Summary

This report is a continued work of a paper that investigates the applicability of exploiting the Bitcoin network's address propagation and churn in order to gain a topology favorable to an adversary. We further investigate how the Bitcoin network's address propagation and churn can be used to increase the number of inbound connections to stable adversary nodes and how damaging this potentially can be to the Bitcoin network.

We propose The ChurnAddr strategy, which dictates the adversary to setup and maintain a certain amount of highly stable nodes, which must offer a complete set of features as found in the Bitcoin Core application. Moreover, part of the strategy imposes that the adversaries can exploit the Bitcoin Core protocol implementation for propagating addresses in order to increase the amount of inboudn connections. This is done by frequently broadcasting an ADDR message containing the IP addresses of the adversary nodes, where the ADDR message is constrained to maximum contain 10 IP addresses. We hypothesize that performing this strategy in a long-term period of more than 3000 hours will let the adversary obtain a favorable topology of the network. During this favorable topology, we suggest attacks that should become applicable for the adversary, and we discuss how the general health of the Bitcoin network might decrease.

We analyze and describe the data necessary to prove this hypothesis, and we carefully delimit the problem domain to a scope, which is both appropriate for proving the hypothesis and delimiting a substantial part of the Bitcoin Core application.

We propose various methodologies to obtain an insight in the applicability and effectiveness of the ChurnAddr strategy, in which we examine ethical and resource-related issues for testing the hypothesis on the real network. We find that real network testing is impractical, hence, we further examine simulation and modelling methodologies to obtain the desired insight. We find that given the project's context and limitations, a carefully constructed simulator is the best suited choice for this task. In addition, we find that the simulator in previous paper has fundamental issues in its perception of time, which makes its result less trustworthy and makes reuse a non-trivial task. Hence, we construct an improved simulator, which is data-driven in accepting behavioral data from the real Bitcoin network to increase accuracy, it has an accurate and finely granular perception of time, and it achieves an impressive performance from being event-based and thoroughly optimized.

As we choose to construct a simulator that should be both be accurate and perform a long-term strategy, a significant part of this study's work is put into optimizing the simulator and its Bitcoin Core implementation's code without losing critical accuracy. We achieve the performance optimization with minimal accuracy loss from attentively reviewing and reverse engineering the Bitcoin Core source code Using pseudocode as an intermediate translation from the source code to the simulator implementation, we obtain a conceptual model that can be used as a verification and assessment medium of the simulator's accuracy.

We assess the possibilities of gathering existing topology measurements of the Bitcoin network and applying these to the simulator. Although topology measurements do exist, we find that they are missing critical details required by the ChurnAddr strategy. To obtain a plausible and trustworthy reconstruction of the Bitcoin network's topology in the simulator, we collect and use churn and latency data from the real Bitcoin network, hence, the simulator is data-driven and features dynamic topology generation as well. In addition, the simulator supports deterministic randomness shared in all its internal random procedures meaning that it features experiment replication and performing tests with varying randomness seeds.

While collecting the simulator's needed latency data, we investigate multiple approaches for implementing an accurate node-to-node latency. In summary of the methodologies used for discovering the network's latency, we investigate ICMP Echo Request, Bitcoin Ping, TTL, RTT, and geographical distance with the purpose of finding a useful correlation that can be used for deducing a node-to-

node latency in the simulator's network. Although we do not find any strong evidence for the needed correlation, we do come up with a distribution of operating systems used in the Bitcoin network, which we to the best of our knowledge is the first public insight of this.

We assess parts of the ChurnAddr strategy from running multiple scenarios and configurations of the simulator. The results includes how churn alone is a factor in increasing an adversary's inbound connections, which is shown for scenarios with a varying amount of adversary nodes. Furthermore, we present results showing how block propagation is influenced in cases where the adversaries either participates in the propagation or attempts to hinder the propagation. Although we may propose multiple use cases for this adversary behavior, we simply follow a use case of a miner that has recently mined a block, for which the adversary has the intention of slowing down the block propagation in order to gain an advantage in the block race.

In conclusion, we make several contributions on the fundamentals required for constructing a Bitcoin network simulator featuring a dynamic topology, data-driven capabilities, allows long-term experiments, and featuring automatic result generation. We assess the applicability of performing the ChurnAddr strategy and how damaging its influence is on the network. We disprove the applicability of using the ChurnAddr strategy as part of an eclipse attack, which the previous work suggested might be possible. Instead, we find that the strategy can be used to disrupt block propagation from select network participants.

ChurnAddr Strategy in the Bitcoin Network

Assessing the Applicability of using Churn and Address Propagation Exploits in the Bitcoin Network to Perform Adversary Actions

Alex Grøndahl Frie, Rune Willum Larsen

June 2018

Contents

1 Introduction

The Bitcoin network has been in deployment and developed since 2008 and while still being in beta with the newest version being 0.16.0 it is still growing in size. Being a pioneering application for the relatively new data structure, blockchains, and still holding strong as an accepted currency [1], there seems to be no indication that it will become unimportant in the near future.

Since the Bitcoin network is fully open for public participation as a peer-to-peer (P2P) network, several precautions have been taken to ensure that people cannot cheat or obtain an unfair advantage. Indeed, Bitcoin is well-equipped to handle these issues through its use of blockchain, cryptography, and computational proofs. However, we propose that the Bitcoin network protocol has potential weaknesses that combined with a knowledge of simple network behaviors can be exploited.

This report is a continuation of a previous project, which investigates the applicability of exploiting the Bitcoin network's address propagation enabled by the network's protocol through churn. The challenge of demonstrating this claim, is that it requires long-term measurements of the Bitcoin network, which was obtained using a tick-based simulator in previous study [2]. Although this simulator was carefully constructed to replicate selected parts of the Bitcoin Core application's behavior, we question its trustworthiness due to its rough implementation of handling simulation time.

Our motivation originates from the previous study [2], which revealed how churn can drastically influence the topology of the Bitcoin network. While this previous study focused at achieving a connection monopolization of a victim to perform an eclipse attack [3], we noticed a significant increase of an adversary's inbound connections from other nodes in the simulated Bitcoin network, which ultimately created a curious topology. As a result, we find it relevant to investigate whether this topology has a consequence on the Bitcoin network, and if it can be exploited by an adversary to gain an advantage in the network.

1.1 Problem Analysis

In this section, we will establish the founding problem behind this study. We will shortly present the implications of Eclipse attacks. This attack is a shared problem between this study and a previous study [2], which we will henceforth refer to as "the BitcoinChurn project". We will critically approach the BitcoinChurn project and reflect on its findings to unveil a new potential threat to the Bitcoin network. Furthermore, we will establish the potential pitfalls of the BitcoinChurn project, and see if the result could be achieved in a more precise way.

1.1.1 Eclipse Attack and its Impact

The Eclipse attack proposed by Heilman et al. [3] bears a serious threat to the Bitcoin network if successfully executed. In a nutshell, the attack can be used to partition victim nodes out of the network in such a way that the adversary can completely monitor and control the information to and from the victim. The attack exploits the Bitcoin network's behavior as defined by the Bitcoin Core application, which is also by far the most dominant Bitcoin application on the Bitcoin network making up for almost 95 % of the visible nodes at the time of writing [4].

This can be a problem for different stakeholders in the Bitcoin network. Firstly, Bitcoin traders can have an interest in broadcasting transactions quickly in the network, as the Bitcoin value during a transaction's propagation can otherwise decrease or increase significantly changing the profit or loss gained from a transaction [5, 6]. Secondly, Bitcoin miners are dependent on broadcasting mined blocks in order to retrieve the mined block's reward. Furthermore, Bitcoin miners are dependent on receiving information about newly mined blocks, as the time spent on mining atop of an old block is not profitable. If we consider a scenario in which one of the mentioned stakeholders are eclipsed, it becomes clear that an eclipse attack can cause disruption and financial damage to its target, as the attacker can control the transactions and blocks to and from the stakeholder.

1.1.2 The BitcoinChurn Project and Outbound Monopolization

The BitcoinChurn project [2] proposes an alternative adversary strategy to partly obtain the state in which an Eclipse attack becomes feasible. Specifically, it focuses on monopolizing the victim's outbound connections in such a way, that the victim is unaware of being a potential target of an Eclipse attack. During the BitcoinChurn project's measurements, it was discovered that using the aforementioned strategy had side effects on the whole network topology, as other nodes in the network had a tendency of connecting to the adversary nodes as seen in Figure 1.1.

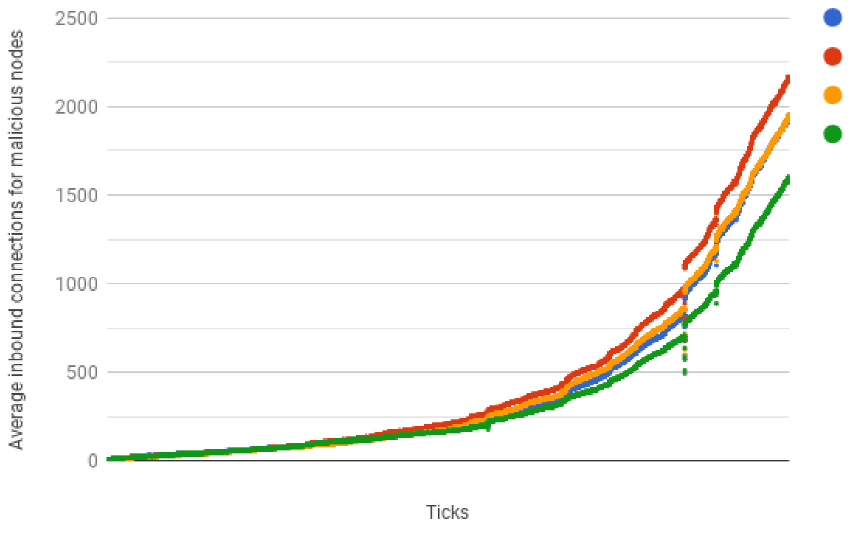


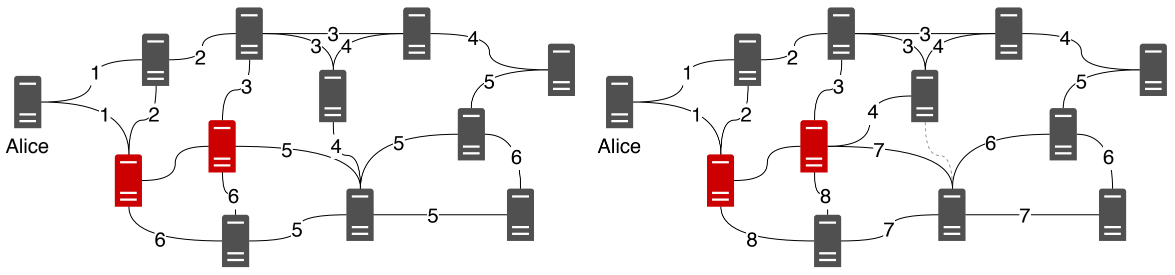
Figure 1.1: Average amount of inbound connections to adversary nodes of four simulations with different seeds from BitcoinChurn [2]

We reflect on the aforementioned results and raise the question if a general increase of adversary inbound connections is a potential threat to the Bitcoin network. We propose an extreme scenario, where the adversaries have gained enough influence to monopolize all nodes' outbound connections. We know that a Bitcoin Core node has an outbound connection limit by default, which means that in the proposed scenario, we would in addition to the full network outbound monopolization have a situation where only the adversaries have inbound connections as well. What the scenario provides is an extreme case example and an initial intuition of what the impact is of having a high amount of inbound connections. In relation to an Eclipse attack's requirements, the adversary is no longer required to monopolize the inbound connections to a victim in order to carry out an Eclipse attack.

1.1.3 Limiting Network Outbound Capacity

We recognize that fully monopolizing the full Bitcoin network is indeed unrealistic, however, approaching the scenario while not constraining it to just enable an Eclipse attack opens up a new possibility. We rethink the scenario with a nuance, where only a subset of the network's 'pool' of total outbound connections have been monopolized, and we propose that this still carries a potential threat to the participating stakeholders yet is more practical for an adversary to accomplish.

Let us think of the Bitcoin network as having a limited capacity of outbound connections and engage ourselves in establishing a healthy network, meaning that the stakeholders are protected, and the features of the Bitcoin network are intact and supported by the underlying network topology. Firstly, we realize that in order to achieve a healthy network topology, we have to spend an amount of our outbound connections–after all, a network without connections is essentially useless and cannot propagate information. Secondly, we must attempt to randomize and promote a decentralized network, as this provides efficient information propagation and equal opportunity to the network participants [7, p. 3]. Lastly, to provide efficient propagation of transactions and blocks, we have to consider the geodesic distance between nodes, as this has a significant influence on how fast information is propagated [8, p. 9]. An example of such a network can be seen in Figure 1.2a.



(a) Adversaries with poor influence

(b) Adversaries have gained an extra connection

Figure 1.2: Graphs showing hop distance for block and transaction propagation from Alice, where red nodes represent adversaries controlled by Chuck

We introduce a distributed adversary to our thought experiment and let Alice be a Bitcoin participant and stakeholder, who has just mined a new block and wants to broadcast this block in order to gain her reward. As seen in Figure 1.2a, Alice can broadcast her block to the whole network within 5 hops, even though if the placed adversaries (red), refuse to propagate Alice's block. We propose that if the adversaries somehow managed to attract more connections, hence reducing the total amount of connections available to construct a healthy network, then the adversaries could be more effective. We illustrate such a scenario in Figure 1.2b, where one adversary has attracted an additional connection from a nearby node. In this case, we realize that if the adversaries prior to this topology configuration had a way of attracting the extra connection, they could effectively degrade Alice's block propagation by increasing the hop distance to 7.

1.1.4 The BitcoinChurn Project's Pitfalls

The BitcoinChurn project, which this study continues, proposed exploiting mechanisms of the Bitcoin Core application's address propagation along with network churn, where churn expresses the join-leave cycles of nodes in the network. This exploit highlights a strategy, which adversaries can use to gain the aforementioned favorable topology, where the applicability of an Eclipse attack gradually becomes more practical with time. What we add is that although an Eclipse attack may not be immediately possible, the adversaries' increased influence can potentially damage the information propagation in the network.

The simulator used during the BitcoinChurn project has potential pitfalls, which begs the question: Are the results trustworthy? The main problem with this simulator is that it uses a very simplistic simulation engine. The simulator is tick-based and configured to interpret a tick as a 5 to 6 minute period, which makes the concept of time inaccurate and might be a factor that neutralizes the conclusion gained from the BitcoinChurn project. The reason for this is that it can be safely assumed that the address propagation active in the real network and the real network's latency operate on a time span granularity that is finer than 5 minutes.

1.2 Problem Statement

In summary, we establish that the Bitcoin Core application's protocol for address propagation along with churn is a potential threat to the Bitcoin network. We propose that a distributed adversary can potentially exploit this to disrupt the network in terms of transaction and block propagation, and in worst case enable or partly enable the applicability of an Eclipse attack. Although some of these findings have been presented as plausible [2] using a tick-based simulator, we critically question this approach and we raise the question, of whether it is possible to show the same results using a more accurate approach.

We stress that it is favorable to enable gathering these results within a short period of time, as the exploit requires a long-term active participation in the Bitcoin network. Finally, we settle on the problem statement:

> "How can the Bitcoin network's address propagation and churn be exploited to increase the amount of inbound connections to an adversary over time, and can this exploit ultimately enable adversary actions?"

1.2.1 Contributions

This work comprises multiple contributions, which we list:

- We propose and define a novel strategy, the ChurnAddr strategy, which utilizes simple Bitcoin protocol and network mechanics with the purpose of obtaining a topology that is favorable for adversary actions
- We analyze, delimit, and present the problem domain required for understanding, assessing, and executing the ChurnAddr strategy
- We propose and compare various evaluation methodologies and existing simulators for assessing our proposed claim
- We perform and analyze latency measurements relevant to the Bitcoin network and our simulator, in which we curiously discover new insight on the operating systems participating in the Bitcoin network
- We attentively review and reverse engineer the Bitcoin Core source code and denote what optimization issues impact long-term experimentations in a Bitcoin network simulator
- Using our simulator, we disprove the applicability of using the ChurnAddr strategy to enable Eclipse attacks
- Using our simulator, we present results on how the amount of inbound connections increases over time using parts of the ChurnAddr strategy, and we present results on how the topology obtain by this strategy can influence block propagation.

2 Strategy and Requirements

In this chapter, we will present the ChurnAddr strategy in further details, which in the remainder of this chapter will be used to reason about what results are appropriate and necessary to conclude on the problem statement. Moreover, we will discuss what information about the problem domain is necessary to understand, and we will broadly elaborate on what information has been delimited for the reason of achieving a well-defined project scope.

2.1 ChurnAddr Strategy Description

We describe the underlying reasoning of why and how the ChurnAddr strategy works based on a simplified mental comprehension of how the Bitcoin Core application works. Details of the Bitcoin Core application are presented in Section 4.3.

The purpose of a ChurnAddr strategy is to occupy enough of the outbound connections available in the Bitcoin network to enable attacks like eclipsing or disrupting information propagation. By 'outbound connections available', we refer to the total amount of outbound connections available summed up across all nodes in the network. For instance, if we presume that all nodes are running the Bitcoin Core application with the default outbound connection limit of 8, the total amount of outbound connections available will simply be evaluated as $\mathsf{NumberOfNodes} \cdot 8$.

As its name suggests, the strategy fulfills this purpose by the means of exploiting network churn and address propagation. These exploits have the objective of popularizing the adversary nodes, meaning the adversary nodes are known by a comprehensive part of the network and that they are likely to be chosen as an outbound connection candidate by the individual nodes.

Exploiting churn simply means that the adversary waits for nodes to leave the network. Let Alice be a node that has a healthy connection to another node, Bob, making this connection an outbound connection from the perspective of Alice. Given that our adversary nodes controlled by Chuck are intended to overtake Alice's outbound connection to Bob, Chuck is first and foremost dependent on the healthy connection to be dropped. Chuck simply relies on the fact that Bob will at some point of time disconnect, which likelihood in relation to Bob's online time is expressed by Bob's specific churn rate. We will present general churn data of the Bitcoin network in Section 4.2.

Given that Bob disconnects, Alice will try to establish a new outbound connection, for which she evaluates her list of *known* nodes to find an outbound candidate. To find the suitable candidate, Alice randomly chooses a node from her knowns list, which likelihood of being chosen are weighted based on the following parameters (elaborated in Section 4.3):

1. Nodes found in local area network are completely omitted
2. Nodes not using the default port 8332 are down-prioritized
3. Nodes not providing Alice's desired *services* are down-prioritized, e.g. nodes that do not provide Bloom filtering
4. Nodes that have appeared more frequently in ADDR messages are more likely to be chosen

Parameters 1 and 2 can trivially be avoided by Chuck while parameter 3 can be avoided by simply using the latest Bitcoin Core version. Parameter 4 is where the address propagation exploit comes in.

An ADDR message is used to propagate information about what nodes are potentially online and the message can contain multiple IP addresses. The general idea is to exploit that there is no penalty for spamming nodes with ADDR messages and that ADDR messages containing less than 10 IP addresses will continuously be flooded in the Bitcoin network until it reaches a 10 minutes time limit. Each time a node receives an ADDR message, the enlisted IP addresses have a certain chance of increasing its weight in the receiver's list of known addresses, which in turn also increases the chance of that IP address getting picked as a candidate for a suitable outbound connection. Hence, Chuck can use this exploit to increase the chance of his distributed nodes getting picked as the next outbound connection for Alice. The ADDR message and its mechanics will be elaborated in Section 4.3.

2.2 Required Project Results

As previously stated, the goal is to establish a strategy to "increase the number of inbound connections", which we do by exploiting the Bitcoin Core application's protocol for address propagation and the network's churn influence. From this context, we will determine what results are required to support the claim of this strategy.

Let us recall Chuck, who wants to increase his amount of inbound connections as the means to achieve a weakened topology of the Bitcoin network. Our first required result is trivial. We must show Chuck's amount of inbound connections over time, as this is the result that concludes whether the ChurnAddr strategy has an effect on increasing the amount of inbound connections.

Let Alice be a Bitcoin network participant, who has recently mined a block or has an urgent transaction, for which in both scenarios Alice has the intention of broadcasting these messages efficiently in the network. Hence, we must show results of how efficient the messages are propagated, where we define "hop efficiency" as how many hops it takes to propagate a message, and we define "time efficiency" as the time it takes to propagate a message to the rest of the network.

2.3 Problem Domain Characteristics

We establish and elaborate the information needed to generate the required results. This section will present key characteristics of the Bitcoin network and the Bitcoin Core application, which are relevant to this study. We also stress, that during this section a notion of a 'simulator' will appear. The reader should know that we will later determine to use a simulator to generate the needed results, and that this influence what part of the Bitcoin network we include in our problem domain.

2.3.1 Bitcoin Network Characteristics

The Bitcoin network remarks itself as a heterogeneous P2P network comprised of thousands of participating nodes [4]. We note its heterogeneity by stressing that the participants use various clients and client version, and that a participant can freely choose to offer select features like Bloom filters, wallets, full blockchain history, mining, et cetera. This study is focused on the Bitcoin's networking features, which dictate the network's topology and propagation of transactions and blocks. Hence, we delimit the project scope to only include features that involve these. Moreover, as 95 % of the Bitcoin network are using the Bitcoin Core application [4], we will only consider this application.

Network Participants: Nodes and Clients

We define two categories of Bitcoin network participants: Clients and nodes. Clients are participants, which for some reason have a restricted level of network interaction capability. For instance, a participant which is located behind a NAT is considered a client, as this makes the participant incapable of accepting inbound connections. Nodes on the other hands have a full level of interaction and have a public IP available making inbound connections possible. We also describe nodes as being 'visible' or refer to them as visible nodes, while a client can be said to be a non-visible node.

To reduce the problem's complexity and delimit the project scope, we only include nodes. As clients cannot accept inbound connections, they naturally cannot be eclipsed. Moreover, we find no techniques for discovering clients in the network, hence we attribute them as being 'non-visible'. Should we consider if a ChurnAddr attack can ultimately influence the information propagation efficiency for clients, we would of course have to include clients in the study.

External Factors: Churn and Latency

As with other networks, the Bitcoin network has certain *external* characteristics that influence the time for propagating information and its topology, where a characteristic is external if it originates from something that is not defined by the Bitcoin protocol.

Firstly, we must include the Bitcoin network's churn. The Bitcoin network is comprised of thousands of participating nodes, in which hundreds of nodes joins or leaves the network during a single day [4]. The join-leave cycles of nodes over time is called *churn* and is a phenomena that is essential to a ChurnAddr attack, as we predict a high churn rate, or in other words a more frequent join-leave cycle, will increase the attack's effectiveness and vice versa. When an arbitrary node initiates an outbound connection, it will keep this connection until the connected node leaves the network. In another perspective, if the Bitcoin network participants never disconnects, then the ChurnAddr attack will not have any effect.

Secondly, we must also consider including latency in the measurements. Trivially, latency have an impact on the time for fully propagating a message to the full network. Moreover, the nodes' handling of address messages (ADDR messages) is dependent on the age of the message (10 minutes since first seen). Specifically, after a given message age the nodes will stop propagating the address message, which in relation to latency will happen faster if the latency is high. We also note that latency was omitted in the BitcoinChurn project, hence we do not know how this can influence the networks topology or information propagation.

2.3.2 Bitcoin Core Application Characteristics

We have to carefully select the Bitcoin messages relevant to this study. The latest version (0.16.0) of the Bitcoin Core application offers 26 different network messages each with their unique purpose [9]. This is significantly more messages compared to other P2P networks like BitTorrent and Gnutella, which respectively support 13 and 6 messages according to their latest draft specifications [10, 11]. Furthermore, unlike Gnutella and BitTorrent, there exist no formal specification of a Bitcoin protocol, which instead depends on consensus (node majority decides protocol) [12] making the process of selecting the relevant messages to focus on more delicate.

In Section 2.1, we present parameters influencing the Bitcoin Core application's choice of a new outbound connection, in which the ADDR message is an essential component. Also related to address propagation, we have to investigate the GETADDR message, which in broad terms is used for querying a node in the network to send an ADDR message back. These messages summarizes the address propagation features of the Bitcoin Core application.

We briefly consider the inclusion of block messages to our study. Whenever a given Bitcoin Core application node receives a connection request (an inbound connection) and it already has reached the limit of maximum inbound connections, the node will try to *evict* (disconnect) an existing connection, where it will partly prioritize keeping nodes that provide new blocks the fastest. We only deem this feature slightly relevant to our study, as it is not critical for adversaries to have specific outbound connections in the ChurnAddr strategy. Indeed, this feature might have an influence on how the network's topology evolves over time, but we find in our simulator that evictions are uncommon, and we simply assume that all nodes provides new block equally fast. Additionally, disregarding block propagation during the simulator's topology generation avoids severe complexity issues. We elaborate further on the eviction feature in Section 4.3.3.

Nevertheless, as the ChurnAddr strategy targets block and transaction propagation, we have to investigate the message for these as well. We settle for only applying these messages in-between stages of topology generation.

Lastly, we must investigate how Bitcoin Core handles and prioritizes the choice of connections, as this is an essential part of the ChurnAddr strategy, which should support that mass propagating addresses increases the chances for the adversaries to be picked.

3 Comparing Evaluation Strategies

In this chapter, we will propose and investigate different methodologies, which can be used for assessing the ChurnAddr strategy's claim.

3.1 Experiments on the Real Bitcoin Network

Before we decide to use a model of the Bitcoin network to assess the ChurnAddr strategy, we consider using the real Bitcoin network as well as its test network. In this section, we will present the issues and risks associated with performing the ChurnAddr strategy on these platforms.

3.1.1 Ethical Issues when Interacting with the Network

Indeed, there exist plenty of work that do perform topology measurements directly on the Bitcoin network, and we cannot reject that this is still possible. However, it is evident that these measurements are bound to get more difficult to perform with time as the Bitcoin Core development team continuously releases updates that protects the network against certain data collection activities to protect the users' privacy. For instance, a technique called AddressProbe proposed by Miller et al. [7] can be used to discover a great deal of information about the topology of the visible part of Bitcoin network, but has been made ineffective after release 0.10.1 of the Bitcoin Core application. An additional technique called TxProbe has been suggested by Miller [13], which has a similar use as AdressProbe. Unfortunately, TxProbe is considered invasive as it is expected to negatively influence the network's transaction rate [14].

We realize that executing the ChurnAddr strategy in the Bitcoin network can potentially cause disruptions. We consider a scenario, where we have gained a substantial amount of inbound connections. We can expect to have a high centrality in the network, and we might unintentionally have the adversary nodes collect sufficient connections for a minimum cut [15, p. 147]. Hence, we risk being responsible for reliably propagating information, and we cannot know for sure if a mistake, i.e. a sudden simultaneous disconnect from our part, will disrupt the Bitcoin network. In extent of potentially causing disruptions, we also have to realize that it is in the nature of the blockchain technology that data change is irreversible [16]. Thus, causing a disruption that somehow affects the Bitcoin blockchain may never be reverted.

3.1.2 Resource-related Issues when Performing the ChurnAddr Strategy

We identify time as a critical issue concerning real-life measurements of the Bitcoin network. We should expect that the ChurnAddr strategy requires long-term interactions with the network, which we also find in the BitcoinChurn project that uses a similar strategy [2]. Although some long-term measurements might be applicable to perform on the real Bitcoin network, it is without a doubt an advantage if same measurements could be performed in a shorter period of time but still represent a long-term study. Lastly, in context with this project's time constraints, this issue becomes a severe concern and risk, because the expected time period for real network experimentation (2592 to 3082 hours $\approx$ 4 month [2]) almost exceeds the allocated project time of about 4 to 5 months.

The cost of performing topology measurements of the Bitcoin network varies. Some work only require a single or a handful of nodes in order to carry out the intended measurements or demonstrate a concept [7, 17], while others may be dependent on several and even up to a hundred nodes [3, 2].

While we may discover a cost-efficient methodology for measuring the topology, we deem this too risky.

3.1.3 Using the Bitcoin Test Network and Summary

We briefly presume that we can overcome the ethical issues of the ChurnAddr strategy by testing it on the Bitcoin test network, Testnet3 [18]. Disrupting Testnet3 might be considered ethically okay, as this network is purely for implementation testing. Unfortunately, we find little information about the network's topology and churn, which are essential to the ChurnAddr strategy. This amplifies the resource-related issues, as we would additionally have to investigate Testnet3's topology and churn. Moreover, Testnet3 and the real Bitcoin network do not have the same participants and due to their different purposes, we should expect that Testnet3 participants can have different behaviors introducing diffidence to the results.

In summary, we find too many issues and high risks using either the real Bitcoin network or Testnet3 as platforms for testing the ChurnAddr strategy. While measurements on the real Bitcoin network would be preferred to obtain the highest accurate results possible, we deem the risks and challenges associated with this too high. Lastly, we find it advantageous to enable topology-related measurements outside the real Bitcoin network, as ethical concerns and large-scale orchestration of physical Bitcoin nodes are less of an issue.

3.2 Synthetic Bitcoin Network

To avoid misconceptions of the word 'model' when we elaborate on details, we use 'synthetic' as a term associated with the general act of modeling, which in context of synthesizing the Bitcoin network means 'enabling measurements outside the real Bitcoin network through a model'. In details, synthesizing means that the object under investigation, the Bitcoin network, is replicated in a fashion that allows experimenting with said replica without affecting the source object, and that the experimentation should provide some insight of the source object.

As there exist many ways of doing this, we must engage an investigation of what kind of synthesize technique is best suited. We propose three general approaches of doing this:

- **Formal modelling and verification**: Using a timed automata and providing queries to answer certain questions about the model.
- **Emulation**: Setting up an environment that can essentially replace the Bitcoin network and enables investigation of the emulated environment.
- **Simulation**: Carefully selecting components of the Bitcoin network to be included in a model, which is then investigated to generate predictions.

In this section we compare and evaluate the three proposed approaches to find the one best suited for the context of the ChurnAddr strategy. We also note, that the target environment is not simply a Bitcoin Core application – it is the whole Bitcoin network. For instance, an emulation of the Bitcoin Core application alone does not cover our targeted environment fully.

We also stress that there exist many explanations on the difference between simulators and emulators [19, 20, 21, 22, 23], some of which are more alike than others. Hence, we will elaborate on their differences while comparing them.

3.2.1 Formal Modeling

Formal modeling is strong in exploring branches of possibilities for *select* components of the source target. It checks not only for a single possible outcome, but it evaluates all possible outcomes making it possible to query comprehensive question like "Is it guaranteed that a stable Bitcoin node will eventually have an ingoing connection from all other nodes in the network?"

The weakness of formal modeling arises as the state-space grows, where the state-space comprises the set of variables in the model that can change over time during checking. Put in perspective, as a model increases its state-space, the model checking activity will increase its run-time exponentially. This is also known as the state-space explosion problem [24].

Modeling the Bitcoin network with even just a fraction of the nodes' features and a fraction of the total amount of nodes on the network imposes impractical run-times, which was the conclusion from previous experiments [2]. It must be mentioned, that these experiments followed guidelines like applying aggressive space reduction, reducing variables, and reducing randomness.

3.2.2 Emulation

An emulation reproduces the exact same external behavior as the targeted system and adhere to all its rules, meaning it must have the exact same inputs and outputs as the targeted system. In other words, an emulation is an exact replica of the targeted system that is running in a separated environment. Emulation is strong in creating a fully-featured synthetic environment, which enables a large set of queries to be asked and answered with potentially high confidence.

We realize that an emulation by its definition can be complicated to optimize, as modifying its behavior to achieve better performance contradicts its requirement to have the exact same behavior as the targeted system. Indeed, it is not necessarily impossible to speed up an emulation, as there do exist emulators that takes control of the clock time input, e.g. Visual Boy Advance [25] and DOSBox [26]. However, the Bitcoin network is many magnitudes larger and more complex than DOS and a GameBoy Advance making this a daunting and possibly impractical endeavor.

In relation to formal modeling, a bare bones emulation does not allow exploring all possibilities, which is an issue as the Bitcoin network incorporates randomness in parameters like latency and connection choice. In this case, it is necessary to run the emulation multiple times with different randomness seeds to establish a confident prediction.

3.2.3 Simulation

A simulation reproduces a similar behavior as the targeted system and adheres to a subset of its rules, meaning it only covers a subset of the same inputs and outputs as the targeted system. In relation to an emulation, a simulation does not provide an exact replica of the targeted system, however, it provides an indication of some of the targeted system's behavior.

Compared to an emulation, a simulation offers more flexibility in terms of selecting what should be synthesized, and it gives more room for optimizing as long as the resulting behavior is similar to the targeted system. Compared to formal modelling, a simulation does not have to follow a strict mathematical structure, which has the downside of not allowing powerful queries that grants guarantees, but in return allows more freedom to the structure.

We reflect on the simulator created during previous project[2], for which its strengths were focused on optimization and performing some topology-relevant measurements while allowing the whole network of many thousands of Bitcoin Core nodes to be simulated. Its shortcoming was a low level of confidence as many parts of the simulator's structure were greatly simplified to achieve practical run-times, and the simulator was tick-based with a coarse granularity making the concept of time inaccurate.

3.2.4 Choice of a Synthetic Environment

We recall our intentions of using a synthetic environment:

1. Invasive actions in the environment must not influence the real Bitcoin network, for which all approaches trivially fulfills this criteria.

2. It must allow topology-related measurements, for which all approaches fulfills this criteria while the simulation and formal modeling approaches can be simplified to focus on exactly this feature.
3. It should include an indication of result confidence, for which formal modeling naturally satisfies this while both an emulation and a simulation must implement this separately.
4. It must be capable of performing multiple months of real-time network behavior, for which a simulation and formal modeling allows more flexibility for performance optimization than an emulation.

We reflect on these comparisons and constitute a high-level summary of the three approaches' weaknesses in Table 3.1.

	Performance	**Code Analysis**	**Result Confidence**
Formal Modeling	Using the full power of verification, this becomes difficult	Is required	Is included
Emulation	Is inherently difficult to optimize	Not required	Is not included
Simulation	Allows more freedom to optimize	Is required	Is not included

Table 3.1: High-level comparison of synthetic environment approaches

In summary, no approach fits all criteria perfectly but all are possible candidates while posing different sets of challenges. We deem simulation as the best suited approach, because it is seemingly the most cost-effective approach with regards to its challenges, and because it mitigates the risk of rendering the synthetic environment impractical to use due to performance issues.

3.3 Existing Bitcoin Network Simulators

There exist some work that attempt to generate synthetic measurements of the Bitcoin network. A formal model in UPPAAL made by Chaudhary et al. [27] is used to investigate how malicious nodes can affect the applicability of a double-spending attack. A Bitcoin plugin by Miller and Jansen [28] for the Shadow simulator is used for investigating performance-related attacks in the Bitcoin network and running private Bitcoin networks. Both have made success and are part of workshops.

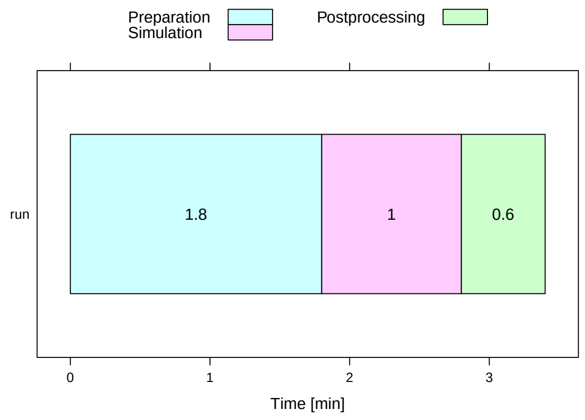
Other frameworks are available, such as Simcoin [29] and BitCoin Server Emulator [30], which also indicates that there is a certain attraction for constructing synthetic environments of the Bitcoin network. Except for BitcoinChurn [2], we have to the best of our effort not found a synthetic environment, which specializes in the topology of the Bitcoin network as influenced by the way a Bitcoin Core application evaluates and chooses connections.

We find three simulators which can potentially be used to demonstrate the ChurnAddr strategy: Simcoin [29] and Shadow [28], for which both emulate the Bitcoin Core application using different techniques. Moreover, BicoinChurn [2], simulates the Bitcoin network by simulating a small subset of the Bitcoin Core application's behaviors. We examine their use for this study.

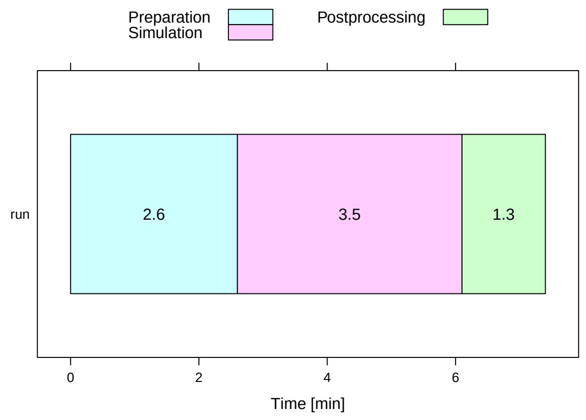
3.3.1 Simcoin

We engage experimentation with Simcoin. In summary, we find that the simulator is indeed of good quality, as running it requires little effort, and the Simcoin authors did a great job in allowing a Bitcoin Core application to be used with minimal code modifications. Moreover, it has been developed for almost two years with three contributors, which is a good indicator for its maturity and quality. While Simcoin is a promising tool, our goal's needs are skewed in relation to Simcoin's strengths:

- The Bitcoin Core application is emulated. Indeed, a fully emulated Bitcoin Core application is attractive, but we only need to focus on a small subset of its functionality, namely its features affecting the topology.
- Being constrained to emulation makes optimization a critical and difficult endeavor. We wish to simulate multiple months of Bitcoin network run-time, and optimizing Simcoin's performance to this need would ultimately entail modification of the Bitcoin Core code.
- While Simcoin's allowance of network configuration and its use of Docker should indeed make customized network orchestration possible, we have no easy way of predicting if this has unseen issues. In order to assess this, we would have to go more in-depth or even commit to Simcoin and Docker, which we deem is too risky in relation to the project's time constraints.



(a) 24 nodes, generated by Simcoin



(b) 36 nodes, generated by Simcoin

Figure 3.1: Simcoin run-times for 1 minute of emulation with varying amount of nodes

In evalution of Simcoin's performance, we find that it does not immediately allow speed up of the Bitcoin Core application emulation. We find this apparent in Figure 3.1a, where the simulation/emulation time is the same as Simcoin's run-time. We also discover that a Bitcoin Core application emulation proves too complex to reach acceptable run-times, which we find in Figure 3.1b where we emulate 36 nodes. For reference, we would need to emulate more than 126.000 nodes, which indicates that even if we speed up the emulation, the Bitcoin Core application emulation will still be a bottleneck.

3.3.2 Shadow

We investigate the applicability of using Shadow. At first glance, the Bitcoin-enabled Shadow simulator is a promising candidate as it can run about 6000 Bitcoin Core nodes [28, p. 6]. Moreover, the Shadow framework has been developed for multiple years with multiple contributors and has a Tor plugin as well, which again indicates it has good maturity and quality. Unfortunately, we find no indications of how good the run-time is, and our requirements are not aligned with Shadow's current capabilities:

- The Shadow Bitcoin plugin is 7 years behind the current latest version of the Bitcoin Core application [31], for which we must update this in order to create useful results.
- The topology is static meaning it cannot change during Shadow's simulation and nodes do not leave or join the network once the simulation has started [28, p. 5]. Unfortunately, these are non-negotiable features, which our goals require.

3.3.3 BitcoinChurn

We examine the simulator, BitcoinChurn [2]. This simulator is specialized in representing the topology-related features of the Bitcoin Core application and was used to take measurements of connection monopoly providing different adversary strategies. The simulator is part of a work, which conclusion stresses that the Bitcoin network is susceptible to a churn exploit. Indeed, the conclusion is backed up by its experiments, however, it is unclear whether these results are trustworthy as the simulator is greatly simplified to achieve acceptable run-times.

Although BitcoinChurn is specifically customized for topology measurements, it has a set of issues that must be addressed:

- it has a weak result credibility due to it is missing network features such as latency.
- it is unclear whether its simplified implementation of the Bitcoin Core application is correct.
- the use of a tick-based simulator with a very coarse timespan granularity is indeed great for performance, but it likely decreases accuracy significantly making the results less appealing.

3.3.4 Summary

We realize that although there already exist Bitcoin network simulators, they fall short of our intentions. Hence, we determine that constructing a Bitcoin network simulator with the purpose of allowing topology-related measurements and implementing the Bitcoin Core application's topology-related behavior is necessary. Fortunately, BitcoinChurn already has some of the work done, to which we choose to base a new simulator from this work. Although this alleviates the amount of work required to construct a suiting simulator, we stress that BitcoinChurn requires fundamental changes, which does induce a significant workload.

3.4 Simulator Structure

We plan the construction of a simulator capable of simulating the Bitcoin network's topology and message propagation. From the existing work of BitcoinChurn, we approach a series of issues that we account for during this section.

Following modelling and simulation compendiums and guidelines by various authors [32, 33, 34], we lay out the broad constituents of our needed simulator from the perspective of improving the BitcoinChurn simulator.

3.4.1 BitcoinChurn: A Tick-based Simulator

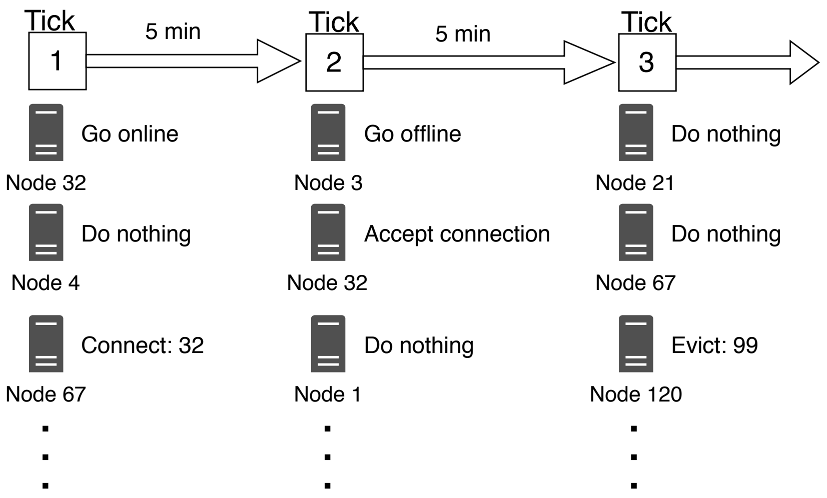


Figure 3.2: BitcoinChurn simulator tick illustration showing an arbitrary sequence of nodes being invoked for every tick

Our first step is to engage the issues of the BitcoinChurn simulator. As previously mentioned, one of the biggest issues with the BitcoinChurn simulator is its concept of time. In details, the simulator works by running a finite number of 'ticks', for which each tick represents a rigidly defined time period or 'slice of time'. Specifically, the BitcoinChurn simulator operates a tick as a 5 to 6 minute time period, which effectively means that anything modeled in the simulator is forced to await this time period between every action performed. We illustrate this in Figure 3.2, where the simulator sequentially and randomly invokes all nodes in a tick before proceeding to the next tick.

The simulator must be able to operate with a time granularity of milliseconds accuracy, reason being that we will find the network's latency as being the smallest unit of time in Section 4.5. Indeed, this is also possible with a tick-based simulator. In Figure 3.2, we can imagine setting the tick from a 5 minutes timespan to a 1 millisecond timespan. Unfortunately, this provides very poor performance, because each node is invoked every millisecond although they are only going to *Do nothing*.

3.4.2 The Event-oriented Simulator Paradigm

We part simulators in two different paradigms as proposed by Matloff [34]. We shortly define these paradigms:

Activity-oriented The simulator runs in terms of jobs or 'ticks', which are scheduled with a constant time span. For each tick, the simulator checks if any of its simulated entities wishes to perform an action, and the simulator will let an entity perform its job if so desired.

Event-oriented Instead of the simulator checking its entities between ticks, the entities become responsible for adding future events to the simulator. In this case, the simulator works as an event queue sorted by the timing of the events, and it always executes the next event in its queue.

We note that the paradigms mentioned here are high-level and flexible terms, and that there exist variations of each paradigm as well as a process-oriented paradigm, which we will not go into details with. The paradigm of a simulator is fundamental to its implementation and changing a simulator from one paradigm to another is not a trivial task.

As mentioned, we derive from a tick-based simulator, namely the BitcoinChurn simulator [2]. A tick-based simulator is in the category of an activity-oriented simulator. In some circumstances, an activity-oriented simulator can be considered as providing the best accuracy compared to the other paradigms, because it can be used as a continuous simulator as seen in works requiring analytically or numerically solutions, e.g. a predator-prey simulation [32, p. 255]. This level of accuracy will be very difficult to apply to the BitcoinChurn simulator, because the behavioral information we will need to collect from the Bitcoin network essentially have to be expressed as a differential equation as found in continuous-time dynamic system [32, p. 247-266]. For reference, this kind of modeling is often applied in experiments related to the laws of physics [32, p. 247-249].

We propose using an event-oriented approach as a better alternative. A simulator in this paradigm is also known as a 'discrete event-driven simulator', and it is in fact the same paradigm used by Shadow [35]. Performance is one of the big highlights of the event-oriented approach. As the simulator executes the events in a chronological order, where we assume that the simulation state does not change in-between events, the simulator will jump its internal representation of time to that of the next event in order. Effectively, we gain an accurate and dynamic representation of time during simulation, and we avoid having to compute the Bitcoin nodes' idle behavior.

We illustrate this concept in Figure 3.3, where an event-based simulator is currently letting node 201 do work at time 21:30. Node 201 is then planning to go online at 21:43, which the simulator adds to its chronologically ordered event queue. We see that node 76 at 21:41 is able to initiate a connection to node 2 before node 201's planned event, and we see that the simulator is able to jump in time between events while allowing all necessary node actions to be performed.

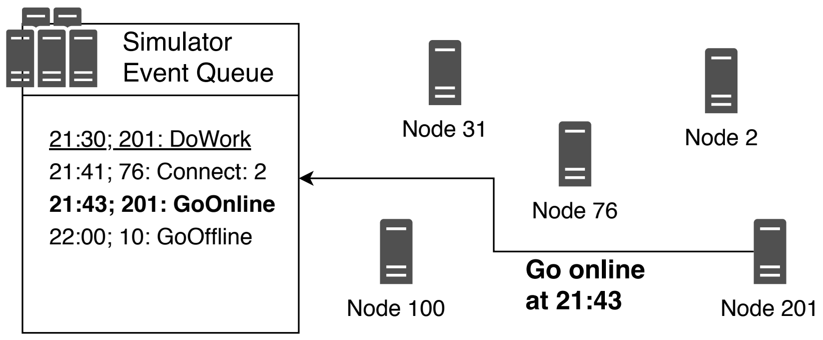


Figure 3.3: Concept of an event-based Bitcoin network simulator, where node 201 enqueue a future event

4 Network Measurements and Implementation

We have achieved, that we must construct an event-based simulator of the Bitcoin network, for which we must employ some level of accuracy in order to get trustworthy results. We do this by implementing the simulator according to the specifications in Chapter 2, where we present the ChurnAddr strategy and analyze the problem domain for the information required to demonstrate the strategy. From the aforementioned chapter, it is clear that our proposed simulator must be data-driven, meaning that it must accept and perform according to behavioral information gained from the real Bitcoin network.

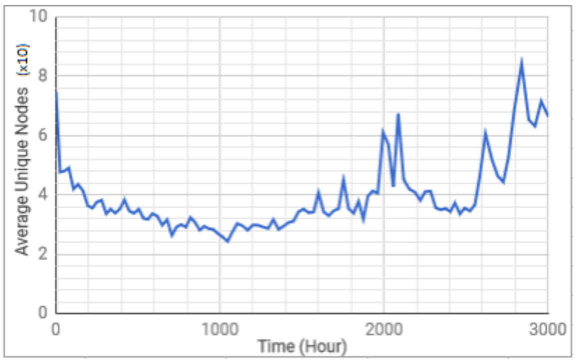
This chapter gathers the information required by the simulator to generate trustworthy results, which we present in parts. We will present findings and methodologies for gathering data of the network's topology, churn, and latency. The Bitcoin source code is presented as pseudocode, meaning we translate it to be easier read, which is an important step in constructing the simulator, as this translation can be used to pinpoint potential flaws in the simulator or verify that our model is correct. Furthermore, we present how we implement this in our simulator along with performance optimizations and assumptions.

4.1 Reconstructing the Bitcoin Network Topology

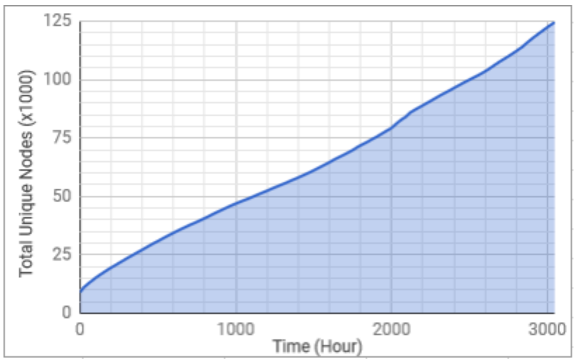
As trivial as it may be, our proposed simulator must start at some point, for which we must construct a trustworthy start state. We find that creating an appropriate starting topology as essential for accuracy. Although this requirement may be trivial, its execution is not, as we mention in Section 3.1.1 on the ethical and resource demanding topics of measuring the Bitcoin network's topology. We recall from this section, that there is indeed a proposed methodology (TxProbe) of measuring this, however it is considered unethical [13]. Indeed, the topology of the Bitcoin network has once been studied and partly mapped by Miller et al. [7], however this was in 2015, and we presume that the topology have changed since then.

In terms of cost-benefits, we may indeed get an extra benefit from using the aforementioned topology study to generate our simulator's starting state, however we cannot ascertain what is the head and tail node from the connections [7]. Moreover, we will not know how to map the topology with our churn data presented in Section 4.2, as these data sets contain information about a specific node's churn rate. Hence, we use the same approach as in the BitcoinChurn simulator [2], where the topology is randomly generated as part of a 'priming' state in the simulator before the actual simulation starts.

We prime our simulator with a topology, which also accepts different random seeds for comparative analysis. We prime by collecting a list of nodes that are online from the very start of the simulation and let these discover each other through a seeding function that collects a randomized subset of all nodes online. The nodes during prime are then allowed to interact with each other to initiate their needed connections and to get the knowledge of some of the other nodes in the network. Lastly, to increase the robustness of the starting state topology, we allow the simulator to simulate 10 hours of simulation-time before any extra-ordinary nodes (i.e. victims or adversaries) are inserted and started. We use 10 hours, as we found in preliminary runs of the simulator, that the simulated nodes get 'stable' at around an 8 hours of simulation-time, meaning that the nodes have gained a steady amount of known nodes, and that they are receiving and propagating a steady amount of messages. Steady means that the amount does not deviate much from its current value as the simulation time progresses.



(a) Average uniques per hour



(b) Total uniques over time

Figure 4.1: Unique IP addresses joining over time from BitcoinChurn [2]

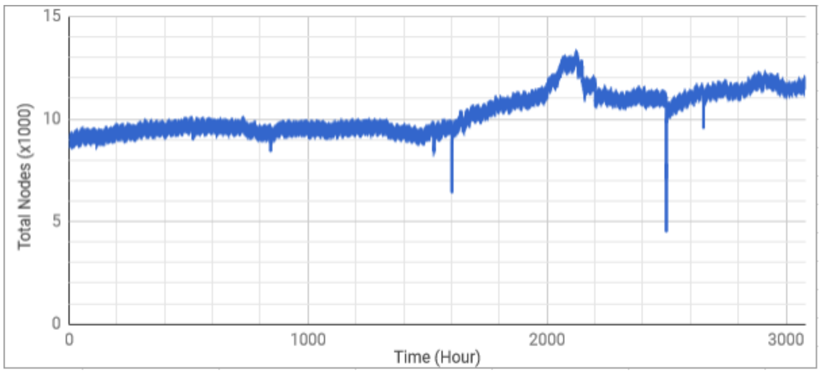


Figure 4.2: Peers in total over time from BitcoinChurn [2]

4.2 Collecting and Using Churn Data

As we find in Chapter 2, getting real churn data of the Bitcoin network is essential to the ChurnAddr strategy, as the strategy depends on this characteristic in order to work, and because we expect that a higher churn rate will influence the strategy's results letting the strategy seem more effective.

The BitcoinChurn project [2] collects the necessary churn-related data of the Bitcoin network for a period of 128 days using the Bitnodes API [4]. These churn measurements can be seen in figure 4.1 and 4.2. As a side note, we try to collect additional data, as this will allow an even longer period of simulation or at least enable the option to test the ChurnAddr strategy with different sets of real-life churn data. However, Bitnodes have since then changed their data policy and now only provides snapshots for the last 60 days, which unfortunately is not enough for our use case, and it leaves a time gap of about one month between the BitcoinChurn project's churn data and the newly acquired data. Hence, we choose to reuse the churn data obtained during the BitcoinChurn project.

We construct our simulator to strictly follow the churn data acquired from Bitnodes. This is done at the simulator's priming stage, where each node is configured to either go online or offline according to the data. From this data, we acquire that during the 128 days time period 126,217 unique nodes join and leave the Bitcoin network.

4.3 Code Review

To get a deep understanding of how messages are used in the Bitcoin network, we carefully review Bitcoin Core source code. This section will start by describing the type of messages the Bitcoin network uses, and generally how two nodes connect to each other. After a general overview, this section will go into depth with how a new connection is chosen and how a node chooses a connection to terminate if the node reaches its maxed allowed connections. Lastly this section will explain how nodes react when a ADDR message is received and how the addresses in the message are stored, as this is one of the main aspects on how addresses are shared across the network.

Throughout this section, pseudocode will be used to explain Bitcoin cores functionality. It should however be noticed that Bitcoin core includes a lot of checks and edge cases that have been omitted together with several other details in favor for readability and understand-ability. We use the pseudocode as an intermediate translation from the Bitcoin Core source code to our simulator implementation. Moreover, this serves as a conceptual which can be used as a tool for verifying the simulator's accuracy. Since no detailed documentation of the Bitcoin Core code exist, details in this section is translated directly from the Bitcoin Core source code [36].

4.3.1 Bitcoin Core Connection Establishment and Maintenance

We analyze, reverse engineer, and implement the Bitcoin Core application's procedure for initiating and creating a connection as well as maintaining a connection. As mentioned in Section 2.3.2, the Bitcoin Core application uses 26 different messages to interact with the Bitcoin network. Although, we only focus on the ADDR, GETADDR, and block propagation messages, we have to temporarily expand our collection of focused messages to achieve a deeper understanding of the connection establishment and maintenance procedures.

We introduce six Bitcoin Core messages that are used when a new connection is established or maintained:

- *VERSION*: This message contains, among other things, the node's protocol version, features to be enabled for this connection, a random nonce, and the user agent.
- *VERACK*: This message is sent as a reply to a VERSION message and does not contain anything except a message header with the string 'VERACK'.
- *PING*: This message is used to ensure that the TCP connection is still valid. The message contains a nonce to keep track of what PING message the PONG message is a response to.
- *PONG*: This message is a response to a PING message with the corresponding nonce.
- *ADDR*: This message contains information about known nodes on the network. The message can at maximum contain 1000 IP addresses.
- *GETADDR*: This message is used as a request to a node for information about known peers. One or more ADDR messages are expected as response.

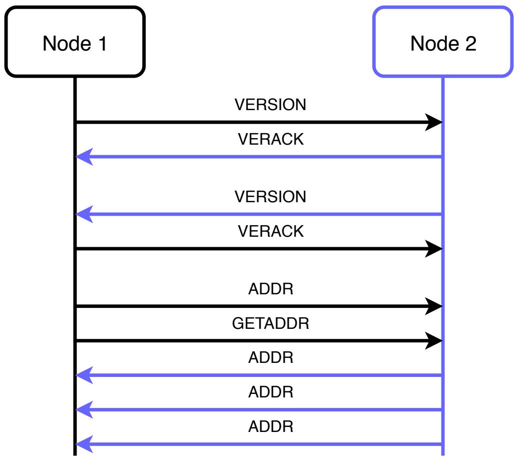


Figure 4.3: Messages when two Bitcoin nodes establish a connection.

A handshake is initiated by a Bitcoin Core node if the node does not have the desired amount of outbound connections, where a Bitcoin Core node will by default strive to establish and maintain 8 outbound connections while allowing 117 inbound connections. Whenever a handshake is initiated, the involved nodes perform a series of message exchanges, which we present as a sequence diagram in Figure 4.3. After a connection is established there is no difference in the behavior regarding transaction, address, and block propagation on an inbound or outbound connection.

For a complete overview of a Bitcoin Core connection establishment, we elaborate Figure 4.3. We define the first stage of a connection establishment. Node 1 sends version details to Node 2 and awaits an acknowledgement and version details from Node 2. At this point, Node 1 presumes that Node 2 is online and ready, to which Node 1 simply replies an acknowledgement of the connection being desired and accepted. At the second stage of an establishment, Node 1 will send an ADDR message that solely includes Node 1's address, which in accordance with the Bitcoin Core application will be propagated by Node 2 and henceforth flooded by the rest of the Bitcoin network. At the third stage, Node 1 will request Node 2 for information about known nodes in the network, to which Node 2 will at maximum reply with up to 3 ADDR messages each containing 1000 IP addresses. Additionally, Node 2 will at maximum reply with 23 % of its known addresses. Node 1 will not propagate these addresses and will simply store them for own use. Lastly at the fourth stage, which is not depicted in Figure 4.3, Node 1 and Node 2 will continue to maintain the connection using PING and PONG messages, and they can propagate transactions, blocks, et cetera as long as the connection is maintained. We refer to the last stage as the 'keep-alive stage'

Simulator Implementation Details and Optimization

We identify the connection establishment procedure as being essential to our simulator. However, we also find that it is unnecessarily complex for our study, because it consists of messages that are purposed for handling issues regarding real life networking such as connection drop and imperfect information about foreign nodes in the network.

We disregard the VERACK and VERSION messages. Hence, we presume that all simulated nodes posses the full range of Bitcoin Core features, and that they are all running the latest version of the Bitcoin Core application. We remove a substantial amount of complexity from doing this, as we can implement the simulator to only replicate the default behavior during a connection establishment, in which all aforementioned preconditions hold.

We also disregard PING and PONG mesages. We assume that our simulated Bitcoin network is fully reliable in terms of connectivity. Hence, a connection can only be dropped if one of the involved nodes intentionally disconnects or goes offline. Again, we reduce the complexity from doing this, because we will not have to model the ping/pong information between nodes, and we will not have to model events representing unintentional connection drops. In essence, this removes the whole keep-alive stage from the simulation implementation, and we simply assume that given a node should disconnect, its connected nodes will be notified of this immediately.

We keep the complete third stage of the connection procedure. We find that this is essential for the network topology, as it directly implicates address propagation. Lastly, we find no more room for optimization, as both the ADDR and GETADDR will be implemented in accordance with the Bitcoin Core code, meaning we also implement the logic that covers the edge cases found during the third stage of a connection procedure.

4.3.2 Choosing a New Outbound Connection

As we found in previous section on the connection establishment and maintenance procedures, there is a point of time, where a node have to choose a node from its known nodes and establish a connection to this node. The Bitcoin Core code uses the mental model of having 'buckets' of known nodes, in which the application can insert up to 64 known nodes or 'IP addresses' in each bucket. Additionally, the application has two ranges of buckets:

1. A bucket range comprised of 1024 buckets storing *new* IP addresses, which we refer to as the NewBuckets
2. A bucket range comprised of 256 buckets storing *tried* IP addresses, which we refer to as the TriedBuckets.

From carefully reviewing the Bitcoin Core code, we find that these buckets are used during three distinct events:

1. When an ADDR message is received, the contained IP addresses are potentially stored in the buckets following a certain logic
2. When a 'feeler connection' is performed on a specific IP address, this IP address is moved to the TriedBuckets
3. When a candidate for an outbound connection is to be chosen, the buckets provide a biased randomness to the choice of the candidate.

We also note, that once a Bitcoin Core nodes has obtained its desired outbound connections, the thread responsible for performing this condition changes strategy to exercise the mentioned 'feeler connections'. This behavior helps the node to obtain more detailed information about the network and its known nodes, in which this behavior tests whether a given node is online. If this condition holds, the given node's IP address is moved to the TriedBuckets, where it in general will have a higher chance of being chosen as a candidate for an outbound connection. We recognize that the details associated with the bucket logic is complex, thus we abstract from these details when presenting the pseudocode for choosing a new address to connect to as seen in Algorithm 1.

We elaborate on the procedure in Algorithm 1. The procedure runs repeatedly as a separate thread until interrupted. The procedure initially checks if the Bitcoin Core application has bootstrapped, meaning it has obtained a collection of known nodes, where line 3 to 5 is the fallback method if the bootstrapping is unsuccessful.

When bootstrapping, the Bitcoin Core application defaults to querying six hard-coded DNS servers for a list of known nodes. Once a query has been successful, the application performs a 'one-shot connection' to some of the retrieved nodes, meaning the application sends a GETADDR message and disconnects immediately after receiving the expected ADDR messages.

If the application is unsuccessful in querying all the DNS servers, the fallback method on line 3 to 5 is called, which retrieves a list of 1525 hard-coded addresses to be used as one-shot connections instead. Should both fail, new addresses must be added manually.

On line 6 to 9, it is decided if the new connection is a regular outbound connection attempt or a feeler connection. As long as the application has less than 8 outbound connections, it will try to establish a new connection. The inner while loop on line 10 to 30 selects a random IP address or 'node' and tests if the selected node meets all of the requirements for a new outbound connection. If not, the loop will restart and select a new random address. If the address do fulfill all the requirements, the application will attempt to establish a connection on line 32.

Simulator Implementation Details and Optimization

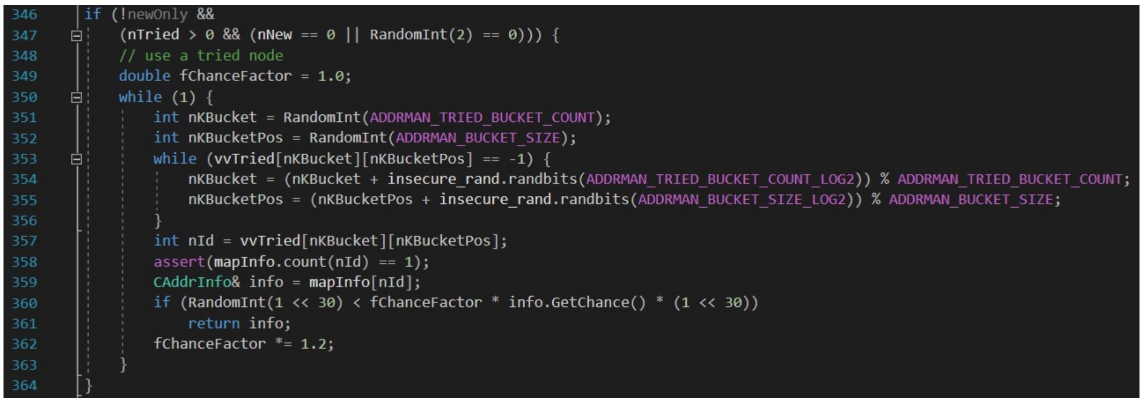
We realize that the presented procedure for choosing a new outbound connection is essential to the ChurnAddr strategy, as it is this part of the Bitcoin Core application that is targeted to be exploited. This is also one of the most challenging parts to implement, because its use of buckets is performance intensive. In Figure 4.4, we present a screenshot of the actual code responsible for choosing an outbound connection candidate from the buckets implemented by the Bitcoin Core application.

Biased Randomness and Hash Table Performance

We present the first performance issue. On line 351 and 352, a random bucket and a random position in that bucket is chosen. From line 353 to 356, a new random bucket and position is chosen as long

Algorithm 1 Choose Outbound Connection

```
procedure ThreadOpenConnections
    while not Interrupted do
        if number of known addresses $==0$ and $currentTime-$time in this loop $> 60seconds$ then
            Add fixed seed nodes as DNS doesn't seem to be available.
        end if
        $nOutbound \leftarrow$ Number of outbound connections
        if $nOutbound >= 8$ && $timeSinceLastfeelerConnection >$ 120 seconds then
            $fFeeler \leftarrow true$
        end if
        while not interrupted do
            if $fFeeler$ then
                $addr \leftarrow$ random address from buckets of *new* addresses
            else
                $addr \leftarrow$ random address from *tried* or *new* addresses.
            end if
            if $addr$ is local or $addr$ is in same group as another outbound then
                break
            end if
            if inner loop ran more than 100 times then
                break
            end if
            if $addr$ does not have all required services then
                continue
            end if
            if $addr$'s port is not default and inner loop ran less that 50 times then
                continue
            end if
            *addrConnect* $\leftarrow$ *addr*
            break
        end while
        if *addrConnect*$!= null$ then
            Create a connection to $addr$.    ▷ If it is a feeler connection (fFeeler == true), a small random delay is introduced before connecting.
        end if
    end while
end procedure
```



```
346 if (!newOnly &&
347     (nTried > 0 && (nNew == 0 || RandomInt(2) == 0))) {
348     // use a tried node
349     double fChanceFactor = 1.0;
350     while (1) {
351         int nKBucket = RandomInt(ADDRMAN_TRIED_BUCKET_COUNT);
352         int nKBucketPos = RandomInt(ADDRMAN_BUCKET_SIZE);
353         while (vvTried[nKBucket][nKBucketPos] == -1) {
354             nKBucket = (nKBucket + insecure_rand.randbits(ADDRMAN_TRIED_BUCKET_COUNT_LOG2)) % ADDRMAN_TRIED_BUCKET_COUNT;
355             nKBucketPos = (nKBucketPos + insecure_rand.randbits(ADDRMAN_BUCKET_SIZE_LOG2)) % ADDRMAN_BUCKET_SIZE;
356         }
357         int nId = vvTried[nKBucket][nKBucketPos];
358         assert(mapInfo.count(nId) == 1);
359         CAddrInfo& info = mapInfo[nId];
360         if (RandomInt(1 << 30) < fChanceFactor * info.GetChance() * (1 << 30))
361             return info;
362         fChanceFactor *= 1.2;
363     }
364 }
```

Figure 4.4: Code Snippet of Select_ function in bitcoin core

as the chosen position in the given bucket is empty. While testing this snippet in our simulator, we find our first performance issue, as this dominated a major part of the simulator's CPU usage, and we can see why. In worst case, if the node only knows about one address, it risks staying in the loop until it has tried all 1024 buckets $\cdot$ 64 positions $= 65536$ positions. Indeed, the actual cases in the simulator has better conditions, and this is most apparent in newly joined nodes, which had a small collection of *knowns*. Nevertheless, the simulator shows run-times longer than the simulation-time, which is unacceptable for our purpose.

To increase the performance of this snippet, we implement the buckets as a .NET dictionary, which is implemented as a hash table providing almost O(1) complexity at look-up [37]. We further associate each address with is own 'weight', which represent the bias for the respective address, where a higher weight will make the given address more likely to be chosen. This implementation avoids the aforementioned while loop making sure that a random IP address will be chosen at the first attempt, and that it is still possible to implement a bias in the randomized selection.

Removing Unnecessary Randomness

We present the second performance issue from Figure 4.4. Given that previous snippet finds an IP address, there is only a slim chance that it will be returned. The condition for returning the found IP address can be seen on line 360 to 362. For each time we go through the outer while loop, the chance of returning a found IP address increases. During this snippet, the *info.GetChance()* method returns a constant value respectively to the IP address found, which is then used as a bias. Simply put, this makes the procedure discard multiple otherwise valid, randomized, and biased IP addresses before returning, and we are unsure of the purpose for this.

We find that the important part of this snippet to the ChurnAddr strategy, is that known nodes are biased according to how many times they appear in the buckets and how recently they have been connected to (*info.GetChance()* provides this information). Hence, we implement additional functionality to our address dictionary, which now in addition to a weight also provides the *info.GetChance()* modifier. Doing this, we avoid the outer while loop giving us the ability to choose a randomized outbound connection at first attempt, where the choice is biased towards nodes that appear in the buckets multiple time and nodes recently connected to.

4.3.3 Eviction of Inbound Connections

Upon receiving a VERSION message, the Bitcoin Core application interprets there is an incoming connection request. Given that our Bitcoin Core application has filled its maximum allowed inbound connection, which is at 117 by default, the application will initiate an attempt to evict existing inbound connections. In general, the application will evaluate whether to evict its existing connections based on a set of parameters, which is part of ensuring that an adversary will have difficulties in forcing evictions of otherwise non-adversary nodes.

Algorithm 2 Try to evict a connection to make room for a new

```
1: procedure AttemptToEvictConnection
2:     for all nodes $n$ we are connected to do
3:         if $n$ is inbound or $n$ is white listed then
4:             continue
5:         end if
6:         add $n$ to $vEvictionCandidates$
7:     end for
8:     Remove the 8 nodes with the lowest minimum ping time from $vEvictionCandidates$.
9:     Remove the 4 nodes that most recently sent us transactions from $vEvictionCandidates$.
10:    Remove the 4 nodes that most recently sent us blocks from $vEvictionCandidates$.
11:    Remove half of the remaining nodes which have been connected the longest from
       $vEvictionCandidates$.
12:    Remove all nodes from $vEvictionCandidates$ that is not part of the ADDR group with most
       nodes. If two or more ADDR groups have the same number of nodes, the group with the newest
       node is selected.
13:    if $vEvictionCandidates$ contains any elements then
14:        Evict the node that most recently connected to us from $vEvictionCandidates$.
15:    else
16:        No nodes can be evicted.
17:    end if
18: end procedure
```

We elaborate on the eviction evaluation in Algorithm 2. On line 2 to 7, a list of all the node's inbound connections is generated as eviction candidates. On line 8, we remove the 8 candidates with the lowest minimum ping. Line 9 to 11 remove candidates that are best suited to reliably propagate information and maintain the blockchain.

After this and if a suitable candidate is found, the most recently connected candidate is selected for eviction, which will make room for other inbound connections. If no suitable candidate is found, the node will not acknowledge the received VERSION message with a VERACK message.

Simulator Implementation Details and Optimization

We find it trivial to implement most of the eviction logic in the simulator, as it does not posses any difficult performance issues. However, we find a few details in this pseudocode that will require very complex Bitcoin Core features in order to work, namely block and transaction propagation, which both uses an aggressive flooding strategy. Hence, we do not implement the candidate evaluation that prioritizes nodes which have recently sent transactions or blocks.

We expect this to have an insignificant influence on the accuracy of the simulator. Firstly, with regards to the ChurnAddr strategy, there is no point where the adversary is dependent on forcing connection eviction. Secondly, in our simulator we find that evictions rarely happen, and this seems more likely to happen in the start of the simulator, where the adversaries have gained only a few or none inbound connections.

4.3.4 Receiving ADDR Messages

ADDR messages are used for propagating information of what nodes are potentially online, hence it covers the feature of peer discovery in the Bitcoin network. We identify an ADDR message being used in three different scenarios.

1. For every Bitcoin Core node in the network, an ADDR message will be sent every 24 hours as a self-announcement. The node in question sends an ADDR containing its own IP address to all of its connections, for which the message is then continuously propagated to the network.

2. Whenever a new connection is successfully established, the connection initiator will request ADDR messages of the other node in the connection by sending a GETADDR message. The initiator can expect up to three ADDR messages with 1000 IP addresses in each message.

3. Whenever receiving an ADDR message, the receiving node will evaluate if some of the contained IP address should be forwarded.

Algorithm 3 Received an ADDR message

```
procedure ProcessMessage
    vAddr ← addressees in ADDR message
    if vAddr contains more than 1000 addresses then
        punish sender with 20 and return.
    end if
    for all addresses addr in vAddr do
        if addr.timeStamp <= 100000000 || addr.timeStamp > current time + 10 minutes then
            addr.timeStamp ← current time - 120 hours
        end if
        AddAddressKnown(addr)                                  ▷ Function is explained later
        if addr.timeStamp > current time - 10 minutes && cAddr contains less than 11 then
            if is the addr one we think we can connect to? then    ▷ For example Thor network, Local, IPV4, IPV6
                nRelayNodes ← 2
            else
                nRelayNodes ← 1
            end if
            Forward address to nRelayNodes random connections. Uses deterministic randomness to send to the same nodes for 24 hours.
        end if
    end for
end procedure
```

We elaborate on how ADDR message are forwarded in Algorithm 3. On line 3 to 5, given that the sender should violate the 1000 IP address limit, the receiver 'punishes' the sender by adding a 20 points penalty score to its internal information about the sender. If a node obtains a penalty score of 100, the punishing node will ban this node by disconnecting it and ignoring connection requests from it for the next 24 hours. This punishment is purposed to ensure that nodes that do not behave appropriately (i.e. adversary nodes spamming ADDR messages) are rejected by the network. We also note that the penalty feature can be used by adversaries to force rejection of other nodes, e.g. Tor nodes, in the network [38].

We examine all IP addresses found in the ADDR message. On line 7 to 9, the IP addresses timestamps are normalized, and if they are too old or in the future, they are set to be 120 hours old as a form of penalty. On line 10, the IP address is attempted to be stored in the application's buckets as previously discussed. We will elaborate on the bucket storing functionality in Section 4.3.5. On line 11, if the timestamp is no older than 10 minutes and the received ADDR does not contain more than 10 addresses, we forward it. On line 12 to 17, we decide what node or nodes should receive the forwarded addresses, which we name relay nodes. The relay nodes are chosen and remembered for every 24 hours and are selected randomly from our application's pool of connections. This is also the functionality that allows self-announcements to be propagated. We realize that the logic for receiving ADDR messages is essential to the ChurnAddr strategy, hence we implement this logic fully in our simulator.

4.3.5 Adding Addresses to the Buckets

Should a Bitcoin Core application attempt adding an IP address received from an ADDR message, it will check for various conditions of the received address. This is done by the *Add_* function presented as pseudocode in Algorithm 4, which is responsible for placing its call parameter, an IP address, in the correct bucket and position or reject it if certain conditions hold. On line 2 to 5, we check if the received address is a self-announcement. If this is not the case, the address will have a 2 hours time penalty. Coarsely explained, this makes the address less trustworthy and worsens its chances of being stored.

On line 6 to 33, we check if the address has already been stored, and if this is the case we update our already stored address' associated information and we potentially store another instance of the address. On line 13 to 18, given that our received address is 'new enough', we update our stored address' timestamp. If this condition does not hold, we simply return, because we do not intent to store an address older than the one we already have stored. On line 19 to 21, we check if the address is in the TriedBuckets, for which if this is true, then we do not store it and simply return – Addresses are allowed to be stored only once in the TriedBuckets, and we are not allows to store addresses in the NewBuckets, which are already stored in the TriedBuckets. On line 22 to 25, we make sure to store the address in the NewBuckets if and only if we have not already stored it 8 times, for which we would otherwise just return. On line 26 to 32, we decide by random if our address should be stored, where the chance of this decreases as it has been stored more times. On line 34 to 45, we try to save the address to our NewBuckets. This is done by selecting a random position in the new buckets, for which the address is stored given that the position is empty. Otherwise, we check if the existing address appears multiple times in the buckets, and if our new address does not appear at all in the buckets. Given that these conditions hold, we overwrite the existing address on our given position with our new address.

Simulator Implementation Details and Optimization

We realize that the logic for adding addresses is essential to part of the ChurnAddr strategy, hence we implement this logic in our simulator. However, we have to implement this logic carefully, as we have partly removed the notion of having buckets. What we find an important notion here regarding the ChurnAddr strategy, is that as a node has many known nodes in its buckets, the likelihood of adding a new known, e.g. an adversary node, becomes smaller.

4.4 Slim Node Implementation

Having implemented the behavior from the Bitcoin Core source code to the simulator as described in the previous section, we test the simulator's run-time and identify performance issues.

To assess the run-time, we execute a simulation of 126.217 nodes with a timespan of 168 hours (7 days). The simulation was put to a stop after reaching 21% in 7 hours and 20 minutes. This corresponds to a 35.28 hours simulation-time executed in 7 hours and 20 minutes run-time. This means that the simulator is $35.28/7.33 \approx 4.82$ times faster than real time. While this is impressive taking the amount of nodes and messages into account, it is not fast enough to simulate our target of three month.

We use Visual Studio's Performance Profiler to identify the performance issues, from which we find that it is the implementation of Bitcoin Core's address flooding protocol, which accounts for a majority of the CPU usage. Specifically, the diagnostic reveals that most of the CPU time was used when handling received ADDR messages. As the functionality for handling ADDR messages is already thoroughly optimized, we determine to disable this in simulator. Unfortunately, this makes it inapplicable to test the ChurnAddr strategy's exploit of address propagation, however, the simulator will not be usable otherwise.

Disabling the ADDR messages, we implement a slim version of the Bitcoin Core implementation. Since the slim node has removed ADDR messages, it will not be usable to show how frequently sent self-announcements can increase the amount of inbound connections over time. However, the simulator can still show the part of the strategy regarding churn exploit, where we predict that a stable node will increase its inbound connections over time by simple staying online. Furthermore, the slim

Algorithm 4 Add Address to Known

```
procedure CAddrMan::Add_(addr)
    nTimePenalty ← $2hours$
    if addr == sender of ADDR message then                    ▷ Self announcement
        nTimePenalty ← 0
    end if
    if we know about the addr then
        $pinfo$ ← the address from known correlating to the new addr
        if current time − addr.timeStamp < 24 hours then
            $nUpdateInterval \leftarrow 1hour$
        else
            $nUpdateInterval \leftarrow 24hour$
        end if
        if pinfo do not contain a time stamp || pinfo.timeStamp < addr.timeStamp − nUpdateInterval − nTimePenalty then
            pinfo.timeStamp ← addr.timeStamp − nTimePenalty
        end if
        if addr.timeStamp <= pinfo.timeStamp then
            return false
        end if
        if pinfo is in tried buckets then
            return false
        end if
        nRefCount ← number of times pinfo is in new bucket
        if nRefCount == 8 then
            return false
        end if
        nFactor ← 1
        for $n \leftarrow 0; n < nRefCount; n++$ do      ▷ stochastic test: $2^{nRefCount}$ times harder to increase it
            $nFactor* = 2$
        end for
        if nFactor > 1 && random number between 0 and nFactor ! = 0 then
            return false
        end if
    end if
    BucketPos ← random position in new bucket
    if BucketPos does not contains addr then
        fInsert ← is bucket position free
        if !fInsert then
            if address in BucketPos is in new bucket more than once && addr is not in new bucket then
                fInsert = true
            end if
        end if
        if fInsert then
            BucketPos ← addr
        end if
    end if
end procedure
```

node implementation still allows measurements of how a high number of inbound connections to an adversary can influence block propagation in the network.

For the rest of this project, the slim node will be used for simulations as the removed ADDR message resulted in an acceptable run-time while still making it possible to measure the aforementioned results.

4.5 Collecting Latency and Latency Analysis

As mentioned in Section 2.3.1, latency influences the network's efficiency in network propagation, and it is relevant to some of our required results for assessing the ChurnAddr strategy's efficiency. Furthermore, the latency also influence how many hops a self-announcement can perform before propagation stops.

When measuring latency to be used for our simulator, we wish to find a correlation of the nodes we have information of from our existing churn data and their latency between each other, as we would be able to accurately map the exact latency between nodes in the simulator. Unfortunately, this information is not available, so we investigate if we can deduce the latency from the nodes' geographical location, which we do have available. Moreover, we investigate if the round trip time (RTT) can be said to have a correlation with latency. RTT is the time it takes to send a message plus the time it takes for the acknowledgement to be received, which we can use to give an indication of the time it takes to send a message back and forth in our simulator.

We also predict that the Bitcoin Core application will have an 'internal latency', meaning we will have to take a processing delay into account. If the node is heavily loaded, the handling of the message might be queued resulting in a greater delay, and we expect that some messages received need processing before they are reacted upon. This means that a connection to a busy node might have a low RTT, but still respond slowly to new messages. Moreover, we are also interested in measuring the time to live (TTL) between nodes, simply because this might reveal a useful insight that might be correlated with latency or the geographical location.

What we wish to obtain in this section:

- Find a correlation between latency or TTL and geographical distance.
- Compare the latency with studies of other P2P networks
- Find a correlation between Bitcoin ping and Internet Control Message Protocol (ICMP) echo. This way we can measure the latency of nodes not allowing pings.

4.5.1 Data Collection Methodology and Results

To measure the RTT and TTL, we obtain a list of all online nodes using the BitNodes API [4]. We note that since BitNodes scans the network periodically about every 5 minute, they will not have any false positives at the individual scans, however, we can expect to not have a 100 % complete list of online nodes.

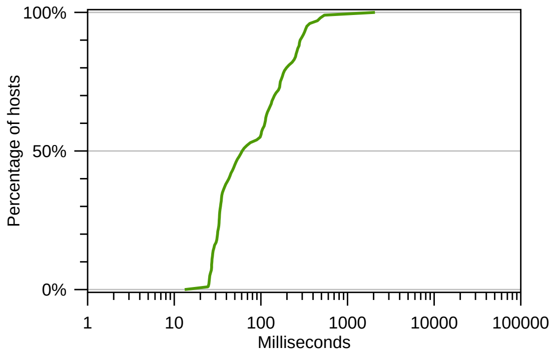
For each online node on the network we send 10 ICMP Echo Requests and 10 Bitcoin Ping messages, which gives us a collection of RTT, TTL and Bitcoin Ping time for each node. To ensure that we do not add additional delay to the measurements, we only use 5 threads to send and await responses. Furthermore, each thread waits for a response for 5 seconds before sending a new echo request, in which case we assume that the node has gone offline. Since our measurement application is not implemented as a distributed system, it takes about 72 hours to collect the data, which we highly expect to influence our measurements. Because of the churn on the network, which we present in Section 4.2, we can expect that a significant amount of the nodes will have gone offline during the 72 hours, which also showed to be the case during the measurements.

Out of the 10525 online nodes we obtained from the BitNodes API, 320 of them were Tor nodes, which we choose to ignore. Out of the remaining 10205 online nodes, we were able to send an

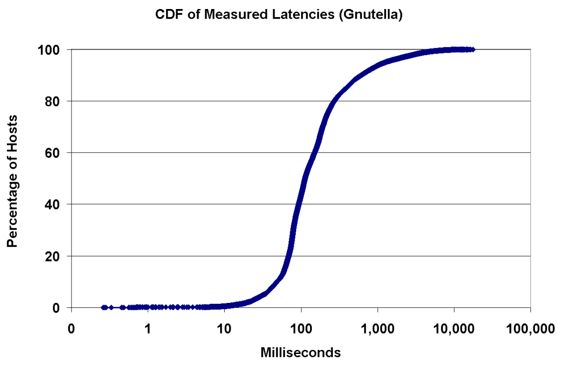
ICMP Echo Request and receive responds from 5501 of these nodes. As mentioned, this is very likely influenced by the network's churn, and we also note that its possible that some nodes have disabled ping responses. We were able to Bitcoin Ping 8177 out of the 10205 nodes. Again this is likely due to nodes having gone offline, and we do indeed find that some nodes accepts Bitcoin Pings while not accepting ICMP Echo Requests.

4.5.2 Latency compared to Gnutella P2P network

To get a perspective of how the Bitcoin network's ping is in relation to the Gnutella network's ping, we compare them as seen in Figure 4.5.



(a) Latency CDF on the Bitcoin network



(b) Latency CDF on the Gnutella network from Saroiu, Gummadi, and Gribble [39]

Figure 4.5: CDF charts of Bitcoin and Bitcoin latencies

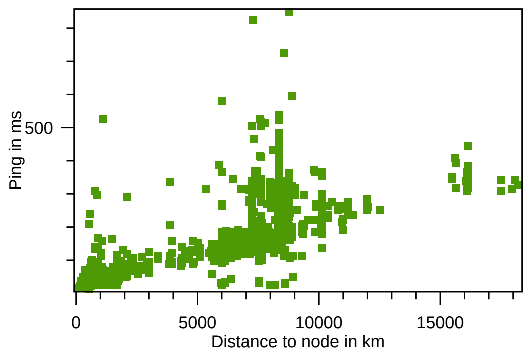
The Gnutella network's latency in Figure 4.5b is from Saroiu, Gummadi, and Gribble [39]. While this chart should show the same type of result, the authors do not mention how the results are measured and handled. Hence, we are unsure if these measurements shows an average per node, and we are unsure on how the latency measurements below 1 millisecond are deduced. Comparing Figure 4.5a and Figure 4.5b, we find that the Bitcoin network's latency is in general lower than the Gnutella network's latency. Additionally, we note that measurements above 5 seconds in the Bitcoin network's measurements are ignored, which might bias the Bitcoin network's latency to seem lower than the Gnutella network's latency.

4.5.3 Latency and Geographical Distance Correlation

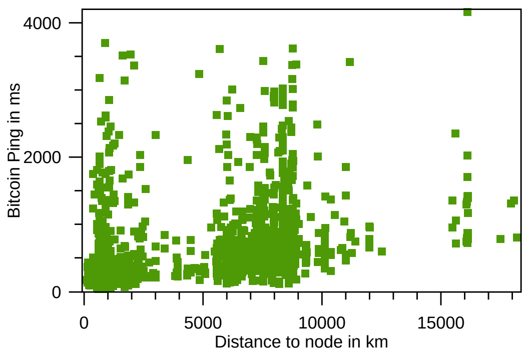
Similar to our ping measurements, we obtain a list of online nodes from the BitNodes API and in addition, we gather their latitude and longitude for each node's IP address using *GeoIP® Databases & Services: Industry Leading IP Intelligence* [40]. Using this, we are able to calculate the geographical distance and compare it to the latency and TTL in an attempt to find a correlation.

To calculate the geographical distance, we choose to use an implementation of Vincenty's formulae to calculate the distance between two points. We also consider using Haversine, which is often faster and might be more precise for distances below 0.8 km. However, for this task the performance is irrelevant, and we expect much longer distances between nodes, for which Vincenty is better suited [41].

In Figure 4.6, we present the relation between the geographical distance and Bitcoin Ping as well as ICMP Echo Request. In Figure 4.6a, we find a faint pattern emerging, which slightly indicates a linear correlation between ICMP Echo Requests and the geographical distance. Unfortunately, adding a linear trend line we can only obtain an R-squared value of 0.732, which we deem is too low to safely assume that there is an accurate correlation. This can be due to a lot of factors influencing the latency such as bandwidth, package hops, and computational latency.



(a) Relation between ICMP Echo Requests and geographical distance



(b) Relation between Bitcoin Ping and geographical distance

Figure 4.6: Relation between latency and geographical distance

Since we find a too weak correlation between latency and distance, we investigate if there is a correlation between geographic distance and TTL. We do this in Figure 4.7, where we unfortunately cannot discern a strong correlation between the geographical distance and TTL either. However, we do make an interesting observation from Figure 4.7, where we see that the measurements are aligned within three different groups of TTL measurements. This is very likely caused by the nodes using different operating systems, as we will elaborate on in the following section.

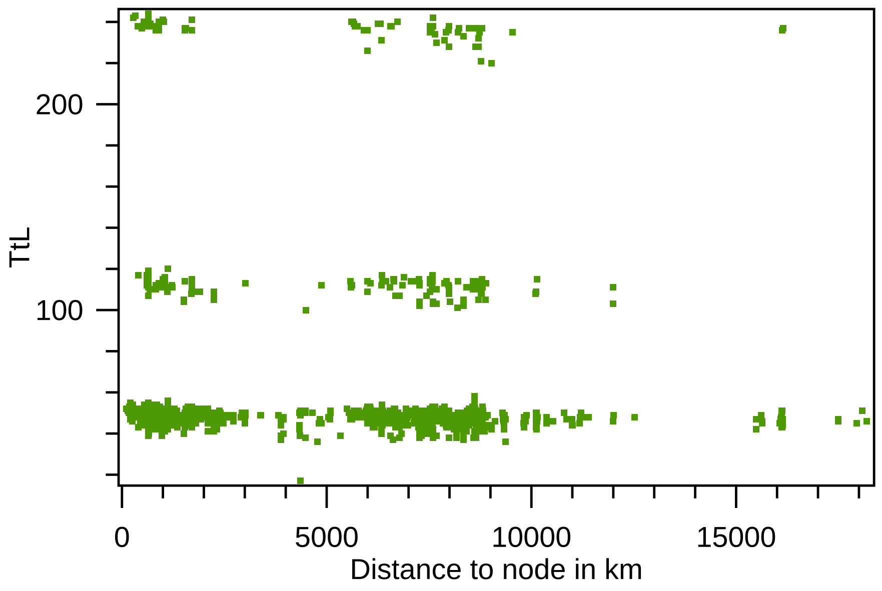


Figure 4.7: Relation between TTL and geographical distance.

Deducing Node Operating System from its TTL

According to a network consulting Engineer at Cisco [42], an IP packet will in the very worst case travel about 50 TTL. Presuming this statement is true, we can safely assume that none of the measurements in Figure 4.7 will fall under the wrong TTL group due to having traveled far on the internet, which would otherwise cause the TTL to decrease significantly. With this assumption, we can use our TTL measurements to deduce and categorize the Bitcoin network nodes' operating system [43], which we present as a distribution in Figure 4.8.

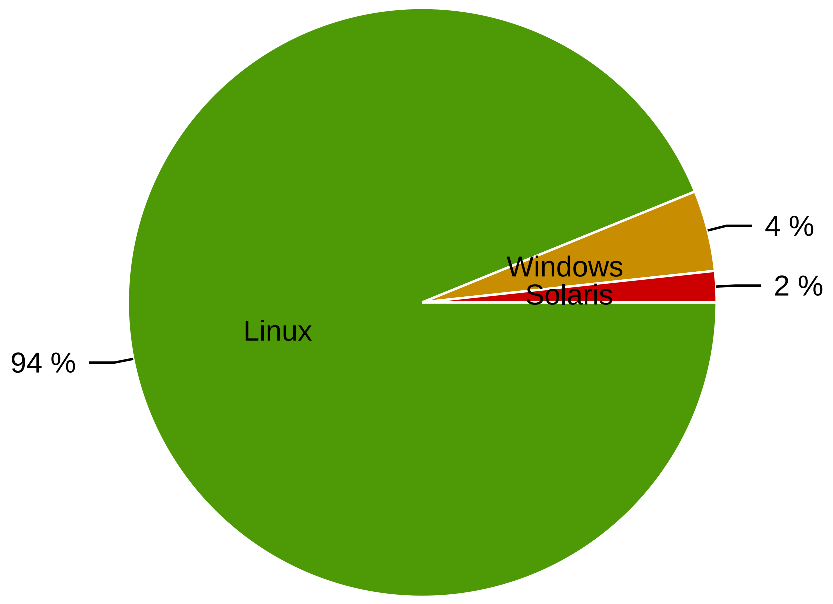


Figure 4.8: Distribution of Bitcoin network nodes' operating systems

While we were only able to gain TTL from 5501 out of the 10525 nodes in the Bitcoin network, this is to our knowledge the first insight of the distribution of operating systems in the Bitcoin network. Sites like `https://coin.dance`, `https://bitnodes.earn.com` and `https://blockchain.info` are some of the largest public sites with information of the Bitcoin network, but neither mention anything about operating systems. In Figure 4.8, we see that 94 % of Bitcoin nodes are running a UNIX-based operating system, 4 % are running Windows, and 2 % are running Solaris [43].

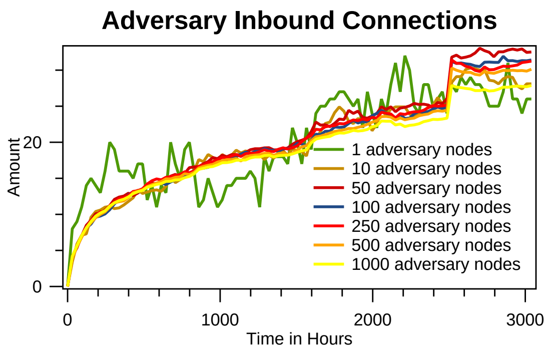
5 Results

In this chapter, we will present the results of the simulations. We will present results of how an adversary's amount of inbound connections changes over time by simply being online for a long period of time. Lastly, we present how a high amount of connections to an adversary can influence block propagation.

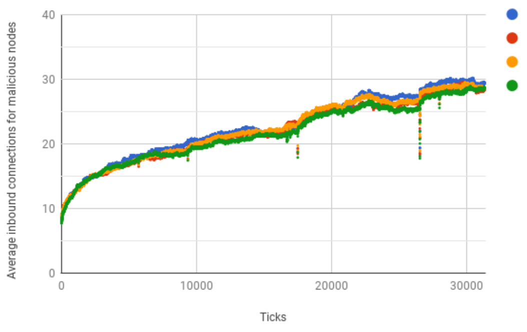
5.1 Inbound Connections Increase in the Simulator

We recall our hypothesis. A stable node with no limit of inbound connections will increase its amount of inbound connections over time, and will eventually posses an influence, which can be used for adversary setting. To assess this hypothesis, we first test how an adversary can increase its inbound connections over time by simply being online. The test is performed by running multiple simulations each with different amounts of adversary nodes, while measuring their average amount of inbound connections over time. We present these results in Figure 5.1a, where we see that the average amount of inbound connections slowly increases over time. In Figure 5.1a, the simulations implementing a single adversary show a much more unstable amount of inbound connections compared to the other simulation, which is due to the simple fact that this simulation calculates the average samples of a size of 1.

From these results, we see that an adversary node can on average be expected to obtain around 30 inbound connections in 3082 hours. We also see that the average amount of inbound connections between the 10 and 1000 adversary samples are very similar. Hence, within a period of 3082 hours we can expect that the total amount of inbound connections in relation to the amount of adversary nodes scales very close to linearly. We do note, that past the 1000 hours measurements, we start to see indications of a diminishing return, as the simulations with a larger amount of adversary nodes tend to get a slightly lower average of inbound connections. At the 2456 hours mark, the average of inbound connections varies with 2.085 between the 1000 and 10 adversary samples, which accounts for a 8.2 % to 9 % variance.



(a) Average amount of inbound connections to adversary nodes



(b) Average amount of inbound connections to adversary nodes of four simulations with different seeds from BitcoinChurn [2]

Figure 5.1: Average amount of inbound connections to 100 adversary nodes over time shown from this study's simulator and the BitcoinChurn simulator [2]

We compare the results in Figure 5.1a representing this study's results with the results from the

BitcoinChurn simulator [2] in Figure 5.1b. The 3082 hours of event-based simulation in Figure 5.1a roughly corresponds to the 30000 ticks of tick-based simulation in Figure 5.1b, as they both reach about 30 inbound connections by the end of their measurements, and they both reach about 25 to 26 inbound connections at the 2456 hours mark. We find the corresponding 2456 hours mark in Figure 5.1b by looking for the results at the spike around the 26000 ticks mark. We note that it is not trivial to map these results to each other due to their different perceptions of time, hence we have to use approximations like this.

5.2 Block Propagation Time

As stated in the ChurnAddr strategy, we hypothesize that an adversary having a high amount of inbound connections can threaten the block propagation in terms of hop and time efficiency as described in Section 2.2. To asses this hypothesis, we run multiple simulations of a scenario, where the adversary have 8 nodes in control. We use exactly 8 nodes, as this is also the default maximum limit for a Bitcoin Core application's outbound connections, which we target to gradually monopolize in our experiment.

We rephrase our hypothesis as a question: "How effective is an adversary in disrupting the network, given he has a certain acquired topology?" We attempt to answer this using a simulation with a given topology, which we do by forcefully creating outbound connections from the network's nodes to the adversary's nodes. We do this by forcing the connections in the starting stage of the simulations, hereafter we let the simulator run for 3082 hours of simulation-time. In addition, we give every node the knowledge of all other nodes in the network, such that we can assure that healthy connections are still being maintained in the network during the simulation.

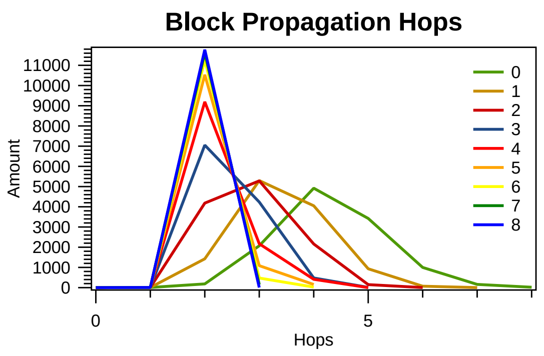
By the end of the normal simulation operation of 3082 hours, we enter a new stage of the simulator, where we pick 5 highly stable nodes as our 'victims'. Each victim is made highly stable to increase its inbound connections. In other words, the victims competes with the adversary on gaining inbound connections using the same passive churn strategy. We make two kinds of simulations:

1. Each victim broadcasts a block, which all nodes including the adversary propagates in the network
2. Each victim broadcasts a block, which all nodes excluding the adversary propagates in the network, hence the adversary attempts to attack the victims by disallowing the victims' block propagation.

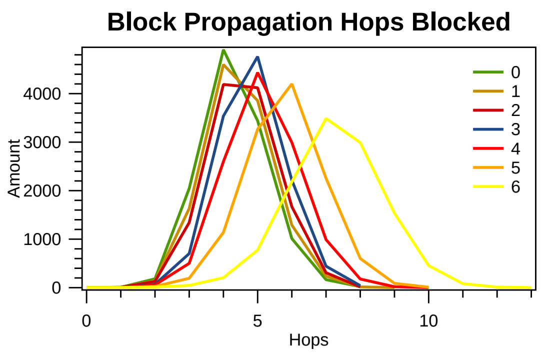
While performing these simulations, we take measurements of how many hops and how long time it takes to propagate the block to the full network. We also note here, that we do not consider the nodes' processing time and extra latency which is associated with propagating blocks. Hence, we should expect a better propagation time efficiency compared to real Bitcoin network measurements. The hop efficiency remains untouched by this. Our time propagation measurements should not indicate the actual Bitcoin network propagation time, but it should indicate how effective the adversary's disrupting behavior is in the perspective of our previously mentioned *simulation kinds*.

We present the hop efficiency in Figure 5.2, which shows the results from both simulation kinds, where we change the amount outbound connections each node has to an adversary. In Figure 5.2a, the adversary nodes participates in the block propagation, for which we find that the adversary actually helps the hop efficiency when the adversary obtain more connections. This is expected, as the adversary nodes functions as 'hubs' due to their high degree centrality, meaning that once the adversary nodes receives the block, they immediately propagate to a majority of the other nodes on the network. In Figure 5.2b, the adversary nodes refuse the victim's block, which corresponds to executing a blocking propagation attack in the final stage of the ChurnAddr strategy.

By looking at figure 5.3, we can see the same measurement, but including those simulation where we forced 7 and 8 connections to an adversary. By forcing 7 connections to an adversary, 237 nodes out of 11779 were actually eclipsed and did not receive the block at all. With all 8 connections to an adversary, the block was of course never propagated and all nodes were eclipsed.



(a) Adversaries do not refuse propagation



(b) Adversaries refuse propagation

Figure 5.2: Number of hops for a propagated block before it reaches a node. The legend indicates the amount of outbound connections to an adversary

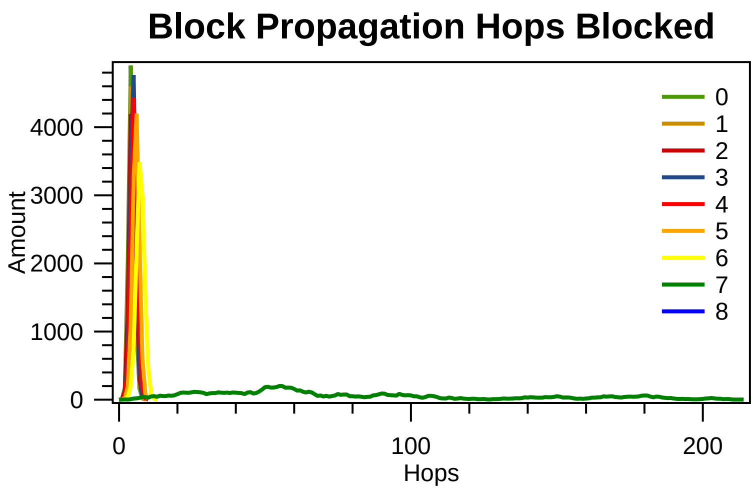


Figure 5.3: Number of hops for a propagated block before it reaches a node. Legends is the number of outbound connections to a adversary.

To get an alternative view of how this attack influences the network, the sum of hops which the nodes have to wait before receiving the block is presented in Figure 5.4. For each aforementioned scenario, the sum of hops is measured when propagating from the five different victims, which is marked with a green color. The average of these five measurements is shown by the black columns. The average amount of hops and percentage increase for the aforementioned scenarios are as follows:

- Scenario 0 connections: 50844,6 hops
- Scenario 1 connection: 51877,4 hops $\approx$ 2% increase
- Scenario 2 connections: 54554,4 hops $\approx$ 7.3% increase
- Scenario 3 connections: 57463,6 hops $\approx$ 13% increase
- Scenario 4 connections: 61419 hops $\approx$ 20.8% increase
- Scenario 5 connections: 67821,6 hops $\approx$ 33.39% increase
- Scenario 6 connections: 85472,6 hops $\approx$ 68.11% increase

From these results, we can expect that the time for each individual node to learn about a new block is increased as the number of connections to an adversary is increased. As the propagation time is increased this will also increase the fork rate on the network [8]. The exact time and increase fork rate depends on each hop's latency, the block size, and how fast each node validates the block.

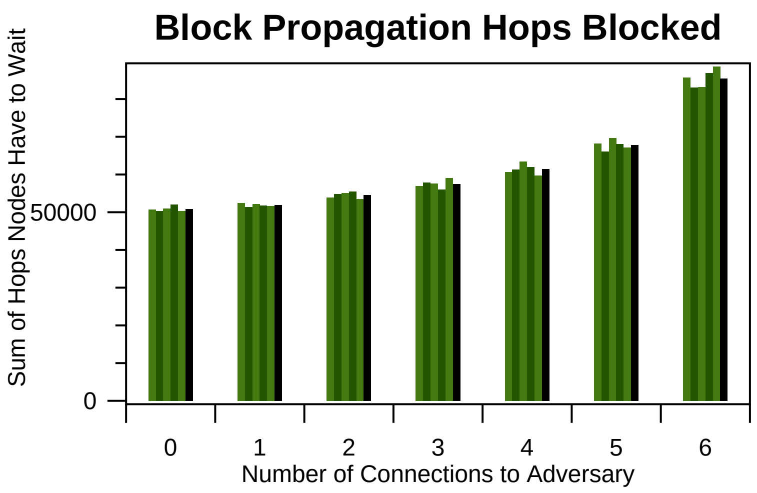


Figure 5.4: The sum of hops for a propagated block before it reaches a node

6 Discussion

6.1 From Tick-based to Event-based Simulation

In section 3.4, we chose an event-based simulator as the better alternative over the previous tick-based BitcoinChurn simulator [2]. The decision is based on the tick-based simulator having a flawed perception of time and poor capabilities in adopting latency features. Ultimately, we find that in order to fix these 'flaws' the tick-based simulator will be extremely slow, which made the option of reuse undesirable.

The downside of this decision is that the Bitcoin Core simulator implementation has to be largely rewritten as well. This did entail an additional workload, which was also expected. However, after thoroughly describing the Bitcoin Core application in pseudocode and reviewing it more comprehensively, we found missing details in the previous simulator, which would otherwise have to be implemented as well to achieve a more credible simulator. Hence, we evaluate this decision and still remark it as the better choice in relation to the workload and risks implied given our knowledge at that time.

After generating the event-based simulator's results, we evaluate and reflect these to the results of the tick-based simulator. Curiously, we find in Section 5.1 that both simulators generate somewhat similar results – they show the same tendency but with different perceptions of time. We speculate whether the tick-based simulator can be deemed inappropriate in terms of accuracy for our purpose, which may very well not be the case. This opens an ever-appearing question and challenge in modelling: "is the model accurate and correct?" to which it may be that both simulators replicate the desired behavior accurately and correctly. Obviously, a model will by its definition never achieve an exact replication of the modelled object, but it can replicate subsets of the objects behaviors, properties, et cetera with some degree of accuracy and correctness. What both the tick-based and event-based simulators have in common, is that they are closely constructed to replicate small parts of the behavior of the Bitcoin network, where the event-based simulator has additional modelling precision.

Unpleasantly, it may also be that both simulators are lacking accuracy and correctness, however, what we have offered in this study is a comprehensive presentation of the Bitcoin Core application in pseudocode, which acts as a conceptual model – an intermediate translation between the source object and the model implementation. This can be purposed as a tool of verification, in which the reader can verify or criticize aspects of the model, to which the accuracy of the simulator can be evaluated.

6.2 Alternative Use of the Simulator

As the reader may have noticed, the simulator is used for more than generating the final results. This study has followed an evolution-based development approach, where we use prototypes and intermediate versions of the simulator to assess certain behaviors of the simulated Bitcoin network. We do this mostly in relation with performance optimization, where we use diagnostics to measure the performance of separate Bitcoin Core features, for which we carefully optimize the features that require most attention. Moreover, we use intermediate versions of the simulator to assess whether certain Bitcoin Core features are used more often than other, for which if a feature is rarely or never used, we can remove or simplify it with a less concern of accuracy loss. Hence, the simulator serves more than a result generator.

We design the 'simulator engine' with this in mind, for which the simulator engine has achieved a generic, lightweight, and easily maintainable structure. As a fortunate side-effect, the simulator can easily be implemented for similar P2P network studies, which are focused on topology generation, or simply other Bitcoin network topology studies. What the simulator offers is the ability to perform long-term measurements and predictions in a relatively short time.

We find that simulating the topology and address propagation is in itself a heavily CPU intensive task for the simulator, which makes the idea of having a single simulator that can handle the complete logic and behavior of the Bitcoin Core application impractical. We do note however, that by segmenting the implementation of the Bitcoin Core application, it might be possible to extend the simulator to support all logic while letting the simulator user enable or disable the logic desired to be simulated such that the simulator keeps its strength in performance. We believe that such a simulator is beneficial for the development of the Bitcoin Core application, and that this study's simulator could be a potential candidate for this task, as we have a large part of the topology logic implemented.

7 Conclusion

We recall the problem statement:

> "How can the Bitcoin network's address propagation and churn be exploited to increase the amount of inbound connections to an adversary over time, and can this exploit ultimately enable adversary actions?"

We approach to answer this statement by constructing an event-based and data-driven simulator, which features an accurate perception of time. The simulator's Bitcoin Core implementation is capable of sending and receiving all 6 Bitcoin Core messages, which we identify as essential for the Bitcoin network's topology generation.

We construct the simulator, by reviewing and reverse engineering the Bitcoin Core source code, where we analyze 5 threads with over 30 functions. During this process, we create a conceptual model consisting of pseudocode of the 4 most influential functions in relation to address propagation and topology generation, for which the conceptual model can be used as a tool for verifying the simulator.

With a highly accurate implementation of the Bitcoin Core behavior including the address flooding protocol, the simulator obtains a run-time which is 4.82 times faster than the simulated time while simulating a churn-enabled network of 126.217 unique nodes. We obtain a high simulator accuracy from injecting the Bitcoin network's actual churn rate and latency into the simulator and by thoroughly reviewing the Bitcoin Core source code.

We assess the aforementioned problem statement by dividing it into three main questions, for which we use our simulator to answer each question in details.

1. How can the Bitcoin network's address propagation be exploited to increase the amount of inbound connections?
2. How can the Bitcoin network's churn be exploited to increase the amount of inbound connections?
3. Can the ChurnAddr strategy, which coveres the aforementioned exploits, enable adversary actions?

Unfortunately, due to removing the address propagation feature in the simulator in favor of run-time performance, we are not able to conclude if the Bitcoin network's address propagation can be exploited by the ChurnAddr strategy to increase the amount of inbound connections.

From the simulators results, we conclude that churn can be exploited to increase the number of inbound connections to an adversary over time. Furthermore, we show that increasing the amount of adversary nodes only entail a slight long-term diminishing return, which makes the strategy's effectiveness scale close to linearly as more adversary nodes are included.

While we disprove the applicability of performing an Eclipse attack using the churn exploit, the network does indeed become more susceptible to attacks targeting the hop and time efficiency of block and transaction propagation. Additionally, the network likely weakens in terms of increased fork rates, as a longer propagation time will result in this [8].

Curiously, the results also show that if an adversary choose to help the network by eagerly forwarding blocks, the average block propagation time is decreased strengthening the network.

We therefore conclude that churn do increase the number of inbound connections to stable nodes, and that a high amount of inbound connections gives the possibility to slow down block propagation hereby harming the network.

Bibliography

[1] *Regulation of Bitcoin in Selected Jurisdictions*. Jan. 2014. URL: https://www.loc.gov/law/help/bitcoin-survey/.

[2] Alex Grøndahl Frie and Rune Willum Larsen. *The Churn, the Bitcoin, and the Monopoly*. 2018.

[3] Ethan Heilman et al. "Eclipse Attacks on Bitcoin's Peer-to-Peer Network." In: *USENIX Security Symposium*. 2015, pp. 129–144. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-heilman.pdf.

[4] *Bitnodes*. Jan. 2017. URL: https://bitnodes.earn.com/.

[5] *Bitcoin Ticker*. Jan. 2018. URL: http://bitcointicker.co/.

[6] Sean Williams. *How Volatile Is Bitcoin?* May 2018. URL: https://www.fool.com/investing/2018/05/10/how-volatile-is-bitcoin.aspx.

[7] Andrew Miller et al. "Discovering bitcoin's public topology and influential nodes". In: *et al.* (2015). URL: https://allquantor.at/blockchainbib/pdf/miller2015topology.pdf.

[8] 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, pp. 1–10.

[9] *Bitcoin Core Message Types Reference*. Mar. 2018. URL: https://en.bitcoin.it/wiki/Protocol_documentation#Message_types.

[10] *BitTorrent Protocol Specification v2, DRAFT*. May 2017. URL: http://bittorrent.org/beps/bep_0052.html.

[11] *Gnutella Protocol Specification 0.6, DRAFT*. June 2002. URL: http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.

[12] *Bitcoin Core Developer Reference*. May 2018. URL: https://bitcoin.org/en/developer-reference.

[13] Andrew Miller. *Understanding Bitcoin's Network Topology*. 2015. URL: https://scalingbitcoin.org/montreal2015/presentations/Day1/10-Andrew-Miller-Coinscope_Shadow-ScalingBitcoin.pdf.

[14] Andrew Miller. *coinscope-andrew-miller Transcript*. 2015. URL: https://diyhpl.us/wiki/transcripts/scalingbitcoin/coinscope-andrew-miller/.

[15] Linton C Freeman, Stephen P Borgatti, and Douglas R White. "Centrality in valued graphs: A measure of betweenness based on network flow". In: *Social networks* 13.2 (1991), pp. 141–154.

[16] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[17] Philip Koshy, Diana Koshy, and Patrick McDaniel. "An analysis of anonymity in bitcoin using p2p network traffic". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2014, pp. 469–485. URL: https://www.ifca.ai/fc14/papers/fc14_submission_71.pdf.

[18] *Bitnodes*. May 2018. URL: https://en.bitcoin.it/wiki/Testnet.

[19] Zak Cole. *Network simulation or emulation?* 2018. URL: https://www.networkworld.com/article/3227076/lan-wan/network-simulation-or-emulation.html.

[20] DifferenceBetween team. *Difference Between Emulator and Simulator*. 2018. URL: http://www.differencebetween.com/difference-between-emulator-and-vs-simulator/.

[21] stackoverflow community. *Simulator or Emulator? What is the difference?* 2018. URL: https://stackoverflow.com/questions/1584617/simulator-or-emulator-what-is-the-difference.

[22] StackExchange Software Engineering community. *What's the difference between simulation and emulation*. 2018. URL: https://softwareengineering.stackexchange.com/questions/134746/whats-the-difference-between-simulation-and-emulation.

[23] Quora community. *What are the differences between simulation and emulation?* 2018. URL: https://www.quora.com/What-are-the-differences-between-simulation-and-emulation.

[24] Fuchun J Lin, PM Chu, and Ming T Liu. "Protocol verification using reachability analysis: the state space explosion problem and relief strategies". In: *ACM SIGCOMM Computer Communication Review*. Vol. 17. 5. ACM. 1987, pp. 126–135. URL: %5Curl%7Bhttps://s3.amazonaws.com/academia.edu.documents/30835963/Forum18.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1528024603&Signature=lzHenzqWXhjVcfyGCyXjkC3L18w%5C%3D&response-content-disposition=inline%5C%3B%5C%20filename%5C%3DDelay_insensitive_chip-to-chip_interconn.pdf#page=44%7D.

[25] *Visual Boy Advance*. 2018. URL: https://github.com/visualboyadvance-m/visualboyadvance-m.

[26] *DOSBox*. 2018. URL: https://www.dosbox.com/.

[27] Kaylash Chaudhary et al. "Modeling and verification of the bitcoin protocol". In: *arXiv preprint arXiv:1511.04173* (2015). URL: https://arxiv.org/pdf/1511.04173.pdf.

[28] Andrew Miller and Rob Jansen. "Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-threaded Applications." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 469. URL: http://soc1024.ece.illinois.edu/shadow-bitcoin.pdf.

[29] Simon Mulser, Andreas Kern, and Aljosha Judmayer. *Simcoin*. 2015. URL: https://github.com/sbaresearch/simcoin.

[30] Mat Holroyd. *BitCoin Server Emulator*. 2012. URL: https://github.com/matholroyd/bitcoin-server-emulator.

[31] "Shadow-Bitcoin". In: (2018). URL: https://github.com/amiller/bitcoin.

[32] Louis G Birta and Gilbert Arbez. *Modelling and simulation*. Springer, 2013. URL: https://campusvirtual.univalle.edu.co/moodle/pluginfile.php/1167280/mod_resource/content/1/Modelling_and_Simulation_Book.PDF.

[33] Averill M Law, W David Kelton, and W David Kelton. *Simulation modeling and analysis*. Vol. 3. McGraw-Hill New York, 2007.

[34] Norm Matloff. "Introduction to discrete-event simulation and the simpy language". In: *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* 2.2009 (2008). URL: https://www.informs-sim.org/wsc98papers/050.PDF.

[35] Rob Jansen and Nicholas Hopper. "Shadow: Running Tor in a Box for Accurate and Efficient Experimentation". In: *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*. Internet Society, Feb. 2012.

[36] *Bitcoin Core*. Jan. 2007. URL: https://bitcoin.org/en/bitcoin-core/.

[37] *.NET Dictionary*. May 2018. URL: https://msdn.microsoft.com/en-us/library/xfhwa508.aspx.

[38] 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, pp. 15–29. URL: https://arxiv.org/pdf/1405.7418.pdf.

[39] Stefan Saroiu, P Krishna Gummadi, and Steven D Gribble. "Measurement study of peer-to-peer file sharing systems". In: *Multimedia Computing and Networking 2002*. Vol. 4673. International Society for Optics and Photonics. 2001, pp. 156–171.

[40] *GeoIP® Databases & Services: Industry Leading IP Intelligence*. May 2018. URL: https://www.maxmind.com/en/geoip2-services-and-databases.

[41] Alex Grøndahl Frie et al. *aSTEP Outdoor Project*. 2016.

[42] Aditya Ghanekar. *Why is the maximum TTL value in an IP header 255?* 2017. URL: https://www.quora.com/Why-is-the-maximum-TTL-value-in-an-IP-header-255.

[43] *Default TTL (Time To Live) Values of Different OS*. Apr. 2014. URL: `https://subinsb.com/default-device-ttl-values/`.