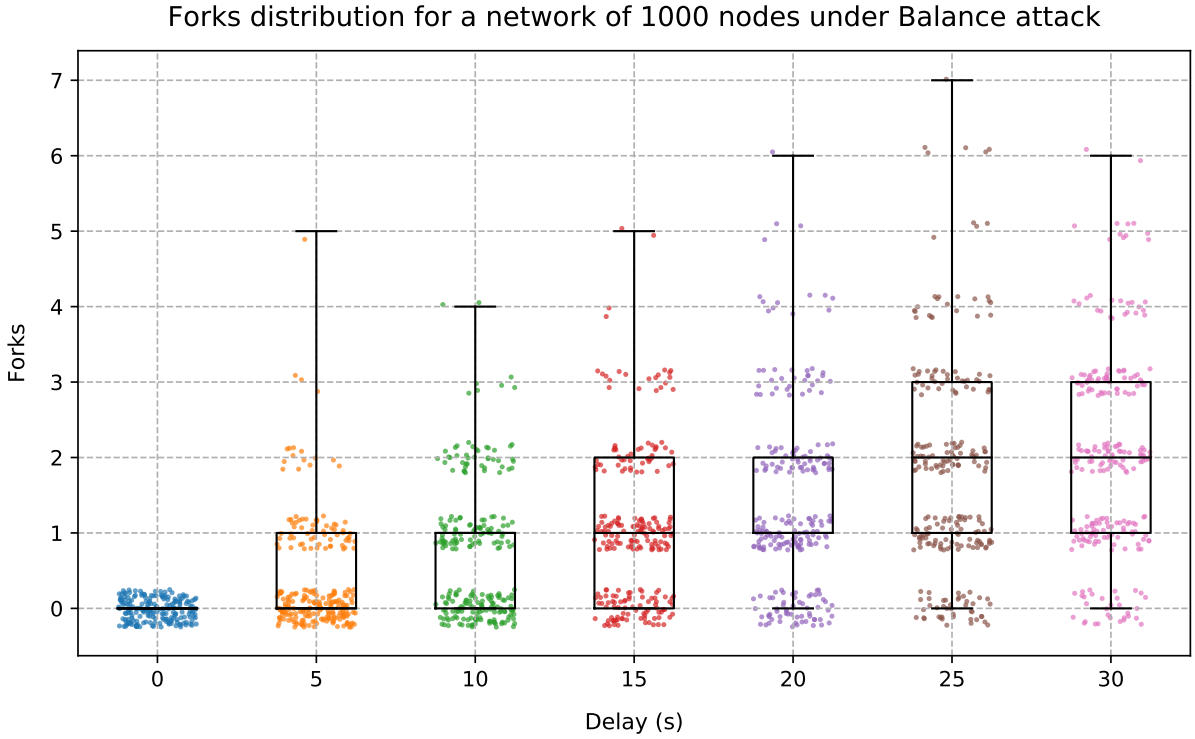


(b) Distribution of forks for a network of 9000 nodes with different delays. With zero delay, blocks are immediately distributed to all nodes and no fork is generated. As for smaller networks, even a small delay of 5 s starts to create some forks, and longer delays produce an higher number of forks.

Figure 6.3: Forks distribution for a network of 9000 nodes with different delays.

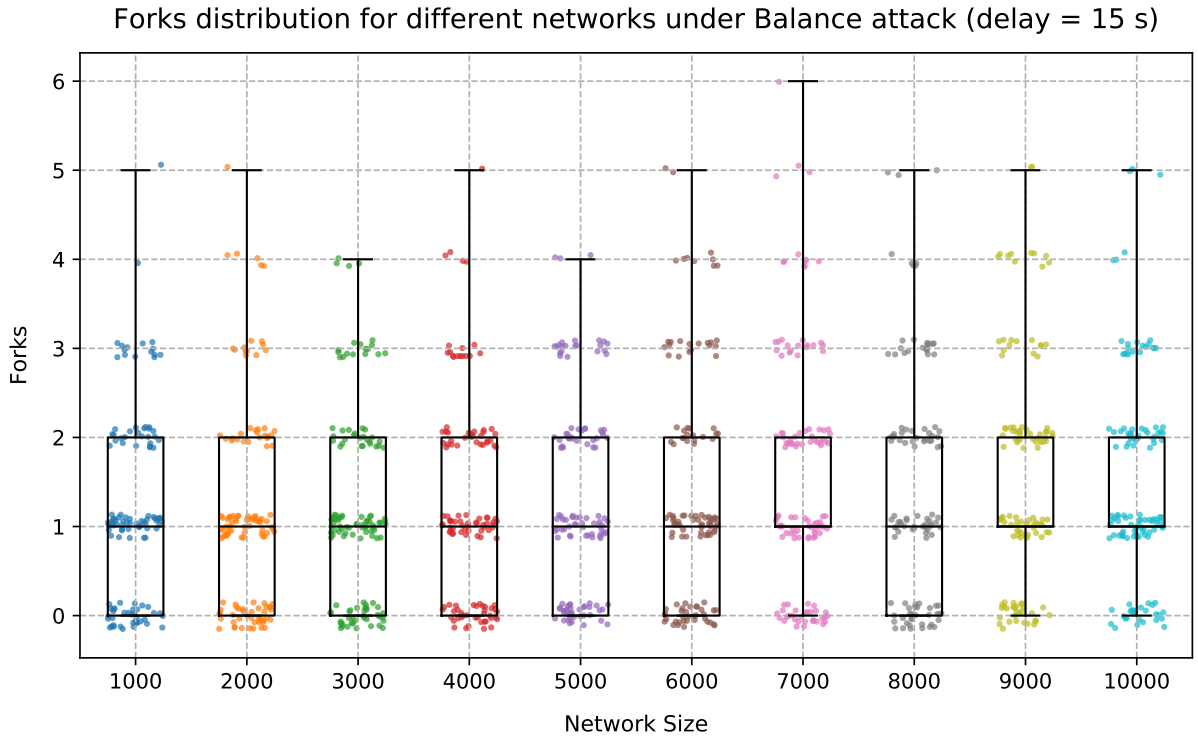


(a) Average forks number for a network of 1000 nodes under Balance attack. Relatively small delays of 10 s on the propagation of `Block` messages between nodes belonging to different partitions cause 0.7 fork on average at the end of the 3 hours simulation. With a larger delay of 30 s, the average number of forks raises to 2.

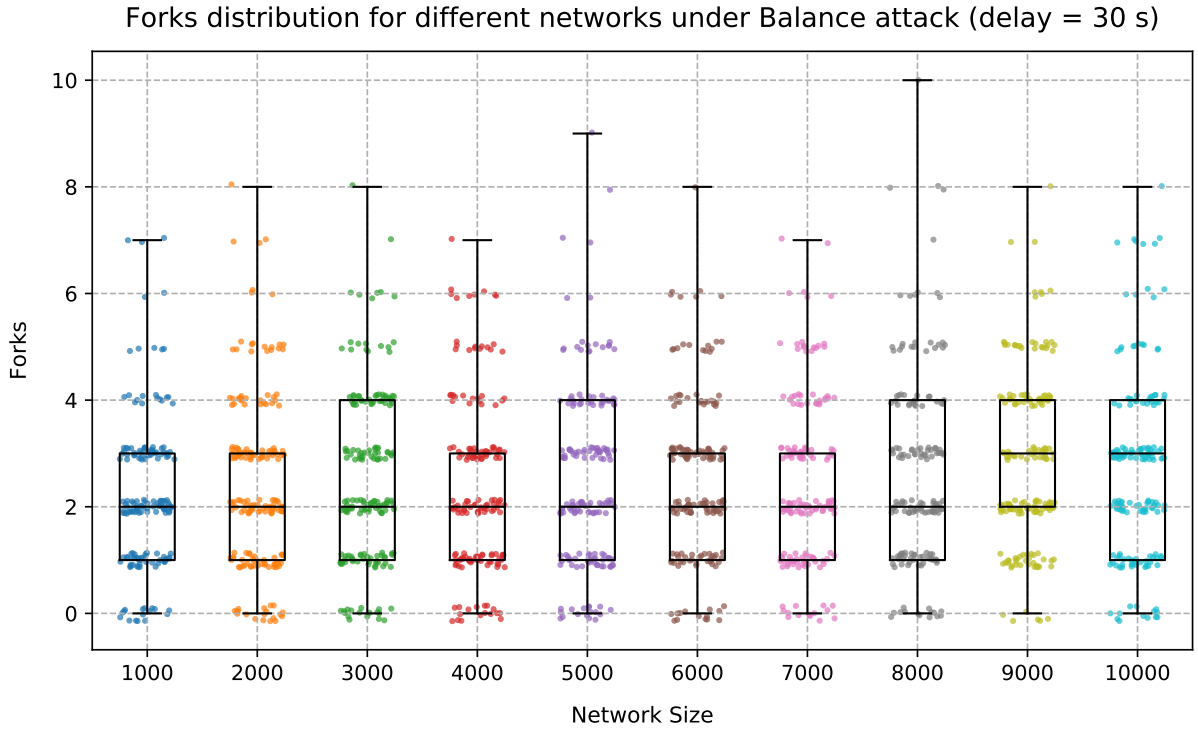


(b) Distribution of forks for a network of 1000 nodes under Balance attack. Many simulations with delay of 10 s generate 1 fork, while some have reached 2 or 3 forks. The distribution of forks shifts towards higher numbers with larger delays. With a delay of 30 s, the network generates up to 4 - 6 forks.

Figure 6.4: Forks distribution for a network of 1000 nodes under Balance attack.

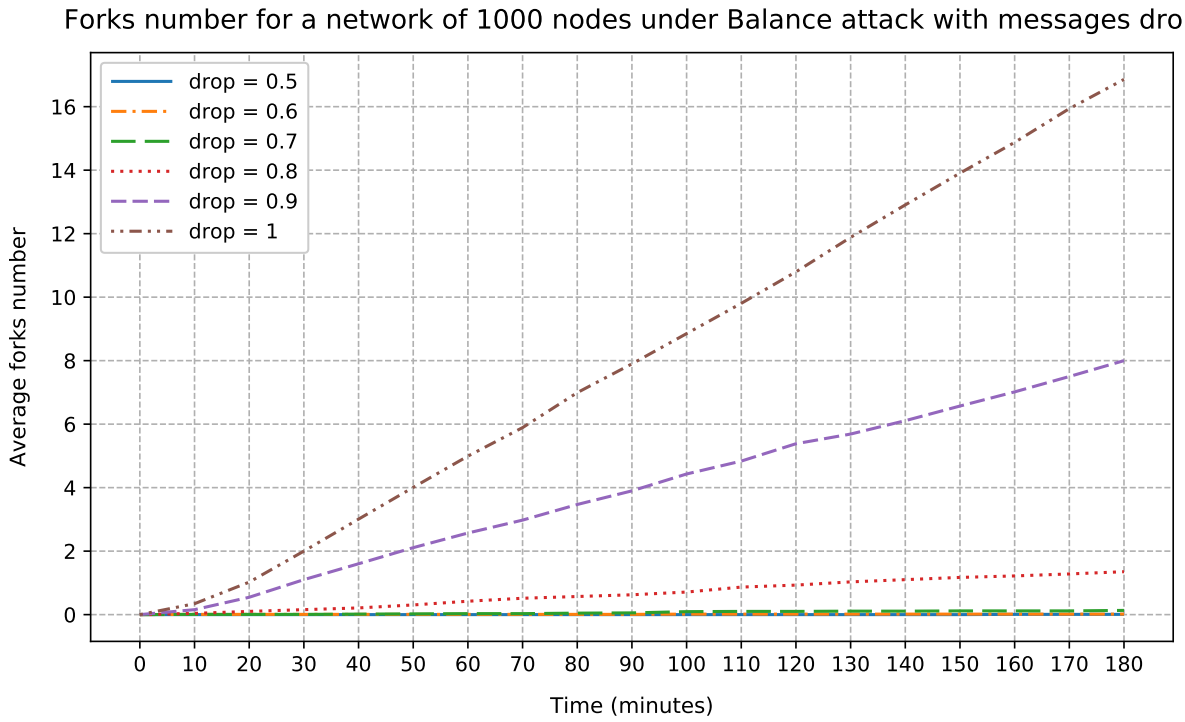


(a) Distribution of forks for different networks under a Balance attack that delays all `Block` messages by 15 s.

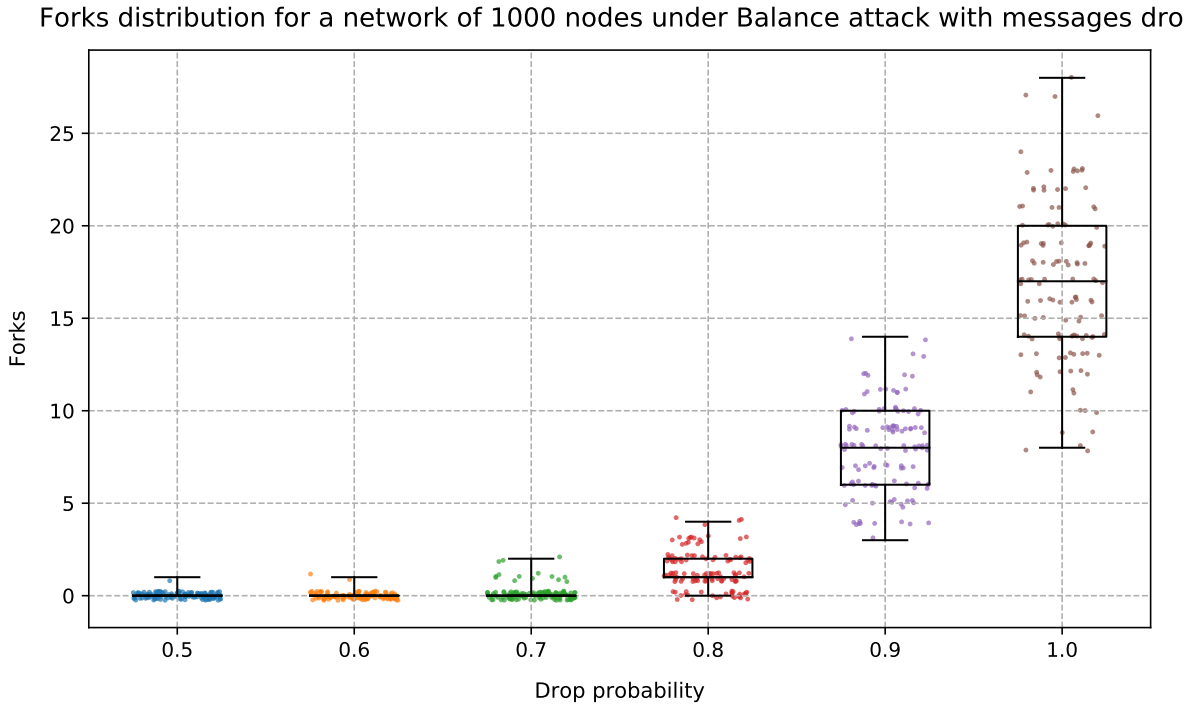


(b) Forks distribution for different networks under a Balance attack that delays all `Block` messages by 30 s.

Figure 6.5: Distribution of forks for different networks under a Balance attack that delays all `Block` messages by 15 or 30 s. The size of the network does not seem to influence the effect of the attack for both tested delays.



(a) Average forks number for a network of 1000 nodes under a Balance attack with message different drops.



(b) Forks distribution for a network of 1000 nodes under a Balance attack with message different drops.

Figure 6.6: Forks distribution for a network of 1000 nodes under a Balance attack with different message drops. The attacks works well only for drop probabilities of 0.8 to 1.

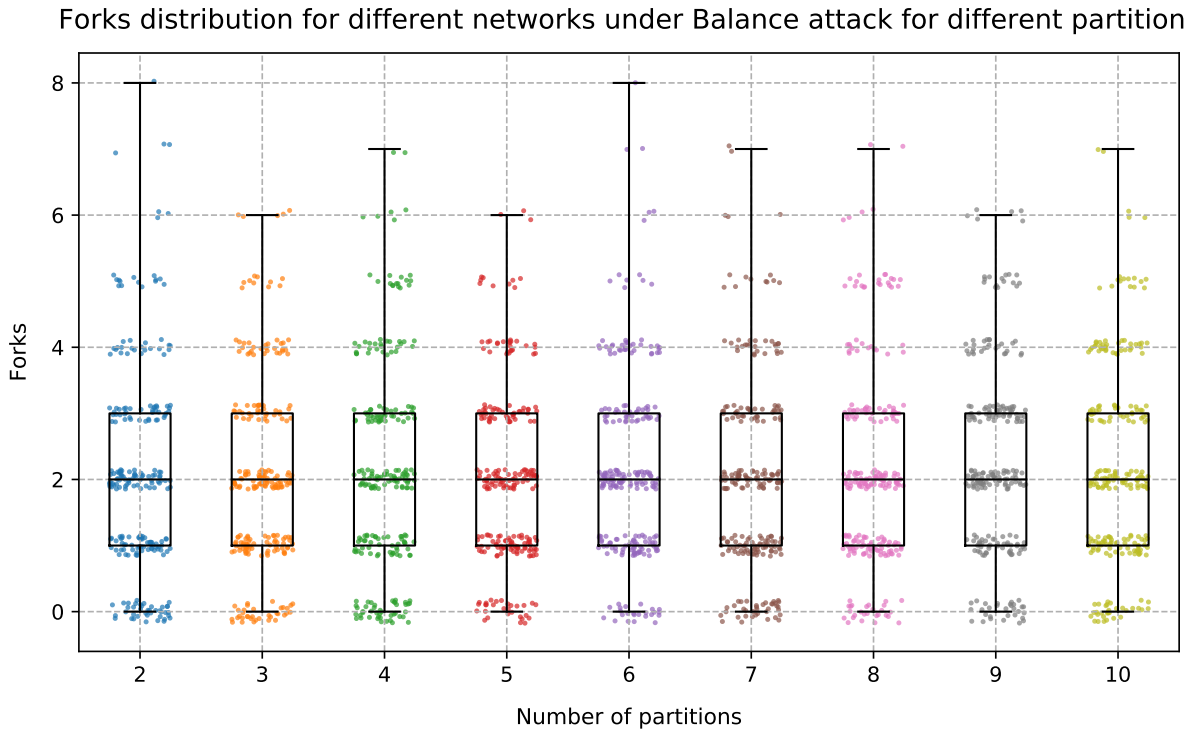


Figure 6.7: Forks distribution for a network of 1000 nodes under Balance attack with different numbers of partitions and `balance_attack_delay` of 30 s. There is not significant difference in the number of forks generated for the different number of partitions.

Partitions

We run some experiments to evaluate if the number of partitions created by the Balance attack influence its effectiveness. Figure 6.7 shows the distribution of forks for a network of 1000 nodes during a Balance attack with `balance_attack_delay` of 30 s with different number of partitions. According to our experiment, there is no significant difference in the effect of the attack for a number of partitions between 2 and 10: the forks are distributed around 6, with most points concentrated around 4 and 8. The results are in line with the previous experiments.

6.4 Evaluation

The Balance attack is very effective in creating forks in the Bitcoin's blockchain. Delaying `Block` messages between different partitions of about the same size is much more effective than introducing random delays in the entire network. Dropping random packets between nodes of different partitions is ineffective, thanks to the gossip protocol implemented by Bitcoin to propagate blocks and transactions: as soon as a single `Block` message manages to reach the other partition, it is immediately broadcasted to the neighbors, which themselves propagate the information to their peers, until all nodes in the network have received the information. The attack seems to have the same effect for a different number of partitions (2 to 10), provided they all have about the same computational power.

Chapter 7

Conclusions

In this work, we implemented an event driven simulator of the low-level Bitcoin protocol using the PeerSim framework. We used the simulator to run Bitcoin networks of different sizes under various network conditions: at rest, with generalized delays and under attack. In particular, we focused on the Balance attack, which consists in partitioning the nodes into 2 groups by delaying communications between peers in different partitions. Finally, we implemented some variants of the original attack and evaluated their performances.

Our experiments show the Bitcoin protocol is robust under normal network conditions and in presence of random message losses, thanks to the gossip-style underlying protocol. However, the network produces a significant number of forks in presence of diffuse delays. The situation is even worse under a Balance attack: delays on blocks broadcast between nodes of different partitions higher the probability of forks in the network.

The variants of the Balance attack we tested do not improve the attack's effect. Random message dropping is totally ineffective, unless the attacker drops at least 80% to 90% of all messages exchanged between nodes in different partitions. If such an attack was feasible, the attacker could simply drop all messages, isolating completely the 2 groups from each other.

Higher number of partitions seems to have the same effect as the base attack: the forks distribution for a number of partitions up to 10 does not differ significantly from the one with just 2 of them. An attacker might consider to create more than 2 groups of nodes if this were easier in a particular context.

Balance attacks are very effective, but difficult to achieve in practice, since an attacker needs to control a significant number of the links that connect Bitcoin nodes over the Internet. Still, similar attacks have been performed in the past against pools of miners: weaknesses in the BGP allowed an attacker to hijack connections between miners and pool coordinators, taking control of the victims' mining power. Since all communications in Bitcoin are not encrypted and use a simple TCP connection, the same idea can be easily used by a powerful attacker to mount a Balance attack.

7.1 Future work

As stated in the original paper [57], all *forkable* blockchains are potentially vulnerable to a Balance attack. Some new cryptocurrencies try to solve most problems of Bitcoin and similar digital currencies by almost completely eliminating the possibility of forks. For example, Algorand [29] defines a new byzantine consensus algorithm based a novel cryptographic technique called Verifiable Random Functions to build a fast protocol that can confirm transactions with a small latency and avoid the generation of forks even under attack. Possible future works include extending our simulator to the Algorand protocol and experimentally verifying if the claimed properties yield in practice.

Bibliography

[1] 51% Attack. URL: https://www.investopedia.com/terms/1/51-attack.asp (visited on 08/05/2018).

[2] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. *2017 IEEE Symposium on Security and Privacy (SP)*:375–392, 2017.

[3] Adam Back. Hashcash - A Denial of Service Counter-Measure. 2002. URL: http://hashcash.org/papers/hashcash.pdf (visited on 07/29/2018).

[4] BGPStream. URL: https://bgpstream.com/ (visited on 08/10/2018).

[5] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pages 15–29.

[6] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pages 15–29.

[7] Alex Biryukov and Ivan Pustogarov. Bitcoin over Tor isn't a good idea. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pages 122–134.

[8] bitcoin. URL: https://bitcoin.org/en/faq (visited on 07/29/2018).

[9] Bitcoin Developer Guide. URL: https://bitcoin.org/en/developer-guide (visited on 07/29/2018).

[10] Bitcoin Developer Reference. URL: https://bitcoin.org/en/developer-reference (visited on 07/29/2018).

[11] Bitcoin dips below $10,000 for first time since December. *BBC*, January 17, 2018. URL: https://www.bbc.com/news/technology-42717639 (visited on 07/29/2018).

[12] Bitcoin Vocabulary. URL: https://bitcoin.org/en/vocabulary (visited on 08/04/2018).

[13] Bitnodes. URL: https://bitnodes.earn.com/nodes/ (visited on 08/18/2018).

[14] CoinCheckup. URL: https://coincheckup.com (visited on 07/29/2018).

[15] CoinMarketCap. URL: https://coinmarketcap.com (visited on 07/29/2018).

[16] CoinRanking. URL: https://coinranking.com (visited on 07/29/2018).

[17] Common Vulnerabilities and Exposures - Bitcoin Wiki. URL: https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures (visited on 08/09/2018).

[18] Confirmation. URL: https://en.bitcoin.it/wiki/Confirmation (visited on 08/05/2018).

[19] Nicolas T. Courtois and Lear Bahack. On Subversive Miner Strategies and Block Withholding Attack in Bitcoin Digital Currency. *CoRR*, abs/1402.1718, 2014.

[20] CryptoCompare. URL: https://www.cryptocompare.com (visited on 07/29/2018).

[21] CVE-2013-5700. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-5700 (visited on 08/09/2018).

[22] Christian Decker and Roger Wattenhofer. Information Propagation in the Bitcoin Network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*. IEEE, 2013, pages 1–10.

[23] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th annual ACM Symposium on Principles of Distributed Computing Systems (PODC'87)*. ACM, 1987, pages 1–12.

[24] Ittay Eyal and Emin Gün Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. *CoRR*, abs/1311.0243, 2013.

[25] Ittay Eyal and Emin Gün Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[26] Cyrus Farivar. Bitcoin pool GHash.io commits to 40% hashrate limit after its 51% breach. 2014. URL: https://arstechnica.com/information-technology/2014/07/bitcoin-pool-ghash-io-commits-to-40-hashrate-limit-after-its-51-breach/ (visited on 08/05/2018).

[27] GHash.IO and double-spending against BetCoin Dice. URL: https://bitcointalk.org/index.php?topic=327767.0 (visited on 08/05/2018).

[28] GHash.IO is open for discussion. URL: https://blog.cex.io/news/ghash-io-51-percent-5180 (visited on 08/05/2018).

[29] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. In SOSP '17. ACM, Shanghai, China, 2017, pages 51–68. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132757. URL: http://doi.acm.org/10.1145/3132747.3132757.

[30] Nermin Hajdarbegovic. Bitcoin Miners Ditch Ghash.io Pool Over Fears of 51% Attack. URL: https://www.coindesk.com/bitcoin-miners-ditch-ghash-io-pool-51-attack/ (visited on 08/05/2018).

[31] Has there ever been a successful double spend attack on the Bitcoin network? If not, is it really necessary to wait for confirmations? URL: https://bitcoin.stackexchange.com/questions/722/has-there-ever-been-a-successful-double-spend-attack-on-the-bitcoin-network-if (visited on 08/05/2018).

[32] Ethan Heilman. How many IP addresses can a DNS query return? URL: http://ethanheilman.tumblr.com/post/110920218915/how-many-ip-addresses-can-a-dns-query-return (visited on 08/19/2018).

[33] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *USENIX Security Symposium*, 2015, pages 129–144.

[34] Stan Higgins. GHash Commits to 40% Hashrate Cap at Bitcoin Mining Summit. 2014. URL: https://www.coindesk.com/ghash-commits-40-hashrate-cap-bitcoin-mining-summit/ (visited on 08/05/2018).

[35] Garrick Hileman and Michel Rauchs. 2017 Global Blockchain Benchmarking Study, 2017.

[36] Joel Hruska. One Bitcoin group now controls 51% of total mining power, threatening entire currency's safety. 2014. URL: https://www.extremetech.com/extreme/184427-one-bitcoin-group-now-controls-51-of-total-mining-power-threatening-entire-currencys-safety (visited on 08/05/2018).

[37] Alnitak (https://stackoverflow.com/users/6782/alnitak). How many A records can fit in a single DNS response? URL: https://stackoverflow.com/a/6860337 (visited on 08/19/2018).

[38] Irreversible Transactions. URL: https://en.bitcoin.it/wiki/Irreversible_Transactions (visited on 08/04/2018).

[39] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pages 906–917.

[40] Ghassan Karame, Elli Androulaki, and Srdjan Capkun. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. *IACR Cryptology ePrint Archive*, 2012(248), 2012.

[41] Joshua A Kroll, Ian C Davey, and Edward W Felten. The economics of Bitcoin mining, or Bitcoin in the presence of adversaries. In *Proceedings of WEIS*. Volume 2013, 2013, page 11.

[42] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[43] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2017.

[44] Pat Litke and Joe Stewart. BPG Hijacking for Crytocurrency Profit. August 2014. URL: `https://www.secureworks.com/research/bgp-hijacking-for-cryptocurrency-profit` (visited on 06/15/2018).

[45] Majority attack. URL: `https://en.bitcoin.it/wiki/Majority_attack` (visited on 08/05/2018).

[46] Ralph C Merkle. Protocols for public key cryptosystems. In *Security and Privacy, 1980 IEEE Symposium on*. IEEE, 1980, pages 122–122.

[47] Andrew Miller, James Litton, Andrew Pachulski, Neal Gupta, Dave Levin, Neil Spring, and Bobby Bhattacharjee. Discovering Bitcoin's Public Topology and Influential Nodes, 2015.

[48] Alberto Montresor and Márk Jelasity. PeerSim: a scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*. Seattle, WA, September 2009, pages 99–100.

[49] Moonstats. URL: `https://www.moonstats.com/` (visited on 07/29/2018).

[50] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2009. URL: `http://bitcoin.org/bitcoin.pdf` (visited on 07/29/2018).

[51] Satoshi Nakamoto and "contributors". Bitcoin. https://github.com/bitcoin/bitcoin. 2013.

[52] Satoshi Nakamoto and "contributors". chainparams.cpp. 2013. URL: `https://github.com/bitcoin/bitcoin/blob/1f470a8916aabbb7f03afda26d41ebfd8d1a2263/src/chainparams.cpp#L132-L138` (visited on 08/19/2018).

[53] Satoshi Nakamoto and "contributors". chainparamsseeds.h. 2013. URL: `https://github.com/bitcoin/bitcoin/blob/1f470a8916aabbb7f03afda26d41ebfd8d1a2263/src/chainparamsseeds.h#L10` (visited on 08/19/2018).

[54] Satoshi Nakamoto and "contributors". net.h. 2013. URL: `https://github.com/bitcoin/bitcoin/blob/1f470a8916aabbb7f03afda26d41ebfd8d1a2263/src/net.h#L40-L43` (visited on 08/19/2018).

[55] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[56] Christopher Natoli and Vincent Gramoli. The Balance Attack Against Proof-Of-Work Blockchains: The R3 Testbed as an Example. *CoRR*, abs/1612.09426, 2016.

[57] Christopher Natoli and Vincent Gramoli. The Balance Attack or Why Forkable Blockchains Are Ill-Suited for Consortium. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 2017, pages 579–590.

[58] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 2016, pages 305–320.

[59] Alessio Netti. An introduction to PeerSim. April 2018. URL: `http://www.cs.unibo.it/babaoglu/courses/csns/slides/peerSim.pdf`.

[60] Orphaned Blocks. URL: `https://www.blockchain.com/btc/orphaned-blocks` (visited on 08/05/2018).

[61] PeerSim: A Peer-to-Peer Simulator. URL: `http://peersim.sourceforge.net/`.

[62] Pool Stats - BTC.com. URL: https://btc.com/stats/pool?pool_mode=month3 (visited on 08/18/2018).

[63] Pete Rizzo. Ghash.io: We Will Never Launch a 51% Attack Against Bitcoin. URL: https://www.coindesk.com/ghash-io-never-launch-51-attack/ (visited on 08/05/2018).

[64] Meni Rosenfeld. Analysis of hashrate-based double spending. *arXiv preprint arXiv:1402.2009*, 2014.

[65] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pages 515–532.

[66] Satoshi Client Node Discovery. URL: https://en.bitcoin.it/wiki/Satoshi_Client_Node_Discovery (visited on 08/19/2018).

[67] Robert E Shannon. Introduction to the art and science of simulation. In *Proceedings of the 30th conference on Winter simulation*. IEEE Computer Society Press, 1998, pages 7–14.

[68] Assaf Shomer. On the Phase Space of Block-Hiding Strategies. *IACR Cryptology ePrint Archive*, 2014:139, 2014.

[69] Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

[70] Pierangelo Soldavini. Bitcoin sotto 10mila dollari: -50% in un mese. Chiude una piattaforma. *Il Sole 24 ORE*, January 17, 2018. URL: http://www.ilsole24ore.com/art/finanza-e-mercati/2018-01-17/bitcoin-quota-10000-dollari-50percento-un-mese-chiude-piattaforma-111939.shtml (visited on 07/29/2018).

[71] Stratum Mining Protocol. URL: https://slushpool.com/help/manual/stratum-protocol (visited on 08/13/2018).

[72] Stratum mining protocol. URL: https://en.bitcoin.it/wiki/Stratum_mining_protocol (visited on 08/10/2018).

[73] Ole Tange. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1):42–47, February 2011. DOI: http://dx.doi.org/10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[74] Target. URL: https://en.bitcoin.it/wiki/Target (visited on 08/20/2018).

[75] James Titcomb. Bitcoin price falls below $10,000 as dramatic sell-off enters second day. *The Telegraph*, January 17, 2018. URL: https://www.telegraph.co.uk/technology/2018/01/17/bitcoin-price-falls-10000-dramatic-sell-off-enters-second-day/ (visited on 07/29/2018).

[76] Weaknesses. URL: https://en.bitcoin.it/wiki/Weaknesses (visited on 08/05/2018).

[77] Wikipedia contributors. Autonomous system (Internet) — Wikipedia, The Free Encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Autonomous_system_(Internet)&oldid=850854570 (visited on 08/10/2018).

[78] Wikipedia contributors. Border Gateway Protocol — Wikipedia, The Free Encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Border_Gateway_Protocol&oldid=854290407 (visited on 08/10/2018).

[79] Wikipedia contributors. Cryptographic hash function — Wikipedia, the free encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function&oldid=850041198 (visited on 07/29/2018).

[80] Wikipedia contributors. Discrete event simulation — Wikipedia, the free encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Discrete_event_simulation&oldid=851197489%22 (visited on 08/26/2018).

[81] Wikipedia contributors. Ghash.io — Wikipedia, the free encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Ghash.io&oldid=852351321 (visited on 08/05/2018).

[82] Wikipedia contributors. Osi model — Wikipedia, the free encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=OSI_model&oldid=856294265 (visited on 08/26/2018).

[83] Wikipedia contributors. Poisson point process — Wikipedia, the free encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Poisson_point_process&oldid=855129336 (visited on 08/30/2018).

[84] Wikipedia contributors. Simulation — Wikipedia, the free encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Simulation&oldid=854900884 (visited on 08/26/2018).

[85] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[86] Ed. Y. Rekhter, Ed. T. Li, and Ed. S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC (4271). RFC Editor, January 2006, pages 1–104. URL: http://www.rfc-editor.org/rfc/rfc1654.txt.