UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Low-Resource Eclipse Attacks on Alternative Ethereum Clients

Leichtgewichtige Eclipse Angriffe auf Alternative Ethereum Clients

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Gordon Dahlke

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Lübeck, den 9. Oktober 2019

Abstract

In this thesis, we explore the possibilities of eclipse attacks on Parity Ethereum, a popular client for the Ethereum network. Eclipse attacks separate a network participant from the rest of the network, making him vulnerable to further attacks. We achieve an eclipse by overloading the victim with attacker connections, rendering it unable to fetch information from other network participants. This results in us being the victim's only source of information. Next, we show that table poisoning attacks are possible on Parity Ethereum: We insert our own adversarial network information into the network tables of victim clients in an efficient and controlled manner. Besides providing detailed information about the circumstances under which these attacks are successful, we also present uses for eclipse attacks, taking into account the amount of cryptocurrency controlled by the adversary.

Kurzreferat

In dieser Arbeit analysieren wir Parity Ethereum, einen bekannten Ethereum Client, auf Schwachstellen, die Eclipse [1] Angriffe ermöglichen. Eclipse Angriffe trennen einen Netzwerkteilnehmer von dem Rest des Netzwerks und dienen so als Grundlage für weitere Angriffe. Wir erreichen eine *Eclipse* durch Überlastung des Opfer-Clients mit Verbindungen, die von Angreifern ausgehen. So kann sich der angegriffene Client keine Informationen mehr von anderen Netzwerkteilnehmern einholen und die Angreifer sind seine einzige Informationsquelle. Danach zeigen wir, dass sogenannte Table Poisoning Angriffe auf Parity Ethereum Clients möglich sind: Wir fügen unsere eigenen Verbindungsdaten effizient und kontrolliert in die Netzwerktabelle der Opfer-Clients ein. Außer detaillierten Informationen über die Umstände, die solche Angriffe ermöglichen, zeigen wir auch, wofür Eclipse Angriffe in Abhängigkeit von der Menge Kryptowährung, die der Angreifer kontrolliert, genutzt werden können.

[1] deutsch: Finsternis

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 9. Oktober 2019

Acknowledgements

We [2] would like to thank Professor Eisenbarth, Ida Bruhns, Jan Wichelmann, Thore Tiemann and Gorka Irazoqui for their great support during this project and for making the security-related courses of our studies not only informative, but also very interesting. We are also very grateful for our parents' support throughout our whole lives. Finally, we would like to thank the Docker developers, the Ethereum founders and the Parity Ethereum developers as well as Yuval Marcus, Ethan Heilman and Sharon Goldberg [26] for making this thesis possible.

We included a list of useful software we appreciate in *Appendix D*.

[2]Please refer to the title page for information on who "we" are.

Contents

1 Introduction

In this thesis we will analyze the Parity Ethereum client [3] for vulnerabilities allowing low-resource eclipse attacks. Parity Ethereum is a client for the Ethereum network, which in turn provides the Ethereum cryptocurrency. It claims to be "the fastest and most advanced Ethereum client", "built for mission-critical use" serving "miners, service providers, and exchanges" [22]. Because of these claims and its popularity, it will be interesting to analyze Parity Ethereum for critical security vulnerabilities regarding eclipse attacks. An eclipse attack separates a network participant (the victim) from other network participants, making the attacker the only *peer* of the victim, allowing the attacker to manipulate the way the victim perceives the network and information stored and exchanged therein. Here is an example of a practical *mis*use of eclipse attacks:

1. Merchant Alice is selling physical apples for Ether.

2. Eve uses 50 own Ethereum clients to eclipse Alice by monopolizing all connections of Alice's Ethereum client. (Now Eve has control over Alice's view of upcoming transactions.)

3. Eve sends Alice five Ether, while sending those exact same five Ether to herself again (double-spending).

4. Eve broadcasts the first transaction to Alice, while broadcasting the second transaction to the regular Ethereum network.

5. The network accepts and validates Eve's second transaction, while Alice only sees Eve's first transaction and hands five apples to Eve.

6. Eve leaves with the five apples.

7. If Alice later uneclipses herself and broadcasts Eve's first transaction to the Ethereum network, it will be rejected because Eve's second transaction is already validated.

8. Now Eve has both the Ether and the apples, while Alice has no Ether and no apples.

[3] `https://github.com/paritytech/parity-ethereum`

1.1 Related Work

This thesis is inspired by the 2018 paper "*Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network*" by *Marcus, Heilman and Goldberg* [26]. In the paper, low-resource eclipse attacks on the *Geth* (Golang Ethereum) client [4] (versions prior to 1.8) were analyzed. Three eclipse attack vectors were identified:

1. Connection Monopolization: An attacker can quickly fill up all possible connections of a restarting node with many node *ids*, using only two nodes himself.
2. Table Owning (or Table Poisoning): An adversary can use special node ids to make a restarting victim connect to him on its own.
3. Time Manipulation: Manipulation of a victim's local clock via attacks on the Network Time Protocol (NTP) could make it eclipse itself.

These findings were then shared with the Geth developers, who incorporated some eclipse attack countermeasures in Geth version 1.8. Some exploitable design decisions in the Geth client remain unchanged. Groundwork for current eclipse attacks and more related papers follow in the next subsections.

1.1.1 General Peer-to-Peer (P2P) Networks

Douceur presents "Sybil attacks" "in which a small number of entities counterfeit multiple identities so as to compromise a disproportionate share" of a p2p system in his 2002 paper [6]. The possible impact of Sybil or eclipse attacks on Kademlia[49]-based networks [5] is analyzed in 2007 by *Steiner et al.* [42] and 2009 by *Kohnen et al.* [25]. In their 2002 paper [41], *Sit and Morris* categorize various attacks on p2p distributed hash tables (DHTs) including routing attacks and attacks causing partitioning. In 2006, *Singh et al.* stress the importance of eclipse attack countermeasures, and propose an "efficient" countermeasure using "anonymous auditing" [40]. *Castro et al.* describe partitioning attacks on structured p2p networks in 2002 [4]. They show that custom node id selection or bulk random node id acquisition make such attacks easier / possible. *Urdaneta et al.* publish a *Survey of DHT Security Techniques* [45] against Sybil and eclipse attacks (2011).

1.1.2 Bitcoin

A strategy called *selfish mining* (keeping mined blocks secret at first to maximize profit beyond the fair share) was found to be profitable in 2013 by *Eyal and Sirer* [17]. *Gervais et al.*

[4]`https://github.com/ethereum/go-ethereum`

[5]The Ethereum network is based on Kademlia.

show that a selfish mining adversary "which commands less than 34% of the computing power in the network" could (under certain circumstances) "control the entire network" using attacks on block delivery [20] (2015). In the same year *Heilman et al.* publish a paper [23] on Bitcoin eclipse attacks and countermeasures. *Nayak et al.* describe further attacks combining mining attacks with eclipse attacks in 2016 [28]. In [1] (2016), *Apostolaki et al.* publish partitioning attacks on the Bitcoin network that are based on "hijacks" of the internet routing protocol BGP [47].

1.1.3 Ethereum

In early 2018, Cisco Talos published a vulnerability [43] in Aleth [6] (a C++ Ethereum client). This vulnerability allowed malformed JSON requests to crash the client, possibly allowing an eclipse attack upon restart. Eclipse attacks on Ethereum along with countermeasures are described in the 2016 tech. report by *Wüst and Gervais* [50]. Double-spending attacks utilizing mining attacks and eclipse attacks are published by *Natoli and Gramoli* in 2017 [27]. *Gencer et al.* analyze the "decentralization" of Bitcoin and Ethereum in [19] (2018).

1.2 Room for Research

One important question that is not covered by previous work is the following:
Are low-resource eclipse attacks on alternative [7] Ethereum clients possible?
To answer this question, we want to

- develop an easy-to-use / easy-to-deploy eclipse attack setup (see section 1.3),
- and explore the possibilities of low-resource eclipse attacks on current versions of alternative Ethereum clients.

Besides *Geth*, there are other Ethereum clients, for example *Parity Ethereum* [8] and *Aleth* [9]. These may also have vulnerabilities that allow low-resource eclipse attacks. (For example, one open GitHub issue for Parity Ethereum [21, open as of 6th Oct. 2019] references the paper mentioned above.) Because Parity Ethereum is the only alternative client with a non-negligible usage share (Go Ethereum 75.4% and Parity Ethereum 23.5% [10]), analyzing it for the *connection monopolization* vulnerability will be our first goal.

[6] `https://github.com/ethereum/aleth`
[7] unofficial / non-reference
[8] `https://github.com/paritytech/parity-ethereum`
[9] `https://github.com/ethereum/aleth`
[10] on 27th March 2019 [15] / for current client usage stats see `https://etherscan.io/nodetracker`

1.3 Eclipse Attack Setup

To further familiarize ourselves and others with eclipse attacks, we developed the model depicted on figure 1.2. Without the eclipse attack (figure 1.1), the victim (III) would fill up its connection slots (II) with connections to regular Ethereum nodes (I). But during an eclipse attack (figure 1.2), the attacker fills all victim connection slots with connections to attacker nodes (IV). If the victim is only connected to attacker nodes, the attacker can manipulate the victim's view of the blockchain (and transactions therein) to his liking.

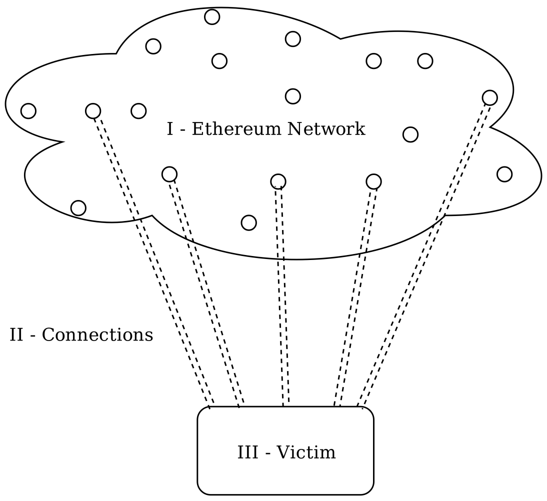

Figure 1.1: Standard Scenario

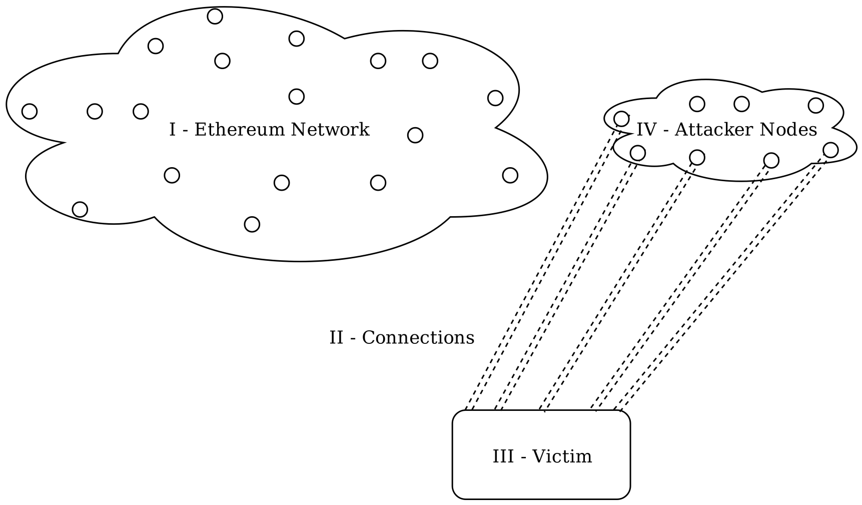

Figure 1.2: Eclipse Attack Scenario

We will realize this model with the following Parity Ethereum Docker containers running as light clients [11] on our local machine (see section 1.4 for details regarding Docker):

- one victim node (III)
- n attacker nodes (IV)

After a vulnerability is found, the nodes will be separated to multiple machines to validate the vulnerability with real network latency. Our usage of light clients rules out active manipulations of the victim's view of the blockchain, because light client attacker nodes can only deny the victim access to information about the blockchain, while they can not *serve* the victim relevant information (see section 4.1).

1.4 Performance with Docker

Docker [12] is a virtualization / containerization tool that allows simultaneous operation of multiple instances of Ethereum clients on a single host, although these clients are designed to operate *one at a time*. To test whether the scheme described above is realizable with many nodes, we used *Docker-Compose* [13] to start 100 isolated [14] instances of Parity Ethereum on our local machine with an 4-core i5 and 8 GB of RAM running Ubuntu 18.04. The processor and RAM were not maxed out during idle operation of these 100 nodes. This suggests that our testing scheme is indeed realizable with enough nodes to simulate a realistic environment (even on a single machine).

1.5 Project Goals

1. Create a flexible, easy-to-use Docker setup (see section 1.3).
2. Attack a Parity Ethereum victim client utilizing the Docker setup in a realistic environment (with victim and attacker clients on separate machines).
3. Test Parity Ethereum for other vulnerabilities like table poisoning.

[11] a faster synchronization mechanism [33]
[12] `https://www.docker.com/`
[13] `https://docs.docker.com/compose/`
[14] Each client was configured to not communicate with any other client.

2 Classical Uses of Eclipse Attacks

This chapter spotlights attacks that are enhanced by a victim being eclipsed, although the attacks can be carried out on a non-eclipsed victim with a smaller success probability.

2.1 Double Spending

A successful eclipse attack will allow the attacker to perform a double spending attack of at least the amount of ether he *actually* controls. If the attacker controls 100 ether and buys a car for 100 ether (double spending those to himself), the car salesman now thinks that he received 100 ether, and the attacker has no ether left. To be able to spend his ether again during the same eclipse attack, the attacker would have to convince the salesman that the double spending transaction was accepted, while the original attacker$\rightarrow$salesman transaction was declined by the network. This would open up two possibilities: The salesman could either realize that he was betrayed, ending or even reverting the attack, or not realize this, and the attacker could repeat the aforementioned steps over and over, accumulating more and more goods, independent of his ether credit.

2.2 Transaction Malleability Attacks on Bitcoin

These are the attacks associated with the Bitcoin exchange *Mt. Gox* [5].
Signed Bitcoin transactions can be altered, creating equivalent transactions with different hashes [5]. This makes the following attack possible (notice how the roles of the vendor and client are switched): The attacker sells a car to the victim, which in turn publishes a transaction transferring 100 ether from the victim to the attacker. The attacker publishes a modified version of this transaction, resulting in a different transaction hash, and manages to get the network to accept his modified version, rejecting the original transaction [15]. The victim checks the hash of the original transaction and sees that it was rejected. This opens up two possibilities: Either the victim simply rebroadcasts the original transaction, preventing exploitation by the attacker, or the victim creates another, non-equivalent transaction to pay the attacker and actually transfers different funds. The second possibility would result in the attacker being paid two times (or more times, if the attacker successfully repeats the attack).

[15]This step is trivial if the victim is eclipsed and broadcasts all transactions through (only to) the attacker.

2.3 Transaction Malleability Attacks on Ethereum

This section is shorter than the last one, because a fundamentally different transaction design removes the possibility of such attacks on Ethereum [44]. But Ethereum users (just like Bitcoin users) may be vulnerable to even simpler attacks described in the following section.

2.4 Transaction Drop Attacks on Ethereum

They work as follows: The attacker sells goods to an eclipsed victim, which in turn publishes a transaction paying the attacker 100 ether. The attacker shares the transaction with the main network, but convinces the victim that its transaction was dropped. Now, a normal victim would rebroadcast the first transaction, giving the attacker no advantage. But if the attacker tricks the victim into publishing another, non-equivalent transaction, the attacker can also "redeem" the second transaction with the main network, and get paid two (or more) times.

3 Usefulness of Eclipse Attacks for a Fundless Attacker

This chapter discusses how an attacker without ether credit, and thus unable to perform classical attacks out-of-the-box, may still profit from eclipse attacks. This is made possible by an exploitation of the difficulty adjustment algorithm, which will be explained below.

Ethereum is still [16] using a Proof of Work consensus algorithm, as opposed to a Proof of Stake algorithm. In simple terms: **Proof of Work** uses a lot [17] of computational power to validate transactions and generate reward currency (proportional to computational power). In **Proof of Stake**, participants can *vote with their money* on what they consider valid or invalid transactions, and thereby also generate rewards (proportional to staked currency).

3.1 Proof of Work

The work of a so-called Ethereum miner consists of the following (simplified) steps:

1. Check whether some new transactions are valid or invalid.

2. Try to find a new block including those transactions. The chance of finding such a block is $\frac{1}{\text{difficulty}}$ per try. With more computational power, a miner can execute more tries per time unit.

3. Publish the block and claim the reward if no other miner was faster. Start over with step 1 again.

The difficulty from step 2 is tweaked for each new block to achieve a desired average block finding time. In June 2019, for example, the average block finding time was about 13.2 seconds [14]. If some miners stopped trying to find new blocks (less computational power in the whole network, less total tries), this time would go up (less total tries, more time needed to find a block with a given difficulty), but eventually settle down again (because the difficulty would decline as well) [18]. Simplified and taken to the extreme:

[16]estimatedly until January 2020 [10]

[17]According to Digiconomist, Bitcoin and Ethereum Proof of Work calculations combined consume more electricity than countries like Chile, the Philippines or Austria. [11]

[18]Go Ethereum's implementation of this algorithm (note that the "difficulty bomb" was ignored in our simplified explanation): `https://github.com/ethereum/go-ethereum/blob/4c90efdf57ce87edf0d855c8cec10525875a6ab1/consensus/ethash/consensus.go`

If all except a single miner would stop their work, this single miner would eventually find a new block every 13.2 seconds on average.

3.2 Can an Eclipse Attacker Print Money?

If an attacker with no ether credit manages to eclipse a victim, this may not be of much use for him because even for double-spending, the attacker needs to have some ether to double-spend. To test whether an attacker could exploit the difficulty adjustment from the last section and efficiently mine blocks during an eclipse attack, we ran Python simulations based on Go Ethereum's aforementioned implementation (uncle blocks [19] can be ignored in this single-miner scenario). All simulations are based on real values collected between June 11 and 13, 2019. Each scenario was simulated at least ten times and the median 20% (regarding total mined reward) of those simulations were selected. From these selected simulations, we chose the one with the smallest sum of squared block time deviation from each found block to the next one. These simulations assume that mined reward ether can be spent instantly. While Ethereum does not have a *maturity* rule like bitcoin [2], Ethereum clients may wait for n additional blocks on top of block x before they accept spending the coinbase transaction of block x. A value of 10 for n is mentioned by *Vitalik Buterin* [3].

3.2.1 Three Year Eclipse

The first simulation illustrates that an attack by a single person without ether credit will most likely be impractical: Even if the attacker owns 10 AMD Radeon RX 570 graphics cards (worth approx. USD 1340 [7] [16]), and manages to eclipse a victim for three whole years, he may have only mined approx. USD 6100 during this time. The mining difficulty and his rewards are plotted in figure 3.1.

3.2.2 Mining Pools

Because solo mining [20] is greatly based on luck, miners often unite their computational power to find blocks and split rewards on success [21]. According to [24, 11th June 2019], one such pool had a hash rate of 44.89 terahashes per second (hashes per second $\widehat{=}$ tries per second). If someone used this much computational power to mine blocks during an

[19] Uncle blocks are blocks that were not chosen to be part of the main chain (i.e. because a competing block was published faster), but are instead acknowledged by future miners and thus partially rewarded [37].

[20] solo mining means mining alone / not as part of a pool

[21] Note that this does not make mining more lucrative with time going against infinity, but it rather makes mining income more steady.

eclipse attack, he could *print* ether worth thousands to millions of USD. This is summarized in table 3.1 along with referenced figures.

3.2.3 Power Costs

While some countries have relatively low electricity costs compared to Germany, electricity costs have to be taken into account for eclipse mining calculations. Electricity costs for the preceding scenarios are displayed in table 3.2 with the following assumptions for the power usage calculation: The mining pool consists of RX 570s only, using 150 watts each [35]. Additional power consumption from computers or cooling is ignored. USD per kWh values are taken from `https://www.globalpetrolprices.com/electricity_prices/` [8, June 2018]. For the Reward/Cost-Column, it is assumed that all mined Ether can be exchanged for goods during the eclipse attack.
The Reward/Cost-Column in table 3.2 illustrates the great impact of electricity costs on the economic benefit of eclipse attacks. While a three-year eclipse attack with 10 RX 570s would result in a financial loss for the attacker in Germany and the UK, it would lead to a financial gain in the USA, China and Egypt. A 24-hour eclipse attack using a mining pool consisting of RX 570s would provide a small financial gain for an attacker located in Germany, while an attacker located in Egypt would generate Ether worth more than 17 times the electricity cost.

Table 3.1: Block Rewards Generated During an Eclipse Attack (Pool Hash Power)

Eclipse Attack Duration	Mining Reward	Fig.
1 hour	45.000 USD	3.2
3 hours	150.000 USD	3.3
6 hours	370.000 USD	3.4
12 hours	980.000 USD	3.5
24 hours	2.600.000 USD	3.6

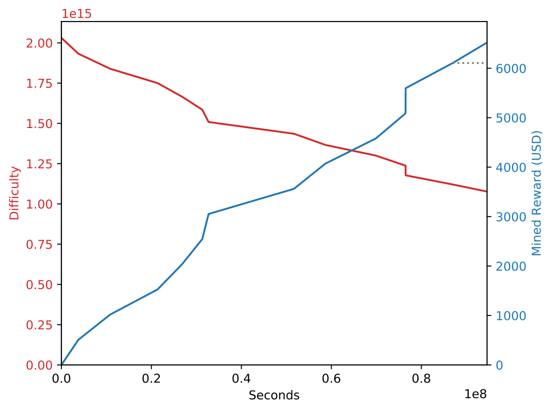

Figure 3.1: Three Years of Eclipse Solo Mining with Ten Budget Graphics Cards

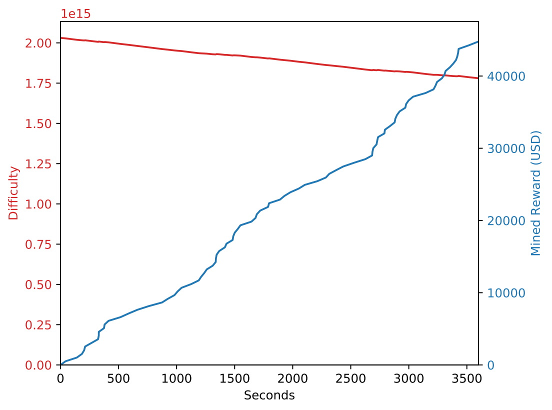

Figure 3.2: One Hour of Eclipse Mining with Pool Hash Power

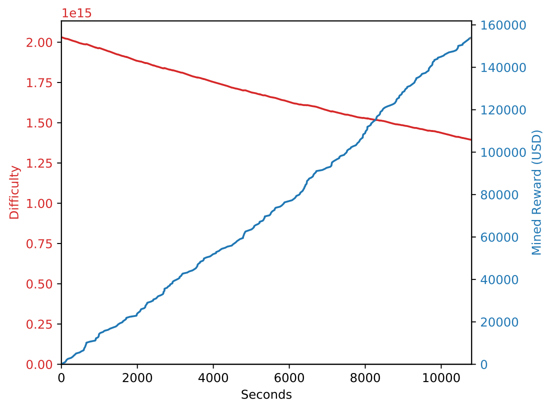

Figure 3.3: Three Hours of Eclipse Mining with Pool Hash Power

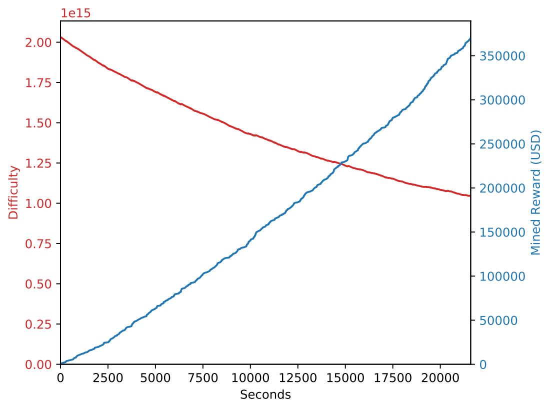

Figure 3.4: Six Hours of Eclipse Mining with Pool Hash Power

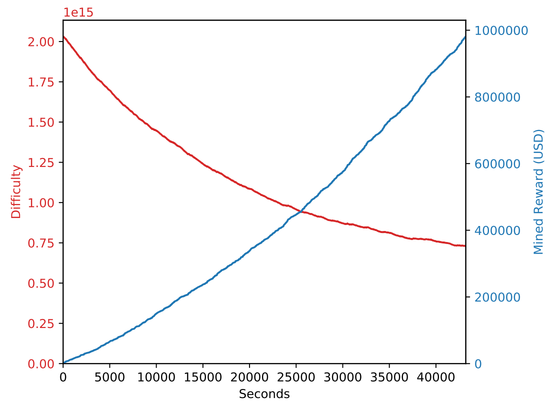

Figure 3.5: Twelve Hours of Eclipse Mining with Pool Hash Power

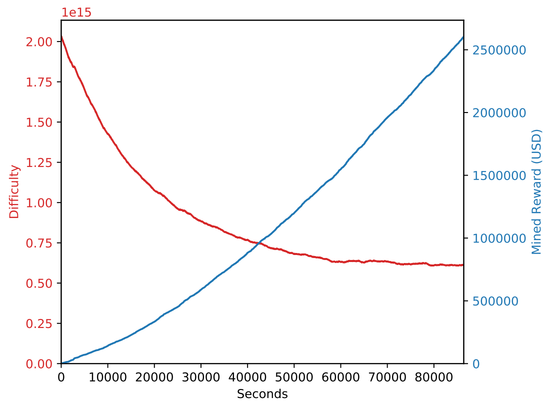

Figure 3.6: Twenty-Four Hours of Eclipse Mining with Pool Hash Power

Table 3.2: Electricity Costs of Eclipse Mining in Different Countries: USD 100 Mining Reward and USD 50 Electricity Cost $\widehat{=}$ Reward/Cost Ratio of 100/50 = 2

Country	**Mining Hardware**	**Duration**	**≈ Electricity Cost**	**Reward / Cost**
Germany	10x RX 570	3 Years	13 000 USD	0.47
	Pool of RX 570s	1 Hour	100 000 USD	0.43
	Pool of RX 570s	3 Hours	310 000 USD	0.48
	Pool of RX 570s	6 Hours	630 000 USD	0.59
	Pool of RX 570s	12 Hours	1 300 000 USD	0.78
	Pool of RX 570s	24 Hours	2 500 000 USD	1.04
UK	10x RX 570	3 Years	8 300 USD	0.74
	Pool of RX 570s	1 Hour	66 000 USD	0.68
	Pool of RX 570s	3 Hours	200 000 USD	0.75
	Pool of RX 570s	6 Hours	400 000 USD	0.93
	Pool of RX 570s	12 Hours	800 000 USD	1.23
	Pool of RX 570s	24 Hours	1 600 000 USD	1.63
USA	10x RX 570	3 Years	5 100 USD	1.19
	Pool of RX 570s	1 Hour	41 000 USD	1.09
	Pool of RX 570s	3 Hours	120 000 USD	1.22
	Pool of RX 570s	6 Hours	250 000 USD	1.50
	Pool of RX 570s	12 Hours	490 000 USD	1.99
	Pool of RX 570s	24 Hours	990 000 USD	2.64
China	10x RX 570	3 Years	3 200 USD	1.93
	Pool of RX 570s	1 Hour	25 000 USD	1.78
	Pool of RX 570s	3 Hours	76 000 USD	1.98
	Pool of RX 570s	6 Hours	150 000 USD	2.44
	Pool of RX 570s	12 Hours	300 000 USD	3.23
	Pool of RX 570s	24 Hours	610 000 USD	4.28
Egypt	10x RX 570	3 Years	790 USD	7.74
	Pool of RX 570s	1 Hour	6 300 USD	7.12
	Pool of RX 570s	3 Hours	19 000 USD	7.91
	Pool of RX 570s	6 Hours	38 000 USD	9.75
	Pool of RX 570s	12 Hours	76 000 USD	12.92
	Pool of RX 570s	24 Hours	150 000 USD	17.13

4 Eclipse Attacks on Parity Ethereum

Here, we present eclipse / denial-of-synchronization attacks on Parity Ethereum. We list relevant pull requests from Parity's GitHub repository in *Appendix B*.

4.1 The Sunshine Setup: Package Leaflet

The two following sections (4.2 and 4.3) describe attacks which block a victim node from accessing the Ethereum network. Apart from subsection 4.3.6, they do **not** include an active adversary serving the victim relevant information.
External nodes' knowledge of the victim's IP address and node id were ignored in our testing, although eclipse attacks may be harder to accomplish if external nodes have already exchanged information with the victim during the preceding test runs, even if the victim client is resetted before each run.
Before subsection 4.3.6, we used the `parity_netPeers` [22] JSON-RPC method, only viewing the victim as eclipsed if it has not completed the handshake with any normal node. This strict requirement was loosened starting from subsection 4.3.6, where a more fine-grained analysis allowed us to view the victim as eclipsed if it does not receive *light protocol request responses* [23], even if it receives certain light protocol messages that contain information about the blockchain state, for example message type 0 (status) containing "the hash of the best [...] known block". Thus, an eclipsed victim may still receive information that could be used to check whether *something is wrong* during an eclipse attack.

4.2 Good Weather in Dockerland

We implemented the attack scenario from figure 1.2 as proposed in section 1.3. The result is a Docker-Compose setup that is called *Sunshine* (sunshine probably helps against eclipse attacks). The Attacker Nodes (IV) from the scenario figure are running on a desktop machine, and the victim is running on a notebook. These two computers are connected wirelessly via a router, with no additional hardware in between. Fifty attacker nodes and the standard victim peer range of twenty-five to fifty are used [29]. All nodes are Parity Ethereum version 2.4.5, which is the latest stable release on 14th May 2019 [32], and are

[22] `https://wiki.parity.io/JSONRPC-parity-module.html?q=traceTransaction#parity_netpeers`
[23] `https://wiki.parity.io/The-Parity-Light-Protocol-(PIP)#request-response`

based on the Docker Hub image [24], which is in turn based on Ubuntu 16.04.6 LTS (Xenial Xerus) [25]. The victim has port 30303 for peer-to-peer networking open for both TCP and UDP, although the standard router configuration disables incoming access from the internet to the victim host. The victim is also configured to be unaware of its public IP address, which is the standard configuration, although awareness could be realized with a service like *Ipify* [26]. An attack counts as successful if the victim client is eclipsed on startup and stays eclipsed for at least five minutes. On the other hand, an attack is unsuccessful if the victim client establishes a connection (completes the handshake) with any *normal* node during the first five minutes after startup. Five minutes were chosen because that would probably be enough time to execute a real-world attack like the one mentioned in chapter 1. Additionally, one would expect a smart Ethereum client to detect that it is only connected to useless nodes (providing no relevant information about the blockchain) during a duration of five minutes, and force connections to other peers.

To ease the communication between the victim and attacker host, and also preserve statistics after container restarts, we used the official Redis Docker image [27]. Redis [28] is a key-value store that is easy to set up and provides various convenience methods. Redis was integrated into our Python code with redis-py [29].

4.3 Mixed Weather in Parityland

Our testing led to different results depending on the victim synchronization state. The results are summarized in table 4.1 and described in the following subsections.

Table 4.1: Success Probabilities of Eclipse Attacks in Different Scenarios

Last Victim Synchronization	**Probability of Successful Attack**
Never / First Start	100 %
Last Sync $<$ 7 days ago	0 %
Last Sync $\geq$ 7 days ago	35 %

[24]`https://hub.docker.com/r/parity/parity/`
[25]The latest Parity images on Docker Hub are still based on Ubuntu 16 on October 7, 2019.
[26]`https://www.ipify.org/`
[27]`https://hub.docker.com/_/redis`
[28]`https://redis.io/`
[29]`https://pypi.org/project/redis/`

4.3.1 Fresh Start

A fresh victim client (just downloaded / first start) performed worst, and was eclipsed every time (20/20). The fresh client knows no other network participants (except the bootnodes) and accepts the incoming attacker connections. It does not connect to any normal node (whose IP address and further info could be obtained from a bootnode), even though the attacker nodes do not provide any valuable information about the blockchain state.

4.3.2 Average Joe

A typical Parity client is probably synchronized, and therefore immune to our eclipse attacks. We repeatedly tested with a fresh client that was synchronized and then restarted. This led to an attack success rate of zero (0/20). Our next test consisted of an attack on a victim client that was synchronized, turned off for half a week (3.5 days) and then restarted. This also led to an eclipse rate of zero (0/20).

4.3.3 Taking Breaks is Bad

Clients that were synchronized but went offline for a week were prone to eclipse attacks with a non-negligible success rate of 35% (7/20). This is possibly due to a mechanism regarding Parity's `NodeTable`: Information about nodes that have been contacted (successfully or not) more than a week ago is discarded, and only nodes that have not been contacted yet are kept [31].

4.3.4 Wrong Forecasts

The successful eclipse attack on a fresh client is not realistic, because the attacker needs to know the victim's enode public key during the victim's first start, but this key is just generated during the victim's start. Thus, the attacker has no way of knowing it beforehand. The Parity Wiki [34] lists two ways to obtain the enode public keys: By RPC or by monitoring the victim's console output. Both ways do not fit into the attack scenario. This makes our attack on *paused* clients from the last subsection the only successful and realistic attack. If the attacker knows the victim's IP address and the victim has connected to the Ethereum network at least once, the attacker may be able to look up the victim's public key with a node explorer, for example `https://www.ethernodes.org/`. Alternatively, the attacker could run such a service himself, collecting as many public key / IP address pairs as possible.

4.3.5 Differences to Go Ethereum and “Low-Resource Eclipse Attacks on Ethereum’s Peer-to-Peer Network”

Go Ethereum is not vulnerable to a simple connection monopolization attack anymore, because it forces outgoing connections [26]. The success of our attack proves that Parity does not successfully force any outgoing connections, implying that Parity does not force outgoing connections before allowing incoming ones. Parity’s behavior does not have to originate from a design flaw, but could also arise from a general application overload caused by the attacker clients, resulting in unspecified behavior.
According to [26, section II C], an attacker needs to maintain `maxpeers` connections to a Go Ethereum victim in order to eclipse it. Our testing shows that a standard [30] Parity light client victim only performs 29-33 handshakes [31] with attacker nodes (50 in total) during the five-minute testing period. This indicates that an eclipsed Parity light victim is not connected to `max_peers` nodes in total.

4.3.6 Serving the Victim

Light clients do not fetch blockchain information from other light clients, only from full clients [38]. To test whether our setup can be used to serve the victim actual information about the blockchain state, we added three full attacker nodes and updated our setup to Parity version 2.6.2 beta. These full nodes did not need to be up-to-date, because we configured the victim client to start its synchronization at block zero instead of a hard-coded higher block [32]. This drastically reduces setup time as well as storage requirements. We started our eclipse attack just like before (only with light attacker nodes). Then we launched the full attacker nodes and stopped some light nodes to allow connections between the full nodes and the victim. This resulted in a proof of our attack concept: We verified [33] that the victim did only get *light protocol request responses* [34] from attacker nodes. The victim was allowed to receive other messages as described in section 4.1.

4.3.7 Taking Docker out of the Equation

To check whether our attacks are only possible on victim clients running as Docker containers, we proceeded our testing with the victim client running directly on the host OS

[30]`min_peers = 25, max_peers = 50` [29]

[31]These numbers are still valid if the victim is running without Docker. Three tests in the context of subsection 4.3.7 showed 33, 34 and 33 handshakes.

[32]With the `-no-hardcoded-sync` CLI flag. `https://wiki.parity.io/Configuring-Parity-Ethereum#cli-options-for-parity-ethereum-client`

[33]With the `tcpdump` packet sniffer.

[34]Message Type 3 `https://wiki.parity.io/The-Parity-Light-Protocol-(PIP)#request-response`

(Ubuntu 19.04 Disco Dingo). First, we measured the time that passes before a non-attacked victim client receives its first *request response*. To reduce the impact of previous tests on the current one, we randomized the node id before each test. Our findings are displayed in table 4.2, ranging from 0 seconds to 2 minutes and 5 seconds, with a mean value of 41.8 seconds. These results confirm that repeated 5 minute eclipse periods are enough to prove that something is wrong with a Parity Ethereum client. The next setup consisted of 50 light attacker clients running as Docker containers on one host, and a light victim client running without Docker on another host. We performed ten test runs, changing the victim node id before each run. Every time, the victim client stayed eclipsed for the whole test duration of 5 minutes, proving that our attack is not only successful on victim clients running as Docker containers, but also on victim clients running directly on the host OS. Lastly, we ruled out general network overload (router or host) as a factor leading to the victim's eclipse: While the victim (running without Docker) was attacked by 50 light clients, we started three light clients (running as Docker containers) on the victim host. These three clients were fetching blockchain information from other network peers while the victim was eclipsed, showing that the whole network stack was intact and responsive.

Table 4.2: Time Spent Before the First Request Response Is Received

Test #:	1	2	3	4	5	6	7	8	9	10	Ø
Time in Seconds:	61	**0**	21	16	17	52	**125**	25	91	10	**41.8**

5 Table Poisoning

The way Parity handles peer discovery [35] allows, at least theoretically, so-called *table poisoning* attacks: An adversary tries to insert his own *(poisonous)* nodes into the victim's node *table*. If he succeeds, the victim may later connect to the adversary by itself, easing attacks like connection monopolization.

5.1 Setup

Parity Ethereum has the ability to print verbose logs for many sub-modules [36]. This allowed us to write a log analyzer in Python, generating and printing a live view of Parity's internal state. We ported relevant code to Python and developed a *manager* that generates attacker node ids on-the-fly and inserts them into a Redis queue, giving multiple attacker clients access to them. We started with Parity Ethereum version `2.6.2-beta`, the latest Docker Hub [37] release on 13th September 2019.

5.1.1 Details

A Parity node can be identified by several related attributes:

- **Secret Key:** This 256 bit / 32 byte key can be passed to the client using the `--node-key` CLI option (hex formatted). If no key is passed to the client and the client starts for the first time, a key will be randomly generated and saved to `network/key` [38]. This saved key will be loaded when the client is restarted and no other key is specified.
- **Public Key / Node ID:** This 512 bit key is calculated by multiplying the secret key with the base point of the "Bitcoin" elliptic curve *secp256k1* [39] (Python: [18]).
- **ID Hash:** The 256 bit Keccak hash of the public key (Python: [46]).

Bucket insertion is based on the `distance()` function which returns the base two logarithm of the xor distance between the own id hash and the inserted node's id hash. See tables 5.1 and 5.2 for example distance values.

[35]See *Appendix C* for more details.
[36]`https://wiki.parity.io/FAQ#how-can-i-make-parity-ethereum-write-logs`
[37]`https://hub.docker.com/r/parity/parity/tags`
[38]On Linux, the `network` directory is located in `~/.local/share/io.parity.ethereum`.

Table 5.1: `distance()` Examples

ID Hash A	ID Hash B	`distance()`
0x0	0x0	*None*
0x0	0x1	0
0x0	0x2	1
	0x3	1
0x0	0x4	2
	...	...
	0x7	2
0x0	0x8	3

Table 5.2: More `distance()` Examples

ID Hash A	ID Hash B	`distance()`
0x0	0xf...f	255
	...	...
	0x80...0	255
0x0	0x7f...f	254

5.2 Local Networking Done Wrong

To test our setup in a controlled environment (and to reduce avoidable network load on regular Ethereum nodes), we instructed both the attackers and the victim to restrict outbound connections to local IPs [39]. Our attack did not work in this environment: Although the light client victim was importing blocks from full attacker nodes, not a single attacker id was inserted into the victim's table.

The following log snippets that appeared together repeatedly gave a hint:
`Starting discovery, Starting round Some(0), Completing discovery`
If the victim's peer discovery process finishes instantly, not a single bucket is filled. Source code instrumentation revealed: There are no *nearest nodes* at the beginning of the discovery loop [40] (because `nearest_node_entries()` [41] returns zero nodes). Thus, the loop is skipped and discovery is stopped [42]. But why is the discovery node table empty, although the victim is fetching information from the local attacker nodes?

The log showed that the victim tried to ping each attacker at the default discovery port 30303. Because the attacker clients were running in Docker containers without published ports on the attacker host, the victim's pings did not reach anyone and expired.

[39]Using the `--allow-ips=private` CLI option: `https://wiki.parity.io/Configuring-Parity-Ethereum#cli-options-for-parity-ethereum-client`

[40]`https://github.com/paritytech/parity-ethereum/blob/61a7c30ed5a093c7c2a26a93aeb1ed5fda3cb017/util/network-devp2p/src/discovery.rs#L328`

[41]`https://github.com/paritytech/parity-ethereum/blob/61a7c30ed5a093c7c2a26a93aeb1ed5fda3cb017/util/network-devp2p/src/discovery.rs#L325`

[42]`https://github.com/paritytech/parity-ethereum/blob/61a7c30ed5a093c7c2a26a93aeb1ed5fda3cb017/util/network-devp2p/src/discovery.rs#L342`

5.3 Local Networking Done Right-ish

To address these issues, we assigned each attacker client a unique port on the attacker host and instructed the clients to use their new port. This resulted in partial success: The attacker client bound to port 30303 was successfully pinged by the victim, replied with a pong and was added to the victim's node table. But the ping requests to the other attackers expired. Why?

The answer is summarized in this source code comment: *"We can't know remote listening ports, so just assume defaults and hope for the best"* [43]. For each attacker that was not bound to port 30303, Parity still *assumed* port 30303 and directed its ping request there. This resulted in the attacker on port 30303 getting all requests and answering all requests, but only a single reply was deemed valid by the victim, and all other replies were labeled *unexpected* because their node id did not match the expected node id.

Parity's behavior is somewhat incomprehensible because Parity manages to fetch blockchain information from the most exotic ports, but always directs its pings to port 30303. But this protects Parity from a simple table poisoning attack using an unmodified Parity Ethereum client [44], although an attack using a heavily modified Parity client or a program imitating Parity's peer-to-peer behavior seems viable.

The findings from this section were confirmed using a victim client with all outgoing connections allowed, although seemingly random ping / pong drops that could lead to the removal of an attacker node from the victim's table were witnessed.

5.4 Modified Attacks

To test whether the aforementioned attack using a modified Parity client is possible, we modified Parity version 2.6.3 beta according to table 5.3. First of all, we instructed the attacker to (slowly) cycle through a set of node ids, pinging the victim each time. The victim then sent a ping message to the attacker. Because the attacker needed to answer these messages with the right node id to get into the victim's node table, we utilized two different strategies: Answer the ping with the last node id (most likely the right one; configurations A and B), or answer the ping with every node id (definitely including the right one; configurations C and D). These two strategies led to the same results (a minimum of six concurrent attacker table entries with ten attacker node ids and a minimum of zero entries with thirty ids). To stay in the victim's node table, we also needed to answer the

[43] `https://github.com/paritytech/parity-ethereum/blob/05f9606bf2fc9fb51a7dd303 3d9fc2ce55f1a2e9/util/network-devp2p/src/host.rs#L793`

[44] Simply cycling through many node ids with a single attacker client will not suffice, because the old node ids need to stay online to be able to accept a connection from the victim and to reply to victim messages.

victim's *FindNode* requests with the right node id, so we chose to answer each *FindNode* request once for every node id [45]. We made a last distinction between two sets of node ids, one containing ten node ids, and the other containing thirty node ids.

Table 5.3: Parity Ethereum Source Code Modifications

Configuration:	A	B	C	D
Sends Attacker Pings with:	Multiple Cycling Node IDs			
Answers Victim Pings with:	Last Node ID		All Node IDs	
Answers Victim *FindNode* Requests with:	All Node IDs			
Number of Attacker Node IDs:	10	30	10	30
Concurrent Attacker Table Entries (minimum):	6	0	6	0

Our testing indicates that this attack scheme (one attacker client with many node ids) is not feasible, because generally each attacker node id needs to stay *live* [46] to stay inside the victim's node table. B and D, the configurations with a thirty node id set, show that simply spamming answers is not the way to go, because both configurations dropped to zero attacker node ids in the victim's table during our testing. The "working" configurations A and C dropped to six attacker node ids, which is also far from a satisfactory result.

5.5 Separate IPs

To resume our testing with *one IP per attacker*, we started a Docker-Compose setup with victim and attackers on the same host. Docker assigned the victim and each attacker container a unique local IP address. Each Parity client was instructed to connect to local IP addresses only.

A test run with 150 attackers resulted in the victim inserting 64 of them into its node table, before all those containers rendered our system (along with Docker's network stack) unusable. As a shutdown of all containers freed up system resources, the victim inserted 6 additional attacker ids into its table before exiting itself.

To test whether the inserted attacker node ids stay in the victim's table, we repeated our test. This time, we started only 50 attacker containers to ensure a (relatively) responsive system. All 50 attackers were quickly inserted into the victim's node table, and most of them stayed there for the test duration of one hour. Only 5 removals and re-insertions did

[45] We instructed the attacker to always return the current node id as the only known "neighbor".

[46] The attacker cannot determine the right node id from a victim request, because requests like Ping and FindNode do not include a recipient node id.

occur, with a minimum of 48 attacker nodes always staying in the table. We verified that our attack works with a "normal" victim by lifting the victim's restriction and allowing outgoing connections to all peers. The result was better than before: All 50 attackers stayed in the victim's node table for one hour, without a single removal. But is there a realistic use case for our attack, where every attacker needs a separate IP address?
Many IP addresses can be obtained by having many cheap devices, like the USD 10 Raspberry Pi Zero W [36]. But even acquiring many cheap devices will quickly exceed the financial bounds of a *low-resource* attack. There is another possibility to obtain many IP addresses: By *simulating* many devices, for example using MAC spoofing. This could allow an adversary to bind many IP addresses to a single computer. Then he could pass incoming victim packets to Docker containers running Parity, or modify Parity to map target IPs to node ids.

5.6 Persistent Poison

We validated that our poisonous table entries persist between normal docker container restarts using a two-phase test. Phase 1: We used 20 poisonous attacker clients to insert their node ids into the victim's table. We stopped the attacker and victim containers after that was accomplished. Phase 2: We restarted the victim only and checked its network log. It indicated that the victim sent pings to our attacker nodes. Because our attacker nodes were offline during the second phase, they could not influence the victim, proving that the attacker entries in the victim's node table persisted between restarts of the victim container.

6 Conclusions

6.1 Summary

After providing an introduction into our thesis in chapter 1, we spotlighted uses of eclipse attacks like double spending in chapter 2. Because a low-resource attacker may also have a low amount of Ether to spend, we analyzed the impact of Ethereum's difficulty adjustment algorithm on mining rewards during an eclipse attack (chapter 3). We also took electricity costs in different countries into account. Chapter 4 contains the heart of our thesis: eclipse attacks on Parity Ethereum. We showed that eclipse attacks are possible on new clients and on clients that have been offline for at least a week. While we heavily relied on Docker for most of our tests, we verified that eclipse attacks can not only be performed on victim clients running as Docker containers. In chapter 5, we showed that Parity's networking algorithms allow table poisoning attacks, in which an adversary inserts his own node ids into the victim's internal node table. We hope that our findings will not only help the Parity Ethereum developers in securing their client, but also benefit future researchers analyzing it.

6.2 Discussion and Open Problems

As explained in subsection 4.3.4, our attack on a new client relies on the attacker's knowledge of the victim's node id, which will most likely only be generated during the first start of the victim client. This makes our attack less dangerous, to say the least. Further research could result in attacks that overcome this obstacle, for example by combining the eclipse attack with an attack on the JSON-RPC or other APIs which could leak the victim's node id to the attacker.
While a full-fledged eclipse attack with faked transactions, miners confirming them in fake blocks etc. was out of scope for this thesis, further research and collaboration with the Parity Ethereum developers could result in advanced measures against eclipse attacks (beyond simple network rules).
There are also many possibilities to extend our table poisoning attack: Creative setups utilizing local network attacks like MAC spoofing (see section 5.5) leave plenty of room for additional research. If the basic eclipse vulnerability from chapter 4 is fixed, multiphase attacks utilizing table poisoning can be developed without accidentally eclipsing

the victim with many poisonous attackers.
Speaking of fixing: The focus of our work was on attacking Parity Ethereum. However, locating and fixing the vulnerability that allows eclipse attacks is also an important task. The removal of vulnerabilities that allow table poisoning is a more complex topic, because there is a direct correlation between desired traits of Ethereum clients and their vulnerability to table poisoning [26, section V].

References

[1] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. *arXiv:1605.07524 [cs]*, May 2016. arXiv: 1605.07524. URL: `http://arxiv.org/abs/1605.07524`.

[2] Protocol rules - Bitcoin Wiki, June 2019. URL: `https://web.archive.org/web/20190624153749/https://en.bitcoin.it/wiki/Protocol_rules`.

[3] Vitalik Buterin. On Slow and Fast Block Times, September 2015. URL: `https://web.archive.org/web/20190624155458/https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/`.

[4] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure Routing for Structured Peer-to-peer Overlay Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314, December 2002. URL: `http://doi.acm.org/10.1145/844128.844156`, `doi:10.1145/844128.844156`.

[5] Christian Decker and Roger Wattenhofer. Bitcoin transaction malleability and mtgox. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, pages 313–326, Cham, 2014. Springer International Publishing.

[6] John R. Douceur. The Sybil Attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, Lecture Notes in Computer Science, pages 251–260. Springer Berlin Heidelberg, 2002.

[7] SAPPHIRE Pulse Radeon AMD RX 570 4gb Gddr5 OC günstig kaufen | eBay, June 2019. URL: `https://web.archive.org/web/20190613115355/https://www.ebay.de/p/SAPPHIRE-Pulse-Radeon-AMD-RX-570-4gb-Gddr5-OC/14028844310?iid=173890059372`.

[8] Electricity prices around the world, June 2018 | GlobalPetrolPrices.com, June 2019. URL: `https://web.archive.org/web/20190620204657/https://www.globalpetrolprices.com/electricity_prices/`.

[9] Ether — Ethereum Homestead 0.1 documentation. URL: `http://www.ethdocs.org/en/latest/ether.html`.

[10] Ethereum 2.0's Phase Zero Scheduled to Launch on January 3, 2020: Devs, July 2019. URL: `https://web.archive.org/web/20190701124455/https://cointelegraph.com/news/ethereum-20s-phase-zero-scheduled-to-launch-on-january-3-2020-devs`.

[11] Ethereum Energy Consumption Index (beta) - Digiconomist, June 2019. URL: `https://web.archive.org/web/20190613102532/https://digiconomist.net/ethereum-energy-consumption`.

[12] Introduction — ENS 0.1 documentation. URL: `https://docs.ens.domains/en/latest/introduction.html`.

[13] Ethereum Project. URL: `https://www.ethereum.org/`.

[14] Ethereum Average Block Time Chart, May 2019. URL: `https://web.archive.org/web/20190508121409/https://etherscan.io/chart/blocktime`.

[15] Ethereum Node Tracker, March 2019. URL: `https://web.archive.org/web/20190327163104/https://etherscan.io/nodetracker`.

[16] 1190 EUR to USD | Convert Euro to US-Dollar | XE, June 2019. URL: `https://web.archive.org/web/20190613115708/https://www.xe.com/de/currencyconverter/convert/?Amount=1.190&From=EUR&To=USD`.

[17] Ittay Eyal and Emin Gun Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. *arXiv:1311.0243 [cs]*, November 2013. arXiv: 1311.0243. URL: `http://arxiv.org/abs/1311.0243`.

[18] fastecdsa — fastecdsa 1.7.4 documentation, September 2019. URL: `https://web.archive.org/web/20190925080938/https://fastecdsa.readthedocs.io/en/latest/fastecdsa.html#fastecdsa.keys.get_public_key`.

[19] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in Bitcoin and Ethereum Networks. *arXiv:1801.03998 [cs]*, January 2018. arXiv: 1801.03998. URL: `http://arxiv.org/abs/1801.03998`.

[20] Arthur Gervais, Hubert Ritzdorf, Ghassan O. Karame, and Srdjan Capkun. Tampering with the Delivery of Blocks and Transactions in Bitcoin. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 692–705, Denver, Colorado, USA, 2015. ACM Press. URL: `http://dl.acm.org/citation.cfm?doid=2810103.2813655`, `doi:10.1145/2810103.2813655`.

[21] Improvements to discovery implementation · Issue #8757 · paritytech/parity-ethereum. URL: https://github.com/paritytech/parity-ethereum/issues/8757.

[22] GitHub - paritytech/parity-ethereum: The fast, light, and robust EVM and WASM client., August 2019. URL: https://web.archive.org/web/20190813074353/https://github.com/paritytech/parity-ethereum.

[23] Sharon Goldberg, Aviv Zohar, Alison Kendler, and Ethan Heilman. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *Eclipse Attacks on Bitcoin's Peer-to-Peer Network*, pages 129–144, 2015. URL: https://www.usenix.org/node/190891.

[24] Ethereum Mining Pools Rating | Investoon, June 2019. URL: https://web.archive.org/web/20190611112329/https://investoon.com/mining_pools/eth.

[25] Michael Kohnen, Mike Leske, and Erwin P. Rathgeb. Conducting and Optimizing Eclipse Attacks in the Kad Peer-to-Peer Network. In Luigi Fratta, Henning Schulzrinne, Yutaka Takahashi, and Otto Spaniol, editors, *NETWORKING 2009*, Lecture Notes in Computer Science, pages 104–116. Springer Berlin Heidelberg, 2009.

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

[27] Christopher Natoli and Vincent Gramoli. The Balance Attack or Why Forkable Blockchains are Ill-Suited for Consortium. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 579–590, Denver, CO, USA, June 2017. IEEE. URL: http://ieeexplore.ieee.org/document/8023156/, doi:10.1109/DSN.2017.44.

[28] K. Nayak, S. Kumar, A. Miller, and E. Shi. Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 305–320, March 2016. doi:10.1109/EuroSP.2016.32.

[29] Parity Config Generator, June 2019. URL: https://web.archive.org/web/20190619145320/https://paritytech.github.io/parity-config-generator/.

[30] Parity Ethereum - util/network-devp2p/src/host.rs#L761, August 2019. original-date: 2015-11-23T11:07:32Z. URL: https://github.com/paritytech/parity-

ethereum/blob/66e4410be72da38dd001b7ad6c9e99b18cfae9ad/util/network-devp2p/src/host.rs#L761.

[31] Parity-ethereum: util/network-devp2p/src/node_table.rs, June 2019. original-date: 2015-11-23T11:07:32Z. URL: https://github.com/paritytech/parity-ethereum/blob/1786b6eedda88c1a6e87b5ee7cd8c013515ece6c/util/network-devp2p/src/node_table.rs#L328.

[32] Releases · paritytech/parity-ethereum · GitHub, May 2019. URL: https://web.archive.org/web/20190514143523/https://github.com/paritytech/parity-ethereum/releases.

[33] Parity Documentation - Light Client. URL: http://wiki.parity.io/Light-Client.html.

[34] Demo PoA-Tutorial · Parity Tech Documentation, June 2019. URL: https://web.archive.org/web/20190619150936/https://wiki.parity.io/Demo-PoA-tutorial.html.

[35] Radeon RX 580 und RX 570 im Test: AMDs Grafikkarten sind schneller und sparsamer - Golem.de, June 2019. URL: https://web.archive.org/web/20190620205546/https://www.golem.de/news/radeon-rx-580-und-rx-570-im-test-amds-grafikkarten-sind-schneller-und-sparsamer-1704-127204.html.

[36] Raspberry Pi Zero W ID: 3400 - $10.00 : Adafruit Industries, Unique & fun DIY electronics and kits, September 2019. URL: https://web.archive.org/web/20190925162019/https://www.adafruit.com/product/3400.

[37] Fabian Ritz and Alf Zugenmaier. The impact of uncle rewards on selfish mining in ethereum. *CoRR*, abs/1805.08832, 2018. URL: http://arxiv.org/abs/1805.08832, arXiv:1805.08832.

[38] Thibaut Sardan. What is a light client and why you should care?, July 2018. URL: https://www.parity.io/what-is-a-light-client/.

[39] Secp256k1 - Bitcoin Wiki, September 2019. URL: https://web.archive.org/web/20190925080902/https://en.bitcoin.it/wiki/Secp256k1.

[40] Atul Singh, Tsuen wan 'Johnny' Ngan, Peter Druschel, and Dan S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*, 2006.

[41] Emil Sit and Robert Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, Lecture Notes in Computer Science, pages 261–269. Springer Berlin Heidelberg, 2002.

[42] Mori Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Exploiting KAD: possible uses and misuses. *ACM SIGCOMM Computer Communication Review*, 37(5):65, October 2007. URL: `http://portal.acm.org/citation.cfm?doid=1290168.1290176`, `doi:10.1145/1290168.1290176`.

[43] TALOS-2017-0471 || Cisco Talos Intelligence Group - Comprehensive Threat Intelligence. URL: `https://web.archive.org/web/20190113160838/https://www.talosintelligence.com/reports/TALOS-2017-0471`.

[44] r/ethereum - Transaction Malleability: Does Ethereum have this issue? URL: `https://www.reddit.com/r/ethereum/comments/3o9ru0/transaction_malleability_does_ethereum_have_this/`.

[45] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. A survey of DHT security techniques. *ACM Computing Surveys*, 43(2):1–49, January 2011. URL: `http://portal.acm.org/citation.cfm?doid=1883612.1883615`, `doi:10.1145/1883612.1883615`.

[46] Utilities — ETH-Utils documentation, September 2019. URL: `https://web.archive.org/web/20190925083201/https://eth-utils.readthedocs.io/en/latest/utilities.html#keccak-bytes-int-bool-text-str-hexstr-str-bytes`.

[47] Border Gateway Protocol, December 2018. Page Version ID: 872170057. URL: `https://en.wikipedia.org/w/index.php?title=Border_Gateway_Protocol&oldid=872170057`.

[48] Decentralized autonomous organization, November 2018. Page Version ID: 868070421. URL: `https://en.wikipedia.org/w/index.php?title=Decentralized_autonomous_organization&oldid=868070421`.

[49] Kademlia, November 2018. Page Version ID: 867855490. URL: `https://en.wikipedia.org/w/index.php?title=Kademlia&oldid=867855490`.

[50] Karl Wüst and Arthur Gervais. Ethereum Eclipse Attacks. Technical report, ETH Zurich, 2016. URL: `http://hdl.handle.net/20.500.11850/121310`, `doi:10.3929/ethz-a-010724205`.

Appendices

Appendix A: Glossary

- **Ethereum** is a decentralized *"blockchain app platform"* [13]. In contrast to Bitcoin or Litecoin, apps can run on Ethereum using *smart contracts* that are executed by miners. This allows for *decentralized autonomous organizations* [48], services like the *Ethereum Name Service* [12] or simple apps like a coin flip gambling contract.
- **Ether** is the cryptocurrency of the Ethereum blockchain. It has multiple denominations like *Wei* (10^{-18} Ether) or *Gwei* (10^{-9} Ether) [9]. Ether can be sent to smart contracts and Ethereum addresses and can also be used to pay for *gas*.
- **Gas**: "Gas Fee is effectively the amount of Gas needed to be paid to run a particular transaction or program (called a contract)." [9] These fees are collected by miners who perform the computations specified in smart contracts.
- **Eclipse Attack**:

  > "In an Eclipse attack [...], a modest number of malicious nodes conspire to fool correct nodes into adopting the malicious nodes as their peers, with the goal of dominating the neighbor sets of all correct nodes. If successful, an Eclipse attack enables the attacker to mediate most overlay traffic and effectively "eclipse" correct nodes from each other's view. In the extreme, an Eclipse attack allows the attacker to control all overlay traffic, enabling arbitrary denial of service or censorship attacks."
  >
  > *Eclipse Attacks on Overlay Networks: Threats and Defenses [40]*

  A node (network participant) is eclipsed if it is only connected to malicious nodes, thus having no connections to the *good* nodes making up most of the network. The malicious nodes can then withhold transactions from the eclipsed node, or broadcast transactions *only* to the eclipsed node. Regarding Ethereum: Those transactions still have to be cryptographically sound (valid and signed), but the funds can be double-spent as illustrated in chapter 1.

Appendix B: Parity Ethereum Pull Requests

These are some interesting pull requests from Parity Ethereum's GitHub repository. The first one is directly related to "Low-Resource Eclipse Attacks on Ethereum's Peer-to-Peer Network" [26]. The following ones are / may be relevant for further analysis of Parity.

1. Incoming Connections are limited to
   `max(max_peers - min_peers, min_peers / 2)` [47]
   Standard Values 25 and 50 → `max(50 - 25, 25 / 2) = 25 = min_peers`
   This seems to imply that the client *wants* 25 connections, has 25 incoming connections, and is *satisfied* now [29]. But Parity manages connections differently, as is explained in *Appendix C*.
2. Unknown nodes are removed if table / nodes.json is too big. [48]
3. Light clients choose random peers for requests. [49]
4. Light client requests timeout after 7 seconds and are then reassigned. [50]
5. Node Table is sorted by last contact. [51]
6. Full and light client have separate database directories. [52]
7. General P2P Discovery Changes [53]

This Parity Substrate issue [54] (open as of 6th October 2019) describes the following untargeted eclipse attack vector: An attacker crafts many node ids *close* to each other and waits. If any node randomly picks one of the crafted node ids for discovery, the attacker could flood this node with his crafted node ids to eclipse the node.

[47] `https://github.com/paritytech/parity-ethereum/pull/8060`
[48] `https://github.com/paritytech/parity-ethereum/pull/7716`
[49] `https://github.com/paritytech/parity-ethereum/pull/7844`
[50] `https://github.com/paritytech/parity-ethereum/pull/7848`
[51] `https://github.com/paritytech/parity-ethereum/pull/8541`
[52] `https://github.com/paritytech/parity-ethereum/pull/9064`
[53] `https://github.com/paritytech/parity-ethereum/pull/9526`
[54] `https://github.com/paritytech/substrate/issues/332`

Appendix C: Parity Ethereum's Connection Management

Number of Allowed Ingress and Egress Connections

This subsection is based on Parity's `util/network-devp2p/src/host.rs` module [30]. Standard Configuration [29] implies:

- `min_peers = 25, max_peers = 50`

-
  ```
  max_ingress = max(max_peers - min_peers, min_peers / 2)
              = max(50 - 25, 25 / 2) = 25
  ```
- `ingress_count` is the amount of connections started by the Parity client itself. Ingress connections are allowed if `ingress_count <= max_ingress = 25`.
- `egress_count` is the amount of connections started by another network peer. Egress connections are allowed if `egress_count <= min_peers = 25`.

Thus, a standard Parity client allows 25 ingress plus 25 egress connections.

About Buckets

`Discovery` is initialized in `Host::init_public_interface()` [55] with the current node's public and private key pair. Notice the following assignment in the `Discovery` constructor: `id_hash: keccak(key.public())`
To calculate the bucket mapping, `Discovery::update_bucket_record()` utilizes the `Discovery::distance()` function [56], which operates on two id hashes. An attacker can calculate which node ids will be inserted into a specific victim bucket, because the id hashes are based on the victim's *public* key. *Marcus, Heilman and Goldberg* exploited this vulnerability in their table poisoning attack on Geth [26].

[55] The considered source files are located in `util/network-devp2p/src/`.
[56] See subsection 5.1.1 for further details and example values.

Appendix D: Software We Are Grateful For

This section lists software we used while working on this thesis, which was not mentioned yet and may benefit future students and researchers.

Ethereum:

- Network Visualization / Status
  `https://github.com/cubedro/eth-netstats`
- Parity Config File Generator
  `https://paritytech.github.io/parity-config-generator/`

Python:

- JSON "Cleaner"
  `https://github.com/getify/JSON.minify/tree/python`
- JSON-RPC
  `https://github.com/pavlov99/json-rpc`
- Plotting Library
  `https://github.com/matplotlib/matplotlib`
- Colored Terminal Output
  `https://github.com/tartley/colorama`

Various Docker setups and dockerfiles:

- `https://github.com/paritytech/parity-ethereum/tree/master/scripts/docker`
- `https://github.com/Capgemini-AIE/ethereum-docker`
- `https://github.com/konradkonrad/docker-pyeth-cluster`

Other:

- LaTeX Table Generator
  `https://www.tablesgenerator.com/`
- SED Command List
  `http://sed.sourceforge.net/sed1line.txt`