Understanding and Hardening Blockchain Network Security Against Denial of Service Attacks

Kai Li
Syracuse University

Follow this and additional works at: https://surface.syr.edu/etd

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Li, Kai, "Understanding and Hardening Blockchain Network Security Against Denial of Service Attacks" (2022). Dissertations - ALL. 1390.
https://surface.syr.edu/etd/1390

This Dissertation is brought to you for free and open access by the SURFACE at Syracuse University at SURFACE at Syracuse University. It has been accepted for inclusion in Dissertations - ALL by an authorized administrator of SURFACE at Syracuse University. For more information, please contact surface@syr.edu.
ABSTRACT

This thesis aims to examine the security of a blockchain’s communication network. A blockchain relies on a communication network to deliver transactions. Understanding and hardening the security of the communication network against Denial-of-Service (DoS) attacks are thus critical to the well-being of blockchain participants. Existing research has examined blockchain system security in various system components, including mining incentives, consensus protocols, and applications such as smart contracts. However, the security of a blockchain’s communication network remains understudied.

In practice, a blockchain’s communication network typically consists of three services: RPC service, P2P network, and mempool. This thesis examines each service’s designs and implementations, discovers vulnerabilities that lead to DoS attacks, and uncovers the P2P network topology. Through systematic evaluations and measurements, the thesis confirms that real-world network services in Ethereum are vulnerable to DoS attacks, leading to a potential collapse of the Ethereum ecosystem. Besides, the uncovered P2P network topology in Ethereum mainnet suggests that critical nodes adopt a biased neighbor selection strategy in the mainnet. Finally, to fix the discovered vulnerabilities, practical mitigation solutions are proposed in this thesis to harden the security of Ethereum’s communication network.
UNDERSTANDING AND HARDENING BLOCKCHAIN NETWORK SECURITY AGAINST DENIAL OF SERVICE ATTACKS

By

Kai Li
B.E., Nanjing University of Aeronautics and Astronautics, 2015

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Syracuse University
May 2022
ACKNOWLEDGMENTS

First and foremost, I am so fortunate to be advised by Prof. Yuzhe Tang and to be his Ph.D. student. As my advisor, he spent so much effort guiding me in research and life. From him, I learned how to conduct high-impact research, identify open problems, think from a high-level standing point, propose research ideas, and refine paper presentations. I couldn’t tell how much I enjoyed the time that we brainstormed and discussed research problems together. His passion for research has motivated me to pursue a Ph.D. degree, and his encouragement cultivated my confidence in dealing with failures and frustration. Without his invaluable support and guidance, I doubt that I could complete the degree and become an independent researcher.

I would like to extend my sincere gratitude to many of my collaborators. First, I learned and benefited greatly from the collaborations with researchers from other universities, including Prof. Jianliang Xu, Prof. Xiapu Luo, Prof. Xiaofeng Wang, Prof. Ting Chen, Dr. Cheng Xu, and Dr. Ben Kim. Their expertise has helped me gain the breadth yet the depth of knowledge in Computer Science. Besides, I am fortunate to work with many colleagues, including Jiaqi Chen, Yibo Wang, Yuxuan Zhou, Qi Zhang, Xianghong Liu, Zhehu Yuan, Qiwu Zou, Ju Chen, and others. The discussions and chats with them brought much fun to my Ph.D. study.

In addition, this thesis cannot be finished without the support of my dissertation defense committee members Prof. Joon S. Park, Prof. Wenliang Du, Prof. Vir V. Phoha, Prof. Jae C. Oh, and Prof. Bryan S. Kim. Their supportive and rigorous attitude has set an excellent example for me to pursue my future career in academia. Meanwhile, I also
appreciate the support from the faculty, staff, and students at Syracuse University.

None of my achievements would be possible without the love and support from my family. I want to convey my deepest appreciation to my parent and my younger sister. They have been so supportive and inspiring throughout my life. My parent always trusts me and gives me the opportunities to make decisions independently, and my sister always stands with me and shares with me her real feelings. I am thankful for their continuous love and trust.

Last but not least, I am proud to acknowledge my wife, Songjie Liu, for her continuous love and encouragement during my Ph.D. journey. Whenever I felt self-doubt and frustrated, she always showed her confidence in me and comforted me with so much patience. Without her accompany and effort in making this long journey enjoyable, I could not accomplish so much. Words cannot describe my admiration for her kindness, courage, and devotion. For these, I will be forever grateful. I look forward to our adventure on various trips together in the future.
# Table of Contents

**Acknowledgments** iv  
**List of Tables** x  
**List of Figures** xi  

## 1 Introduction 1  
1.1 Blockchain Communication Network 2  
1.2 Related Work 3  
1.3 Open Research Problem 3  
1.4 Overview of Research Findings 5  
1.5 Contributions 6  
1.6 Organization of Dissertation 7  
1.7 Bibliographic Notes 7  

## 2 Background 9  
2.1 Public Blockchain 9  
2.2 Ethereum Blockchain 9  
2.3 Ethereum Transactions 10  

## 3 Denial of RPC Service via Eth_call 12  
3.1 Introduction 12  
3.2 Preliminary: Blockchain as a DApp Platform 18
3.3 The DoERS Attack Exploiting Eth_call ........................................... 20
3.4 Measurements of Exploitability on RPC Services ......................... 22
   3.4.1 Measuring Blackbox Load Balancers: Methodology .................... 23
   3.4.2 Measurement Results: Load Balancers ................................... 25
   3.4.3 Attack Strategies Evading Load Balancer ................................. 28
   3.4.4 Measuring Gas Limits: Methodology .................................... 30
   3.4.5 Measurement Results: Gas & Rate Limits ................................. 30
   3.4.6 Attack Strategies Evading Gas Limit .................................... 31
   3.4.7 Summary of Attack Strategies ........................................... 31
3.5 Evaluation of DoERS Attacks ..................................................... 32
   3.5.1 Evaluation on Deployed Services ......................................... 33
   3.5.2 Evaluation on a Local Full Node ......................................... 38
   3.5.3 Evaluating DApp Response Time under DoERS ......................... 40
3.6 Countermeasures ......................................................................... 42
   3.6.1 Analyzing Known Countermeasures .................................... 42
   3.6.2 Proposed Countermeasures ................................................ 43
3.7 Related Work ............................................................................. 46
3.8 Responsible Disclosure ............................................................. 47
3.9 Summary .................................................................................... 48

4 Measurement Study on P2P Network Topology ............................... 49
   4.1 Introduction ............................................................................. 49
   4.2 Preliminary ............................................................................. 54
   4.3 Motivation: Significance of Knowing Blockchain Topology ............ 56
      4.3.1 Implication to Blockchain Security .................................... 56
      4.3.2 Implication to Blockchain Performance ................................. 57
   4.4 Related Work .......................................................................... 58
      4.4.1 TxProbe’s Applicability to Ethereum .................................... 60
4.5 TOPOSHOT Measurement Methods ........................................... 61
   4.5.1 Profiling Ethereum Clients’ Behavior .................................. 61
   4.5.2 Measurement Primitive .................................................... 64
   4.5.3 Parallel Measurement Framework ...................................... 69
4.6 Measurement Results .............................................................. 73
   4.6.1 Measurement Validation ................................................. 73
   4.6.2 Testnet Measurement Results ......................................... 76
   4.6.3 Mainnet Measurement Results ....................................... 80
4.7 Ethical Discussion ................................................................. 83
4.8 Summary .................................................................................. 83

5 Denial of Mempool Service at Low Cost ...................................... 85
   5.1 Introduction .......................................................................... 85
   5.2 Preliminary ........................................................................... 92
   5.3 Threat Model ....................................................................... 95
   5.4 DETER Attacks at a txpool .................................................... 96
      5.4.1 Observing txpool’s Eviction Behavior .............................. 97
      5.4.2 DETER-X: Exploit Future Transactions ......................... 99
      5.4.3 DETER-Z: Exploit Latent Invalid Transactions .............. 101
      5.4.4 Evaluation on a Local txpool ..................................... 103
   5.5 Attack Strategies at Miners ................................................... 104
      5.5.1 Proposed Strategies ..................................................... 104
      5.5.2 Evaluation on a Local Miner Node ................................. 107
   5.6 Measuring Deployed Networks’ Exploitability ....................... 109
      5.6.1 Design Rationale ......................................................... 109
      5.6.2 Measuring Testnets’ Exploitability .............................. 110
      5.6.3 Measuring Mainnet’s Exploitability ............................. 114
   5.7 Attack Mitigation ................................................................. 118
5.8 Related Work ................................................................. 121
5.9 Responsible Disclosure .................................................. 123
5.10 Summary ................................................................. 124

6 Conclusion and Future Research ........................................... 125

6.1 Summary ................................................................. 125
6.2 Other Research ........................................................ 126
  6.2.1 Cost Optimization on Smart Contract Data Feeds and Invocations  . . . . . . 126
  6.2.2 Enhanced Cloud Security by Blockchain  ........................................ 126
6.3 Future Research ........................................................ 127

References ................................................................. 128
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Characterizing the load balancing of RPC services.</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Minimally effective attack parameters.</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Existing works on blockchain topology measurement and TOPOSHOT’s distinction.</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Notations of parameters in mempool.</td>
<td>62</td>
</tr>
<tr>
<td>4.3</td>
<td>Profiling different Ethereum clients’ transaction eviction and replacement policies.</td>
<td>62</td>
</tr>
<tr>
<td>4.4</td>
<td>Graph properties of the Ropsten testnet</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>Detected communities in Ropsten testnet</td>
<td>76</td>
</tr>
<tr>
<td>4.6</td>
<td>Connections among critical nodes</td>
<td>82</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary of measurement studies on the testnets/mainnet.</td>
<td>83</td>
</tr>
<tr>
<td>5.1</td>
<td>Notations of parameters for observing txpool’s eviction behavior.</td>
<td>97</td>
</tr>
<tr>
<td>5.2</td>
<td>Profiling different Ethereum clients under tests and DETER attacks.</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>DETER &amp; baseline attacks</td>
<td>107</td>
</tr>
<tr>
<td>5.4</td>
<td>Gas price bumps in different Ethereum clients.</td>
<td>115</td>
</tr>
<tr>
<td>5.5</td>
<td>Critical mainnet nodes: Discoverability and exploitability.</td>
<td>118</td>
</tr>
<tr>
<td>5.6</td>
<td>Evaluation of miners’ revenue and security across mitigation schemes.</td>
<td>119</td>
</tr>
<tr>
<td>5.7</td>
<td>Transaction statistics during the evaluations on mitigation schemes.</td>
<td>120</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Typical workflow of users’ transactions in a blockchain</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>The system of blockchain RPC service</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>The exploitable smart contract to exhaust the computing resources.</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Measure Gas limit of an RPC node</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>RPC service model</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Benchmarks to characterize load balancing in an RPC service.</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Load balancing w.r.t. request timing</td>
<td>27</td>
</tr>
<tr>
<td>3.7</td>
<td>\texttt{exhaustCPUs} attacks to RPC services</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Targeted attacks to RPC services</td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td>A single-request attack exploiting \texttt{exhaustMem} to nodes without gas limits</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>Evaluate DoERS attacks on a local node</td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>DoERS attacks on a metamask-based DApp</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>P2P overlay networks on Ethereum.</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>\textsc{TOPOSHOT}'s measurement primitive.</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>\textsc{TOPOSHOT}'s parallel measurement protocol.</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Measurement validation results.</td>
<td>74</td>
</tr>
<tr>
<td>4.5</td>
<td>Speedup of \textsc{TOPOSHOT}'s parallel measurement over the serial measurement</td>
<td>75</td>
</tr>
<tr>
<td>4.6</td>
<td>Node degree distribution in Ropsten</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>Threat Model: DETER a remote mempool (\texttt{txpool})</td>
<td>86</td>
</tr>
<tr>
<td>5.2</td>
<td>Ethereum transaction workflow</td>
<td>93</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Public blockchains have shown to be a promising technique to support transactions between mutually untrusted parties. A public blockchain is a distributed system that stores a ledger of “transaction” history. Notable examples of public blockchains are Bitcoin, Ethereum, and EOS. Beneath the surface, a public blockchain is a peer-to-peer (P2P) network where different miner nodes join the network and agree on which transaction to be included in a block. The P2P network is designed to admit anyone on the Internet without identification (i.e., open membership) and to provide incentives in cryptocurrency rewards to miner nodes who “mine” in the network. Mining means that all nodes race to solve a puzzle, and the winner gains permission to include transactions in the next block. Once a transaction is included in a block, it is deemed confirmed and cannot be reverted after a certain number of blocks.\[1\]

In order to deliver transactions from users to miner nodes inside a blockchain, the blockchain must establish a communication network between these two parties to achieve the goal. The security of the communication network against any Denial-of-Service (DoS) attacks is of particular importance. For instance, if an attacker is able to cause failures to the communication network, it can corrupt the blockchain ecosystem and render the blockchain unable to process users’ transactions. To that end, this thesis focuses on examining the security of a blockchain’s communication

\[1\]6 blocks in Bitcoin and 25 blocks in Ethereum
network under DoS attacks.

1.1 Blockchain Communication Network

A blockchain relies on a communication network to deliver transactions. As depicted in Figure 1.1, on the one hand, users send their transactions (TX) to the communication network; on the other hand, miners receive transactions from the network and include them in a block. In practice, a blockchain’s communication network typically consists of three network services, namely an RPC (Remote Procedure Call) service, P2P network, and mempool.

Specifically, the RPC service (or transaction relay service) is a web service that handles users’ web requests by translating the requests into transactions and relays the transactions to the P2P network. After that, the P2P network propagates received transactions to miner nodes. Since miner nodes can be placed anywhere in the P2P network, every node in the network thus broadcasts the received transactions to their neighbors. Such propagation process repeats until all nodes have received users’ transactions. Once transactions arrive at each individual node, they are buffered in a data structure called mempool. In the last step, the miner accesses transactions from its hosting node’s mempool and includes them in the next block.

Fig. 1.1: Typical workflow of users’ transactions in a blockchain

We use “RPC service” and “transaction relay service”, interchangeably in this thesis.
1.2 Related Work

Given the wide use of blockchain and intensive investment in cryptocurrency, studying the security of blockchain systems is an important research topic, as shown in the existing literature.

Previous research has examined blockchain’s security at different system layers, including the P2P network [1, 2, 3, 4], mining incentives [5, 6, 7], transaction processing [8, 9, 10, 11], and applications such as smart contracts [12, 13, 14], etc. Among them, the most relevant work on the RPC service is a measurement study of cryptocurrency stealing attacks [15]. In this attack, an adversary exploits the time window between an account-unlocking request and a transaction-sending request so that she can gain unauthorized access to an unlocked account on an RPC service.

At the P2P network layer, existing work such as eclipse attack [1, 2] aims to isolate a victim node from the rest of the network, and routing attacks [3, 4] assume a powerful Internet Service Provider (ISP) or top-tier Autonomous System (AS) can manipulate the network traffic to partition the P2P network. At the mempool layer, Bitcoin stress testing [8] is a work that measures the impact of the 2015 Bitcoin Spam attacks. Recently discovered Front-running attacks [16, 17] show that an attacker can monitor unconfirmed transactions in the mempool and send high-priced transactions to front-run a victim transaction and gain illicit profit. Another form of attack on the mempool is Miner Extractable Value or MEV [18, 19, 16], which shows a miner can inject, exclude transactions, or re-order the transactions within a block to make profits.

1.3 Open Research Problem

Although existing work has studied the security of a blockchain at different system layers, the DoS security of the blockchain’s communication network remains to be systematically studied. This thesis aims to examine the security of blockchain’s communication network under DoS attacks. More specifically, we focus on the security of each network service involved in the communication network.

Studying the DoS security of a blockchain’s communication network is a critical research task.
because a secure and reliable communication network is vital to both blockchain users and miners. Considering that a blockchain’s communication network is disabled, the immediate consequence will be that transactions cannot be delivered to miners. Subsequent consequences are that users cannot make their transactions confirmed in the blockchain, and miners cannot make profits from including users’ transactions. Ultimately, the blockchain will lose both users and miners, which eventually renders the blockchain vulnerable to 51% attacks [20].

As illustrated in Figure 1.1, a blockchain’s communication network consists of three network services (RPC service, P2P network, and the mempool) that collaboratively deliver transactions between users and miners. Any failures in these network services can prevent the inclusion of users’ transactions. To study the DoS security of the blockchain’s communication network, we thus formulate the following three research questions:

1. How secure is the RPC service against DoS attacks?

2. How secure is the P2P network against DoS attacks?

3. How secure is the mempool against DoS attacks?

Specifically, the RPC service in Ethereum exposes many APIs to the public except for accepting transactions, such as accepting queries of blockchain state and speculative execution of transactions. The DoS security of these APIs remains to be studied. For the P2P network, obtaining the network topology is important to understanding its DoS security. For example, if the P2P network topology appears to be a star type of topology, such a network would particularly be vulnerable to DoS attacks because an attacker can concentrate her power to disable the central node in the network and launch network partitions or eclipse attacks [1, 2]. Existing work has uncovered Bitcoin, Monero’s network topology [21, 22]. However, to the best of our knowledge, uncovering Ethereum’s network topology is still an open research problem. Besides, although existing work has studied Bitcoin’s mempool [8] and discovered Front-running attacks in Ethereum’s mempool [16, 17], the DoS security of Ethereum’s mempool is yet to be examined.
1.4 Overview of Research Findings

This thesis aims to answer the three research questions listed in §1.3. Towards the goal, the thesis analyzes the security of each network service in the Ethereum communication network.

Specifically, in the RPC service, we examine its implementation of APIs that were exposed to the public. A vulnerable API \texttt{(eth\_call)} discovered can lead to a zero-cost DoS attack. Despite the initial purpose of supporting smart contract testing with no cost, this API can be misused to execute a resource-consuming smart contract that would exhaust the RPC service’s computing resource, leading to service outages to blockchain users. We name this attack as Denial of Ethereum RPC service or DoERs attack [23]. To assess the exploitability of the DoERs attack, we propose a novel method to characterize the RPC service’s load balancing policies. The measurement results on real-world RPC services reveal that they adopted a consistent load balance policy, which exaggerates the severity of the DoERs attack. We also propose mitigation strategies for RPC services to defend against the DoERs attack, including setting a limit on the resource usage by \texttt{eth\_call} and adopting an unpredictable yet consistency-preserving load balancing policy.

To study the security of the P2P network, obtaining the network topology is a critical research task. Towards obtaining the P2P network topology, we propose a novel technique named TopoShot [24]. TopoShot leverages the handling of replacement transactions in Ethereum to uncover the links in the network topology. We apply TopoShot to measure real Ethereum networks and obtain full topologies of testnets and sub-topology of the mainnet. The results in testnets show Ethereum networks are resilient to network partition attacks. The results in the mainnet suggest that critical services (e.g., RPC services, mining pools) adopted a biased neighbor selection strategy, implying a degree of centralization.

In the mempool, we inspect the admission control enforced by Ethereum mempool (or txpool). We discover the mempool adopted a risky design that prioritizes the miner’s revenue while relaxing the security constraint. For instance, it allows an invalid transaction of a high price to evict a valid transaction of a low price, hoping that the invalid transaction will turn into valid and increase the miner’s revenue. However, such a risky design makes the mempool vulnerable to DoS attacks. We
name these attacks Denial of Ethereum Txpool sErvice (DETER) attacks [25]. DETER can evict normal users’ transactions from mempool or prevent them from entering mempool, leading the miner to read invalid transactions and produce empty blocks. Compared to the baseline DoS attack in which the adversary sends high-priced valid transactions to fully occupy a block, DETER incurs much less cost to the adversary (zero-cost or low-cost). By evaluating DETER in local nodes, we verify DETER attacks have a low cost in all known Ethereum clients and major testnets. We also design non-trivial measurement methods in the mainnet and conduct probe tests to confirm that popular mainnet services are exploitable. We also propose mitigation schemes to reduce DETER attacks’ success rate down to zero while preserving the miner’s revenue.

1.5 Contributions

This thesis makes the following contributions.

**New DoS attacks:** We identify vulnerabilities and design flaws in Ethereum’s RPC services and mempool. Based on the identified vulnerabilities, novel DoS attacks are discovered, including DoERs and DETER attacks.

**New understandings of network topology:** We propose a novel technique TopoShot to uncover Ethereum’s P2P network topology. We apply TopoShot to real Ethereum networks and obtain full network topologies of testnets and sub-network topology of the mainnet. The results show testnets are resilient to network partition attacks, and critical services in the mainnet adopt a biased neighbor selection strategy.

**Systematic security evaluation:** We systematically evaluate the security of real-world network services in the Ethereum ecosystems under DoERs and DETER attacks. The results suggest that real-world network services are vulnerable, and an attacker can exploit the discovered vulnerabilities to corrupt the Ethereum ecosystems.

**Practical mitigation:** We propose practical solutions to mitigate the discovered vulnerabilities in Ethereum network services. For example, to mitigate the DoERs attack in RPC services, we
propose to enforce a limit on the resource usage when serving the API (eth_call) and suggest an unpredictable while consistency-preserving load balancing policy for RPC service operators. To mitigate DETER attacks in the mempool, we propose practical mitigation by preventing invalid transactions from evicting valid transactions. We also responsibly report all discovered vulnerabilities to affected network services and the Ethereum developer community, the vulnerabilities have been confirmed, and short-term fixes have been deployed.

1.6 Organization of Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 introduces the necessary background of the Ethereum blockchain. Chapter 3 presents the details of the vulnerable API discovered in the RPC services and security evaluation results in real-world services. Chapter 4 illustrates the novel technique proposed to uncover Ethereum’s P2P network topology and measurement results of real Ethereum networks. In Chapter 5 the discovered vulnerabilities in mempool and the identified DoS attacks and the evaluation results in real-world Ethereum networks are discussed. Chapter 6 concludes the thesis.

1.7 Bibliographic Notes

Most of the research work appearing in this dissertation has already been published at various venues and has appeared in the publications listed below.

Publications related to the thesis

Conference Papers:


• Kai Li, Y. Wang, and Y. Tang, "DETER: Denial of Ethereum Txpool sERvices", ACM CCS 2021.

Other related contributions

Conference Papers:


CHAPTER 2
BACKGROUND

2.1 Public Blockchain

A public blockchain is a distributed system that stores a ledger of “transaction” history on a P2P network. The P2P network is designed to scale, by admitting anyone on the Internet without identification (i.e., open membership) and by providing incentives in cryptocurrency reward to the nodes who “mine” in the network. Mining means that all participating nodes race to solve a puzzle and to decide which transactions to be included in the next block. Based on these mechanisms, real-world blockchains, including Bitcoin, Ethereum, EOS, etc., see a large operational P2P network (e.g., thousands to hundreds of thousands of peers) and enjoy a higher degree of trust decentralization than conventional systems.

2.2 Ethereum Blockchain

In addition to transactions of cryptocurrency, modern blockchains have evolved to support executing programs (a.k.a, smart contracts) and enabled a new computing paradigm, namely decentralized applications (DApp). A notable example is Ethereum, the largest smart contract blockchain as of this writing. It allows users to implement their logic in smart contracts and deploy them into
the blockchain. After that, users send transactions to trigger the smart contract execution, and all nodes in the P2P network will then validate the program executions and provide trustworthy results to the users.

**Smart contracts and Gas:** A smart contract is a user program running on the Ethereum blockchain. While Bitcoin’s contract, Script [26], is domain specific, more extensible blockchains, including Ethereum and EOS, support running Turing-complete smart contracts. On Ethereum, a client can request to run a smart contract with the provided arguments, if she pays a certain amount of fees known as Gas. The purpose of the Gas mechanism is to prevent denial of service to any Ethereum full nodes. A smart contract (more precisely, the bytecode of the contract) is replicated to all Ethereum nodes and the execution of a smart contract is triggered by a transaction that propagates the invocation information to the Ethereum network.

### 2.3 Ethereum Transactions

An Ethereum transaction binds a sender account to a receiver account, where an account is a public key of an Ether owner.

**Nonce:** Ethereum supports “hasty” transaction sending, that is, an account can send a transaction, say \( tx \), without waiting for the confirmation of the transaction \( tx \) depends on. To enforce a total-order among hastily sent transactions, each Ethereum transaction is associated with a nonce, which records a monotonically increasing counter value per each sender account. In a mempool, a transaction is of state pending, if its nonce equals one plus the maximal nonce of the transactions of the same sender in the mempool (i.e., equal to \( n + 1 \)). Otherwise, if the nonce is strictly larger than \( n + 1 \), the transaction is a future transaction. A future transaction can be a result of the Ethereum network propagating hasty transactions out-of-order.

**Gas price:** In Ethereum, each transaction needs to specify a Gas price, that is, the amount of Ether the sender is willing to pay to a miner for each unit of “work” it does for including the

---

1 The fiat currency in Ethereum
transaction into the blockchain. Here, the work refers to the basic transaction validation workload and that for executing the smart contract invoked by the transaction. The work unit is Gas. The higher the Gas price is, the faster the blockchain network executes the transaction.

**Validity and priority checks:** An unconfirmed Ethereum transaction goes through several mempool operations before being included in a block, such as admission, eviction, and replacement. In each of these operations, transaction \( tx \) is checked on various conditions to determine its validity and priority (against other competing transactions) on the admission to the next stage. Validity checks produce a binary decision, whether the transaction is deemed valid. Commonly, transaction validity is decided by size (e.g., does the amount of Gas it consumes exceed the block Gas limit?), the condition on double-spending or overdraft, the verifiability of its signature, etc. Priority checks produce a score that determines a transaction’s priority against other competing ones during the admission to the next stage. Commonly, an Ethereum transaction’s priority is based on its Gas price; the transaction with a higher price can evict the one of lower prices from the mempool.

In Ethereum, only a *pending* transaction admitted into mempool will be further propagated by an Ethereum node to its neighbors. In addition, only *pending* transactions are available for miners to include in the next block and their transaction fees are collectible to miners.
CHAPTER 3
DENIAL OF RPC SERVICE VIA ETH_CALL

3.1 Introduction

With the advent of operational blockchains, decentralized applications (DApps) running atop these systems are gaining popularity, providing decentralized finances (DeFi), online gaming, information-security infrastructures, etc. A typical DApp, as illustrated in Figure 3.1 is archi-
tected in three layers: DApp clients running inside web browsers send requests to a Remote Procedure Call (RPC) service that translates the clients’ requests to cryptocurrency transactions or queries to a blockchain P2P network. Such an RPC service operates on a blockchain full node maintained directly by the DApp owner (i.e., an in-house RPC node) or a set of nodes hosted by a third party (i.e., a third-party RPC service) intended to ease DApp deployment. Given the ever-
growing blockchain states (e.g., 130 GB and 1.8 TB for a fully synced and an archived Ethereum node, respectively, as of 2018), the RPC service plays an increasingly important role in the DApp ecosystem, scaling DApp clients to low-end mobile devices and web browsers. Major blockchains today flock to roll out RPC supports, which spawn a good number of services in practice, in-
cluding nine service providers (as is evaluated in this work) supporting the Ethereum’s JSON-
RPC interface [27], blockchain.info [28] with Bitcoin’s JSON-RPC [29], dfuse.io [30] and greymass.com [31] with EOSIO’s Chain API [32], stellar.org [33] with Stellar Horizon [34],
etc. These services host the majority of DApps; for instance, at least 63% of Ethereum based DApps use one RPC service [35].

![Diagram of blockchain RPC service]

Fig. 3.1: The system of blockchain RPC service

Despite its importance, the RPC service is less decentralized (one to hundreds of nodes) than the blockchain network (of tens to hundreds of thousands of nodes) and therefore could become a single point of failure should a denial of service (DoS) attack happen, which could lead to the collapse of the whole DApp ecosystem. It is important to note that DoS is known to pose a significant threat to the blockchain ecosystem, particularly in Bitcoin exchanges and mining pools [36, 37, 38]. We believe such an attack can also be launched against a victim RPC service, allowing a service competitor to steal customers from the victim. Besides, the perpetrator who denies an RPC service can illicitly manipulate the transaction order of a financial DApp [39, 16] and gain profit. For instance, in an auction for registering an Ethereum domain [40] or purchasing a CryptoKitty [41], a bidder can delay others’ bidding transactions through denying the RPC service they use to win the auction at an unfairly low price. As another example, a client depositing to a hash-time-lock contract (HTLC), is widely used in blockchain applications (e.g., atomic intra-chain or cross-chain swaps [42] and payment channels [43, 44]), can defer the withdrawal of the deposit after the expiration of the lock (so-called grieving [39]) by denying the RPC service used by the withdrawal, thus retaining the deposit. With such significant implications, this security risk, however, has never been studied before.

Menace of DoERS. Our research shows that indeed this risk is realistic and serious: today’s RPC services are vulnerable and can be easily disabled by a new type of DoS attacks (which we call DoERS or Denial of Ethereum Rpc Service) that exploit the free execution capability they expose. More specifically, blockchain systems support the Gas-free execution of a smart contract on an individual RPC node, such as Ethereum’s eth_call RPC [45] (and eth_estimateGas [46] running the same code path). An eth_call can be triggered to run any smart contracts, including those...
reading and/or updating their states. Unlike the transaction-triggered smart contract execution, the `eth_call`-triggered execution occurs locally on the recipient RPC node, and its state update, if any, will not be propagated to or reflected in the global blockchain state. The purpose of such a capability is to enable a variety of real applications in pre-production contract testing (e.g., estimating the smart contract cost before DApp deployment by `estimateGas`), “stored-procedure” like database analytics on blockchain (e.g., the GraphQL queries [47] and decentralized financial analysis [48]), and others. Most importantly, `eth_call` is often free: the charge for its execution is not mandated by Ethereum and instead left to individual RPC services, which tend to waive the expense for attracting DApp clients, as observed in practice. Therefore, the adversary can deploy on the blockchain an attack smart contract involving a resource-consuming procedure (e.g., an infinite loop of hashing computations) and then trigger it through `eth_call`. This attack is shown in our research to effectively stop a node from performing critical operations for all DApps it hosts, including block/transaction synchronization, serving RPC requests, etc.

Notably, DoERS is different from other DoS attacks on the blockchain network, as studied in the prior research [49, 50, 51, 5]. First, it aims at disrupting the communication channel between a blockchain and its DApps by blocking third-party RPC services (in Figure [3.1]), not taking down the blockchain itself as the other attacks do. Second, our attack exploits a unique weakness – Gas-free contract execution on RPC-enabled Ethereum nodes (Section 3.2), while existing DoS attacks seek under-priced instructions for attacking replicated smart contract execution [49, 50, 51] or misusing mining mechanisms [5].

The attack is non-trivial to carry out. We need to overcome the protection already in place on each Ethereum node, such as limiting each call’s Gas (i.e., Gas limit) and time (i.e., timeout), through strategically delivering continuous queries at an alarmingly low rate below the victim’s rate limit (§ 3.3). Also, it is less clear how extensively RPC interfaces are open to the public on the nodes operated by the DApp owners. Even more challenging is the use of third-party RPC services, which typically run a load balancer in front of RPC nodes. Such a balancer hides the node(s) serving a specific DApp and spreads out its clients’ requests using undisclosed strategies.
Understanding how it works is critical to the success of an attack targeting a specific DApp or a specific client of the DApp. For this purpose, we perform an analysis and measurement study on Ethereum.

**Measurement and findings.** More specifically, there are three types of nodes in an Ethereum network: the nodes not accepting any RPC requests (non-RPC nodes), the nodes with public RPC ports, responding to the requests from any web clients (public RPC nodes), and the nodes with private RPC ports, only communicating with specific web servers (private RPC nodes). The majority of the private nodes are the backends of third-party RPC services such as SrvR1[1] and SrvR6. Since most DApps rely on these well-established services to connect to Ethereum [35], such private nodes play a critical role in controlling connections between DApp clients and the Ethereum network. Thus, our research focuses on measuring these private nodes.

We first detect the presence of Gas limits on the private RPC nodes in nine leading RPC services. We propose a detection technique that makes `eth_call`S with varying Gas amounts and performs a binary search to find out the Gas limit. The measurement reveals that five out of the nine major RPC services do not configure Gas limits of any kind and the other four set a rather nonrestrictive limit of more than 1.5 block gas.

Further, we look into the load balancers deployed by the nine third-party RPC services. To reverse-engineer their operations, we develop a novel probing technique based upon *orphan transaction*, which stays on one node without being propagated to others. Our approach delivers one orphan transaction through a given RPC service and then sends in the second one that attempts to double-spend the first. If both are assigned to the same node by the balancer, the second transaction will fail (as it double-spends the first one), and otherwise, it will also go through (as the two transactions reside on different nodes). By observing the transaction outcome, we systematically analyze 9 popular services (§ 3.4.1). Our study reveals different load-balancing strategies, assigning requests to nodes according to the client’s IP (e.g., SrvR6), service API key (e.g., SrvR5), or timings of RPC calls (e.g., SrvR1). Based upon the discoveries, a DoERS attack can be adjusted

---

[1] In this chapter, we refer to the nine services by numbered names from ServiceX1 to ServiceX9. We intentionally avoid use their real names to protect their identities and operations.
to target a DApp, a client or the client’s visit to a given DApp, depending on their RPC services’ balancing strategies (§3.4.1). Among all 9 services, we conclude that 5 can be exploited to block a specific client or a DApp, without fully taking down the whole services. Our study of Gas limit reveals that another five RPC services out of the nine don’t configure any Gas limit. Also interesting is our measurement of rate limits, which turns out to be less aligned with the public information (e.g., the measured rate limit of SrvR6 is twice the one published on their website).

**Attacks.** In addition to the above targeted attacks, we design DoERS strategies that are specific to the measurement findings. For nodes without Gas limits, DoERS strategically sends a single RPC request exploiting the Ethereum Virtual Machine’s (EVM’s) CODECOPY instruction such that it evades all other known protections (in timeout, load balancing and rate limiting) until it crashes the victim node. This is caused by the EVM design of atomically executing instructions that even a thrown timeout cannot interrupt. For nodes with Gas limits, DoERS sets the “payload” size of each RPC call below the specific Gas limit and increases the request rate to cause visible damage. Because of the innate computation asymmetry between sending an RPC on the client and executing programs on the RPC node, the required request rate remains low, as observed in our evaluation.

To verify the impacts DoERS will have on real Ethereum peers and services, we perform a carefully-designed experiment on these services, in a way the effectiveness of the attacks can be observed without significantly downgrading their services (§3.5.1). Our study reveals that all nine tested services are vulnerable to the DoERS attacks that cause noticeable performance degradation, increasing the service latency by $2.1X \sim 50X$. Notably, sending a single DoERS request can cause the latency to increase by $10X$ ($30X$) on SrvR6 (SrvR4), without triggering any exceptions. In addition, the evaluation study verifies that the proposed attack strategies can effectively evade the deterministic load balancing in services, such as SrvR6 and SrvR5.

We conduct experiments on an Ethereum peer under our control. The controlled experiments allow us to explore more extensively the combinations of attack parameters and to evaluate the effectiveness of attacks in the presence of out-of-gas, timeout and other exceptions. The study

---

We will use “nodes” and “peers” interchangeably in this chapter.
shows the attack can slow down block synchronization, in addition to causing latency increases. Particularly, we found sending DoERS RPCs at a rate as low as 150 per second under a very restrictive limit of 0.65 block gas was adequate to slow down block synchronization on a blade-server class machine by 91%.

**Mitigation.** Mitigating the DoERS threat without undermining the usability of the RPC service is nontrivial. For example, setting a low Gas limit for each request *alone* does not work well, as the protection can still be evaded by the attack using multiple DoERS requests. An exceedingly low Gas limit could also complicate the design of a DApp and downgrade its usability [52]. Indeed, we observe in our measurement study that no RPC service adopts a Gas limit below 1.5 block gas. So instead of capping the Gas usage for each request, we propose to limit the Gas for each service client, to defeat the multi-request attack. For this purpose, we address the challenge of identifying the requests from a specific client in an open-membership scenario using its performance profile and further develop the technique to capture performance anomalies.

Another direction is to make the behavior of the frontend load balancer unpredictable (independently assigning each request to a randomly selected peer) so as to weaken a DoERS attack on a specific set of backend peers. The challenge here is that the balancer for today’s DApp processes both RPC and transaction requests, and the latter may require certain dependency relations to be preserved: e.g., an ERC20 token’s *approve* call and a subsequent *transferFrom* call should be assigned to the same peer to keep their order. This challenge has been addressed in our research with a DoERS-secure load balancer that differentiates transaction requests from RPC queries (including *eth_call*), so that the queries that could lead to DoERS are distributed independently across peers while the transaction requests are handled under their dependency constraints (e.g., order preservation).

In general, DoERS is enabled by an open-membership RPC service that allows for free execution of arbitrary smart contract programs on its peers shared by different DApps. Invalidating any condition here can defeat the attack, but may also affect the fundamental security-usability trade-off expected from practical protection. Such trade-offs have been extensively dis-
cussed in this work. Our proposed mitigation techniques can be built into an RPC service and are transparent to its clients (§ 3.6).

Contributions. The contributions of this work are outlined as follows:

- **New attack.** We identify a new denial of service weakness in today’s blockchain, showing that the widely existing free query calls enable potential resource depletion attacks on RPC services, the weakest link of the DApp ecosystem. Also we implement the attack on Ethereum incurring zero Ether costs and demonstrate the real-world impact of the threat across leading RPC services.

- **New understanding.** We perform a systematic measurement study on the nine leading RPC services that control the connection between most DApp clients and the Ethereum network. Our measurement on leading RPC services’ load balancers, using a novel orphan-transaction based prober, has brought to light the hidden strategies they take, which enables targeted attacks on DApps and clients they serve.

- **Mitigation.** We also study the potential mitigation on the new threat, identifying a few promising solutions, including the ones that selectively penalize the DApps or clients consuming a large amount of resource on a node.

Roadmap. The rest of the chapter is organized as follows: the preliminary is introduced in Section 3.2, the research formulation is presented in Section 3.3, the measurement of DoERS exploitability among RPC services are presented in Section 3.4, attack evaluation is presented in Section 3.5, countermeasures are described in Section 3.6. Related works are presented in Section 5.8 before discussion on responsible disclosure in Section 5.9 and conclusion in Section 5.10.

3.2 Preliminary: Blockchain as a DApp Platform

DApp platform: The blockchain is widely used as a source of trust decentralization and underpins today’s decentralized web applications, known as DApps. A DApp is typically a javascript program residing in a webpage that accesses information on the blockchain by invoking DApp-specific smart contracts. For instance, the CryptoKitties DApp [41] is a market for digital pets sales. Its
system consists of an off-chain website that runs DApp javascript code [41] and five smart contracts on the blockchain [53]. Likewise, Melon terminal [54] is a financial DApp that runs financial-analysis in smart contracts on Ethereum and presents statistics on an off-chain webpage.

**RPC services:** To bridge the DApp web clients and blockchain, remote procedure call (RPC) services are essential. An RPC service accepts the JSON requests sent from a DApp client inside a web browser and translates them into queries or transactions. To do so, an RPC service internally runs one or a group of blockchain full nodes. Ethereum’s RPC interface [27] includes 43 open queries and 3 privileged operations such as `sendTransaction`. A valid privileged operation must be sent from a cryptocurrency owner for signing the operation using her private key, while an RPC query can be open membership in that it can be sent by anyone on the Internet without identification. Unlike transactions propagated to the blockchain network, an RPC query is served locally within the service.

**Speculative smart contract execution (eth_call):** The particular capability of interest to DoERS is Ethereum’s speculative smart contract execution in `eth_call` (and `eth_estimateGas` on the same code path in both Ethereum clients Geth [55] and Parity [56]). `eth_call` speculatively runs any smart contracts, in a different way from the conventional contract execution triggered by transactions. The difference is two-fold: 1) `eth_call`-triggered execution runs only on the serving RPC node and is not being propagated. This capability applies not only to the so-called pure/view functions [57], but also to any state-updating functions, as we tested in Ethereum Virtual Machine (EVM). The updated state however is not propagated and not reflected in the global blockchain state (hence the name, speculative execution). 2) `eth_call` does not mandate charging fees from a contract execution. Instead, such a decision is left to the hands of a service provider. In most practical RPC services, `eth_call` is offered free, as a means to attract DApp clients and developers, which is essential for growing their customer base.

Gas limit is a feature in Ethereum clients (e.g., Geth and Parity) that bounds the amount of Gas an individual `eth_call` invocation can consume.
3.3 The DoERS Attack Exploiting Eth_call

Our threat model involves three actors: an attacker sends one or multiple malicious, crafted RPC requests to an Ethereum RPC service that also serves the regular RPC requests from a benign client. In practice, the benign client can be a DApp. The goal of the attacker is to deny the RPC service to the benign client, for instance, increasing its RPC response time. The Ethereum RPC service can be a single Ethereum node choosing to accept RPC requests (the basic model) or a group of Ethereum nodes behind a frontend infrastructure (e.g., load balancing) to accept RPC requests (the third-party service model). This section considers the basic setting while the third-party service model is presented in §3.4.

```
1 contract DoERS-C {
2  function exhaustCPU(uint256 payload_size1) public returns (bool) {
3    bytes32 target = 0xf...f;
4    for (uint256 i = 0; i < payload_size1; ++i) {
5      target = keccak256(abi.encodePacked(target));
6    return true ;}
7    bytes32[] storage;
8  function exhaustIO(uint256 payload_size2) public returns (bool) {
9    for (uint256 j = 0; j < payload_size2; ++j) {
10       storage.push(0xf...f);
11    return true ;}
12  function exhaustMem(uint256 payload_size3) pure public returns (bool) {
13     bytes32[] memory mem = new bytes32[](payload_size3);
14     mem[payload_size3-1] = 0xf...f;
15     // "CODECOPY" allocate memory
16     return true ;}
}
```

Fig. 3.2: The exploitable smart contract to exhaust the computing resources (in CPU, memory allocation, etc.) of the victim node

The DoERS attack is constructed based on an exploitable smart contract that contains resource-consuming procedures. In this chapter, we use the DoERS-C contract in Figure 3.2 as an example, while there can be many alternative designs — how to design the most “effective” smart contract for the attack is out of the scope of this chapter. Contract DoERS-C includes three

```
1 float rpc_gasLimit(IP rpcNode) {
2  int lengthLower=0; int lengthUpper =500; //0/500 block gas
3  while (lengthUpper - lengthLower > 1){
4    arrayLength = (lengthLower + lengthUpper) / 2;
5    try{
6      rpcNode.eth_call(exhaustMem, arrayLength)
7    } (Exception e) {
8      if(e instanceof OutofGasException){
9        lengthUpper = arrayLength;
10      } else { //no gas limits
11        return 0;}
12    } else {
13      lengthLower = arrayLength;}}
14  return localNode.estimateGas(exhaustMem, arrayLength);
}
```

Fig. 3.3: Measure Gas limit of an RPC node.
exploitable functions that aim at depleting CPU, memory and IO resources, respectively, on the victim node. Specifically, function `exhaustCPU` runs a loop of hashing computation. Function `exhaustIO` runs a loop of storage updates in order to incur IO operations; note that Variable `storage` is persisted in the smart contract’s storage. Function `exhaustMem` runs a single operation (EVM instruction `CODECOPY`) to allocate a large array in memory. The three functions all take an argument called payload size, which controls the number of iterations of the loop (in `exhaustCPU` and `exhaustIO`) and the size of the array (in `exhaustMem`). This argument is essentially a knob for tuning the level of resource consumption incurred by the smart contract.

The DoERS attack is executed in two steps: 1) The attacker client deploys the DoERS–C smart contract to Ethereum by sending a transaction. This step costs a small amount of Ether. 2) The attacker sends one or multiple `eth_call` RPCs to the victim node to trigger one of the three `exhaustXX` functions in DoERS–C. By specifying a large payload size, the execution of these functions incurs a large amount of resource consumption on the victim node. The purpose here is to cripple the node’s functionality in block/transaction synchronization, serving co-siding RPCs, blockchain mining, etc. Since `eth_call` does not charge Ether (the main currency unit of Ethereum), the cost of the attack is low. We also describe a zero-Ether DoERS in §3.6.1 that eliminates the Ether cost in the first step.

In practice, the configurations of Ethereum nodes may thwart the above basic attack. For instance, Ethereum’s Gas limit, if configured, would limit the amount of computation that can be incurred by each DoERS request. To evade the protection, a sophisticated attacker should lower the payload size to avoid triggering the Gas limit, and instead send multiple such smaller DoERS requests at a certain rate to make the service unavailable to the victim. Also, other protective measures could be in place to raise the bar for a successful DoERS attack, such as timeout, rate limiting, load balancing, as well as other unknown mechanisms inside the black box RPC services (e.g., performance isolation, hypothetically). Based upon this observation, we set the goal of our research as follows:

**The goal of our research** is to understand the risk of DoERS across deployed Ethereum RPC
nodes and services. Particularly, we analyze the private nodes serving the backend of third-party RPC services to measure their susceptibility to the attack, motivated by the fact that most DApp clients are connected to the Ethereum network through such third-party RPC services [35].

Towards the goal, 1) we conduct a systematic measurement study on nine leading RPC services on the market to analyze the behaviors of their load balancers, Gas limits and rate limits; 2) we design the strategies that evade the protection discovered, in order to make the attack more effective (§ 3.4) and 3) we evaluate the impacts of our low-cost strategies on existing services and local nodes (§ 3.5).

3.4 Measurements of Exploitability on RPC Services

This section describes our measurement study including methodology and results on real-world third-party services. Our goal is to understand the internal of a blackbox RPC service by measuring service features in load balancing, Gas limits and rate limiting, as modeled next.

**Modeling an RPC service**: An RPC service runs web servers on the frontend to accept JSON-RPC requests and runs several Ethereum RPC nodes on the backend to process those requests.
Each frontend web server may run rate-limiting and load balancing on the received requests. The service model is illustrated in Figure 3.4.

3.4.1 Measuring Blackbox Load Balancers: Methodology

Goals

To characterize a load balancer, we first describe a detailed model. A load balancer receives JSON-RPC requests sent from DApp clients’ web browsers. The DApp of a JSON-RPC request is identified by one or a few API keys. The JSON-RPC request can also be identified by the IP address where the browser resides. Given an incoming request, the load balancer makes a decision regarding which RPC peer on the service backend should the request be forwarded to. The goal here is to characterize a load balancer in terms of its forwarding policy. Specifically, we aim at answering the following questions:

LB0. Given two RPC queries from the same IP and with the same API key, does the load balancer forward them to the same RPC peer?

LB1. Given two RPC queries with different API keys, does the load balancer forward them to the same RPC peer?

LB2. Given two RPC queries from different IPs, does the load balancer forward them to the same RPC peer?

LB3. Given two RPC queries with the same API keys and same IP but sent with $TT$ seconds apart, does the load balancer forward them to the same RPC peer?

Methods

The key technique to enable answering the above questions is whether one can detect the presence of a load balancer. Specifically, given two incoming RPC requests, the presence of a load balancer entails the two requests being forwarded to different RPC peers in the service backend.
Design rational: To detect load balancing in a blackbox service, our key idea is to exploit the way that Ethereum clients including both Geth and Parity handle orphan transactions. Recall that each Ethereum transaction is associated with a count, called nonce, from its issuing client. Given the nonce of the latest transaction of a client, an orphan transaction is a transaction sent from the same client and with a nonce no smaller than nonce + 2. An Ethereum peer receiving an orphan transaction handles it in the following manner: It will store the transaction locally and evict it under one of the two conditions: 1) If a transaction with nonce + 1 is received, the orphan transaction becomes unorphaned and, together with the transaction of nonce + 1, will be propagated to the entire P2P network. 2) If no transaction with nonce + 1 is received, the peer will drop the transaction after a timeout, say $O_t$. A subtle fact is that an orphan transaction can be replayed and updated before it becomes unorphaned. Specifically, an Ethereum peer, upon receiving two orphan transactions of the same nonce, will fail the second one if its gas price is lower than 110% of the gas price of the first transaction. In an RPC service, if one can send such two orphan transactions and observe the success of the second transaction, she can infer whether there is a load balancer forwarding the second transaction to a different backend peer than the first one.

Measurement mechanisms: Based on the above idea, we design a benchmark program to detect the presence of a load balancer inside a blackbox service. The program, namely detectLB_byOrphan in Figure 3.5, works as follows: It first sends an orphan transaction with nonce + 2 and gas price to a target RPC service (nonce is the nonce of latest confirmed transaction) and observes the returned hash txHash (Line 3). It then sends the second orphan transaction, with the same nonce + 2 but paying gas price − 1. If the second transaction fails (Line 4), it implies the second transaction is forwarded to the same backend peer as the first transaction; no load balancing is detected. Then, it further sends RPC queries to eth_getTransactionByHash(txHash) (Line 7). After waiting for a time period specified in the argument stall, it sends the second eth_getTransactionByHash(txHash) RPC query (Line 9). If both getTransaction(txHash) RPCs return successfully, it means no load balancing is detected for RPC queries. What's noteworthy in our benchmark design is that we additionally require sending two RPC queries at different
time points to confirm the absence of load balancing. In our preliminary design without the two 
getTransaction(txHash) queries, we found certain RPC services may exercise different load-
balancing policies for different types of RPCs (i.e., privileged RPCs like sendTransaction() and 
open queries like getTransaction(txHash)).

To double-check the measurement result, we design a second detection mechanism based on 
RPC queries getBlockNumber. The benchmark, namely detectLB_byBlockNo in Figure 3.5 
works as follows: It sends a series of getBlockNumber queries, one every two seconds, to a target 
RPC service and observes the sequence of block numbers returned. If an “anomaly” is detected, 
it implies the presence of a load balancer in RPC queries. Here, the anomaly is defined as a 
getBlockNumber query sent earlier in time that returns a block number larger than a later query. 
This reasoning here is that if all getBlockNumber queries are forwarded to the same RPC peer, 
the block number returned should monotonically increase with time.

The purpose of using two benchmarks is to complement each other (either confirm or dispute 
the results of each other), as the detectLB_byOrphan can be accurate on the case of asserting no 
load balancing and detectLB_byBlockNo can be accurate on the case of detecting load balancing.

3.4.2 Measurement Results: Load Balancers

We conduct a series of experiments in order to answer questions LB0, LB1, LB2 and LB3.

For LB0, we set up the benchmark programs in such a way that all RPC requests are sent 
out with a single API key and from a single client (of a single IP). We run both benchmark pro-
grams, detectLB_byOrphan and detectLB_byBlockNo, against the nine RPC services. For each 
service, we collect the results (true or false) of the two benchmarks and crosscheck them before 
determining whether load balancing is present. In particular, we set the RPC rate (interval in 
benchmark detectLB_byBlockNo) to the maximal value right below the service rate limit (see the 
measurement result in § 3.4.5).

In experiments, the two benchmarks mostly agree with each other. That is, when 
detectLB_byOrphan detects load balancing (no load balancing), detectLB_byBlockNo con-
firms the same. The only exception is SrvR9, on which detectLB_byOrphan detects load balancing but detectLB_byBlockNo does not (i.e., no observed case of decreased block numbers). detectLB_byBlockNo on SrvR9 also return many failed results (i.e., block numbers being 0) which may be the culprit of the inaccuracy.

For LB1, we send RPC requests with two API keys and from the same client IP. For LB2, the RPC queries are sent with the same API key and from two client IPs. In the two cases, the different RPC requests apply to Line 7 and Line 9 in Figure 3.5 for detectLB_byOrphan. In detectLB_byBlockNo, each step would send out two different RPC queries at the same time, and the “block-number-decreased” anomaly is detected on the combined sequences of the two query results.

<table>
<thead>
<tr>
<th>Type</th>
<th>RPC services</th>
<th>1IP-1key (LB0)</th>
<th>1IP-2key (LB1)</th>
<th>2IP-1key (LB2)</th>
<th>Gas</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>SrvR2</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>SrvR3</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>SrvR4</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>50</td>
</tr>
<tr>
<td>ii</td>
<td>SrvR5</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>SrvR6</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>iii</td>
<td>SrvR1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>SrvR7</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>SrvR9</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td></td>
<td>SrvR8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Table 3.1: Characterizing the load balancing of RPC services (✗ means no load balancing detected or no gas limit, either making the service exploitable. XIP-Ykey means sending two requests from X IPs and with Y API keys to detect load balancing.)

**Results:** The measurement results are presented in Table 3.1. We find three types of load-balancing behaviors w.r.t. LB0, LB1 and LB2: Type i) No load balancing of any sort, represented by SrvR2, SrvR4 and SrvR3, (ii) Deterministic load balancing that entails two subtypes: Type ii-a) No load balancing detected when RPC queries are sent with the same API key and from different IPs; this is represented by SrvR5, Type ii-b) No load balancing detected when RPC queries are sent from the same IP but with different API keys; this is represented by SrvR6, and Type iii) Comprehensive load balancing detected, when RPCs are sent from the same IP and with the same
API key; this is represented by SrvR1, SrvR9, SrvR7 and SrvR8.

**LB3:** We further conduct a measurement study for LB3; we design a simple benchmark to do so, which issues a series of `sendTransaction` RPCs, with each two $TT$ seconds apart. We run the benchmark against all services with load balancing (i.e., except for Type-i services). Most load balancers do not exhibit dependency on timing. The exceptions are SrvR1 and SrvR9 gateway.

**Results:** In SrvR1, with $TT = 5$ seconds, we consistently observe the following behavior of SrvR1’s load balancer: The decision which backend peer to forward a request to is made based on the timing of the request. The experimental result is reported in Figure 3.6a. The result shows a four-minute period (where our raw result is more than an hour) when a series of transactions from the same `from` address are sent to SrvR1. Every minute SrvR1’s load balancer will forward a transaction to a new peer if the transaction is the first one after the 0th, 10th, or 15th second in the minute. All transactions sent between 15th and 60th seconds of a minute will end up with the same three backend peers in that minute.

The result of SrvR9 is illustrated in Figure 3.6b, which is measured under 10 RPCs per second. In the first 25 seconds of any minute, there is a successful orphan transaction every 5 seconds, Here, it requires the timeline is aligned with the Unix timestamp.
implying a new backend peer is allocated. After that, orphan transactions keep failing until the 40th second.

The deterministic behavior of real load balancers discovered in our research apparently serves an important purpose – maintaining consistency across dependent transactions of DApps: for instance, the order between an approval and a subsequent transferFrom of an ERC20 token should be preserved, so the load balancer always forwards these requests to the same backend node. This property, however, can be exploited to concentrate DoERS attack payloads on a small set of nodes to enhance their effectiveness, as elaborated in §3.4.3.

3.4.3 Attack Strategies Evading Load Balancer

An attacker can leverage the above measurement results to adjust an DoERS attack to specific services. Here, we present sample attack strategies specific to Type-ii and Type-iii services (note that Type-i service essentially runs no load balancer and can be attacked in a straightforward way).

Targeted Attacks to Type-ii Services

The deterministic behavior of a Type-ii load balancer can be exploited to launch a DoERS attack targeted at specific DApp victims. We propose strategies that a DoERS attacker can use to select victims adaptively to the service types, namely Type-ii(a) and Type-ii(b) services.

Targeted attack to Type-ii(a) services: Recall that a DApp web client commonly sends requests to an RPC service, using API keys. As the API key has to be disclosed on DApp websites to all visiting browsers, the DoERS attacker can easily obtain the API key. The attacker then sends DoERS requests with the API key to the service. Recall that a Type-ii(a) service forwards requests with the same API key to the same peer, despite which IPs they are sent from. Thus, the DoERS requests will be processed by the same nodes serving other requests of the same API key. By this means, the attacker can disable the RPC node and further delay the service to other clients of the same DApp. Therefore, the DoERS attack can disable all clients of a victim DApp.
Targeted attack to Type-ii(b) services: Initially, the attacker prepares a “malware” token contract, called M-Token, which encodes the exhaust programs in DoERS-C. For instance, the balance function in the token internally calls exhaustCPU(1000000).

The attacker distributes the malware token M-Token to victim DApp clients. To do so, the attacker can set up a token faucet similar to gitcoin [58], that gives away free M-Tokens and, as a honeypot, attracts victim owners.

Later the victim owner may open her wallet DApp as usual. She will be surprised to find her DApp webpage unresponsive, because the webpage sending RPC requests to a service would make the service run M-Token’s balance function and get stuck. Furthermore, not only M-Token’s balance is not viewable on the victim owner’s webpage, but also the balances of other benign tokens are not responding. Because both the benign RPCs (to run benign tokens’ balance) and the M-Token’s RPCs are sent from the same browser, thus the same IP, the Type-ii(b) service forwards them to the same backend peer. By this means, the M-Token RPCs can denial-of-service the benign tokens’ RPCs.

Exploiting Timing Dependency to Attack Type-iii Services

Recall our measurement results in § 3.4.1 that the load balancers of RPC services exhibit timing dependency: if two requests are sent close in time, the balancer forwards them to the same backend peer, for purposes such as preserving the ordering between the two requests. This predictable behavior can be exploited to direct DoERS requests to just a few peers, undermining their services to some DApps without saturating the entire service backend. This results in a low cost and more effective attack on multi-node RPC services.

Specifically, consider SrvR1 as an example. As revealed from our measurement study (§ 3.4.1), SrvR1’s load balancer forwards all incoming requests received within a minute (with time aligned) to at most three distinct backend peers. So an attacker can exploit this timing dependency to send all its DoERS requests in one minute to land on three specific nodes, which can effectively deny their services to DApps. Particularly, if the attacker knows when a specific DApp or its client issues
requests (e.g., through eavesdropping on its communication or aiming at a known auction deadline when bids would come in), he could produce a few attack requests within the 1-minute window to block the three backend peers serving the DApp. This strategy enables a low-cost attack in which one does not need to overload hundreds of backend peers (e.g., more than 192 peers behind SrvR1), which is very expensive, to undermine the service to some DApps and their clients. Note that such a “flash attack” (e.g., one minute for SrvR1) can still have serious consequences, e.g., frontrunning a competing bid in a decentralized auction. The effectiveness of this attack is evaluated in §8.

3.4.4 Measuring Gas Limits: Methodology

Given an RPC service, the goal is to test the presence of any Gas limit configured on the service’s backend peers. Our test program, named by rpc_gasLimit, is in Figure 3.3. The goal of the test program is to find the maximal argument (arrayLength in function exhaustMem()) that does not trigger the out-of-gas exception, a value that implies the Gas limit. To do so, the program starts with an initial guess on the target arrayLength value, then grows the guess exponentially until the first exception is observed. It then enters the second phase that binary-search the Gas-limit corresponding value of arrayLength. After the target value $V$ is obtained, the program then uses a local RPC node (under our control) to run estimateGas() with function exhaustMem under $V$. The returned value is the Gas limit. Note that our design uses exhaustMem function which consumes Gas faster than the other two exhaustXXs and can finish before Ethereum’s default 5-second timeout.

3.4.5 Measurement Results: Gas & Rate Limits

We measure the Gas limits of the backend peers by using program rpc_gasLimit in Figure 3.3. Here, we assume that different nodes in the same RPC service have the same Gas limit. The results are illustrated in Table 3.1 which shows that four out of nine services configure the Gas limit: SrvR4/SrvR1/SrvR9/SrvR8 respectively set the Gas limit at 50/10/5/1.5 block gas. Through our responsible disclosure, after our study, SrvR6 has set a limit of 10 block gas.
without Gas limits are particularly vulnerable to our DoERS attacks.

In our extended study, we also measure the rate limits deployed on many services’ frontend. The rate limits are intended to protect the service against distributed DoS. However, rate limiting without real-world identities can be easily bypassed by a Sybil attacker who registers multiple service accounts and accumulates much higher rate limits. Yet, requiring the DApp clients (e.g., a web browser surfing a DApp page) to expose real-world identities is impractical. We thus don’t consider rate limiting as effective protection against DoERS. One can essentially bypass all RPC services’ rate limit by using as many API keys or IPs as needed.

3.4.6 Attack Strategies Evading Gas Limit

For the RPC node without Gas limit, we design a single-request DoERS attack that has the power of evading all other protective measures we will observe in the next section (including rate limiting and load balancing). The attack sends a single request with a very large payload size (e.g., $10^9$) to run the `exhaustMem` function in the DoERS-C smart contract. The key observation here is this: `exhaustMem` runs a single EVM instruction, namely `CODECOPY`, to allocate a large memory. Running a single EVM instruction is atomic and is not interrupted, even when there is a timeout. Thus, the DoERS attacker can increase the payload size of an `exhaustMem` invocation to evade the 5-second timeout, causing a higher resource consumption and more severe service damage, as evaluated in §3.5.2.

For the RPC node with a Gas limit, the attacker can send multiple DoERS requests, each with a medium payload size under the Gas limit. If the requests are sent at a sufficiently high rate, there will be visible service interference, as evaluated in §3.5.1 and §3.5.2.

3.4.7 Summary of Attack Strategies

We summarize what an actual DoERS attacker can do, with respect to different real-world situations. We consider the goal of the attacker is to cause maximal damage to the DApp ecosystem, while minimizing her cost.
C1) For nodes or services without a Gas limit, the DoERS strategy is to send a single request invoking `exhaustMem` with a big-number payload size (e.g., $2^{64}$). If this crashes the EVM on the victim node, the attacker waits for 30 seconds and pings the node before sending the request again. This strategy also applies to any services without Gas limits — the single-request attack evades the protection of a load balancer.

C2) For nodes with a Gas limit, the DoERS strategy is to set the payload size of an individual request under the Gas limit and to send multiple such requests at a certain rate. In the case of a very low Gas limit, the attacker can tune up the request rate; because there is an innate asymmetry between the service and the DoERS attacker, the attacker can expect to cause significant damage to an individual node at a low cost (as evaluated in §3.5). This strategy applies to public RPC peers and Type-i services without load balancers.

C3) For Type-iii services with a Gas limit, there can be two DoERS strategies. One (C3a) is to follow the C2 strategy and to increase the rate as necessary to DoS all backend peers in the service. Given the small service scale (tens of peers), the DoERS asymmetry still helps keep the attacker’s cost low (see §3.6.1 for an analysis). The other strategy (C3b) is to predict load-balancing behaviors and design specific attacks, as demonstrated in SrvR1.

C4) For Type-ii services with a Gas limit, the DoERS strategy is to mount targeted attacks. As described in §3.4.3, the target can be a specific DApp client, a DApp, or a web3 library. The targeted attacks can evade the deterministic load-balancing behaviors in Type-ii services.

3.5 Evaluation of DoERS Attacks

In this section, we evaluate the effectiveness and cost of DoERS attacks. The attack effectiveness is measured by service performance degradation in latency increase, block synchronization slowdown, and others. The attack cost is measured by the attack rate. Note that an DoERS attack costs zero Ether by design as described in §3.6.1. Specifically, the evaluation aims to answer the following questions:
• Are real-world RPC services and peers exploitable under DoERS attacks? How much increase in response time will be caused by DoERS with “minimal” payload and rate (i.e., without causing any exception) on the real services? §3.5.1 answers these questions.
• On a local Ethereum node, how much damage can DoERS cause with payload and rate large enough to trigger and bypass exceptions? The damage is measured not only in response-time increase, but also in block synchronization slowdown, mining rate slowdown, etc. §3.5.2 answers these questions.

3.5.1 Evaluation on Deployed Services

Ethics-Driven Evaluation: Methodology

The goal is to verify whether a deployed RPC service is exploitable under DoERS attack. The main challenge comes from designing an effective test on the target services, without attacking them — The intensity of the test needs to be high enough to cause observable effects while it should be low enough to minimize actual performance degradation. The key idea here lies in discovering what we call the “minimally effective” parameters of the DoERS test. A DoERS test is minimally effective if 1) the difference between the response time of regular RPC requests under the test and that without the test is statistically significant, and 2) the response time of regular RPCs increase with the payload size and request rate.

More concretely, in the evaluation, we set up a virtual-machine (VM) instance in Google Cloud Platform (GCP) [59] for probing (a probing node) and another VM instance in Amazon EC2 [60] for measurement (a measurement node). With this setup, the two nodes do not share anything on their paths to an RPC service, and hence minimize performance interference between probing and measurement. During an experiment, we warm up the measurement node (in its network connection) by sending out three regular RPC requests (e.g., eth_getBlockNumber) to the target service. Then the measurement node sends out regular RPC requests at a rate of one request every two seconds. The response time of these regular RPCs is recorded. From the 30-th second after the measurement node starts, we launch the probing node which sends DoERS requests with
minimally effective parameters. The probing node lasts for \( t_a \) seconds and the measurement node continues for another 60 seconds after that.

In order to discover the minimally effective DoERS parameters, we carry out a series of carefully designed pre-tests in a local machine: We set up the local Ethereum node and conduct local tests to find the DoERS parameters such that the response time with and without the test differ by \( 5 \times \) times. During the pre-tests, we vary the attack parameters in payload size, probe rate and contract type. Note that \( 5 \times \) will be an estimate as the hardware spec. on the local node is different on deployed services. The local pre-tests produce several sets of candidate DoERS parameters, each set is a triplet \( \langle type, p, r_x \rangle \) where \( type/p/r_x \) is contract type/payload size/attack rate. For instance, \( \langle \text{CPU}, 20M, 10 \rangle \) means a DoERS attack exploit \texttt{exhaustCPU} function with payload size being \( 20M \) and attack rate being 10 requests per second. In particular, \( r_x = 0 \) means that a single DoERS request is sent out in the entire test process. Based on the above design, on each test, we would send a total of \( 60 \times 2/2 + 3 = 63 \) regular RPC requests plus at most \( t_a \times r_x \) DoERS requests. We set the attack duration \( t_a \) such that the number of DoERS requests can be upper-bounded before the test.

Besides, we avoid directly using large attack parameters (e.g., attack rates, payload sizes), which would have resulted in severe damage to the service. Instead, we test each service with a sequence of smaller and gradually increasing parameters, with the intention to discover the “trend” or how the server response time grows with increasing parameters. Such a trend allows to predict service response time under large parameters without causing the actual damage (see Figure 3.7b). With such measures, we expect each of our tests to affect no more than three nodes (out of hundreds) on the backend of each service for a short period of several minutes, to minimize the impact on its normal operations.

\[ \text{We use } M \text{ and } K \text{ to denote a million and a thousand, respectively.} \]
Evaluation Results

We follow the above methodology and test all nine services. Note that both measurement and probing VM instances do not run Geth, but instead run Curl [61] to send RPC requests. We first describe the experiment with SrvR3 as an example. We run a series of tests described above with different minimally-effective parameters. Each test produces a timeline of RPC response times. For instance, Figure 3.7a reports such a timeline on SrvR3 under DoERS attacks exploiting exhaustCPU with 30,000 payload and at the rate of 30 requests per second. The result shows a moderate $5 \times$ slowdown under the specific attack setting.

From there, we vary the attack rate with payload size fixed at 0.07$M$ (vary payload sizes with attack rate fixed at 18 per second). In each test, we define the attack-effective period by the period that the response time increases by at least $1.2 \times$ than the response time without attacks. Then we calculate the average response time during the attack-effective phase and report it in Figure 3.7b. The result clearly shows that the response time grows with increasing payload size and attack rates. For ethical reasons, we stop our test at maximal payload size 0.15$M$ or maximal rate 30 per second, resulting in a maximal response time of about 100 milliseconds. Also, our experiments observe no timeout or other exceptions thrown. The trend revealed in the figure implies that an actual attacker can use larger parameters than ours to cause a much longer RPC delay towards crashing the service.

**DoERS attacks to Type-iii services:** We conduct experiments on SrvR1 as an example Type-iii service. In the experiment, DoERS requests are sent to exhaustCPU with payload size 1.5$M$...
at the rate of 200 requests per second. The 1.5\textit{M} payload size makes the per-request Gas right below the Gas limit of SrvR1 service. The attack lasts for 20 seconds, and we observe a protective measure taken by SrvR1–15 seconds after the attack starts, the DoERS requests are returned with null. The timeline of measured response time is illustrated in Figure 3.7d. The RPC response time increases from 40 milliseconds before the attack to 160 milliseconds after the attack, leading to a 5\times increase. We suspect two causes: First, SrvR1’s load balancing depends on the timing of requests: all DoERS requests sent within one minute are collocated to the same three RPC nodes. Second, there are hundreds of peers on the backend of SrvR1 and all peers are saturated by the DoERS attacks.

**Targeted attacks to Type-ii services:** Among Type-ii services, SrvR6 is a representative service whose load balancer distinguishes requests based on IPs; recall Table 3.1. We conduct two tests that differ only by where the DoERS requests are sent. The specific result is in Figure 3.8a. If the DoERS requests are sent from a different IP from where the measurement requests are sent (as in our original setup), no increase of response time can be spotted. However, if we send the DoERS requests from the same IP as the measurement requests, the response time clearly increases right after the attack starts at the 5th second in Figure 3.8a. To eliminate the possibility of performance interference between probing and measuring, we conduct an extra test by sending DoERS requests at the same rate but with a much small payload size (e.g., 3 iterations in a loop) and no response-time increase can be observed. The result corroborates our measured load-balancing behavior and
directly shows that the adaptive attack strategy (recall § 3.4.3) is effective on SrvR6. Note that in Figure 3.8a, the attack sends only a single request exploiting `exhaustMem` with $20M$ payload size. The 10X increase of response time is caused by this single DoERS request. We also conduct similar experiments on the other Type-ii service, SrvR5, where DoERS requests are sent with the same/different API key with the measurement requests. The result, presented in Figure 3.8b similarly shows the effectiveness of our attack strategies – under the DoERS with the same API key, a $6\times$ slowdown (from 0.4 seconds to 2.4 seconds) is caused while under the DoERS of the same API key, there is no visible service slowdown.

![Fig. 3.9: A single-request attack exploiting exhaustMem to nodes without gas limits](image)

**Single-request memory DoERS:** For RPC services with no gas limits, the DoERS attacker can send a single RPC request to execute the `exhaustMem` that bypasses any load balancing. On SrvR3, we send a single request with parameters `eth_call(exhaustMem(5 \times 10^7))`, and we report the response times in Figure 3.9a. After the attack starts at the 5th second, the response time grows up by $20\times$ (from 0.1 seconds to 2 seconds). On SrvR6, we similarly a single request with parameters `eth_call(exhaustMem(1 \times 10^9))`. From Figure 3.9b the response time is increased by $150\times$ (from 0.2 seconds to 30 seconds).

**Summary of attack parameters:** Table 3.2 summarizes the effective attack parameters we found on these services. It can be seen that most existing services, with or without Gas limits,
### Table 3.2: Minimally effective attack parameters: Gas* in the number of block gas. In parenthesis are Gas limits.

<table>
<thead>
<tr>
<th>Services</th>
<th>(type, payload, rate)</th>
<th>Time</th>
<th>Gas*</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrvR2</td>
<td>(CPU, 2M, 10)</td>
<td>16×</td>
<td>13</td>
</tr>
<tr>
<td>SrvR3</td>
<td>(CPU, 0.15M, 30)</td>
<td>3.8×</td>
<td>0.2</td>
</tr>
<tr>
<td>SrvR4</td>
<td>(CPU, 3M, 0)</td>
<td>30×</td>
<td>19.5 (50)</td>
</tr>
<tr>
<td>SrvR6</td>
<td>(Mem, 50M, 0)</td>
<td>10×</td>
<td>5000</td>
</tr>
<tr>
<td>SrvR5</td>
<td>(CPU, 0.04M, 30)</td>
<td>4×</td>
<td>0.3</td>
</tr>
<tr>
<td>SrvR1</td>
<td>(CPU, 1.5M, 200)</td>
<td>5×</td>
<td>10 (10)</td>
</tr>
<tr>
<td>SrvR7</td>
<td>(CPU, 5M, 10)</td>
<td>15×</td>
<td>32.5</td>
</tr>
<tr>
<td>SrvR9</td>
<td>(CPU, 0.04M, 30)</td>
<td>2.1×</td>
<td>0.3 (5)</td>
</tr>
<tr>
<td>SrvR8</td>
<td>(CPU, 0.6M, 200)</td>
<td>110×</td>
<td>1.5 (1.5)</td>
</tr>
</tbody>
</table>

can be successfully attacked, causing an observable response-time increase by at least 3.8×. On SrvR3, for instance, the parameters to cause 3.8× increase are (CPU, 0.15M, 30); note that payload size 0.15M amounts to 0.2 block gas. Currently, SrvR3 does not set Gas limits; but our result implies that even if they set a very low Gas limit, like 0.2 block gas (which by the way is unlikely because of interference with service usability as discussed in § 3.6), it is still not enough to defend against DoERS attacks. SrvR4 can be effectively attacked at a Gas limit as low as 19.5 block gas, which is much lower than their current Gas limit (50 block gas). SrvR1 can be attacked with their current Gas limit (of 10 block gas), causing 5× response-time increase. We notice that the minimally effective payload sizes differ in different services and this can be caused by different hardware specs of the machines run in these services.

#### 3.5.2 Evaluation on a Local Full Node

In order to evaluate the damage caused by DoERS more extensively, we conduct experiments on a local machine under our control. The machine is a blade server with a 32-core 2.60GHz Intel(R) Xeon(R) CPU (E5-2640 v3), 256 GB RAM and 4 TB SSD disk. We set up a Geth v1.99 client on the server and fully synchronize it with the Ethereum mainnet. We turn on the RPC on this full node with default settings. The probing node and measurement node are run on the same commodity computer as before (§ 8).
The first experiment evaluates the DoERS’s impact on the block synchronization rate on the victim. In the experiments, we measure the local victim node’s current block height, denoted by $B_v$. To do so, the measurement node sends `eth_getBlockNumber` RPCs to the victim. We also monitor the block height of a regular mainnet node by $B_r$ and record the initial block height before the attack by $B_0$. From there, we report a metric that we call block synchronization slowdown:

$$\frac{B_r(10) - B_v(10)}{B_r(10) - B_0}$$

where $B_r(10)/B_v(10)$ is the block height 10 minutes after the attack starts. In the experiment, we vary the payload size and the attack rate, and report the slowdown in Figure 3.10a.

The result shows that block synchronization slowdown reaches as high as 96% with attack parameter ⟨CPU, 1M, 100⟩. When the payload size is 0.1M which amounts to 0.65 block gas, the DoERS attacker can cause synchronization slowdown by 91%, at the rate of 150 RPCs per second. Note that 0.65 block gas is very restrictive and is lower than any Gas limits we observe on all real RPC services and peers. In the figure, each point is labeled by whether a timeout is triggered during the test. It can be seen the DoERS attack of parameters ⟨CPU, 0.1M, 100⟩ does not trigger timeout yet still causes a 50% synchronization slowdown.

The second experiment shows how timeout can be effectively evaded by `exhaustMem` on nodes without Gas limits; recall the attack strategy C1 in § 3.4.7. In this experiment, we conduct a
series of tests, each of which sends a single `exhaustMem` request with increasing payload sizes. We report the response time of the attack request as in Figure 3.10b. When the payload size increases, the response time grows, first without timeout (in the red line) and then with a timeout (in the blue line). It is clear that after the timeout occurs, increasing payload sizes still leads to the increase of response time. This implies that the `exhaustMem`-based DoERS can essentially evade the timeout and increases payloads to crash the machine. The severe damage is applicable to five RPC services that do not configure Gas limits. The explanation for this attack is the following: `exhaustMem` contains an EVM instruction `CODECOPY` that runs a loop inside EVM to allocate memory of arbitrary length. Executing the instruction is atomic and can not be interrupted in between by a timeout; throwing a timeout has to wait until the completion of the instruction.

### 3.5.3 Evaluating DApp Response Time under DoERS

![Image of DApp response time graphs](image)

(a) DApp response time w/wo attacks  
(b) DApp response time w/wo attacks  
(c) Screenshot of DApp

Fig. 3.11: DoERS attacks on a metamask-based DApp

Our objective is to evaluate the impacts of DoERS on real-world DApps. Recall Figure 3.4, a typical DApp architecture includes user-facing web pages, wallet clients running as browser extensions such as metamask [62], the RPC service it uses and a remote blockchain network. It is known that most DApp webpages rely on metamask and third-party RPC services to communicate with Ethereum [62], which has also been confirmed in a simple measurement study we conducted:
Among the top 26 DApps (in terms of active user number) from dappradar.com, 20 (with combined 201,500 active users in 30 days) use metamask. So we focus on the response time for metamask-based RPC clients.

Specifically, our experiment is based upon a browser running a sample DApp that we developed on top of metamask. The sample DApp is a web button to get the latest block from the Ethereum network, through an RPC query `eth_blockNumber`. Here, metamask is configured to connect to a sample RPC service, namely SrvR9.

Also, we run a javascript code that issues a metamask query every $X$ seconds. We first set $X = 25$ seconds, since RPC results cached by metamask expire every 20 seconds (based on our experiment results). In the experiment, we measure and compare the response times of the “get-Block” button with and without a DoERS attack on the RPC node, as illustrated in Figure 3.11a, under the following parameters: 200 requests per second and a payload of $8 \times 10^5$. As we can see from the figure, the response time perceived by the client becomes significantly longer in the presence of the attack, causing a $10 \times$ slowdown. We present the screenshots of our DApp with/without the DoERS attack in Figure 3.11c which are taken from the full video demo shared on our public website.

We then vary the interval $X$ between 5 seconds and 50 seconds. We rerun the above experiment three times and report the average response time and their variance in Figure 3.11b. The result shows a shorter average response time can be observed if the internal is below 20 seconds, which matches the conjectured effect of result caching in metamask.

6https://dappradar.com/rankings/protocol/eth
7https://sites.google.com/view/doersdemo/
3.6 Countermeasures

3.6.1 Analyzing Known Countermeasures

Effectiveness of Gas Limits

In the Ethereum community, Gas limits are provided as the primary defense to denial of RPC service. Both Geth and Parity provide configuration knobs to set the Gas limit on an RPC instance. Ideally, the service provider should set a Gas limit low enough to protect their nodes from DoS attacks.

In practice, finding a “meaningful” value for the Gas limit is non-trivial if not impossible at all. There are two restrictions/challenges: 1) Setting a low Gas limit could negatively affect the service usability. For instance, a benign DApp wants to send a Google BigQuery-style RPC [63] to the blockchain service which would be blocked by a low Gas limit. This intention between a security-concerning service provider who wants to set a lower Gas limit and a usability-desiring client rooting for a higher Gas limit is real and has been observed [52]. In the end, the service provider often puts customer experience over the service security, by increasing the Gas limit, such as from 2 to 10 block gas in [52]. 2) More fundamentally, blockchain RPCs supporting Turing-complete programs cause asymmetry of computing cost between the client side and service side. That is, in a usable setup, the client-side cost of sending an RPC request is supposed to be much lower than the server-side cost of executing the smart contract. The “perfect” DoERS security will entail equating the client-side cost and server-side cost, which will lead to very restrictive loops (e.g., fewer than ten iterations) and would be detrimental to the service usability.

Empirically, the table in Figure 3.2 shows mixed results: On the one hand, some services, notably SrvR3 and SrvR5, can be effectively attacked even if the Gas limits are set as low as 0.2 and 0.3 block gas. Let alone that a low Gas limit is unlikely to be deployed in practice due to the impacts on service usability. Our experiments with local nodes described in § 3.5.2 also suggest there are effective attack parameters even with low Gas limits as 0.65 block gas. On the other hand, there are services that could mitigate DoERS vulnerability by deploying a reasonably low
Gas limit. For instance, if SrvR6 deploys the Gas limit of 10 block gas (which is the case after our disclosure of the problem to them in May, 2020), it would make the DoERS harder to succeed.

We believe setting a Gas limit is necessary but not sufficient; in other words, complementary defensive measures to Gas limiting are needed to provide effective DoERS protection.

**Contract Banning and Zero-Ether DoERS**

Recall that the first step in DoERS (in §3.3) requires the attacker to deploy a smart contract at her own cost. From our experience with SrvR7, a service provider who monitors the RPC performance can correlate the latency spikes to a malicious smart contract; they can take measures to ban all subsequent RPCs accessing the malicious contract. This will force the attacker to deploy the DoERS-C smart contract to a new address which could increase her cost in Ether.

We propose a zero-Ether DoERS that incurs zero monetary cost to the attacker, as a technique to evade a contract-banning service provider. The zero-Ether DoERS exploits the “state override” extension of `eth_call` in the recent Geth release [64, 65]. This feature allows a client to upload a smart contract at the invocation time of `eth_call`, instead of using a separate transaction. Specifically, the `eth_call` request carries the bytecode of a smart contract in its “state override” argument and invokes to run a certain function in the bytecode on the RPC node.

With this capability, the attacker can mount the DoERS attack in one step without paying any Ether. The attack works by the attacker sending a crafted `eth_call` request that includes the code of `exhaustXX` in its “state override” object and specify the invoked function to be `exhaustXX`. We have tested this zero-Ether DoERS attack on our local RPC node running Geth v1.9.2 [65].

### 3.6.2 Proposed Countermeasures

The root cause of DoERS is an open-membership RPC service that allows for free execution of arbitrary smart contract programs on its peers shared by different DApps. Intuitively, “falsifying” any condition in this root cause should harden the security against DoERS attacks, such as removing open-membership (e.g., by authenticating DApp clients based on their true identities),
charging the contract execution triggered by `eth_call`, limiting the computation expressiveness (e.g., prohibiting loops) and avoiding any sharing of an RPC node among DApps. Along with these design directions, we encounter a fundamental trade-off between DoERS security and service usability. For instance, the service provider can simply refuse to admit any `eth_call` triggering to run loops, which, while eliminating DoERS, comes at the expense of not being able to serve the benign DApps that do rely on loops; there are real-world smart contracts like this, such as financial analysis [48]. Also, requiring DApp clients to present real-world identities would be impractical or against the design of blockchain information transparency. We believe eliminating the DoERS vulnerability without affecting service usability is fundamentally difficult, if not impossible at all. Beyond simply Gas limiting, we propose a variety of mitigation techniques without dropping service usability, by performance anomaly detection, requiring security deposits, secure load balancing, atomic EVM execution (as will be described next), and other feasible defenses such as performance isolation. These techniques can be engineered in an RPC service at the layers of both the service frontend and the underlying EVM.

**Unpredictable yet consistency-preserving load balancing:** We design a secure and practical RPC load balancer that serves two purposes. First, it is expected to preserve the order between dependent transactions, which is important to ensure the correctness and fairness of the target DApp’s operations. Specifically, two transactions issued sequentially from the RPC client need to keep that order in the blockchain’s final transaction history. For instance, for an ERC20 token contract, the call `approve` needs to be followed by `transferFrom`, or otherwise, the execution will fail. Second, the load balancer’s behavior should be unpredictable in the sense that it independently forwards different incoming requests to randomly selected backend peers. Any determinism in load balancing can be exploited to direct the DoERS payloads to a few victim peers, allowing the attacker to overload them at a low cost.

However, preserving cross-request consistency could be in conflict with achieving load-balancing unpredictability. For instance, independent assignment of the `approve` and `transferFrom` calls could cause the calls to be handled by different backend peers, which will
send them independently to miners, rendering the order of their reception on the miners hard to maintain. Note that since approve and transferFrom are transactions issued from different sender accounts, Ethereum’s builtin nonce mechanism does not apply here. Our research shows that the load balancers in existing RPC services are designed to favor consistency preservation (DApp semantics) over unpredictability (§ 3.4.2).

We believe that this challenge is fundamentally caused by the use of a single balancer to process both transactions (write to a block chain) and RPC queries (read from the chain). The former requires cross-request consistency while the latter does not. Since DoERS targets the RPC queries, we could simply separate them from transaction requests, through two load balancers, to protect the RPC peers through an unpredictable assignment of the queries. More specifically, one balancer handles only transactions while the other forwards only RPC queries (including the eth_call’s), independently and randomly selecting a peer from the RPC service for each query (through a uniform distribution). To this end, the load balancer can internally maintain a secret true-random number or the current workload that decides the destination backend peer an RPC query should be forwarded to. In the meantime, the transaction-only balancer distributes the requests under the constraint of preserving consistency, just like what has been done by SrvR1 today (transactions with temporal locality given to the same backend peer).

A limitation of this dual-balancer solution is that it does not ensure transaction-query consistency: that is, the order between a transaction and an RPC query related to the transaction may not be preserved. One way to address this issue could be simply handing over such a transaction-query pair to the transaction balancer, so they can be assigned to the same peer and propagated to the blockchain in the right order.

**Performance anomaly detection plus security deposits:** As we analyzed, simple performance monitoring with contract banning can be evaded by our zero-Ether DoERS. We propose a countermeasure against the zero-Ether DoERS. The key idea is for the service provider to require security deposits from any potential clients, such that a benign client’s deposits will be refunded and a malicious client’s deposits will be confiscated to discourage any further attacks. In the pro-
posed framework, 1) the service provider only processes RPCs from a client having made security deposits. 2) The service provider monitors the performance and detect DoERS requests as performance anomalies. 3) After identifying attackers, the service provider confiscates deposits from attackers and refunds benign clients.

The success of the countermeasure hinges on whether the performance monitor can distinguish malicious DoERS requests from benign RPCs. Here, our assumption is that a DoERS attacker who wants to keep her cost low and evade existing DDoS protection has to make each malicious \texttt{eth_call} cause a significant amount of computations much more than a benign RPC.

**Interruptible EVM instructions**: The success of single-request DoERS can be attributed to atomic EVM instructions that timeout cannot interrupt. To avoid this attack vector, EVM should allow the “long-lasting” execution of a single instruction (e.g., \texttt{CODECOPY} in \texttt{exhaustMem}) to be interrupted by timeout. This may require engineering to change EVM’s instruction scheduling algorithm and to enforce the maximal memory size allocated by a single \texttt{CODECOPY} call.

### 3.7 Related Work

**Blockchain DoS security**: Since the advent, public blockchains have been a target of DoS attacks. A variety of DoSes have been designed and practiced on the different layers of a blockchain system in smart contract execution [13, 14], transaction processing [39, 9, 11], mining-based consensus [6, 5], and the underlying P2P network [1, 2, 3, 4]. For instance, in the P2P network layer, an eclipse attack [1, 2] aims to isolate a DoS-victim peer from the network and a routing attack [3, 4] employs BGP hijacking to intercept network traffic towards partitioning it. Among these attack vectors, of particular relevance are the DoSes that evade the Gas-based mechanism for smart contract execution. Under-priced EVM instructions, notably \texttt{EXCODESIZE} [50] and \texttt{SUICIDE} [51], have been identified and exploited in practice DoS attacks. Ethereum EIP150 [13] fixes the bugs by increasing the Gas associated with these instructions. Broken metering [49] further exploits the runtime variation of an EVM instruction, with the goal to lower contract-execution throughput
(gas per second) at low cost. Defensive mechanisms [66] have been proposed to punish contracts that excessively execute a particular (vulnerable) instruction. Unlike existing DoS attacks, DoERS uniquely targets the RPC-service layer of a blockchain node. DoERS is extremely low-cost and does not incur any Gas or Ether), which differs from existing DoSes that incur significant Gas.

**Blockchain RPC attacks:** In the existing literature, the only research work on the attacks exploiting blockchain’s RPC is a measurement of currency stealing attacks [15]. In the currency-stealing attack, an adversarial client exploits the time window between an account-unlocking RPC request and a transaction-send request, such that she can gain unauthorized access to an account unlocked on an RPC service. DoERS differs from the RPC-based currency stealing attack in that it does not exploit the privileged RPCs (e.g., account unlocking and transaction sending) but focuses on the open RPC queries that allow smart contract execution.

**Blockchain measurements:** Passive measurement [67] reveals various deployment information in Ethereum network (e.g., node distribution, network sizes, etc.). The approach taken is to launch several Ethereum nodes and collect the messages they exchange with their neighbors, which are analyzed to uncover network information. There are other measurement works focusing on Bitcoin network topology [21, 68], Monero P2P network [22], ERC20 token networks [69], etc. The measurement studies in this chapter focus on the DoERS security and leverage a novel measurement method based on orphan transactions that are not taken in existing works.

### 3.8 Responsible Disclosure

We have disclosed the DoERS vulnerability to the developer communities of Geth [70] and Parity/OpenEthereum [71], as well as all tested service providers. The bug reports are sent in May, 2020, leaving tested services at least 9 months to fix the bug before disclosing the vulnerability publicly (in Feb. 2021).

We have received a total of $260 bounty in Ether and are informed by the RPC services that bug fixing is in progress. For instance, our bug report has been acknowledged in Geth v1.9.16
release (July 10, 2020), which sets a new default limit to $25 \times 10^6$ Gas. Also after our reporting, SrvR6 sets a new limit to their service at $25 \times 10^6$ Gas, and invites us for further testing.

### 3.9 Summary

This chapter presents the first measurement study on the security of Ethereum’s RPC-enabled nodes under denial of service attacks. The results reveal that five out of the nine popular services (as of Apr. 2020) have turned on RPCs without configuring any Gas limits. These peers are particularly vulnerable and can be crashed by the proposed DoERS attack that sends as few as a single `eth_call` request at zero Ether cost. While the four other services including SrvR1 have configured Gas limits, the limits are so nonrestrictive that a properly configured DoERS attack can cause a latency increase by $2.1 \times \sim 50 \times$, as verified in our probes. On a local node protected by a very restrictive limit of 0.65 block gas, sending 150 RPC requests per second can slow down the block synchronization of the victim by 91%.

This chapter addresses the challenge of eliminating the DoERS vulnerability without affecting service usability. We propose mitigation beyond simply limiting the Gas; these techniques include unpredictable load balancing, performance anomaly detection, and interruptible EVM instructions. They are easy to be engineered in an RPC service at the layers of both the service frontend and the underlying EVM.
4.1 Introduction

A blockchain system relies on an underlying peer-to-peer (P2P) network to propagate information including recent transactions and blocks. The topology of the P2P network is foundational to the blockchain’s availability under network partitions, its security against a variety of attacks (e.g., eclipsing targeted nodes [1], denial of individual nodes’ service [72, 73], and deanonymization of transaction senders [74, 75]), and its performance (e.g., mining power utilization [76] and the quality of RPC services [77, 78, 79]). Details are in § 4.3. This value has motivated a line of measurement studies on the network topology of popular blockchains including Bitcoin [21, 68] and Monero [80]. However, although Ethereum is the second largest blockchain network (after Bitcoin) and the biggest smart-contract platform, measuring Ethereum’s network topology remains an open research problem. The existing Ethereum measurement studies [67, 81] focus on profiling individual peer nodes, but not the connections among them.

Research goals: Specifically, the operational Ethereum P2P network today runs tens of thousands of nodes and hosts multiple overlays: 1) an underlying P2P overlay, called platform overlay, which
forms a structured DHT network by following Kademlia’s protocols \[82\] for peer discovery (RLPx) and session establishment (DevP2P) \[67\], and 2) a number of application-specific overlays \[83, 67\], among which the dominant ones are Ethereum blockchains for information propagation. In particular, the Ethereum P2P network hosts multiple blockchain overlays with different “networkIDs” including the mainnet and various testnets, such as Ropsten \[84\], Rinkeby \[85\] and Goerli \[86\]. This multi-layer view of Ethereum’s P2P network is depicted in Figure 4.1. In the P2P network, each Ethereum node maintains “peer” connections at these two layers: 1) On the blockchain overlay, a node maintains a list of *active* neighbors through which local information is propagated. 2) On the platform overlay, a node stores the *inactive* neighbors in a DHT routing table, from which live nodes are promoted to active neighbors in the future.

This work aims at measuring the Ethereum P2P network’s blockchain overlay and its *active* links\[1\]. In practice, it is the blockchain overlay’s active links, instead of the platform overlay’s inactive ones, that capture the exact flow of information propagation and are more informative. For instance, a node running the Geth client \[87\] (which is the most popular Ethereum client and is deployed on more than 80% nodes in the mainnet \[88\]) maintains 272 inactive neighbors and around 50 active neighbors, by default. Knowing what these 50 neighbors are is helpful to understand the node’s resilience to eclipse attacks (as information is propagated through the 50 active neighbors, not the 272 inactive ones, and an attacker only needs to disable the 50 active neighbors to block information propagation). Also, knowing whether the 50 neighbors contain nodes from top mining

\[1\] In this chapter, we use terms “links”, “connections” and “edges”, interchangeably.
pools is useful to estimate the timeliness and quality of the blocks (or transactions) received on the node, as well as understand the centralization of the blockchain network.

Measuring Ethereum network’s active links is an open research problem. In the existing literature, 1) the related work that measure Ethereum networks focus on profiling individual nodes [67, 81] or detecting inactive links [89, 90], but not the active connections. Compared to the inactive links that are exposed in Ethereum peer discovery messages (i.e., RLPx’s FIND_NODE packets) and can be directly measured as in [89, 90], active links are hidden information inside remote Ethereum nodes, directly measuring which without inference is impossible as we thoroughly examine Ethereum protocol’s messages. 2) The other related work explore the topology of non-Ethereum blockchains including Bitcoin [21, 68] and Monero [80]. Their measurement approaches exploit features specific to Bitcoin/Monero and are inapplicable to Ethereum, as will be detailed in §5.8. Notably, TxProbe’s approach [21] to infer Bitcoin’s topology cannot be applied to measuring Ethereum topology, as these two blockchains differ in transaction model (account-based versus UTXO-based) and propagation model (direct propagation versus announcement), as will be further explained in §4.4.1.

Measurement methods: In this work, we propose TOPOSHOT to measure an Ethereum blockchain overlay by repurposing Ethereum’s transaction replacement and eviction policies. Briefly, an Ethereum node buffers unconfirmed transactions (prior to mining) in a local data structure named mempool, where an unconfirmed transaction can be replaced or evicted by a subsequent transaction at a sufficiently higher Gas price. Transaction replacement and eviction are standard Ethereum features, widely supported by Ethereum clients (including Geth [87], Parity [91] and others [92, 93, 94]), and highly desirable by real-world applications. For instance, a common practice in blockchain-based decentralized applications is that a user having sent a transaction can posthumously speed up its inclusion into the blockchain by sending replacement transactions at a higher price per computation unit (or the so-called Gas price). Leveraging these features,

The difference between transaction replacement and eviction is that a transaction \(tx\) is replaced by another transaction from the same sender account with \(tx\), and \(tx\) is evicted by another transaction from a different sender from \(tx\) when the mempool is full.
TOPOSHOT runs a measurement node $M$ to detect the connection between two remote nodes $A$ and $B$. In TOPOSHOT, node $M$ propagates a high-priced transaction $tx_A$ on target node $A$, a low-priced transaction $tx_B$ to target node $B$, and a medium-priced transaction $tx_C$ propagated to all other nodes in the same network. It then observes $tx_A$’s presence on node $B$ and, if so, draws the conclusion that node $A$ is actively connected to node $B$. To ensure accurate measurement, when node $A$ is not linked to node $B$, measurement transaction $tx_A$ should not be propagated and do not reach node $B$ (the so-called “isolation” property [21]). One of the key insights in this work is that Ethereum’s transaction replacement policy can be repurposed to enforce isolation property for accurate link measurement. Intuitively, the isolation is ensured by the fact that TOPOSHOT’s high-priced $tx_A$ can replace the low-priced $tx_B$ on node $B$ but not the medium-priced $tx_C$ on other nodes, through which $tx_A$ cannot be propagated to reach node $B$.

To set up the measurement as above, TOPOSHOT further leverages Ethereum’s support of transaction eviction and future transactions, that is, to evict an existing unconfirmed transaction on a node by incoming future transactions (the concept of future transaction in Ethereum is similar to orphan transactions in Bitcoin). Specifically, when using TOPOSHOT to measure the connectivity between Nodes $A$ and $B$, the measurement node $M$ first needs to connect to both nodes, propagates $tx_C$ to all nodes, then sends future transactions to evict $tx_C$ (with other existing transactions) on node $A$ and $B$ before sending $tx_A$ and $tx_B$ to node $A$ and $B$, respectively. This method can be applied to measuring the connectivity among all possible pairs of nodes by the standard approach of launching a “supernode” connecting to all other nodes in the network [67] [81].

The basic TOPOSHOT achieves 100% result precision but not 100% recall, which can be attributed to non-default settings of the target node. We further propose a pre-processing phase retrofittable with TOPOSHOT to profile the actual settings of the target node and to improve the result recall, proactively.

For large-scale measurement on real Ethereum networks, we propose a non-trivial method to parallelize multiple pair-wise measurements, reducing the rounds and overall time of measurement.

**Measurement results:** We systematically evaluate the validity of TOPOSHOT and conduct mea-
urement studies on both testnets and the mainnet. The measurement results uncover, for the first time, the full network topology of Ethereum’s major testnets (including Ropsten, Rinkeby and Goerli) and the inter- and intra-connections among the mainnet’s mining-pools and transaction relay services.

First, we validate the TOPOSHOT’s correctness in terms of recall and precision. We set up a local node under control and a remote node in a testnet, and we use TOPOSHOT to measure the connection between the two nodes. By comparing against the ground-truth of node connection (via querying the local node’s state), we confirm that TOPOSHOT achieves the perfect precision (100%) and a high recall (up to 97%).

Second, we use TOPOSHOT to measure, for the first time, the network topology of major Ethereum testnets including Ropsten, Goerli and Rinkeby. We also analyze the captured graphs which reveal a number of graph-theoretical properties including degree distribution, distances, assortativity, clustering and community structures. Our comparative analysis shows that the measured Ethereum networks have particularly lower modularity than classic random graphs \cite{95, 96, 97}, implying a better resilience against attacks to partition the networks.

Third, we propose enhanced TOPOSHOT configurations to allow lightweight yet effective measurement on the mainnet without ethical concerns. The TOPOSHOT enhancement minimizes the impacts on the target nodes being measured, and particularly ensures that the set of transactions included in the blockchain does not change under measurement. Using the approach, we measure a critical substructure of Ethereum’s mainnet overlay. The result reveals a biased neighbor selection strategy commonly practiced by critical Ethereum services such as mining pools and transaction relay services that prioritize connecting other critical nodes over average nodes. We acknowledge the high cost of our method and avoid measuring the topology of the entire mainnet network which would otherwise cost 60 million USD at the Ether price as of May 2021.

**Contributions:** This work makes the following contributions:

- **Novel methods:** We propose a novel method, named TOPOSHOT, to measure Ethereum network links and topology. TOPOSHOT takes a unique approach by exploiting Ethereum’s handling of
unconfirmed transactions (i.e., transaction replacement and eviction). TOPSHOT is generic and supports all Ethereum clients (including Geth and Parity). TOPSHOT is effective and achieves 100% result precision and high recall (88% $\sim$ 97%).

- **Large-scale measurements**: We address the scalability and ethical challenges raised in measuring large-scale, real Ethereum networks. We propose to schedule pair-wise measurements in parallel for efficiency. We propose workload-adaptive mechanisms to configure TOPSHOT for minimal service interruption on the target nodes/network.

- **New systematic results**: Without TOPSHOT, an Ethereum network’s topology remains hidden information inside blackbox Ethereum nodes, measuring which stays an open research problem. By systematically conducting measurements against a variety of Ethereum networks, we obtain a series of new knowledge on network topology and its graph-theoretic statistics, ranging from full-network topology in popular testnets (Ropsten, Rinkeby and Goerli) to critical sub-network topology in the mainnet. The source code of TOPSHOT is on [https://github.com/syracuse-fullstacksecurity/Toposhot](https://github.com/syracuse-fullstacksecurity/Toposhot).

**Roadmap**: The chapter is organized in the following order: § 5.2 presents the preliminary knowledge. Motivation of this work is presented in § 4.3. § 5.8 surveys the related work and their (in)applicability to measuring Ethereum’s topology. § 4.5 presents TOPSHOT’s measurement primitive, parallel schedule, as well as correctness analysis. § 4.6 presents the measurement results of Ethereum testnets and the mainnet. The ethical aspects of this work are discussed in § 4.7, and the conclusion is in § 5.10.

### 4.2 Preliminary

**Ethereum transactions**: To begin with, we describe the transaction model used in Ethereum. An Ethereum transaction binds a sender account to a receiver account. Each transaction is associated with a *nonce*, which is a monotonically increasing counter per sender. An Ethereum transaction is associated with *Gas price*, that is, the amount of Ether the sender is willing to pay to the miner for
each unit of computation carried out by the miner to validate the transaction.

**Unconfirmed transaction buffer** (**mempool**): Each Ethereum node stores unconfirmed transactions in a local data structure, named **mempool**. In a **mempool**, a transaction \( tx \) is **pending**, if its nonce equals one plus the maximal nonce of the transactions of the same sender in the **mempool** (i.e., equal to \( n + 1 \)). Otherwise, if \( tx \)’s nonce is strictly larger than \( n + 1 \), \( tx \) is a **future transaction**.

When a transaction \( tx \) propagated from other nodes arrives at a node \( N \), node \( N \) determines whether to **admit** \( tx \) into its **mempool**. Admitting a transaction \( tx \) may trigger two more **mempool** events: 1a) **eviction** of an existing transaction \( tx' \) by \( tx \) where \( tx \) and \( tx' \) are of different accounts or nonces, and 1b) **replacement** of an existing transaction \( tx' \) by \( tx \) where \( tx \) and \( tx' \) are of the same sender and nonce.

**Transaction propagation**: When admitting a pending transaction to its **mempool**, an Ethereum node propagates the transaction to its active neighbors. If an incoming transaction is not admitted or the admitted transaction is a future transaction, it will not be propagated.

Normally, an Ethereum node directly **pushes** a pending transaction to its neighbors. That is, it sends a message to its neighbors encoding the transactions it wants to propagate. It may be the case that the propagated transactions are already received by the neighbors. This is the default transaction propagation protocol supported widely in Geth, Parity, and other clients.

Some Ethereum clients (e.g., Geth with version later than 1.9.11) support **announcements** as an optional transaction-propagation protocol. It works in three messages: 1) The node announces its local pending transactions by their hash and propagates the hash to its neighbors. 2) Then within the next 5 seconds, its neighbors will respond with requests if they want to receive the transaction. Within these 5 seconds, the neighbors will not respond to other announcements of the same transaction. 3) The node propagates the transactions to all requesting neighbors. While this is similar to Bitcoin’s transaction announcement as exploited in TxProbe [21], there is an important distinction: Ethereum’s transaction announcement has to **co-exist** with transaction pushing, and Ethereum’s pushing can bypass the blocking of an announcement.
4.3 Motivation: Significance of Knowing Blockchain Topology

The motivation of this chapter is that a blockchain network’s topology is foundational to the blockchain’s security and performance. In this section, we present a non-exhaustive list of “use cases” of blockchain topology knowledge in the hope of justifying its significance.

4.3.1 Implication to Blockchain Security

The knowledge of blockchain network topology is crucial to understanding its security against various attack vectors, with examples listed below.

**Use case 1: Targeted eclipse attacks.** In the network topology, if a blockchain node is found to be of a low degree (i.e., few neighbors), such a node is particularly vulnerable under a targeted eclipse attack \[1\]. That is, such an eclipse attacker can concentrate her attack payload on the few neighbors to disable the information propagation and isolate the victim node from the rest of the network at low costs.

**Use case 2: Single point of failure.** The blockchain’s network topology may reveal the centralization of network connection, leading to a single point of failure. Specifically, there may be supernodes that connect to all other nodes, “bridge” nodes that control the connection to the backend of critical services, and topology-critical nodes removing which may lead to partitioned networks. Directing denial-of-service attacks onto these critical nodes, using attack vectors recently discovered \([72, 73]\), can lead to consequences such as crippled blockchain services and the censorship of individual transactions.

**Use case 3: Deanonymizing transaction senders.** With the knowledge of the network topology, if nodes’ neighbors are distinguishing (i.e., node \(X\)’s neighbors are distinct from another node \(Y\)’s neighbors), the neighbor set can be used to identify/fingerprint nodes and can be further used to facilitate the deanonymization of transaction senders. Specifically, in the deanonymization attack \([74]\), a blockchain “client” node (i.e., a node behind the NAT) is identified by its “server”-node
neighbors (a server node is of public IPs, is not behind the NAT and accepts incoming connections). An attacker then simply monitors the transaction traffic on all server nodes in the blockchain (e.g., a Bitcoin network contains much fewer server nodes than the client nodes, thus lowering the attacker’s costs). The attacker can link a transaction sender’s blockchain address (her public key) to a client node’s IP address, which can be further linked to a real-world identity, thus deanonymizing the blockchain address.

4.3.2 Implication to Blockchain Performance

Blockchain network topology is essential to achieving its performance promises and matters to both miners and client users.

**Use case 4: Mining efficiency and mining pools’ QoS (quality of service).** In a blockchain, the delay in propagating a recently found block from its miner to the entire network is critically important: If the delay of propagating miner X’s block is too long, her block may arrive after another miner Y’s block, leading to X’s block unable to be included in the blockchain and X’s loss of revenue. Thus, a blockchain’s network topology that affects propagation delay can influence a miner node’s revenue and mining-power utilization (i.e., how much mining power spent is useful and is reflected in the main chain’s blocks). Thus, it is unfavorable to have a minor with limited connectivity and incur long propagation delays.

For a client interested in joining a mining pool, she may want to access the knowledge of blockchain topology and make an informed decision to choose the mining pool with better connectivity and lower propagation delay to ensure high mining revenue.

**Use case 5: RPC service’s QoS.** For a client sending transactions through RPC services (e.g., infura.io), she may want to choose a service with better connectivity so that her transaction can be relayed on a timely basis.

In summary, the knowledge of blockchain network topology is critical to understanding its security, performance and decentralization. Given the high market capitalization of today’s blockchains (e.g., $4106 billion USD for Ethereum as of Sep. 2021 [98]), we believe measur-
ing blockchains’ network topology is valuable and worthy even if it costs as much as 60 million USD, described in §4.6.3.

### 4.4 Related Work

In this section, we present the existing measurement studies on public blockchain networks. Existing works can be categorized into three classes: W1) Measuring blockchain nodes, W2) measuring blockchain inactive edges, and W3) measuring blockchain active edges.

**Measuring blockchain nodes (W1):** Kim et al. [67] propose a passive method to characterize the Ethereum mainnet by launching a “supernode” to connect all reachable mainnet nodes and collecting messages exchanged. The result reveals node-wise characteristics including network size, node geo-distribution, clients’ age and freshness, and others.

Neudecker et al. (2019) [99] is a passive measurement study that last four year to characterize the behavior of individual Bitcoin peers and their operators. Their approach is by launching “supernodes” and passively collecting transaction traffic, a method similar to [67].

**Measuring blockchain inactive edges (W2):** Ethereum’s peer discovery protocol (RLPx) has a $\text{FIND\_NODE}$ message through which a node can discover another node’s current routing-table entries (inactive neighbors). Recent research works [89, 90] directly measure Ethereum’s inactive links by sending $\text{FIND\_NODE}$ messages to all nodes in an Ethereum network. This method cannot distinguish a node’s (50) active neighbors from its (272) inactive ones and does not reveal the exact topology information as TOPOSHOT does.

Henningsen et al. [100] measure the Kademlia network topology in IPFS by sending crafted peer-discovery queries. Despite other findings, this chapter reveals IPFS’s network combines a structured Kademlia DHT and an unstructured P2P overlay.

**Inference of blockchains’ active edges (W3):** Coinscope [101] targets Bitcoin’s network topology and infers the links by leveraging the expiration timestamps of Bitcoin’s ADDR messages.

TxProbe [21] infers Bitcoin’s network topology by exploiting Bitcoin’s support of orphan trans-
actions and announcement-based transaction propagation. We will describe how TxProbe works, with more details in § 4.4.1, to understand its applicability to measuring Ethereum networks.

Grundmann et al. [68] present two Bitcoin-topology inference approaches among which the more accurate one exploits Bitcoin’s behavior of dropping double-spending transactions. Neudecker et al. (2016) [102] conducts a timing analysis of Bitcoin transaction propagation to infer the network topology. Despite the optimization, both works are limited in terms of low accuracy.

Daniel et al. [103] propose to exploit block relay mechanisms to passively infer connections among mining nodes and their direct neighbors in the ZCash network.

Cao et al. [80] measure Monero’s P2P network topology by exploiting the timing of neighbors’ liveness probes. Specifically, a Monero node maintains the liveness of its neighbors (the `last_seen` label) by periodically discovering its hop-2 neighbors, probing their liveness by sending PING messages, and selectively promoting them to be hop-1 neighbors. Topology inference methods are proposed to exploit the timing difference of neighbor nodes’ `last_seen` labels. This method is specific to Monero’s liveness-check protocol.

<table>
<thead>
<tr>
<th>Research work</th>
<th>Blockchain</th>
<th>Measurement target</th>
</tr>
</thead>
<tbody>
<tr>
<td>[99]</td>
<td>Bitcoin</td>
<td>Nodes</td>
</tr>
<tr>
<td>TxProbe [21] &amp; others [101, 68, 102]</td>
<td>Bitcoin</td>
<td>Active edges (W3)</td>
</tr>
<tr>
<td>[80]</td>
<td>Monero</td>
<td>Active edges (W3)</td>
</tr>
<tr>
<td>[103]</td>
<td>ZCash</td>
<td>Active edges (W3)</td>
</tr>
<tr>
<td>[67]</td>
<td>Ethereum</td>
<td>Nodes (W1)</td>
</tr>
<tr>
<td>[89, 90]</td>
<td>Ethereum</td>
<td>Inactive edges (W2)</td>
</tr>
<tr>
<td>[100]</td>
<td>IPFS</td>
<td>Inactive edges (W2)</td>
</tr>
<tr>
<td><strong>TOPOSHOT</strong></td>
<td>Ethereum</td>
<td>Active edges (W3)</td>
</tr>
</tbody>
</table>

Table 4.1: Existing works on blockchain topology measurement and TOPOSHOT’s distinction.

The existing blockchain measurement studies are summarized in Table 4.1. In general, existing techniques on W1 and W2 directly measure the target (as the target information of nodes and inactive edges is exposed in collected messages), while measuring active edges (W3) requires inference. Existing topology-inference techniques focus on non-Ethereum blockchains and exploit
blockchain specific features (e.g., Monero’s timing of liveness probes and Bitcoin’s announcement-based propagation) that are absent in Ethereum.

### 4.4.1 TxProbe’s Applicability to Ethereum

To understand how TxProbe works and its (in)applicability to measuring Ethereum network, we first describe the following measurement framework: Suppose a measurement node $M$ is to detect the connection between a pair of target nodes, say $A$ and $B$. Node $M$ can propagate a transaction $tx_A$ to node $A$ and observe $tx_A$’s presence on node $B$. If present, nodes $A$ and $B$ are actively linked. The success of this method depends on the so-called isolation property. That is, when node $A$ and $B$ are not actively linked, $tx_A$ should not be propagated to node $B$. In other words, there is no alternative routing path besides the direct link between $A$ and $B$ that $tx_A$ can take to reach node $B$.

TxProbe [21, 68] materializes this framework to measure active links in Bitcoin and ensures the isolation property by Bitcoin’s transaction announcement mechanism. Briefly, Bitcoin’s transaction announcement works as follows: a Bitcoin node propagates a transaction to its neighbor by first sending a transaction announcement (i.e., a hash value) to the neighbor and, upon neighbor’s acknowledgment, then sending the actual transaction. Bitcoin has a policy that the neighbor node receiving an announcement will ignore the subsequent announcements of the same transaction from other nodes for 120 seconds. TxProbe exploits this policy to ensure the isolation during the 120-second period. This is done by having Node $M$ announce $tx_A$ to all nodes besides $B$ so that these nodes will not relay $tx_A$ when Node $A$ starts to propagate $tx_A$ to $B$, ensuring the isolation property.

However, TxProbe’s method is inapplicable to measuring Ethereum. Ethereum’s transaction propagation only partially depends on announcements. That is, a transaction is announced to some neighbors and is directly sent to other neighbors without announcement. The existence of direct propagation, no matter how small a portion it plays, negates the isolation property, as measurement transaction $tx_A$ can be propagated through the nodes taking direct propagation as the alternative
path to reach node $B$, introducing false positives to the measurement results.

In addition, TxProbe relies on Bitcoin’s UTXO model, which differs from Ethereum’s account model. Directly applying TxProbe to Ethereum risks incorrect measurement.

4.5 TOPOSHOT Measurement Methods

We first present our observation of real Ethereum clients’ behavior in transaction replacement and eviction, which lays the foundation of TOPOSHOT measurement method (§ 4.5.1). We then describe the measurement primitive in TOPOSHOT that detects just one link between two nodes (§ 4.5.2). We will then describe how to use this primitive to measure a network of a large number of links (§ 4.5.3).

4.5.1 Profiling Ethereum Clients’ Behavior

We first describe a parameterized model for mempool and then our measurement study that reveals the mempool parameters of real Ethereum clients.

mempool model: Recall that transaction eviction (replacement) is a mempool process that takes as input the initial state of mempool and an incoming transaction $tx_1$ and produces as output the end state of the mempool and possibly, a transaction $tx_2$ that is of the same (different) sender with $tx_1$ and that is evicted (replaced) from the mempool. To formally describe the process, suppose the initial state is a full mempool consisting of $l$ pending transactions and $L - l$ future transactions, where $L$ is the capacity of the mempool (denoted in Table 4.2). The incoming transaction $tx_1$ is a future transaction with Gas price higher than any transactions currently in the mempool. There are $u$ transactions currently in the mempool that are of the same sender with $tx_1$.

When there is another transaction $tx_2$ in the mempool that has the same sender and nonce with $tx_1$, admitting $tx_1$ to the mempool triggers the replacement of $tx_2$. The generic transaction replacement strategy is that mempool decides to replace $tx_2$ by $tx_1$, if $tx_1$’s Gas price is $1 + R$ of $tx_2$’s Gas price.
Otherwise (i.e., when there is no transaction of the same sender and nonce with \(tx_1\)), admitting \(tx_1\) may trigger transaction eviction. For transaction eviction, the situation of interest to us is the eviction victim \(tx_2\) being a pending transaction. Under this situation, the transaction eviction strategy generally follows the template: \textit{mempool decides to evict a pending transaction \(tx_2\) by \(tx_1\), if 1) \(tx_1\)’s Gas price is higher than \(tx_2\)’s Gas price, and 2) there are more than \(P\) pending transactions existing in the \textit{mempool}, and 3) there are fewer than \(U\) existing transactions of the same sender with \(tx_1\).} The three \textit{mempool} parameters, namely \(R\), \(U\) and \(P\), and their meanings are presented in Table 4.2.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R)</td>
<td>Minimal Gas price difference for an incoming transaction (tx) to replace an existing tx in \textit{mempool}</td>
</tr>
<tr>
<td>(U)</td>
<td>Max number of future txs sent from the same account that can be admitted to a node’s \textit{mempool}</td>
</tr>
<tr>
<td>(P)</td>
<td>Minimal number of pending txs buffered in a node to allow eviction by future txs</td>
</tr>
<tr>
<td>(L)</td>
<td>Maximal number of txs allowed to store in a \textit{mempool} (\textit{mempool} capacity)</td>
</tr>
</tbody>
</table>

Table 4.2: Notations of parameters in \textit{mempool}

<table>
<thead>
<tr>
<th>Ethereum clients</th>
<th>Deployment (mainnet)</th>
<th>Replacement behavior</th>
<th>Eviction behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(R)</td>
<td>(U)</td>
<td>(P)</td>
</tr>
<tr>
<td>Geth</td>
<td>83.24%</td>
<td>10%</td>
<td>4096</td>
</tr>
<tr>
<td>Parity</td>
<td>14.57%</td>
<td>12.5%</td>
<td>81</td>
</tr>
<tr>
<td>Nethermind</td>
<td>1.53%</td>
<td>0%</td>
<td>17</td>
</tr>
<tr>
<td>Besu</td>
<td>0.52%</td>
<td>10%</td>
<td>∞</td>
</tr>
<tr>
<td>Aleth</td>
<td>0%</td>
<td>0%</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.3: Profiling different Ethereum clients’ transaction eviction and replacement policies. The second column refers to the percentage of mainnet nodes running a specific client [88].

\textbf{mempool tests}: The measurement is set up with 1) a measurement node \(M\) running the test and 2) a target node \(T\) running the Ethereum client to be measured. For each test, node \(T\)’s initial state of \textit{mempool} contains \(l\) future transactions and \(L - l\) pending transactions.

We design the first set of tests to trigger transaction replacement and measure \(R\). Specifically, \(tx_1\) has an identical sender and nonce with an existing transaction \(tx_2\) in \textit{mempool}. In each unit test, Node \(M\) sends \(tx_1\) of a certain Gas price to node \(T\), and observes if node \(T\) replaces \(tx_2\) by
tx\textsubscript{1}. We run a series of unit tests with varying tx\textsubscript{1}’s Gas prices, in order to observe the minimal Gas price that triggers the replacement, from which we calculate and report \( R \).

We design the second set of tests to trigger transaction eviction and measure \( U \) and \( P \). Specifically, the mempool contains \( L - l \) future transactions and \( l \) pending transactions, among which there are \( u \) transactions sent from the same account with future transaction \( tx\textsubscript{1} \). As before, in each unit test, node \( M \) sends to node \( T \) \( tx\textsubscript{1} \) at a Gas price higher than any transactions in node \( T \)’s mempool. We run a series of unit tests with varying \( l \) and \( u \). We observe the maximal value of \( u \) that triggers a successful eviction by \( tx\textsubscript{1} \) and report such value by \( U \). We observe the minimal value of \( l \) that triggers a successful eviction by \( tx\textsubscript{1} \) and report such value by \( P \).

**Test results on Ethereum clients:** We conduct the tests on two local nodes: We first set up a local measurement node \( M \) running tests on an instrumented Geth client and a local target node \( T \). The statically instrumented Geth client allows node \( M \) to bypass local checks and to propagate future transactions to node \( T \). We run the two sets of tests against the target node \( T \) running five different Ethereum clients: Geth (Go), OpenEthereum/Parity (Rust), Nethermind (.net), Besu (Java), and Aleth (C++). Here, we discard the Python client (i.e., Trinity) as the incomplete implementation. The distribution of mainnet nodes running the five Ethereum clients is presented in the second column of Table 5.2, where Geth (83\%) and Parity (15\%) are the dominant clients on the mainnet.

The measurement results are reported in Table 5.2. The mempool model and measurement results will guide the design of TOPOSHOT’s method and the configuration of the measurement on different Ethereum clients. Noteworthy here is that Aleth’s and Nethermind’s \( R \) values are both zero (0\%), which renders our TOPOSHOT unable to work (as will be seen, it requires a non-zero \( R \) to enforce the isolation property). Thus, TOPOSHOT currently does not work with Aleth and Nethermind clients. On the other hand, we deem a zero-value \( R \) is a flawed setting that can be exploited to construct low-cost denial of service or flooding. For instance, an attacker can send multiple replacing transactions at almost the same Gas price, consuming network resources by propagating multiple transactions yet without paying additional Ether.
Fig. 4.2: TOPOSHOT’s measurement primitive: Running measureOneLink with $Y = 0.1$ Gwei, $Z = 5120$, $R = 10\%$, $U = 1$.

4.5.2 Measurement Primitive

We consider the basic system model consisting of a measurement node $M$, target node $A$, target node $B$, and the rest of Ethereum network denoted by node(s) $C$. The measurement primitive’s goal is to detect one link, that is, whether Node $A$ and $B$ are actively connected in the Ethereum network. Note that this model assumes a strongly connected Ethereum network without network partition.

**Mechanism:** We denote our measurement primitive by $\text{measureOneLink}(A, B, X, Y, Z, R, U)$, which are parameterized by target nodes $A$ and $B$, target nodes’ mempool behavior $RU$ (recall Table 4.2) and $X/Y/Z$ that will be described below. As depicted in Figure 4.2a, the measurement primitive works in four steps:

1. **Node $M$** sends a pending transaction $tx_C$ with Gas price $Y$ Gwei to $A$, and waits for $X$ seconds (e.g., $X = 10$ in our study as will be described) for $tx_C$ to be propagated to other nodes including node $B$. Setting $Y$ at a low Gas price is intended to slow down or even prevent the inclusion of $tx_C$ in the next block (recall Ethereum nodes decide which transactions to be included in the next block based on Gas/Gas price).

2. **Node $M$** sends to Node $B$ $Z$ future transactions $\{tx_{O1}, tx_{O2}, \ldots tx_{OZ}\}$ at Gas price $(1 + R) \cdot Y$.

---

One Gwei equals $10^{-9}$ Ether.
Gwei. These future transactions are uniformly sent from $\frac{Z}{U}$ accounts (i.e., there are $U$ future transactions sent from each account). Immediately after that, Node $M$ sends a transaction $tx_B$ at Gas price $(1 - 0.5R) \cdot Y$ Gwei to Node $B$. Transaction $tx_B$ has the same nonce with $tx_C$.

3 Node $M$ sends to Node $A$ $Z$ future transactions $\{tx_{O1}, tx_{O2}, ..., tx_{OZ}\}$ which are at Gas price $(1 + R) \cdot Y$ Gwei and sent from $\frac{Z}{U}$ accounts. Immediately after that, Node $M$ sends a transaction $tx_A$ at Gas price $(1 + 0.5R) \cdot Y$ Gwei to Node $A$. Transaction $tx_A$ has the same nonce with $tx_C$.

The purpose of the future transactions is to fill up the mempool on Nodes $A$ (and $B$), to evict $tx_C$ there, and to make room for $tx_A$ ($tx_B$) of the same nonce to $tx_C$.

4 Node $M$ checks if it receives $tx_A$ from Node $B$. If so, it draws the conclusion that Node $A$ is a neighbor of Node $B$, as analyzed in §4.5.2.

**Configuration of $R/U$:** Parameters of the $\text{measureOneLink}$ primitive are configured as follows: On a target Ethereum client, parameters $R/U$ will be set at the client’s value as in Table 5.2. Here, note that Nethermind and Aleth are not measurable by TOPOSHOT due to their zero-value $R$, which is also flawed as explained before. Besu has an infinitely large value of $U$, and Geth has a fairly large $U$. In these two cases, only one account is used to send the future transactions $\{tx_O\}$. Geth/Parity have non-zero $P$, which are fairly small compared with their mempool capability $L$. The working of measureOneLink requires the following condition: The number of pending transactions in the measured mempool should remain larger than $P$ in the entire process of measurement. We verify that this condition holds on the mainnet for all Ethereum clients’ $P$ and $L$.

**Configuration of $X$:** Parameter $X$, which is the time period that Step 1 waits, is set to be large enough so that transaction $tx_C$ is propagated to all nodes in the network. In order to obtain a proper value of $X$ in an Ethereum network, we conduct a test by running several local nodes (e.g., 11 nodes in our study) and joining them to the Ethereum network. Among the 11 nodes, there
are no direct connections. During the test, we send a transaction through one node, wait for $X'$ seconds and observe the presence of the transaction on the other 10 nodes. We conduct a series of such tests with varying $X'$es to obtain such a $X' = X$ that with 99.9% chances, the transaction is present on the 10 nodes after $X$ seconds.

The four steps occur in order. That is, Step 1 occurs $X$ seconds before Step 2, which finishes before Step 3 starts, which is before Step 4. Timing and ordering are essential to the success of our measurement method, as analyzed below.

**Correctness Analysis**

We analyze the correctness of the measurement primitive (`measureOneLink`):

10 seconds after Step 1, Transaction $tx_C$ is propagated to the entire Ethereum network and it is stored in all nodes’ mempools including Nodes A and B.

During 2, when Node B receives $Z$ future transactions $tx_Os$, its mempool becomes full. Based on the eviction policy in Table 5.2, adding a new transaction to a full mempool triggers evicting the transaction with the lowest Gas price. Assuming Gas price $Y$ Gwei is low enough (we will describe how to set $Y$ next), transaction $tx_C$ at $Y$ Gwei will be evicted on Node B. Then, without $tx_C$, transaction $tx_B$ is stored in Node B’s mempool. In other words, Step 2 replaces $tx_C$ with $tx_B$ on Node B. Note that in the process, future transactions $\{tx_O\}$ are not propagated, thus C still stores $tx_C$.

Note that after the arrival of $\{tx_O\}$ but before $tx_B$, there are chances that certain Nodes C can propagate $tx_C$ back to Node B, which, if occurs, would invalidate the efforts of $\{tx_O\}$ and leave $tx_B$ unable to replace (the re-propagated) $tx_C$ on B. In TOPOSHOT, the actual chance of this event is very low and the reason is two-fold: 1) 2 waits long enough (10 seconds) after 1 to start and 2) $tx_B$ is propagated immediately after $\{tx_O\}$. In addition, in our local validation experiment (in §4.6.1), we don’t observe the occurrence of the event.

By a similar analysis, Step 3 can replace $tx_C$ with $tx_A$ on Node A.

Now, we have established that after Steps 1,2 and 3, Node A stores $tx_A$, Node B stores $tx_B$.
and Nodes $C$ store $tx_C$. The snapshot of our measurement system at this timing is illustrated in Figure 4.2b.

We consider two cases: Case 1) $A$ and $B$ are directly connected. In this case, $A$ will propagate $tx_A$ to $B$, which will replace $tx_B$ because of $tx_A$’s $R$ (e.g., 10% for Geth) higher Gas price than $tx_B$. In this case, $A$ will also propagate $tx_A$ to $C$, which however will not replace $tx_C$ as $tx_A$’s Gas price is lower than $R$ (e.g., 10%) of $tx_C$’s price. The property that $tx_A$ is stored only on Node $A$ and cannot be propagated through Nodes $C$ is called isolation. That is, $tx_A$ is isolated on Node $A$. Thus, after a sufficient delay for propagation from $A$ to $B$, $M$ can receive $tx_A$ from Node $B$.

Case 2) $A$ and $B$ are not connected. In this case, $A$ propagates $tx_A$ only to Node $C$. As analyzed, $tx_A$ cannot replace $tx_C$ on Node $C$ because of insufficient Gas price. Also, Node $C$’s $tx_C$ cannot replace $tx_B$ on Node $B$. Thus, $tx_B$ stays on Node $B$, and $M$ does not receive $tx_A$ from Node $B$.

To ensure correctness, TOPOSHOT requires that the mempool on the two measured nodes, namely nodes $A$ and $B$, are full. This condition holds quite commonly in Ethereum mainnet, as observed in our measurement study (99% of the time during our mainnet measurement, the measurement node’s mempool is full).

**Configuration of $Y/Z$:** Pending transactions like $tx_C$ should stay in the mempool of Nodes $C$, in such a way that they are not included in the next block or be evicted. To do so, the Gas price of $tx_C$ should be low enough so that it will not be included in the next block, and at the same time, be high enough to avoid eviction by incoming transactions. To estimate a proper Gas price in the presence of current transactions, we rank all pending transactions in the mempool of Node $M$ by their Gas prices, and use the median Gas price for $tx_C$. In actual measurement studies, the value of $Y$ varies from testnets and at different times. We apply the estimation method before every measurement study and obtain $Y$ dynamically.
**Cost Analysis**

Running the `measureOneLink` primitive, the cost comes from the pending transactions sent (i.e., either $tx_A$ or $tx_C$), assuming their inclusion in the blockchain. In practice, whether these two transactions are included is not deterministic and depends on the state of the miners’ mempool. Also, note that the future transactions $tx_O$ sent during the measurement are guaranteed not to be included in the testnets and mainnet, thus incurring no costs.

**Improving Result Recall**

Based on the above analysis, the TOPOSHOT guarantees that any tested connection is a true positive (i.e., no false positives) but may miss the detection of a connection (i.e., false negative may exist). In other words, the 100% result precision is guaranteed by the protocol but not for the recall. Note that 100% precision/recall means no false positive/no true negative in the measurement result. In the following, we present several heuristics to improve the result recalls in practice.

A passive method to improve the result recall is to repeat the measurements multiple times and use the union of the results. This passive method has limited applicability if the false negative is caused by the non-default setting on the remote Geth node being measured. In the following, we propose a proactive method to improve the recall.

**Handling node-specific configurations by pre-processing:** In Ethereum networks, client configurations (e.g., on mempool) are specific to nodes. This is evident in our field experience where the mempool capacities (i.e., $L$) differ across nodes. Using the same setting of $L$ when measuring different nodes can lead to incorrect results.

To solve the problem, we add a pre-processing phase: Before the measurement, we can launch a speculative $B'$ node locally and use it to connect all other nodes in the network. For each other node, say $A'$, we then run TOPOSHOT between $A'$ and $B'$. Because the local node $B'$ is under our control and its actual neighbors can be known (by sending `peer_list` RPC queries), we compare the measurement result with the ground truth. If there is a false negative, it implies the remote node $A'$ has some non-default setting on its node (e.g., use a mempool larger than the default $Z$).
We then increase the mempool size in additional pre-processing measurements to discover a proper setting of the mempool. The result of the pre-processing can help guide the actual measurement to use a “right” parameter on the connections involving node $A'$.

4.5.3 Parallel Measurement Framework

We previously described the primitive of measuring one connection between a source and a sink node. To measure a network, a native schedule is to serially run the pairwise primitive over all possible pairs, which however incurs a long measurement time in the case of large networks and is not a scalable method. For time efficiency, we propose a parallel schedule that decomposes the set of all possible pairwise connections into subsets and measures all connections within each subset in parallel. In the following, we first describe the parallel measurement primitive (§ 4.5.3) and then the schedule that measures the entire network in parallel by repeatedly using the primitives (§ 4.5.3).

Parallel Measurement Primitive

We consider a pair of nodes whose connectivity is measured consist of a source node and a sink node. For instance, in Figure 4.2b node $A$ is a source node and node $B$ is a sink node. In a parallel measurement, we consider measuring the connectivity between not one pair of source and sink nodes, but multiple such pairs. Specifically, suppose there are $p$ “source” nodes $A_1, A_2, \ldots, A_k, \ldots, A_p$ and $q$ “sink” nodes $B_1, B_2, \ldots, B_l, \ldots, B_q$; note that $k$ ($l$) is the index of a source (sink) node. In this bipartite graph, there are a total of $p \cdot q$ possible edges from a source to a sink. The objective here is to measure $r$ specified edges out of the $p \cdot q$ ones. We denote by $\text{sink}(k, j)$ a sink node which is the $j$-th neighbor of a source node $A_k$. Then, the edge between $A_k$ and $\text{sink}(k, j)$ is “indexed” by $(k, j)$. Initially, assume there are sufficient funds set up in $r$ Externally Owned Accounts (or EOAs).

Node $M$ sends a total of $r$ transactions $\{tx_{C(k,l)}\}$ and propagates them to the Ethereum network. Any two different transactions are sent from different EOAs.
To each Node $A_k$, Node $M$ 1) first sends $Z$ (e.g., 5120 for Geth) future transactions $tx_F$’s followed immediately by 2) sending $\{\ldots, tx_{C(k-1,q_{k-1})}, tx_{C(k+1,1)}, \ldots\}$. 3) It then sends $\{tx_{A(k,1)}, \ldots tx_{A(k,q_k)}\}$. Here, $tx_{A(k,i)}$ spends the same account with $tx_{C(k,i)}$ and its Gas is priced at 1.05Y Gwei. After $p_2$, $tx_{C(k,i)}$ on Node $A_i$ is replaced by $tx_{A(k,i)}$, while other $tx_C$’s stay.

It is noteworthy that after $p_2$, Node $M$ checks whether $tx_{A(k,\cdot)}$ are actually stored on Node $A_k$. It proceeds only if the checked condition holds. Node $M$ carries out the check by observing if $tx_{C(k,\cdot)}$ is propagated from Node $A_k$ before waiting for a timeout.

Fig. 4.3: TOPOSHOT’s parallel measurement protocol; in Figure 4.3b, the four colors represent four $\textit{measurePar}$ iterations where a rectangle with rounded angles is the selected nodes $A$’s in the measurement and the arcs are the collection of edges being measured. For instance, the blue iteration is $\textit{measurePar}(\{n_1, n_2, n_3\}, \{n_4, n_5, n_6, n_7, n_8\}, \{\emptyset\})$.

Node $M$ sends to each Node $B_l$ $Z$ future transactions $tx_F$’s followed immediately by $r$ transactions where the $i$-th transaction is a $tx_B$ transaction (whose Gas is 95% of that of a $tx_C$ transaction) if the $i$-th edge’s sink is Node $B_l$, and otherwise, is a $tx_c$.

For edge connecting $A_k$ and $B_l$, Node $M$ checks if $tx_{A(k,j)}$ (note that $\text{sink}(k,j) = l$) is present on Node $B_l$. If so, $A_k$ and $B_l$ are neighbors.

Note that Ethereum clients, including both Geth and Parity, limit the number of future transactions in their mempool. In our parallel measurements, we ensure the group size is much smaller.
than the limit of future transactions, which further ensures the measurement correctness, since all measurement transactions will be admitted and stored on the participant nodes.

**Example:** We use an example to illustrate the parallel measurement protocol. Among two sources $A_1, A_2$ and two sinks $B_1, B_2$, assume it measures the connections on three edges, that is, $\langle A_1, B_1 \rangle, \langle A_2, B_1 \rangle, \langle A_2, B_2 \rangle$. Figure 4.3a depicts the snapshot of exercising our parallel measure method right after $p_3$.

**Ensuring isolation:** As in the case of measurement primitive, isolation is critical to the success of our measurement method. In the parallel setting, a Node $A$ needs to prevent propagating the $tx_A$ transactions to Nodes $B$’s via not only Nodes $C$’s but also other Nodes $A$’s and Nodes $B$’s. For instance, in the example above, when measuring the connection between Node $A_2$ and $B_1$, it needs to ensure that $tx_A(2, 1)$ is not propagated to Node $B_1$ via Node $A_1$ or $B_2$. This is guaranteed by our measurement method because Nodes $B_2$ and $A_1$ store $tx_C$ transactions and can be treated as a $C$ node when measuring the connection between Node $A_2$ and $B_1$.

**Parallel Measurement Schedule**

Given a network of nodes $\{n_1, n_2, \ldots, n_N\}$, we partition the nodes into $N/K$ groups where each group is of $K$ distinct nodes; for instance, the $i$-th group ($i$ starting from 0) is of nodes $\{n_{iK+1}, n_{iK+2}, \ldots, n_{iK+K-1}\}$.

We schedule the network measurement in the two rounds: The first round runs $N/K$ iterations, where each iteration measures the edges between group $i$ and the rest of the network. The second round measures the edges within a group.

To be more specific, we denote the parallel measurement primitive described in § 4.5.3 by $measurePar(\{A_i\}, \{B_i\}, \{C\})$. 1) Given the $i$-th group, the first round calls $measurePar(\{n_{iK+1}, n_{iK+2}, \ldots, n_{iK+K-1}\}, \{n_1, \ldots, n_{iK-1}, n_{iK+K}, \ldots, n_N\}, \{\emptyset\})$, where $A$ is the $i$-th group, $B$ is the rest of the blockchain network, and $C$ is empty. Each of these iterations sets a goal to measure $K \cdot (N - K)$ possible edges.

2) The second round measures the edges within groups. Specifically, given a group, it maps the
first half of nodes as $A$ and the other half as Nodes $B$. An iteration measures intra-group edges for all groups. It then applies the same splitting respectively for the first and second half of the group. In other words, the next iteration measures the intra-group edges for half of the original groups. This process repeats until the group size reaches 2.

**Example:** Suppose $N = 8$ and $K = 3$. The parallel schedule is of two rounds, each of two iterations, as illustrated by the four curved rectangles (with different colors) in Figure 4.3b.

The first round runs the following two iterations: $\text{measurePar}(\{n_1, n_2, n_3\}, \{n_4, n_5, n_6, n_7, n_8\}, \emptyset)$ which measures the $3 \times 5 = 15$ edges across node group $\{n_1, n_2, n_3\}$ and group $\{n_4, n_5, n_6, n_7, n_8\}$. This is visualized by the horizontal rectangle in blue in the figure. The second iteration is $\text{measurePar}(\{n_4, n_5, n_6\}, \{n_7, n_8\}, \{n_1, n_2, n_3\})$ which measures all $3 \times 2 = 6$ edges and is visualized by the horizontal rectangle in red in the figure.

The second round runs another two iterations: $\text{measurePar}(\{n_1, n_4, n_7\}, \{n_2, n_3, n_5, n_6, n_8\}, \emptyset)$ which measures 5 edges across groups (i.e., edges $(n_1, n_2), (n_1, n_3), (n_4, n_5), (n_4, n_6), (n_7, n_8)$) by the vertical rectangle in green, and $\text{measurePar}(\{n_2, n_5\}, \{n_3, n_6\}, \{n_1, n_4, n_7, n_8\})$ which measures 2 edges (i.e., $(n_2, n_3), (n_5, n_6)$) by the vertical rectangle in orange.

**Complexity Analysis and Configuration of $K$:** On the measurement of a network of $N$ nodes with a group of size $K$, the total number of iterations is $\frac{N}{K} + \log K$ where the first round runs $\frac{N}{K}$ iterations and the second round runs $\log K$ iterations. Roughly, the number of iterations decreases with increasing $K$. However, making the value of $K$ too large would lead to the overflow of mempool as it generates $K \times (N - K)$ transactions in each iteration. In practice, an Ethereum node’s mempool has a capacity of 5120 transactions and to bound the interference, we only use no more than 2000 transaction slots in the mempool. For an Ethereum network of 500 nodes, such as Ropsten, we use $K = 2000/500 = 4$ which results in a total of $500/4 + \log 4 = 127$ iterations.
4.6 Measurement Results

Initially, we run a measurement node $M$ that joins an Ethereum network, such as the Ropsten testnet. The measurement node $M$ is set up without bounds on its neighbors, so it can be connected to the majority of the network.

4.6.1 Measurement Validation

The correct functioning of TOPOSHOT relies on several factors that may vary in a deployed Ethereum network. For instance, TOPOSHOT assumes the default mempool size on Geth nodes (i.e., 5120) that may not hold if an Ethereum node is configured with a different mempool size. The variance would introduce false negatives into TOPOSHOT results and affect the recall. In this subsection, we validate the TOPOSHOT results by evaluating/estimating the result recall.

**Experiment setup:** In addition to the measurement node $M$, we set up a local machine to play node $B$; the node joins the testnet being measured (e.g., Ropsten) and is configured with a number (e.g., 5000) larger than the size of testnet. After staying online for 12 hours in Ropsten, node $B$ connects to 520 nodes, among which 471 nodes run Geth clients. The setup here is external as nodes $A$ and $B$ join a remote Ethereum network.

**Validating measurement primitive** (*measureOneLink*): We then iterate through the 471 nodes, selecting each node as node $A$ to measure the connection between $B$ and $A$ using an unmodified TOPOSHOT. In each iteration, the connection is measured three times. When running the measurement primitive, we verify that $tx_C$ is evicted from nodes $A$ and $B$. This is done by turning on the RPC interface and sending an `eth_getTransactionByHash` query to it. The final result is positive (i.e., there is a connection) if any of the three measurements is positive. For each unit experiment, we report the number of positive connections TOPOSHOT can detect and from there calculate the recall.

We increase the number of future transactions sent in TOPOSHOT and measure the recall using the validation method above. The results are shown in Figure 4.4a. With the increasing number
of future transactions, the recall of TOPOSHOT grows from 84% to 97%. An implication here is that even with a large number of future transactions, TOPOSHOT does not reach 100% recall. We suspect the following culprits: 1) The remote node is configured with a custom mempool size much larger than the default 5120. 2) The node is configured with a custom Gas price threshold other than the default 10%; this threshold determines the mempool’s transaction replacement policy. 3) There are nodes who join Ropsten testnet but do not participate in forwarding transactions, preventing $tx_A$ from being propagated.

**Validating parallel method (measurePar):** In the same experiment, we then validate TOPOSHOT’s parallel measurement method.

Recall parallel TOPOSHOT is parameterized with $p$ and $q$. In this experiment, we use $q = 1$ and vary $p$ (referred to as the group size), that is, a node $B'$ and $p$ nodes $A$’s in a parallel measurement. $p$ is varied between 1 to 99.

Specifically, we set up a new node $B'$ with the default 50 active neighbors and join the Ropsten testnet. It turns out its 35 active neighbors run Geth. We then serially measure the 35 neighbors, which successfully detects 29 neighbors. When running validation of the parallel method, we need to choose $p$ nodes $A$’s. When $p \leq 29$, we choose a subset of the 29 active neighbors of node $B'$ to play nodes $A$’s. When $p > 29$, we choose the 29 neighbors of node $B'$, as well as the nodes that

![Fig. 4.4: Measurement validation results.](image)
do not have connections with node $B'$, to be nodes $A$'s.

For each group size, we run the parallel measurement three times and report a positive result if any of the three returns a positive result. The results are presented in Figure 4.4b. The precision is always 100%. The recall is initially 100% until the group size is larger than 29. It then decreases as the group grows larger. For a group of 99 nodes, the recall is about 60%. The reason for a non-100% recall under a large group is that TOPOSHOT does not guarantee isolation among nodes $\{A\}$, and a larger group increases the chance of non-isolation/interference among nodes $\{A\}$.

![Figure 4.5: Speedup of TOPOSHOT’s parallel measurement over the serial measurement](image)

![Figure 4.6: Node degree distribution in Ropsten](image)

**Measurement speedup of the parallel method:** We also report the time of measuring the same group of nodes with varying group sizes, with the purpose of evaluating possible speedup by the parallel measurement over the serial one. In a similar experiment setup, the measurement target is a group of 100 nodes. With about 4950 edges detected, the measurement times are reported in Figure 4.5. It can be seen that as the group size $K$ increases, the time to measure the same group of nodes (as in the previous experiment) decreases significantly. For instance, with a group size $K = 30$, the measurement time is reduced by an order of magnitude (about $10 \times$ times).
### 4.6.2 Testnet Measurement Results

**Ropsten Results**

<table>
<thead>
<tr>
<th>Community index</th>
<th>No. of nodes</th>
<th>Intra-comm. edges (density)</th>
<th>Inter-comm. edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>92</td>
<td>423 (10%)</td>
<td>1547</td>
</tr>
<tr>
<td>2</td>
<td>142</td>
<td>603 (6%)</td>
<td>1612</td>
</tr>
<tr>
<td>3</td>
<td>107</td>
<td>548 (9.7%)</td>
<td>1827</td>
</tr>
<tr>
<td>4</td>
<td>84</td>
<td>391 (11%)</td>
<td>1505</td>
</tr>
<tr>
<td>5</td>
<td>75</td>
<td>379 (14%)</td>
<td>1704</td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td>127 (10%)</td>
<td>773</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>121 (18%)</td>
<td>840</td>
</tr>
</tbody>
</table>

Table 4.5: Detected communities in Ropsten testnet

We first conduct a measurement study on Ropsten testnet. We use the parallel measurement method with parameter $K = 60$. In particular, the testnet is underloaded and there are not sufficient “background” transactions in mempools. We try to apply TOPOSHOT, as is, to measure Ropsten and found that however low the Gas price we set for $tx_C$ (recall Step 1 in TOPOSHOT), the transaction will always be included in the next block, leaving no time for accurate measurement. To overcome this problem, we launch another node that sends a number of “background” transactions (from a different account than $tx_C$). This effectively populate an operating mempool and help $tx_C$ stay in a mempool for long enough during the measurement period. We encounter the same
situation when measuring Goerli and use the same trick here. Note that more than 95% of peer nodes our supernodes initially connect to stay connected throughout the measurement period.

In the testnet, a target node may run a non-default setting in which they forward future transactions, invalidating the assumption made in TOPOSHOT. Such custom node is avoided in our measurement study as follows: In the pre-processing, one launches an additional monitor node (to the measurement node) to connect to the target node. The measurement node then sends a future transaction to the target node. If the monitor node observes the future transaction from the target node, the target node is removed from the measurement. Besides, the pre-processing phase in TOPOSHOT also avoids unresponsive nodes.

We present a snapshot of the Ropsten testnet taken on Oct. 13, 2020. The precision of the measurement result is 100% and recall is 88% (under group size $K = 60$), using a validation method described above. The network contains 588 (Geth) nodes and 7496 edges among them. This result has the test node and its edges excluded. The degree distribution is plotted in Figure 4.6. Most nodes have a degree between 1 and 60: Particularly, 4% of nodes have degree 10, another 4% have degree 1 and another 4% have degree 12. Omitted in the figure are ten nodes with degree between 90 and 200. This result shows that degrees by active links are much smaller than the default number of inactive neighbors (250).

Table 4.4 summarizes the characteristics of the measured testnet in terms of distances, assortativity, clustering and community structure. 1) For distances, the network diameter, defined as the maximal distance between any pair of nodes, is 5, and the radius is 3. The number of center nodes and periphery nodes, defined respectively as the nodes with eccentricity equal to radius and diameter, are both 36. 2) Degree assortativity, which measures how likely a node connects to a similar node, is -0.1517. 3) The clustering coefficient, which shows how well nodes in a graph tend to form cliques together, is 0.207. The transitivity, which considers the clustering of a particular 3-node substructure, is 0.127. 4) There are 60.748 unique cliques detected in the testnet. The modularity of the testnet, which measures the easiness of partitioning the graph into modules, is 0.0605.
As a baseline for comparison, we generate a random graph following the Erdos-Renyi (ER) model which generates an edge between each pair of nodes using the same probability, independently. It follows a binomial degree distribution and is commonly used as the network-analysis baselines. We use the same number of vertices and edges with the measured Ropsten network (that is, \( n = 588 \) and \( m = 7496 \)) when generating the Erdos-Renyi random graph. We run the graph generation algorithms for 10 times and report the average properties of these random graphs in Table 4.5. Particularly, the density is calculated by the number of measured intra-community edges divided by the number of total possible edges in that community. For instance, the density of a community of 92 nodes and 423 intra-community edges is \( \frac{423}{\binom{8000}{2}} = 0.10 \). Besides, Table 4.5 shows other two random graphs, namely configuration model (CM), and Barabasi-Alber (BA). The former is generated using the same sequence of node degrees with the measured testnet, and the latter is generated using the same number of nodes \( (n = 588) \) and the same average node degree \( (l' = 26) \).

Compared with the ER random graph, the measured Ropsten network has a much larger diameter, a smaller center size, a larger clustering coefficient, and more importantly, fewer cliques and lower modularity. This is similarly the case when comparing Ropsten with CM (except for CM’s comparable diameter) and BA (except for BA’s comparable number of cliques). The implication is that a Ropsten network is much more resilient to network-partition attacks (e.g., eclipse and other DoS attacks) than these random graphs.

We also detect the communities in the Ropsten testnet, using the NetworkX tool implementing the Louvain method described in [105]. The results are in Table 4.5. There are seven communities detected. The biggest community is community number two with 22% of the nodes of the network. The average degree in the community is 19, and 9% of the nodes (i.e., 13 out of 142 nodes) only have a degree of 1. By comparison, community number five contains 12.7% of the nodes with the largest average degree 32.8.
Summary of Rinkeby & Goerli Results

We conduct similar measurements on two other major Ethereum testnets, Rinkeby and Goerli. From our measurement results, Rinkeby has smaller node degrees than Ropsten. Many Geth nodes in Rinkeby are with degrees smaller than 15, and between degrees 15 and 180 the nodes are evenly distributed. In terms of graph statistics, Rinkeby’s modularity (0.0106) is much lower than Goerli’s (0.048) which is comparable with Ropsten’s modularity (0.0605); this result implies that Rinkeby’s the most resilient against network partitioning.

Explaining the results: In the measurement results, we consistently observe smaller modularity in testnets than that in random graphs. Full explainability of the measurement result is challenging and out of the scope of this chapter. We take preliminary efforts to explain the measured results as follows.

We suspect the measurement results, particularly the discrepancy to the properties of random graphs and the much lower modularity, are due to the way Ethereum nodes choose/promote active links and the scale of the networks measured. Briefly, in the Ethereum protocol, a node maintains a “buffer” of inactive neighbors from which 50 active neighbors are selected in the case that existing active neighbors go offline. At the first glimpse, the presence of this buffer localizes the selection of active neighbors in a smaller candidate set than all the nodes as in the random graph, and it should facilitate forming the network of higher modularity. However, by looking more closely at the Ethereum protocol, a node $N$’s candidate buffer consists of node $N$’s inactive neighbors and node $N$’s inactive neighbors’ inactive neighbors. For instance, with each node of 272 inactive neighbors by default, the buffer size is $272 \times 272 = 73984$ which is larger than the size of the testnets we measured. Thus, the effect of localization is not materialized in the testnet results. In fact, the deduplication of active neighbors (i.e., Ethereum clients, such as Geth, check if a recently promoted neighbor is already an active neighbor) may contribute to the much lower modularity in the measured testnets. While here we explain the measurement results by qualitative analysis, we leave it to the future work on the quantitative modeling and analysis of Ethereum network-connection protocols.
4.6.3 Mainnet Measurement Results

Measuring the mainnet’s topology raises new challenges: 1) Due to ethical concerns, the measurement should not interfere the normal operation of live mainnet nodes. 2) Due to mainnet’s large scale (about 8,000 nodes and \( \binom{8,000}{2} \approx \frac{1}{2} \cdot 8,000 \cdot (8,000 - 1) \) possible links) and the high price of Ether, measuring the entire network of mainnet incurs high cost, estimated to be more than 60 million USD as will be analyzed.

To tackle the ethical challenge, we propose a TOPOSHOT extension to additionally verify certain conditions and ensure the non-interference to the service of target mainnet nodes. To bypass the high-cost challenge, in this work, we focus on measuring the topology of a small but critical subnetwork, instead of the entire mainnet.

We conduct the measurement study on the mainnet on May 11th, 2021 and have spent 0.05858 Ether (amount to 197.94 USD at the price as of Aug 2021).

**Non-interference extension of TOPOSHOT:** Consider a measurement node \( M \) runs TOPOSHOT against a target node \( S \) in the Ethereum network \( C \) (\( S \) can be either \( A \) or \( B \) in our measurement primitive as in Figure 4.2b). Suppose the measurement starts at time \( t_1 \) and ends at \( t_2 \). Node \( M \) sets \( tx_C \)'s Gas price at \( Y = Y_0 \) and monitors the following two conditions. Only when both conditions are met, the measurement proceeds.

V1) All blocks produced in \([t_1, t_2 + e]\) are full in the sense that the Gas limit of each block is filled. \( e \) denotes the expiration time of an unconfirmed transaction buffered in an Ethereum node. For instance, a Geth node would drop an unconfirmed transaction \( e = 3 \) hours after it is submitted to the node, if it is not mined.

V2) In the blocks produced during \([t_1, t_2 + e]\), the included transactions have Gas prices higher than \( Y_0 \).

This extended TOPOSHOT achieves the following non-interference property: *The Ethereum blocks produced with the measurement turned on include the same set of transactions with the blocks produced with the measurement turned off.*
**Goal: mainnet’s critical subnetwork:** With the above pricing strategy, measuring a pair of nodes on the mainnet costs $7.1 \cdot 10^{-4}$ Ether or 1.91 USD (at the Ether price as of May, 2021). Thus, for the mainnet that consists of more than 8,000 nodes, measuring all $\frac{1}{2} \cdot 8,000 \cdot (8,000 - 1)$ possible links would cost $22.845 \cdot 10^3$ Ether or more than 60 million USD. We thus refrain from directly measuring the entire mainnet in this work.

Instead, we choose a smaller but critical subnetwork of the mainnet to measure. Our observation is that in today’s blockchain networks, essential transaction activities are centralized to few popular “services” that account for a small portion of the nodes in the network, such as popular transaction relay service (e.g., SrvR1 that relays 63% of Ethereum transactions on the mainnet) and mining pools.

We aim to answer the following question: *Do Ethereum mainnet nodes prioritize the critical service nodes as their active neighbors?*

To address the research, we design a measurement study on the mainnet that 1) discovers Ethereum nodes running behind the known popular services (including transaction relay and mining pools) and then 2) uses TOPOSHOT to measure the pair-wise connections among the discovered service-backend nodes.

**Step 1: Discovering critical nodes:** We discover the mainnet nodes on the backend of critical services. Our approach is to obtain the client version of the backend nodes by submitting the standard Ethereum RPC query (i.e., `web3_clientVersion`) through the service frontend and to match the version against the ones in the Ethereum handshake messages received on a local “supernode” joining the mainnet. The latter step (of passively launching a supernode) is similar to the existing measurement study [67].

Using the above method, we discover the following mining-pool nodes on the mainnet: 59 SrvM1 nodes, 8 SrvM2 nodes, 6 SrvM3 nodes, 2 SrvM4 nodes. We also discover the following transaction-relay nodes on the mainnet: 48 SrvR1 nodes and 1 SrvR2 node. When discovering the nodes, we use the codename revealed through the `web3_clientVersion` query.

**Step 2: Measuring topology among critical nodes:** We run the extended TOPOSHOT to
detect whether critical nodes discovered as above are connected with each other. We consider three possible connection types: the inter-connection between a mining-pool node and a relay-service node, the connection between two mining-pool nodes and the connection between two relay-service node. For each case, we select random nodes from each service and measure all possible links. For measuring the connection between “SrvR1- SrvM1”, for instance, we select two random SrvR1 nodes and two random SrvM1 nodes, and measure the four combinations of links. In addition, we select two nodes for SrvM2 and select one node for each of the services: SrvR2, SrvM3, SrvM4. In total, we choose 9 mainnet nodes.

We report the result in Table 4.6. We make the following observation: 1) A node behind relay service SrvR1 connects to all tested mining pools and other SrvR1 nodes. It does not connect to other relay services such as SrvR2. 2) The single node behind relay service SrvR2 does not connect to any mining pools or other relay service. Here, SrvR2’s node may randomly choose neighbors as vanilla Ethereum clients do. 3) Nodes behind all mining pools connect to nodes of the same pool and other pools. They also connect to SrvR1. The only exception is that SrvM1 nodes do not connect to other SrvM1 node.

**Explaining the results:** There are two possible explanations of the results: a) SrvR1 and all mining pools run supernodes internally, which connect to all other nodes. Blockdaemon runs a regular node that declines incoming connection requests once its active neighbors are full. b) SrvR1 nodes prioritize the connection to its own nodes and mining-pool nodes. They don’t prioritize other RPC-service nodes like SrvR2. So are the mining pool nodes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Connection</th>
<th>Type</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrvR1- SrvM1</td>
<td>✓</td>
<td>SrvM1- SrvM1</td>
<td>✗</td>
</tr>
<tr>
<td>SrvR1- SrvM2</td>
<td>✓</td>
<td>SrvM1- SrvM2</td>
<td>✓</td>
</tr>
<tr>
<td>SrvR1- SrvM3</td>
<td>✓</td>
<td>SrvM1- SrvM4</td>
<td>✓</td>
</tr>
<tr>
<td>SrvR1- SrvM4</td>
<td>✓</td>
<td>SrvM1- SrvM3</td>
<td>✓</td>
</tr>
<tr>
<td>SrvR2- SrvM1</td>
<td>✗</td>
<td>SrvM2- SrvM2</td>
<td>✓</td>
</tr>
<tr>
<td>SrvR2- SrvM2</td>
<td>✗</td>
<td>SrvM2- SrvM3</td>
<td>✓</td>
</tr>
<tr>
<td>SrvR2- SrvM3</td>
<td>✗</td>
<td>SrvM2- SrvM4</td>
<td>✓</td>
</tr>
<tr>
<td>SrvR2- SrvM4</td>
<td>✗</td>
<td>SrvM3- SrvM4</td>
<td>✓</td>
</tr>
<tr>
<td>SrvR2- SrvR1</td>
<td>✗</td>
<td>SrvR1- SrvR1</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.6: Connections among critical nodes
Summary of measurement costs/time: We summarize the measurement costs/time in Table 4.7, which reports the actual Ether cost spent for measuring the testnets and the estimated cost of measuring the full topology of mainnet. The mainnet cost is estimated by multiplying the pairwise-measurement cost by the number of possible edges in the network (as mentioned before). Note that in the mainnet, the measurement transactions’ Gas prices are set to be higher than at least 10% of the pending transactions in the mempool (for estimation purposes, we assume the target node’s mempool has the same content with the measurement node’s mempool).

<table>
<thead>
<tr>
<th>Network</th>
<th>Size (# of nodes)</th>
<th>Cost (Ether)</th>
<th>Date</th>
<th>Duration (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ropsten</td>
<td>588</td>
<td>0.067</td>
<td>Oct. 30, 2020</td>
<td>12</td>
</tr>
<tr>
<td>Rinkeby</td>
<td>446</td>
<td>2.10</td>
<td>Nov. 15, 2020</td>
<td>10</td>
</tr>
<tr>
<td>Goerli</td>
<td>1025</td>
<td>0.62</td>
<td>Oct. 20, 2020</td>
<td>20</td>
</tr>
<tr>
<td>mainnet</td>
<td>9</td>
<td>0.05858</td>
<td>May. 15, 2021</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 4.7: Summary of measurement studies on the testnets/mainnet. # refers to “number”.

4.7 Ethical Discussion

In this chapter, we use TOPOSHOT to measure testnets. While the approach is an active measurement (to refill underwhelmed mempool in the testnet), the testnets do not run business, and the possible service interruption to the testnets will have limited impacts. We also measure a limited sub-network on the Ethereum mainnet. As analyzed before in §4.6.3, the presence of measurement using the TOPOSHOT extension does not affect what set of transactions are included in the blockchain. We believe TOPOSHOT’s impact on normal transactions when measuring the mainnet is small.

4.8 Summary

This chapter presents TOPOSHOT, a measurement study that uncovers Ethereum’s network topology by exploiting transaction replacement and eviction policies. TOPOSHOT achieves the perfect precision and high recall. A parallel schedule is proposed to apply the pairwise measurement to
large-scale networks. TOPOSHOT uncovers the topology of three major Ethereum testnets, which show their difference with random graphs and high resilience to network partitioning. We also use TOPOSHOT to measure critical service interconnection in the mainnet, which reveals biased neighbor selection strategies by top mining pools and relay service nodes.
CHAPTER 5
DENIAL OF MEMPOOL SERVICE AT LOW COST

5.1 Introduction

Today, operational blockchains have grown into complex ecosystems providing a wide variety of services to decentralized application (DApp) users, such as mining pools, transaction relay, and DApp-specific services (e.g., Gas station in Ethereum), etc. Denying these services is a real threat (e.g., the Nov. 2020 incident that disrupts a popular Ethereum relay service [106], denial of services/DoS attacks among Bitcoin mining pools [36, 37, 38], and Bitcoin spam campaign [8]). Such a threat is of interest to some actual blockchain participants, for instance, a service provider competing with the victim service for the same customer base or a DApp user racing to win an auction over peer users.

Related works: In the existing literature, blockchain DoS security has been examined at different system layers, including P2P networks [1, 2, 3, 4], mining-based consensus [5, 6], transaction processing [8, 9, 10, 11], and application-level extensions such as smart contracts [12, 13, 14] and DApp (decentralized application) services [73]. Despite the extensive research, most existing works consider powerful institutional attackers, such as the ones able to control a significant
portion (e.g., 51% or 21%) of computing power in a large blockchain [5], or the ones who can corrupt the underlying network infrastructure like ISP insiders [3, 4], or the botnet which has tens of thousands of IP addresses at her disposal for an eclipse attack [1], or a Bitcoin mogul willing to spend tens of thousands-USD worth of Bitcoin to launch a spam campaign [8]. Recently proposed are a class of low-cost attacks exploiting miner extractable value (MEV) in which the attacker send crafted transactions to front-run other transactions [18, 19, 16] and/or to bribe miners [107, 108, 109]. This class of attacks assume rational miners and have limited impacts (on few targeted victim transactions, instead of all transactions submitted in a period). The full related works are in § 5.8. It is an open research problem whether an average user can mount a low-cost attack to disable a large-scale blockchain.

**Attack goals:** This work aims at low-cost DoS attacks targeting a few blockchain nodes to cause large-scale impacts on the Ethereum-DApp ecosystem. The observation is that despite the original design to decentralize control, practical blockchain services, notably mining pools and transaction relay services, are highly centralized [110], a potential single point of failure. For instance, Ethereum blockchain’s connection to millions of its active DApp users is controlled by a single relay service (infura.io [77]), whose outage in Nov. 2020 [106] (allegedly triggered by a competing service provider [111]) prompts major exchange platforms to halt Ether withdrawal, leading to a global panic among Ether holders. We thus consider the threat model depicted as in Figure 5.1 where an attacker node sends crafted transactions to a victim node running in a DApp service’s backend (e.g., a mining pool or a relay service) which propagates transactions and blocks with the rest of Ethereum network. The attacker’s goal is to disable the victim node’s service to other nodes and DApp users at a low Ether cost. A baseline design is for the attacker account to send a large volume of transactions to occupy the limited block space and to squeeze out normal transac-
tions there. However, the spam transactions sent in this baseline have to be with high Gas prices, burdening the attacker with high cost (e.g., 12.5 Ether per block as analyzed in § 5.3). Also, this baseline benefits miners by actually increasing their revenue, while this work aims to victimize the miners. Note that the attack goal in this work is to prevent a downstream miner from accessing the content of a mempool (or txpool\textsuperscript{1}) which is distinct from the goal of MEV-exploiting attacks \cite{18,19,16,107,108,109} where the miner can access a txpool content and is incentivized to misbehave on it. Also, MEV-exploiting attacks make assumption on rational miners, which is not necessary for this work which aims to victimize any miners.

**Attack design:** To achieve the threat goals, this work exploits an understudied subject in the existing security literature, that is, denying the txpool service. On an Ethereum node, txpool is an essential data structure that buffers unconfirmed transactions received from other nodes until they are included in the blockchain by miners. A disabled txpool presenting a false empty view of no transactions can cripple the victim node’s mining and transaction propagation, and further lead to a global service disruption, as confirmed by our evaluation.

To disable a txpool, our observation is the following: An Ethereum client’s decision in whether to admit an incoming transaction affects the revenue of the miner running the client. Thus, any Ethereum client developed to attract revenue-hungry miners and increase adoption needs to maximize miners’ revenue. To do so, real Ethereum clients are designed to take loose but risky actions to admit transactions, as we measured. The key idea of this work is to exploit real Ethereum clients’ risky txpool behavior in admission control and to design denial-of-service attacks at low Ether cost, namely Denial of Ethereum’s Txpool sERvice (DETER attacks). More specifically, one can trick a txpool to admit invalid transactions at zero Ether cost that evict and victimize existing normal transactions there. In the following, we describe two types of transactions that we exploit to design DETER attacks.

A future transaction in Ethereum (which is similar to the concept of orphan transaction in Bitcoin terms) is the one that arrives at a node earlier than its logical predecessor (i.e., the transaction

\textsuperscript{1}txpool in Ethereum is the same concept with the mempool in Bitcoin.
of smaller nonces as will be introduced in § 5.2). On a node, a future transaction at the time of arrival, say \(tx_2\) with nonce \(n + 2\), can wind up two outcomes when the node’s miner reads it from the txpool: X1) It remains a future transaction which is invalidated by the miner and whose transaction fee cannot be collected, or X2) it is transformed into a valid (or pending) transaction upon the arrival of a subsequent transaction \(tx_1\) of nonce \(n + 1\). In the latter case, \(tx_2\)’s fee can be collected by the miner. To increase miners’ revenue, all Ethereum clients, including Geth (Go) [87], Parity (Rust) [91], Nethermind (.Net) [94], and Besu (Java) [92] as we measured, support admitting future transactions, in the (optimistic) hope that case X2) will occur along the way and their transaction fees are collectible. In this work, we propose the first attack, named DETER-X, that exploits Ethereum’s falsely optimistic design by deliberately sending future transactions of case X1) and at a high Gas price so that they are admitted to the txpool and evict normal transactions, yet without being charged due to transaction invalidity. This attack victimizes not only the senders of the normal transactions evicted (by \(tx_1\)) but also the miners who end up mining no transactions and get low or even zero revenue. Note that at the time of \(tx_2\)’s arrival, which outcome, X1) or X2), will occur fully depends on the arrival of \(tx_1\), which is unpredictable.

The second type of Ethereum transaction exploited is what we call latent invalid transactions, that is, the transactions that are valid and admitted at the time of arrival, but are transformed into invalid transactions later when the miner reads it. Consider a sequence of transactions, \(tx_0, tx_1, tx_2, \ldots\), sent from the same account \(e\) and each of which spends account \(e\)’s full balance. Similar to the previous attack, this transaction sequence can wind up with two likely outcomes: Z1) without the interference from other transactions, this transaction sequence will be invalidated by the miner except for \(tx_0\), as \(tx_i(\forall i > 1)\) overdrafts \(e\)’s balance. Z2) upon the arrival of subsequent transactions to refill \(e\)’s account, \(tx_i(\forall i > 1)\) may be transformed into valid transactions whose fees are collectible by miners. As we measured, all Ethereum clients deployed on the mainnet take the risk to admit latent overdraft transactions in their txpool. We thus propose the second attack, named DETER-Z, to exploit this risky design by sending overdraft transactions at a high Gas price.

\[\text{In this work, we discard the cases of Aleth (C++) [93] and Trinity (Python) [112] clients, because no mainnet nodes run the client and their code maintenance is discontinued (e.g., as of Apr. 2021 for Aleth).}\]
to evict normal transactions in the txpool, at low Ether cost. In particular, we propose DETER-Z variant optimized for Geth’s admission control so that the latent invalid transactions do not only evict transactions but also occupy the txpool for an extended period of time. We also propose evasion strategies on other clients to bypass their limits on the transactions from the same account.

The DETER vulnerabilities are specific to several unique designs in the Ethereum blockchain, which may render their applicability beyond Ethereum limited. Concretely, the cause of DETER-X vulnerability can be attributed to Ethereum’s support of future transactions, which meet the real DApps’ demands to send transactions hastily, as evidenced in measurement studies such as [113]. Likewise, the DETER-Z vulnerability has the root cause in Ethereum’s account model in which account balances can be arbitrarily updated, which is necessary to support smart contracts desirable by many DApps. By contrast, Bitcoin’s UTXO model features limited updatability in which a transaction output can transition only from the unspent state (i.e., UTXO) to the spent, but not the other way around. The one-way updatability in the UTXO model renders Bitcoin immune to the DETER-Z attacks, as we investigated. Note that the two features exploited by DETER-X/Z attacks, namely future transactions and updatable accounts, are generic in the Ethereum protocol and are independent of specific client implementations.

**Measurements and impacts:** We evaluate both DETER attacks’ impacts systematically on an extensive list of victim services in different settings. First, we evaluate the effectiveness and cost of DETER attacks against four Ethereum clients in a local setting (Geth/Parity/Nethermind/Besu) that show DETER-X achieves a 100% success rate at zero cost on all four clients except for parity with a success rate of 75.6% and zero cost. DETER-Z is with 100% success rate against all four clients and has a cost as low as 0.021 Ether per block, a three-order-of-magnitude saving from the baseline [8]’s cost of 12.5 Ether per block (§5.5).

Second, we evaluate the DETER’s effectiveness in disabling blockchain mining: Against a single miner, we propose attack strategies that guide the timing of sending DETER-X payload by

---

8Sending a transaction, say $tx_2$, hastily means sending it without waiting for the confirmation of other transactions that $tx_2$ depends on. For instance, such a dependent transaction, say $tx_1$, can be such that $tx_2$’s nonce is $tx_1$’s nonce plus one.
predicting the block arrival time based on the Poisson model [114][115]. The attack evaluation on a local network shows that mounting DETER-X/Z attacks at a rate of 4 crafted messages per second against a single miner node can persistently reduce the block size (i.e., total Gas of transactions included in a block) by 88.8%/99.2%, at a 0/0.021 Ether per block. In addition, we evaluate DETER’s effectiveness on a number of txpool-based services beyond mining, which includes transaction propagation, Gas stations [116] and other DApp services. The results confirm DETER’s effectiveness and low costs.

Third, we measure the mainnet and testnets’ exploitability under DETER attacks. Measuring the mainnet in the presence of testnet results is necessary as a network’s exploitability, described next, is specific to the network’s deployment. Given each network, we measure whether critical nodes can be discovered (node discoverability) and whether the discovered node can be successfully attacked (node exploitability). For node discoverability, we propose two complementary measurement methods to improve the result accuracy. Measuring node exploitability in a live network poses challenges, as 1) the target node is a blackbox to us except for a minimal information-propagation interface, and 2) the measurement is heavily restricted by ethical concerns, esp. in the mainnet. We tackle these challenges and propose lightweight probe tests by exploiting Ethereum’s support of transaction replacement. The key idea is that a node’s internal state, such as a transaction’s presence in the txpool, can be detected by the success of an attempt to replace this transaction with the one at a carefully selected Gas price; the success/failure of such an attempt is observable.

Our exploitability study shows all the mainnet nodes we discover are vulnerable under the proposed probes. The result on Ethereum’s Ropsten [84] and Rinkeby [85] testnets shows that mounting DETER-X (DETER-Z) attacks on Rinkeby [85]’s top-5 miners can effectively reduce the testnet’s number of transactions included in a block by \(\frac{1}{77}\times (to \ zero)\) (§ 5.6.2).

**Mitigation:** DETER vulnerabilities are not by accident, and they are caused by the fundamental difficulty of designing a “perfect” txpool in the tradeoff between DoS mitigation and miners’ revenue on Ethereum. Specifically, mitigating DETER attacks perfectly without losing any min-
ers’ revenue is hard, because it requires such a txpool to look into the future just to make an admission decision at present. For instance, suppose an arriving transaction, say tx, is currently a future transaction. A perfect txpool needs to admit tx if the transaction turns into a profitable pending transaction in the future, or otherwise, decline tx. However, whether tx would turn into a pending transaction depends on the arrival of subsequent transactions (e.g., another tx’ from the same sender but with a smaller nonce), and thus is uncertain. With this uncertainty, a practical txpool may have to risk admitting a current pending transaction whose (high) fees may end up being non-collectible and/or declining a current future transaction whose (high) fees can become collectible in the end. The former decision is a DoS exploit while the latter one leads to a loss of miner revenue.

Thus, in this work, we propose heuristics for txpool admission control. In the proposed schemes, we define necessary (but maybe insufficient) conditions to describe a DETER transaction and use them to detect/decline transactions. In other words, the scheme puts attack mitigation over miner revenue preservation. We evaluate the proposed mitigation schemes and show that 1) the mitigation schemes fail all DETER probe tests, resulting in zero attack success rates, and 2) the mitigation schemes well preserve and even increase miners’ revenue under real Ethereum transaction traces. We conduct the cost evaluation by replaying the transaction traces collected from the mainnet.

Contributions of this work are listed as follows.

• New attacks: We discover Ethereum clients’ vulnerability in managing unconfirmed transactions. We design two low-cost attacks, DETER-X/Z, to disable a remote Ethereum node’s txpool service. We propose attack strategies targeting various node services in mining, transaction relay, and Gas station.

• New understanding: We measure and verify DETER-X/Z’s effectiveness and low cost extensively, in the settings of a local node running different Ethereum clients, testnets and the mainnet. We propose a non-trivial method to detect the exploitability of a blackbox mainnet node by exploiting Ethereum’s transaction replacement support. The results show DETER-X/Z vulnerability
widely exists among Ethereum clients, a DETER attack can cause testnets (Rinkeby and Ropsten) to produce empty blocks at zero Ether cost, and mainnet nodes underneath critical services can be discovered and are tested exploitable.

- **Mitigation**: We propose mitigation schemes that can reduce the DETER attack success rate to zero while preserving the miners’ revenue. We verify the properties of security and miners’ profitability under real and synthetic transactions.

**Roadmap**: § 5.2 introduces the background of Ethereum transaction processing and txpool. Threat model is presented in § 5.3. Two DETER attacks are described in § 5.4, and strategies to DETER the mining service are presented in § 5.5. § 5.6 presents the measurement study on the exploitability of real-world Ethereum networks under DETER attacks. Mitigation schemes are presented in § 5.7. Related works are discussed in § 5.8. Responsible disclosure is discussed in § 5.9 with the conclusion in § 5.10.

### 5.2 Preliminary

This section presents the background on Ethereum’s transaction buffer, namely txpool, which is the foundation of understanding DETER attacks. To begin with, we first describe the transaction workflow in Ethereum.

**Transaction workflow**: In Ethereum, the life cycle of a transaction begins from its owner account signing the transaction and sending the transaction to an Ethereum node, say $N_1$, typically through a remote-procedure call (RPC) interface. Node $N_1$, receiving the transaction, conducts checks on transaction validity and priority (as will be elaborated on) before buffering it locally in a data structure called txpool and further propagating it to $N_1$’s neighbors, say one of which is $N_2$. $N_2$ similarly propagates the transaction to its neighbors, and the process repeats until the transaction is propagated to the entire blockchain network. In the network, each miner node selects a group of unconfirmed transactions buffered from its txpool and runs mining algorithms on the selected group of transactions. Miners who find solutions to the mining puzzle prepare a block and
propagate it to the network. A node receiving multiple blocks (at the same height) selects the first one and verifies its mining solution. If it passes, the node removes the transactions included in this block from its local txpool.

**Fig. 5.2:** Ethereum transaction workflow and the role Node $N_1$’s txpool plays in it.

**Txpool operations:** Figure 5.2 illustrates the workflow of transaction and block propagation described above. It takes a txpool-centric vantage point on Node $N_1$. The txpool supports three essential operations: 1) An incoming transaction, either directly sent from the sender account or propagated from a neighbor of $N_1$, is admitted into txpool (transaction admission). Admitting a transaction $tx$ may trigger two txpool events: 1a) eviction of an existing transaction $tx'$ by $tx$ where $tx$ and $tx'$ are of different sender accounts or nonces, and 1b) replacement of an existing transaction $tx'$ by $tx$ where $tx$ and $tx'$ are of the same sender and nonce. 2) An existing transaction in the txpool is read by a downstream component (transaction read); for instance, a group of transactions in the txpool may be selected by a miner to include in the next block. Or transactions in the txpool may be read to determine the validity of an incoming transaction. Or unconfirmed transactions in txpool are read by DApp-specific (decentralized application) services such as Gas stations [117, 116] (to suggest cost-effective Gas prices in real time), DeFi frontrunning bots [118], RPC queries (e.g., the RPC API txpool_content [119]), etc. 3) Unconfirmed transactions in the txpool are emitted to the local miner upon finding or receiving a valid block of transactions.
**Ethereum transactions**: An Ethereum transaction binds a sender account to a receiver account, where an account is a public key of an Ether owner. **Nonce**: Ethereum supports “hasty” transaction sending, that is, an account can send a transaction, say $tx$, without waiting for the confirmation of the transaction $tx$ depends on. To enforce a total-order among hastily sent transactions, each Ethereum transaction is associated with a nonce, which records a monotonically increasing counter value per each sender account. In a txpool, a transaction is of state pending, if its nonce equals one plus the maximal nonce of the transactions of the same sender in the txpool (i.e., equal to $n + 1$). Otherwise, if the nonce is strictly larger than $n + 1$, the transaction is a future transaction. A future transaction can be a result of the Ethereum network propagating hasty transactions out-of-order. **Gas price**: In Ethereum, each transaction needs to specify a gas price, that is, the amount of Ether the sender is willing to pay to a miner for each unit of “work” it does for including the transaction into the blockchain. Here, the work refers to the basic transaction validation workload and for executing the smart contract invoked by the transaction. The work unit is Gas.

**RPC services** such as infura.io, etherscan.io, quicknode.io have been the primary means DApp clients use to communicate with the Ethereum blockchain. For instance, the transactions sent through infura.io alone account for at least 63% of all Ethereum transactions [35]. An RPC service receives from DApp clients JSON data on its frontend and sends a transaction packing the data to the blockchain on the backend. Specifically, the RPC service runs several Ethereum nodes on the backend that propagate transactions/blocks with the blockchain network.

**Mining pools**. Minimally, a mining pool mines a block by decomposing a puzzle into several easier versions, sending to miner participants decomposed puzzles and collecting their solutions (so-called shares) before paying out rewards to miners. Today, most blocks on Ethereum are mined by mining pools, and a typical mining pool is a complex service extended with two frontend capabilities: an own RPC service that directly accepts its customers’ transactions and P2P service that propagates transactions and found blocks.

---

[In this work, we don’t exploit the smart-contract capability of Ethereum and only consider the basic transaction functionality for Ether transfer.]
5.3 Threat Model

The threat model consists of four actors in an Ethereum network. An attacker controls an external owned account (EOA) and a full node running Geth client that is connected to an Ethereum network, such as an Ethereum mainnet. A victim node running the Geth client contains a victim txpool and a series of downstream services that read txpool data. The last actor in the threat model is the rest of the Ethereum P2P network. The nodes in the Ethereum network run client software such as Geth and Parity, and they conform to Ethereum’s protocols for transaction/block propagation. The threat model is illustrated in Figure 5.1.

Particularly, we assume the attacker node has been connected to the victim node as its neighbor and can propagate messages of crafted transactions. The feasibility of this assumption is practically evaluated in both Ethereum testnets (§ 5.5) and mainnet (§ 5.6.3); Very briefly, an attacker can discover the victim node, learn the victim’s IP/port/nodeID via measuring the public Ethereum network, and connect to the victim node via passively waiting for the incoming connections from or via proactively initiating the handshakes to the victim node.

The attacker’s goal is to deny the service of txpool to downstream components. More specifically, when the txpool is read by these components, a falsely empty snapshot of txpool is read and legitimate transactions are discarded from the view of the downstream services or other nodes. To do so, for instance, the attacker may purge for once all the transactions currently residing in the txpool and/or occupy the txpool with her own transactions to deny the service to other senders’ transactions.

Via a few disabled txpool’es, the attacker wants to eventually victimize other accounts whose transactions cannot get propagated to the Ethereum network or included in the blockchain. Also, the attacker aims to victimize the miners and decrease their revenue by tricking them to mine on “empty” blocks. The attacker also aims to disable or manipulate DApp services that depend on unconfirmed transactions in txpool, such as manipulating the Gas price prediction by a real-time Gas station running on txpool.

Besides the attack effectiveness, the attacker’s secondary goal is to lower the cost of her attack.
Specifically, the cost of a DETER attack should be lower than the cost of the baseline attack (described next) by orders of magnitude.

**Analyzing a baseline attack:** A baseline attack works by the attacker account sending a flood of spam transactions to “frontrun” other normal transactions and to occupy the limited space of Ethereum blocks. The attacker can do so by configuring her transactions at a higher Gas price than the normal transactions and thus receiving a higher priority to be processed by miners (i.e., 1000 Gwei). To occupy one Ethereum block without any space for normal transactions, the baseline attacker needs to send spam transactions worth Gas of one block limit (i.e., 12.5 million Gas).

We observe the highest Gas price in the recent 200 blocks (from height 12202391 to 12202591) is consistently below 500 Gwei, based on the Gas station [120]. We use 1000 Gwei to estimate the highest Gas price in a block. Thus, the total cost of a baseline attack occupying a single block is $12.5 \cdot 10^6 \cdot 1000 = 12.5$ Ether. As of this writing (on Apr. 2021), this cost is equal to 24795 USD.

The baseline spam attack does not achieve the threat goal aimed in this work: First, the baseline attack incurs a high monetary cost (12.5 Ether for one block). Second, the baseline attack, while victimizing normal transaction accounts, does not deny the service of Ethereum blockchain itself. Particularly, the miners under the baseline attack still receive high revenue, actually higher than normal due to the high Gas price of the spam transactions in the attack.

### 5.4 DETER Attacks at a `txpool`

In this section, we describe the attacks to disable a victim node’s `txpool`. The next section describes the broader impacts of a disabled `txpool` on the downstream services on the victim node and other nodes in the Ethereum network.

We first describe design motivations based on the observation of profiling `txpool` in §5.4.1. We then present two attack designs, DETER-X and DETER-Z, respectively in §5.4.2 and §5.4.3. We propose attack strategies to victimize a local miner in §5.5 and evaluate the attacks against the local miner in §18.
### 5.4.1 Observing txpool’s Eviction Behavior

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>Max txpool length (e.g., default value 5120)</td>
</tr>
<tr>
<td>$L/M/H$</td>
<td>Symbolic value for high/median/low Gas price</td>
</tr>
<tr>
<td>$V/F/I$</td>
<td>Valid/future/invalid transactions</td>
</tr>
</tbody>
</table>

Table 5.1: Notations of parameters for observing txpool’s eviction behavior

Our attack design exploits the eviction (mis)behavior in Ethereum clients’ txpool. This subsection presents generic test cases against a remote blackbox txpool to characterize its eviction behavior.

**txpool tests:** In a txpool of pre-state $S$, an eviction takes as input an incoming transaction $tx$ and produces as output a post-state $S'$ and an eviction victim $tx'$. We denote an eviction event by state transition $(S, \{tx\}) \rightarrow S', \{tx'\}$. When no transaction is evicted in a full txpool, it could be $(S, \{tx\}) \rightarrow S, \varnothing$.

In our model, an Ethereum transaction $tx$ is with two properties: the transaction state $[VIF]$ and Gas price $[HML]$. Transaction state can take values such as valid pending $V$, invalid pending $I$ and future transactions $F$. The property of Gas price can take symbolic values such as high $H$, medium $M$ and low $L$. For instance, $VL$ denotes a valid pending translation with a low Gas price, and an overdraft pending transaction at high Gas price is denoted by $IH$.

Given a txpool of capacity $n$ (i.e., storing at most $n$ transactions), we design two general tests:

\[
t_1 \text{ Test } t_1(x_1 \cdot VL + (n - x_1) \cdot FM, FH) \rightarrow ✓\checkmark\times.\]

The initial state $S$ contains $x_1$ valid pending transactions at low Gas price ($VL$) and $n - x_1$ future transactions at medium price ($FM$). The incoming transaction is a future transaction at a high Gas price ($FH$). All transactions are sent from different accounts.

The partial success of the test, denoted by ✓, indicates that in the target txpool, a future transaction ($\{tx\} = \{FH\}$) can evict a valid pending transaction ($\{tx'\} = \{VL\}$). Additionally, if the evicting future transaction can persist and is stored in txpool, the test is a full success, denoted by ✓. Otherwise, that is, when there is no eviction or the eviction victim is $FM$, the test is a failure denoted by $\times$. 
Test $t_2$: Test $t_2(n \cdot V L, VH + (x_2 - 1) \cdot IH) \rightarrow ✓\, ✓\, ✓$. The initial state $S$ contains $n$ valid pending transactions at low Gas price ($VL$). The incoming message $\{tx\}$ contains a valid transaction ($VH$) followed by $x_2 - 1$ overdraft transactions ($IH$) sent from the same account. These incoming transactions are at a high Gas price $IH$.

The partial success of the test, denoted by $✓$, indicates that the overdraft pending transaction ($\{tx\} = VH + (x_2 - 1) \cdot IH$) can evict valid pending transaction ($\{tx'\} = x_2 \cdot VL$). Additionally, if the evicting overdraft transactions of the same sender can persist in txpool, the test is a full success, denoted by $✓$. Otherwise (i.e., no eviction by any of the $x_2 - 1$ overdraft transactions), the test is a failure denoted by $✗$.

**Test results of real Ethereum clients:** In our study, we set up 1) a measurement node running instrumented Geth and the test code and 2) a target node running one of the following Ethereum clients: Geth (Go), OpenEthereum/Parity (Rust), Nethermind (.net), and Besu (Java). We statically instrument the test Geth node so that it can bypass local checks and propagate any transactions, including future and invalid transactions to the target node. We consider the four clients that are deployed on Ethereum mainnet. The percentage of nodes running these four clients on mainnet is illustrated in the second column of Table 5.2, from which Geth (83%) and Parity (15%) are the dominant clients on the mainnet.

<table>
<thead>
<tr>
<th>Ethereum clients</th>
<th>Percentage</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>DETER-X</th>
<th>DETER-Z (Ether)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geth</td>
<td>83.24%</td>
<td>✓ ($x_1 &lt; 1024$)</td>
<td>✓ ($x_2 &lt; 4096$)</td>
<td>✓ ($x_1 &gt; 1024$)</td>
<td>✓ ($x_2 &gt; 4096$)</td>
</tr>
<tr>
<td>Parity</td>
<td>14.57%</td>
<td>✓ ($x_1 \geq 2000$)</td>
<td>✓ ($x_2 \leq 81$)</td>
<td>✓ ($x_1 &lt; 2000$)</td>
<td>✓ ($x_2 &gt; 81$)</td>
</tr>
<tr>
<td>Nethermind</td>
<td>1.53%</td>
<td>✓</td>
<td>✓ ($x_2 \leq 17$)</td>
<td>✓ ($x_1 \leq 2000$)</td>
<td>✓ ($x_2 &gt; 17$)</td>
</tr>
<tr>
<td>Besu</td>
<td>0.52%</td>
<td>✓</td>
<td>✓</td>
<td>✓ ($x_1 &lt; 2000$)</td>
<td>✓ ($x_2 &gt; 2000$)</td>
</tr>
</tbody>
</table>

Table 5.2: Profiling different Ethereum clients under tests and DETER attacks. For the two DETER attacks, success rates are reported. DETER-Z’s Ether cost is also reported in parenthesis. Note that the baseline attack incurs a cost of 12.5 Ether per block. The second column refers to the percentage of mainnet nodes running a specific client [88]. Note the Ether cost is tested under a txpool storing transactions of the maximal Gas price 1000 Gwei.
We run the two tests against the four Ethereum clients and evaluate the success of each test. The result is presented in Table 5.2. Test $t_1$ fully succeed on all clients except for certain conditions under Parity and Geth: 1) When $x_1 \leq 2000$, Parity does not evict any pending transaction for an incoming future transaction, rendering a failed $t_1$. 2) On Geth, when $x_1 \geq 1024$, the future transactions do evict existing pending transactions but only 1024 transactions are admitted to the txpool, rendering a partially successful $t_1$. Test $t_2$ is successful on all clients excepts for the case that when $x_2$ is larger than 81 (17), $t_2$ fails on Parity (Nethermind). The failure of these test cases is due to that these clients enforce the limit of transactions from the same account. 3) When $x_2$ is larger than 4096, $t_2$ is partially successful on Geth. That is, sending $x_2 > 4096$ transactions as in $t_2$ to Geth, the txpool exhibits a behavior that evicts $x_2$ existing pending transactions and admits only 4096 transactions (the other $x_2 - 4096$ transactions do evict existing transactions but are not admitted to the txpool by themselves).

5.4.2 DETER-X: Exploit Future Transactions

![Diagram](image)

Fig. 5.3: Workflow of a successful DETER-X attack that purges a victim txpool twice, so that the downstream service reads a falsely empty txpool twice.

**Design motivation:** The property tested in $t_1$ can be exploited to disable a txpool. That is,
property \( t_1 \) implies that a txpool receiving a future transaction of high price chooses an existing pending transaction of low price to evict. One can abuse this property by sending a large number of future transactions to evict all pending transactions on a txpool. To evade certain clients’ limit of transactions from the same account (recall test results \( t_2 \)), the attacker prepares multiple accounts to send the future transactions. Because the future transactions, however high their Gas prices are, do not charge their sender, the attack that only sends future transactions incurs zero Ether cost.

**Basic attack workflow:** Formally, in our threat model, the attacker initially makes a guess about the value \( n' \) that is larger than the victim txpool length \( n \), such as \( n' = 10 \times 5120 \) where 5120 is the default \( n \) in Geth.

\( x_1 \): The attacker sends a crafted message encoding \( n' \) future transactions to the victim node. Each of these future transactions is configured with a very high Gas price, much higher than any existing transactions in txpool (e.g., a practical value is 1000 Gwei).

*The success of the attack (DETER-X)* is defined as that the downstream service on the victim node reads a falsely empty set of transactions from the txpool.

**Attack analysis on Nethermind/Besu:** Among these clients, a single step of \( x_1 \) suffices to evict all pending and future transactions in the txpool. The crafted future transactions will stay in the txpool, preventing the transactions subsequently arriving at the txpool from being admitted. In other words, DETER-X by conducting a single step of \( x_1 \) can *occupy* the txpool forever. Note that those crafted future transactions cannot be removed upon the miner finding the next blocks.

**Attack customization on Geth:** On Geth, a single step of \( x_1 \) can evict all pending and future transactions in the txpool. However, after that, Geth limits the maximal number of future transactions that exist in txpool by \( n_f < n \). Thus, only \( n_f \) (e.g., 1024) crafted transactions sent in \( x_1 \) can persist in the txpool. Subsequently arriving transactions will be admitted to and repopulate the txpool.

Thus on Geth nodes, the attacker needs to periodically run step \( x_1 \) at a certain frequency. Ideally, assume the attacker can know (or predict) the next time point, say \( T_i \), when the txpool will be read by the local miner (e.g., upon finding a block) or another downstream service. Estimating
the next $T_i$ is specific to services and will be described in §5.5.

Knowing $T_i$, the attacker sends her crafted future transaction in $x_1$ right before $T_i$ to reduce the number of subsequent transactions repopulating txpool. After this round, the attacker waits until the next estimated read time $T_{i+1}$. She then repeats the action of sending future transactions in step $x_1$. The entire attack workflow is illustrated in Figure 5.3.

5.4.3 DETER-Z: Exploit Latent Invalid Transactions

We design adaptive attacks to exploit the behavior tested in $t_2$, that is, a txpool may admit invalid transactions by evicting valid transactions. Intuitively, an attacker can send a sufficient number of invalid transactions at a high Gas price to evict existing pending transactions at a lower price, thus purging the txpool. As the invalid transactions don’t charge their senders, this attack is of low cost. We thus propose to construct the basic DETER-Z attack:

$z_1$: In our threat model, the attacker node sends a message of $n'$ pending transactions to the victim node. The pending transactions are sent from the same account of balance $B$, and each of them spends $B$ Ether minus the transaction fee. The nonces of these transactions are consecutive.
We call these transactions latent invalid transactions because each individual transaction is valid but together all \( n' \) transactions except for the first are overdraft transactions. The first transaction with the smallest nonce is denoted by \( tx_0 \). All \( n' \) transactions are with a high Gas price, say \( pc \). \( pc \) is slightly higher than any existing transaction in the txpool.

**Attack analysis on Besu:** Running \( z_1 \) on Besu is effective, as Besu does not limit the number of transactions by the same sender.

**Attack customization on Geth:** Geth allows the eviction of existing pending transactions by latent overdraft transactions, but may not admit these overdraft transactions in the txpool. Running the step \( z_1 \) would leave empty slots in the txpool, which allows the subsequent transactions to be admitted. To avoid this and to make the attack payload occupy the txpool, we propose the following attack workflow.

Against a target Geth node, the attacker first still runs Step \( z_1 \) by sending \( n' \) transactions of the same account. She then takes the second step as follows.

\( z_2 \): The attacker sends a message of \( n' \) future transactions to the victim Geth node. Each \( n_\text{s} \) future transactions of the \( n' \) ones are sent from the same account, so there is a total of \( \frac{n'}{n_\text{s}} \) different accounts. The maximal Gas price of existing transactions in the txpool is 1000 Gwei, which is higher than 99.9% of transactions on the mainnet. The future transactions are configured with the Gas price higher than all existing transactions in the txpool but slightly lower than \( pc \), say \( pc - 1 \). The two-step design of DETER-Z a Geth node is depicted in Figure 5.4.

**Attack analysis on Geth:** After the second step \( z_2 \), the victim Geth node stores the maximal number of future transactions allowed by its txpool. As Gas price \( pc - 1 \) is lower than the pending transactions’ price in \( z_1 \), these future transactions would not evict the latent invalid transactions sent in \( z_1 \).

After the two steps, consider that the victim node receives a legitimately propagated transaction \( tx' \). As a normal transaction, \( tx' \)’s Gas price would be much lower than \( pc \). Thus, it is declined and would not be admitted into the txpool.

Now consider that a block that includes \( tx_0 \) arrives at the victim node or is found by the node
itself. Receiving such a block, the node would evict $tx_0$ and all other pending transactions sent in Step $z1$. Thus, the txpool is fully occupied by the attack payload between $z2$ and when the next block is received.

**Attack customization on Parity/Nethermind**: These two clients limit the maximal number of transactions per sender in their txpool (by $n_s = 81$ and 17, respectively). To evade the limit, the attacker runs a customized Step $z1$ where she creates $n'/n_s$ accounts, and under each account, sends $n_s$ transactions of $VH + (n_s - 1) \cdot IH$. Using multiple accounts increases the cost of the attack.

### 5.4.4 Evaluation on a Local txpool

This subsection presents the evaluation to answer the following question (Q):

**Q1.** How effective and costly is a one-shot DETER attack in disabling a target txpool?

With the same setting (as described in §5.4.1), we run the two DETER attacks. Initially, the target node is loaded with pending transactions to its capacity, that is, $n_p$ transactions sent from $n_p$ different accounts. During the attack, the test node sends a single round of DETER payload (i.e., Step $x1$ in DETER-X and $z1/z2$ in DETER-Z) to the target node running different clients. It then immediately reads the txpool content by issuing a txpool\_content RPC request [119]. We calculate the number of evicted transactions by subtracting from $n_p$ the number of original pending transactions found in the RPC result. The number of evicted transactions over $n_p$ is reported as the attack’s success rate. For instance, an attack of 90% success rate would evict 90% pending transactions that would otherwise be accessible by a downstream service. DETER-X has zero cost by design. For DETER-Z, we then turn on the target node’s mining for long enough and observe which transactions are included in the blocks produced. We report the DETER-Z’s cost by the Ether of the test node’s transactions in all the blocks (i.e., a transaction’s Ether is the product of its Gas and its Gas price).

The results are presented in Table[5.2]. In general, the success rates of DETER-X and DETER-Z are 100% and DETER-Z’s cost is 0.021 Ether per block, with the following notable exceptions:
1) DETER-X against Parity has a 75.6% success rate because Parity disallows the eviction of pending transactions by future transactions when there are fewer than 2000 pending transactions in the txpool. 2) DETER-Z against Parity incurs 2.1 Ether per block, because it limits up to 81 transactions per sender account. A DETER-Z attack has to prepare multiple accounts, each sending 81 transactions, in order to evict all pending transactions. 3) Similarly, DETER-Z against Nethermind incurs 1.28 Ether, because of a similar reason, that is, Nethermind limits a maximal 17 transactions of the same sender.

In summary, a DETER-X attack is always of zero cost (in both Gas and Ether), as future transactions don’t charge fees. A DETER-Z attack’s cost is due to the first transaction, which is always 21000 Gas. In other words, a successful DETER-Z attack can occupy a block space of 16 · 10^6 Gas (i.e., the block Gas limit) by abusing only 21000 Gas. In terms of Ether cost, as a successful DETER-Z attack needs to make its first transaction’s Gas price higher than the prices of all existing transactions in the victim txpool, the DETER-Z’s Ether cost depends on the txpool content. In our experiment, the highest Gas price of existing transactions is 1000 Gwei per Gas, under which DETER-Z attack’s Ether cost is reported as in Table 5.2.

### 5.5 Attack Strategies at Miners

This section presents the attack strategies (§ 5.5.1) and evaluation (§ 5.5) on disabling an Ethereum node’s mining.

#### 5.5.1 Proposed Strategies

Given a successfully disabled txpool, the attacker’s goal is to disable the victim node’s miner, so that the block it finds will be empty.

*Preliminary on miner- txpool interaction:* An Ethereum node follows the workflow below to mine the transactions in the local txpool. Upon receiving a block, say \( b_0 \), the node evicts from txpool the transactions in \( b_0 \) and their dependent ones (e.g., the invalid pending transactions as in DETER-Z). After the miner appends the block \( b_0 \) to the tail of its blockchain, it reads the current
content of txpool to select the batch of transactions, such as based on Gas prices, to mine for the next block. For instance, a miner reading the txpool at time $t_1$ may need period $dt_4$ to find a block, thus the block found at time $t_1 + dt_4$ only contains the transactions in txpool before $t_1$.

**DETER-Z's strategy:** To attack a co-residing miner, one can apply DETER-Z in a straightforward way. That is, consider a victim block found at time $t_v$ and its predecessor block propagated at time $t_p$. A DETER-Z attack can succeed as long as the attacker finishes the attack (Steps $z_1$ and $z_2$) after $t_p$ and before $t_v$.

**DETER-X's strategy:** When attacking a miner, a DETER-X attacker needs to ensure the right timing, that is, the submission time of the attacker's request (in sending $x_1$) should be right before when the miner reads txpool, so that the miner reads an empty txpool that is just purged. Denote the time when the miner reads a txpool by $t_p$. If the attack request occurs after $t_p$ (or long before $t_p$), chances are the miner will read a non-empty txpool that is not yet purged (or that is refilled by subsequent transactions).

The attack strategy is this: The attacker node watches a local clock. If time $T$ passes on the clock, it sends the attack request and resets the clock. It also resets the clock upon the arrival of a block.

**Analysis of attack strategy:** It is known that the sequence of block arrivals in a proof-of-work blockchain can be modeled as a Poisson process, under the assumption of constant difficulty and hash rate [114,115]. Based on the classic probability theory [121], the wait time for the next block is a random variable following an exponential distribution. That is, given the average block time $T_0$, the block arrival rate is $\lambda = \frac{1}{T_0}$. Thus, the wait time for the next block, denoted by $x$, follows the exponential distribution with density function: $f(x) = \lambda e^{-\lambda x}$. The probability that the next block arrives after time $T$ is the cumulative distribution function of $x$:

$$Pr[x > t] = e^{-\lambda t} = e^{-\frac{T}{T_0}}$$
Now consider the initial case that a DETER-X attacker resets his clock. The success rate of the attack is the number of transactions discarded in the miner’s next read of txpool divided by all valid transactions received until the next txpool read.

Fig. 5.5: A process with two block arrivals between which the attacker sends two requests. Assuming independent block arrivals, this can be modeled as a Poisson process.

Generally, suppose there are $n$ periods, each lasting $T$, before the arrival of the next block, as in Figure 5.5. In that txpool read, $\frac{n \cdot T}{x}$ percentage of the transactions are purged by the $n$ attack requests and $1 - \frac{n \cdot T}{x}$ percentage of transactions are read by the miner. This leads to the success rate blow:

$$S(n) = \frac{n \cdot T}{x}$$ (5.1)

The probability this event occurs (i.e., the next block arrives after the end of $n$-th period and before the end of $(n + 1)$-th period) is:

$$Pr(n, T) = Pr[x > n \cdot T] - Pr[x > (n + 1) \cdot T]$$
$$= e^{-\frac{nT}{T_0}} - e^{-\frac{(n+1)T}{T_0}}$$ (5.2)

Hence the expected success rate under attack frequency $\frac{1}{T}$ is:

---

5 There are actually two possible initial cases, the attacker node receiving a newly arrived block or the attacker node having just sent the attack message. As the block arrival events are independent and the exponential distribution is memoryless, we can consolidate the two causes in one initial case.
\[ E[S] = \sum_{n=0}^{\infty} Pr(n, T) \cdot \frac{\int_{x=nT}^{(n+1)T} S(n)dx}{T} \]
\[ = \sum_{n=0}^{\infty} n \cdot \ln (1 + \frac{1}{n}) Pr(n, T) \] (5.3)

\[ X \sim .25 \text{tx/sec} \]
\[ X \sim 2 \text{msg/sec} \]
\[ Z \sim .25 \text{tx/sec} \]
\[ \text{Baseline} \]

Table 5.3: DETER & baseline attacks

5.5.2 Evaluation on a Local Miner Node

The evaluation answers the following question:

**Q2.** How effective and costly is sending DETER attacks continuously using the strategies presented in §5.5.1 in disabling a miner with arriving transactions?

We set up a private network of three nodes: A normal node, an attacker node and a victim miner. The victim is connected to both the normal and attacker nodes. There is no connection between
the normal and attacker nodes. In each experiment to be described below, we configure the normal node to generate normal transactions and send them to the victim, at a rate of 2 transactions per second. The process lasts for 500 seconds, and after that, we turn off the mining on the victim node. The metric we use for attack effectiveness is the number of normal transactions included by the victim miner in each block.

In our evaluation of DETER-X, the attacker sends each message with \( n' = 5120 \) crafted future transactions to the victim. First, we fix the rate at which regular transactions are generated at 2 transactions per second and vary the attack rate between 0.25 and 2 messages \((x1)\) per second. For comparison, we also run the experiment without attacks. We run each experiment with mining 100 blocks and report the cumulative number of transactions included in the blockchain. For instance, at (relative) block height 45, we report the total number of transactions included from the first block to the 45-th block. As mining is a non-deterministic process, we repeat each experiment by three times and report the average cumulative number with its standard deviation in Figure 5.6a. The result shows that as block height grows, the cumulative number of included transactions increases linearly (note the log scale of \( Y \) axis). At the block height of 100, there are 764 transactions included when there are no attacks. By comparison, at the same block height 100, the number of included transactions under DETER-X of 0.25 messages per second (2 messages per second) is 266 transactions (115 transactions), leading to the attack success rate (defined as the percentage of regular transactions discarded in a block) of \( 1 - \frac{266}{764} = 65.2\% \) \((1 - \frac{115}{764} = 84.9\%)\). Statistically speaking, it can be seen over time (at different block heights), there is a consistent disparity between the result with DETER-X attacks and that without. The disparity clearly shows the effectiveness of our DETER-X attack with varying attack rates – The larger the attack rate is, the larger the disparity is.

In our evaluation of DETER-Z, the attacker sends a message of \( n' = 5120 \) pending transactions described in §5.5, as well as another message of \( n' = 5120 \) future transactions, to the victim. The attacker sends these two messages every time it receives a new block from the victim. Figure 5.6b reports the same cumulative metric for DETER-Z, from which one can see DETER-Z is much
more effective than DETER-X. Sending approximately 0.25 messages per second, the DETER-Z attack achieves a success rate of 99.3%.

Table 5.3 summarizes the measured success rates for the two DETER variants from our previous experiment results. We apply Equation 5.3 to DETER-X under rates 0.25 and 2 messages per second. The theoretical results roughly fall within the error rates of the measured results. DETER-Z is expected to achieve 100% success rate. In reality, the measured rate is slightly lower than the perfect 100% rate, which we suspect is due to spurious block production (e.g., two blocks are mined at very close time). We also carry out the baseline attack described in the previous subsection (§5.3). The measured success rate is also lower than the expected 100% success rate due to the same suspected reason. At last, in this table, we show the Ether cost of the attacks. DETER-X relies only on future transactions and does not cost the attacker any Ether. In DETER-Z, the attacker needs to pay the fee of one transaction (the first pending transaction) per block, minimally 21000 Gas at 1000 Gwei (i.e., 0.021 Ether). The cost of the baseline attack is 12.5 Ether as previously described. In summary, DETER-X incurs zero monetary cost and achieves reasonable success rates at a low attack rate. DETER-Z is very effective with an almost 100% success rate and its Ether cost is 1000× cheaper than the baseline attack.

5.6 Measuring Deployed Networks’ Exploitability

This section presents the measurement studies on the exploitability of deployed Ethereum networks, including testnets and the mainnet. We first describe the design rationale in §5.6.1 and then the two studies on testnets in §5.6.2 and on the mainnet in §5.6.3.

5.6.1 Design Rationale

In the previous sections, our evaluation focuses on the attack effectiveness on Ethereum clients with the default configurations. And the evaluation is set up on a single Ethereum node under our control. This section’s goal is to understand a deployed Ethereum network’s exploitability
under DETER attacks. This is necessarily a different goal (from previous sections), because an attacker who can successfully DETER a controlled node may be hindered when attacking a deployed network: First, in a deployed network, the critical nodes to a network’s operation can be hidden from the attacker, leaving her unable to discover the attack targets in the first place. This motivates our first measurement goal, node discoverability of an operational network. Second, even if the attacker can discover and connect to a critical node, the node may be configured to weaken or mitigate the DETER attacks, for instance, txpool can be configured to decline future transactions altogether to be resilient to DETER-X. This motivates our second measurement goal, node exploitability of an operational network.

We measure two subjects, testnets and the mainnet. These two types of Ethereum networks serve different purposes in operation and have different levels of impacts on the real world. Concretely, the purpose of testnets is for testing DApps pre production, and they are mainly used by DApp/blockchain developers. The purpose of mainnet is to serve actual DApps in their business, and it is accessed by hundreds of millions of Ethereum DApp users.

Our measurement studies on testnets and the mainnet differ in their measurement methods. For the testnets, we aim to mount DETER attacks directly (with parameters tuned down) and expect to observe some temporary service degradation. By this means, we can produce definitive results regarding exploitability. For the mainnet, our measurement method is designed with ethical concerns as the first-class citizen and aims at lightweight test probes on some necessary (but insufficient) conditions of DETER vulnerabilities.

Particularly, testing the mainnet is necessary, as its exploitability is specific to the client configurations, running discovery protocols, and other deployment-specific settings on the mainnet. Another network’s measurement results cannot be generalized to infer the mainnet’s exploitability.

### 5.6.2 Measuring Testnets’ Exploitability

This subsection presents the measurement study to answer the following question:
Q3. Can a DETER attacker discover the critical service nodes in a testnet (node discoverability)? Can these discovered nodes be effectively attacked (node exploitability)?

**Measurement Methods**

(a) Top miners in Ropsten  
(b) Screenshot from etherscan.io: red cross indicates the miners under DETER-X attack.

**Fig. 5.7: Evaluation of DETER-X attack on the top-4 miners in the Ropsten testnet.**

**Attack Strategy:** With the goal to disable an Ethereum network, the attacker takes the strategy to first discover critical nodes in the network and then direct her DETER payloads at the nodes. Recall that operational blockchains’ functionalities are centralized in the hands of “top” services, such as transaction relay services and mining pools. A critical node is defined as the Ethereum node serving the backend of the top services.

**Methods to measure node exploitability:** In testnets, node exploitability is tested by directly sending DETER payloads (in a short period) and observing the service interruption, such as the block size when the attacked service is a mining pool or the slowdown of transaction propagation when the target is an RPC service. More challenging is measuring node discoverability, as described next.

**Methods to measure node discoverability:** To discover the critical nodes, we propose to leverage the client version “codename” disclosed through the service’s frontend RPC interfaces. Specifically, given a known “top” service that exposes the RPC interface, we send `web3_clientVersion` RPC queries and obtain the results, from which we select the unique ones or the ones bound to the specific service (e.g., SrvR1’s client codename `omnibus` is unique).
We then launch a sufficient number of “supernodes” to join the mainnet. We configure these supernodes to stripe away their default limit of neighbor numbers. We run these nodes long enough until they are connected to the maximal number of nodes in the network (i.e., their neighbor count becomes stable). After that, the supernodes propagate transactions and blocks with their neighbors. During this entire process, the supernodes, as they are statically instrumented, log all the messages that they receive and send, which include the peer-discovery messages, propagated blocks, and propagated transactions, among others.

On the collected messages, we find the peer-discovery messages that match the known services’ “codenames” (i.e., omnibus). A node $N_x$ that sends a peer-discovery message with a matching codename of service $S$ is on the backend of service $S$.

In addition, when discovering critical nodes behind a mining pool, we analyze the block-arrival messages. Each block-arrival message is a triplet: the block hash, the sender (i.e., the supernode’s neighbor that propagates the block), and the timestamp recording when the supernode receives the block-arrival message. Here, we consider both finalized blocks in the blockchain and uncle blocks that are “reorganized” out of the permanent blockchain. For each block, we find its “home” node by choosing the neighbor who sends the block-arrival message at the earliest time. Then, for each given neighbor, we count the number of blocks whose home nodes match this neighbor’s nodeID. By this means, we can find top miners in the network as the supernode’s neighbors that have produced most blocks in an extended period.

**Measurement Results**

**Discovering top mining pools:** We apply the above methods to first discover top miners in Ropsten. Specifically, we launch two supernodes joining the testnet and run them for 24 hours until their neighbor sets become stable (i.e., stop growing). In this period, the two supernodes are connected to 840 nodes in Ropsten and receive a total of 6200 distinct blocks. We use the data aggregation described above to plot the distribution of blocks over their home miners as in Figure 5.7a.
From the result, we choose the target of DETER attacks (or exploitability tests) the top four miner nodes who jointly produce 88.38% of all blocks. We then respectively mount DETER-X and DETER-Z to attack these selected top miners. Our goal is to understand, by DETER-ing these top miners, how much interrupted the global mining activity of the testnet is.

We similarly set up two other supernodes in connection with 490 nodes in Rinkeby, discover the top miners there and conduct node-exploitability tests.

In the above setting, we configure DETER-X by sending $n' = 5120$ future transactions with a Gas price of 1000 Gwei at the rate of 1 message per second. The DETER-X attacks on the selected top miners are mounted in parallel. To control the service interruption, we restrict the duration of the attack in Ropsten to under 60 seconds. After the attack, we take a screenshot of the monitored block history from etherscan.io [78].

In the screenshot in Figure 5.7b our attack starts at block #9450103 and stops at block #9450106. Before the attack starts (blocks #9450100-9450102), each block includes at least 46 transactions and the used Gas in each block varies between 32.3% and 98.9% (normalized over the Ethereum block Gas limit). During the attack (blocks #9450103-#9450106), each block includes at most 1 transaction and the used Gas per block is below 1.8%. Note that these four blocks are mined by miners under DETER-X attacks. After the attack stops (blocks #9450107-#9450109), for blocks #9450107 and #9450109, each block includes at least 31 transactions and has a used Gas per block above 99%. These two blocks are found by the miners not under our DETER-X attacks. Block #9450108 includes only one transaction and is mined by the miner under attack. Note that even though the attack stops when Block #9450108 is produced, it is likely that its home miner 0x4b0c... reads txpool before the attack stops, and it reads a purged txpool.

Overall, by comparing the numbers before/during/after the attack, we show the DETER-X attack of 0 Ether cost can reduce the block size in Ropsten by $31 - 77 \times$ in terms of the number of included normal transactions, and by $18 - 55 \times$ in used Gas per block.
5.6.3 Measuring Mainnet’s Exploitability

This subsection presents the measurement study to answer the following question:

Q4. Can a DETER attacker discover the critical service nodes in the mainnet (node discoverability)? Can these discovered nodes be effectively attacked (node exploitability)?

**Measurement Method**

**Design rational:** Our goal is to measure the DETER exploitability of an identified mainnet node. A naive approach is to directly run the original test $t_1/t_2$ (recall §5.4) against the mainnet node. Unfortunately, this does not work for a mainnet node of which we don’t have control. Specifically, measuring DETER vulnerability entails setting up the txpool with certain initial transactions and observing an eviction by incoming future/invalid transactions on the target node. Directly carrying out these actions requires privilege (e.g., turning on RPC interface as in $t_1/t_2$), which we don’t have if the target is a mainnet node operated by others. Moreover, the admission of a future transaction (as in $t_1$) and the eviction by a latent invalid transaction may only change the internal state and is not externally observable. Besides, even if an eviction can be detected on a mainnet node, attribution to the right cause can be challenging, as a mainnet node, unlike a local node in a controlled environment (in §5.4.4), needs to also process normal transactions propagated in the “background.” An eviction can be attributed to a background transaction or a test transaction.

To address the measurement challenges, our idea is two-fold: 1) Instead of observing the opaque future transaction directly, we send a pending transaction to be evicted by the future transaction and observe its behavior instead. As the pending transaction is propagated, it is observable across nodes. 2) Instead of relying on RPC service that we cannot configure on the mainnet, we exploit the transaction replacement capability in the standard Ethereum protocol: An incoming transaction $tx_2$ may replace an existing transaction $tx_1$ of the same sender and nonce, if $tx_2$’s Gas price is sufficiently higher than $tx_1$’s (e.g., higher than 110% of $tx$’s Gas price in Geth). The insight

---

[By contrast, admission of a pending transaction to the txpool is externally observable, as that results in the admitted pending transaction propagated to other nodes.]
here is that when sending a tx$_2$ at price lower than 110% of tx$_1$’s Gas price, say 105%, observing the propagation of tx$_2$ on other nodes implies that txpool does not initially contain tx$_1$.

To ensure that 105% is low enough, we profile all existing Ethereum clients. Denote by $R$ the minimal Gas price bump necessary to replace an existing transaction. Table 5.4 shows the profiling results, in which all clients’ $R$ are above 105%, which is also consistent with existing works [122].

<table>
<thead>
<tr>
<th></th>
<th>Geth</th>
<th>Parity</th>
<th>Nethermind</th>
<th>Besu</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>110%</td>
<td>112.5%</td>
<td>110%</td>
<td>110%</td>
</tr>
</tbody>
</table>

Table 5.4: Gas price bumps in different Ethereum clients.

Our design also addresses the new challenges presented by ethical measurement on the mainnet. Instead of purging the entire txpool and victimizing all residing transactions there, our mainnet test is designed to affect only the transactions with a too low Gas price to be included in the blockchain (within the default three-hour drop deadline).

**Mainnet test $t_{1m}$:** Suppose the target is to measure the DETER-X exploitability of a mainnet node $T$. Initially, we set up a measurement node $M$ and an observer node $O$, separately connecting to $T$. We double-check if node $M$’s txpool is full, and only proceed upon a fulltxpool. In ①, Node $M$ sends a pending transaction tx$_1$ of medium Gas price $p$ such that $p$ is higher than at least $s_0$ (e.g., $s_0 = 520$) existing transactions in Node $M$’s txpool. Note the txpool’s capability is $n = 5120$ transactions. Node $O$ checks if tx$_1$ is propagated by node $T$. It proceeds only if this is true. In ②, Node $M$ sends to Node $T$ s future transactions, each at price $p + 1$. The number
of future transactions $s > s_0 = 520$. In our measurement, Node $M$ runs a statically instrumented Geth client so that it can propagate future transactions to $T$. In 3, immediately after the previous step, Node $M$ sends Node $T$ one replacing transaction $tx_2$, whose price is $1.05 \cdot p + 1$. At last, Node $O$ observes if it receives $tx_2$ from node $T$. If so, the test is a success, denoted by $✓$, which indicates that Node $T$ does evict $tx_1$ by one of those future transactions. We denote this test by $t_{1m}(s, p) = ✓/✗$. An illustration of the measurement process is in Figure 5.8.

Measurement effectiveness: Consider the situation right after Step 3. If $tx_1$ is evicted by one of the future transactions, $tx_2$ will arrive as a new transaction which will be admitted to the txpool and propagated to $T$’s neighbors, including Node $O$. If $tx_1$ is not evicted, $tx_2$ will arrive as a transaction attempting to replace $tx_1$ which will fail because of its Gas price $1.05 \cdot p + 1 < 1.1 \cdot p$. Thus, in this case, Node $O$ cannot observe the propagation of $tx_2$.

In practice, when there are normal transactions propagated to the target mainnet node, the normal transactions are assumed to arrive at a volume much lower than the burst of $s$ future transactions in the test. Thus, a normal transaction may evict one of the $s_0$ existing transactions at lower Gas prices but not $tx_1$.

Mainnet test $t_{2m}$: We propose a mainnet version of test $t_2$, named $t_{2m}$. Similar to $t_{1m}$, the measurement node $M$ first propagates to the target mainnet node $x_2$ transactions $tx_{1,0}, tx_{1,1}, \ldots, tx_{1,x_2}$ at price $p$ higher than $s$ other transactions in the txpool, then sends a message of $s$ latent invalid transactions at price $p + 1$, and at last sends $x_2$ replacing transactions $tx_{2,0}, tx_{2,1}, \ldots, tx_{2,x_2}$ at price $1.05 \cdot p + 1$. The observer node $O$ observes the replacing transactions that are propagated to it. If all (or less than) $x_2$ transactions are propagated, the test is a full (or partial) success, denoted by $✓$ (or $✓$). Otherwise, if none of the $x_2$ transactions are propagated, the test fails, denoted by $✗$.

Ethical designs: We take extensive measures to address the ethical concerns. First, the measurement methods presented above limit the prices of crafted transactions to be lower than $p$, thus leaving the existing pending transactions on the tested node (i.e., node $T$) unaffected during and after the test. Recall price $p$ is set s.t. 10% of existing transactions on Node $T$ have lower prices than $p$. 
Second, to minimize the impacts on the 10% pending transactions on the tested node $T$, we send these transactions after each test to “refill” the txpool. To do so, in our studies, we increase the length of the measurement node’s txpool to ensure enough transactions are buffered there.

Third, we check 1) the blocks generated during and right after each test are full and reach the block Gas limit, and 2) the lowest Gas price of the transactions in those blocks is higher than $p$. These two conditions jointly ensure that the presence of the test, which affects at most 10% of the transactions in the txpool, does not affect the transactions included in the blocks, thus leaving no long-term effect on the blockchain.

**Measurement Results**

**Node discoverability in RPC services:** To discover the critical nodes in the mainnet, we use the same method described in §5.6.2 only with different configurations as described next. We send `web3_clientVersion` RPC queries to eight well-known RPC services and find that SrvR1 and SrvR2 nodes bear unique codenames.

We then launch eight “supernodes” to join the mainnet and run them in a 7-day period until the neighbors of each supernode stop growing and are stable. Here, launching eight supernodes increases the node coverage in the mainnet. Using the measurement methods described in §5.6.2, we discover 48 nodes serving the backend of SrvR1 and 1 node of SrvR2, as presented in Table 5.5.

**Node discoverability in mining pools:** To discover the top miners on the mainnet, we reuse the same methods with that of the testnets as described in §5.6.2. We reuse the eight mainnet supernodes in the previous experiment. Specifically, we first check if a mining pool provides an RPC service by visiting its website. We did so for all top mining pools listed on the ranking website [123]. If an RPC service is provided, we use the method described previously to discover the mining pool’s codename. We found mining pools SrvM1’s nodes run Geth clients with codename `turbo`, and SrvM2 runs clients of two codenames, `ethereumsolo` and `ethereumpplns`. Other mining pools’ codenames are strongly suggestive w.r.t. their names. Then, we use the found codename to...
<table>
<thead>
<tr>
<th>Service name</th>
<th># of nodes</th>
<th>$t_{1m}/X$</th>
<th>$t_{2m}/Z$</th>
<th>Client-codename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mining pools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SrvM1</td>
<td>59</td>
<td>✓</td>
<td>✓</td>
<td>Geth-turbo</td>
</tr>
<tr>
<td>SrvM2</td>
<td>8</td>
<td>✓</td>
<td>✓</td>
<td>Geth-ethereum, Geth-ethereumpplns</td>
</tr>
<tr>
<td>SrvM3</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
<td>Geth-XX</td>
</tr>
<tr>
<td>SrvM4</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>Geth-XX</td>
</tr>
<tr>
<td>RPC services</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SrvR1</td>
<td>48</td>
<td>✓</td>
<td>✓</td>
<td>Geth-omnibus</td>
</tr>
<tr>
<td>SrvR2</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>Geth-ethshared</td>
</tr>
</tbody>
</table>

Table 5.5: Critical mainnet nodes: Discoverability and exploitability (We anonymize the two services’ labels by XX).

match the peer-discovery messages collected through the eight supernodes and discover the neighbor nodes that serve the backend of known mining pools. Additionally, the supernodes monitor the block-propagation messages and their timings to verify the discovered top mining-pool nodes.

After searching the codenames in the peer-discovery messages collected, we found 59 nodes in SrvM1, 8 nodes in SrvM2 and 6 nodes of SrvM3, as listed in Table 5.5. Note that in our experiments, if waiting for long enough, the measurement node can always be connected to the nodes of the target mining pool. Particularly, we have not found mining pools that only connect to a prefixed set of nodes.

**Node exploitability:** For each identified critical node (from § 19), we run the above test $t_{1m}$ and $t_{2m}$, respectively for measuring DETER-X and DETER-Z exploitability. As all identified nodes run Geth clients, we use the Geth-default setting in our tests, such as $n = 5120$. For each service, we pick two random nodes to the two tests, and each test is run for three times to ensure the same result is produced. We report the result in the two columns named “$t_{1m}/X$” and “$t_{2m}/Z$” in Table 5.5. All tests are successful, and all tested nodes are vulnerable to both DETER-X and DETER-Z attacks.

5.7 Attack Mitigation

Due to the impossibility result discussed in § 5.1, we propose heuristics for DETER mitigation and miners’ profitability preservation. We propose two mitigation schemes $M_0$ and $M_1$ that respec-
tively add restrictions to transaction admission and eviction policies.

**Mitigation scheme** $M_0$: We add three transaction-admission rules to an underlying txpool: $M_{0a}$) It does not admit any future transitions. $M_{0b}$) It does not admit any invalid transaction. $M_{0c}$) It does not admit a transaction that shares the same sender with another transaction currently residing in the txpool.

**Mitigation scheme** $M_1$: We add three transaction-eviction rules over the underlying txpool: $M_{1a}$) It declines an incoming future transaction if it evicts a valid pending transaction in the txpool. $M_{1b}$) It declines an incoming invalid transaction if it evicts a valid pending transaction in the txpool. $M_{1c}$) It declines an incoming transaction if it evicts another pending transaction and leaves a future transaction in the txpool. For instance, consider two pending transaction $tx_1$ of nonce $n + 1$ and $tx_2$ of nonce $n + 2$. If $tx_1$ is evicted, it makes $tx_2$ a future transaction. Such an eviction is prohibited by $M_{1c}$. $M_{1d}$) Other than the above, it optionally enforces the following eviction priority, that is, $VH > VL > [FI]H > [FI]L$ (valid transaction preceding future or invalid transactions, even their Gas prices disagree). Note that $M_1$ does not restrict transaction admission on a txpool if it does not trigger eviction.

**Security analysis:** With $M_{1a}$, a DETER-X payload will be declined. With $M_{1b}$, a DETER-Z payload will be declined. $M_{1a}$) and $M_{1b}$) work for generic initial state of a txpool. $M_{1c}$) defends against DETER attacks in the special case where incoming transactions trigger transforming existing transactions into future transactions. In addition to security analysis, we evaluate $M_1$’s security by implementing it in our txpool simulator (as will be described) and measure the success rate of DETER attacks on it.

<table>
<thead>
<tr>
<th>Schemes</th>
<th>Miners’ revenue (Ether)</th>
<th>DETER security</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$t_1/X$</td>
</tr>
<tr>
<td>Geth (default)</td>
<td>16.5388 (✓/✓) (Table 5.2)</td>
<td></td>
</tr>
<tr>
<td>$M_0$</td>
<td>15.9506 (−3.56%)</td>
<td>✓</td>
</tr>
<tr>
<td>$M_1$</td>
<td>16.5423 (+0.002%)</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.6: Evaluation of miners’ revenue and security across mitigation schemes and Ethereum clients: All clients are configured with the same txpool length of 5120 transactions.

**Evaluation of mitigation schemes:** We evaluate mitigation schemes in terms of miners’ revenue
and the success rates of DETER attacks. The two mitigation schemes are implemented on Geth and are evaluated under real transaction traces. We use Geth’s default transaction-admission policy as the baseline for comparison.

*For security evaluation,* we run tests $t_1$ and $t_2$ with varying parameters ($x_1$ and $x_2$) and observing the test results on the two mitigation schemes. As shown in Table 5.6, all tests under all parameters fail with $\times$ (i.e., zero success rates), suggesting the effectiveness of mitigating DETER attacks.

*For miners’ revenue,* we launch a mainnet node and collect the Ethereum transactions received there. We then replay the trace against a Geth node on which we build a middleware to simulate an additional txpool and implement the proposed mitigation schemes in the middleware. The txpool simulator’s length is set at 5120, which is the same with Geth’s default txpool length. On different clients/experiments, we replay the same sequence of interleaved transactions and mining actions. Each experiment of the same setting is run three times and we verify that the replayed runs produce the same results deterministically. All produced blocks in the experiments are full and with the same block Gas limits. We report the total Ether of all transactions included in these blocks.

<table>
<thead>
<tr>
<th>Schemes</th>
<th># of declined txs ($F$)</th>
<th># of declined txs ($I$)</th>
<th># of declined txs ($FH \rightarrow VL$)</th>
<th># of declined txs ($IH \rightarrow VL$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geth</td>
<td>1395</td>
<td>1589</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$M_0$</td>
<td>9588</td>
<td>1899</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$M_1$</td>
<td>288</td>
<td>353</td>
<td>287</td>
<td>322</td>
</tr>
</tbody>
</table>

Table 5.7: Transaction statistics during the evaluations on mitigation schemes. “#” is number, “txs” refer to transactions, and conditions $F/I$ mean future/latent invalid transactions being declined. Conditions $FH \rightarrow VL/IH \rightarrow VL$ mean the declined transactions are future/latent invalid transactions with high Gas prices that, if not declined, would have evicted pending transactions. Statistics labeled by $-$ are not collected in our experiments due to the lack of usefulness.

During the experiments, we also collect additional workload statistics to help explain the results. We collect the number of declined future/latent invalid transactions (i.e., $F/I$) and the number of declined future/latent invalid transactions that, had not declined, would have evicted a pending
transaction at a lower Gas price (i.e., \(FH \rightarrow VL\) or \(IH \rightarrow VL\)). These statistics are obtained on the Geth node under our control.

The results of miners’ revenue are presented in Table 5.6. \(M_0\)’s revenue is 3.56\% lower than the baseline of Geth’s default policy, and \(M_1\)’s revenue is almost the same as Geth’s (0.0002\% higher). The statistics in Table 5.7 help understand the revenue results. From the table, \(M_0\) declines much more transactions (both \(F\) and \(I\)) than the vanilla Geth, thus failing to collect the potential revenues from those transactions and resulting in lower revenue than Geth’s. \(M_1\) declines fewer transactions than Geth and may benefit its revenues from those transactions’ fees. Note that among those declined transactions in \(M_1\), the majority are the ones exploitable by DETER attacks, that is, \([FI]H \rightarrow VL\), and are necessary to be declined.

5.8 Related Work

Blockchain DoS security: In the existing literature, there is a body of research work on examining the blockchains’ security under DoS attacks. These DoS attacks exploit the vulnerabilities at a blockchain’s different system layers, such as exploiting under-priced smart-contract execution [13, 14], misusing mining incentives [6, 5], abusing transaction processing [6, 5], as well as partitioning the underlying P2P networks [1, 2, 3, 4]. Specifically, 1) On the mining layer, classic mining algorithms are designed under the assumption that the majority of miner nodes are honest, which may not hold in the case of small-scale blockchains. In practice, 51\% attacks have been successful on smaller blockchains [124, 125], resulting in damages ranging from hard forks to the demise of the entire blockchain. A block-withholding attack works by a miner withholding the block she finds and revealing it later, in a strategic way that wastes the efforts of other miners or mining pools [6]. BDoS [5] exploits selfish mining [7] to incentivize miners to stop mining altogether. In general, the existing mining-based DoS attacks assume institutional attackers who control a significant portion of computing power in a blockchain (e.g., the early-day 51\% and the 21\% in the very recent BDoS [5]), which may not be practical in targeting a large-scale blockchain network. In
practice, there are no successful instances of mining-based DoSes on large blockchains [5]. 2) On the P2P network layer, an eclipse attack [1, 2] aims to isolate a DoS-victim peer from the network. A routing attack [3, 4] assumes corrupted institutional attackers controlling ISP and employs BGP hijacking to intercept network traffic towards partitioning it. 3) On the smart-contract and DApp layer, one can exploit under-priced EVM instructions (e.g., EXCODESIZE [50], SUICIDE [51] or instruction runtime variation as in BrokenMetre [12]) as well as zero-priced Ethereum operations (e.g., eth_call [45] as in the DoERS attack [73]) to cause a large volume of computations on victim EVM instances at low costs to the attacker. In addition, contract execution can be failed by overflowing the call stack [13] or overflowing the block gas limit [14]. Failing a contract call is of interest to, for instance, a malicious auction leader who wants to fail a refund call to the previous leader.

Of particular relevance is the small body of research on 4) the attacks to deny pre-blockchain transaction handling, by sending Bitcoin spams. Bitcoin “stress testing” [8] is a measurement study on the impacts of the 2015 Bitcoin spam campaign. In this campaign, a flood of “spam” transactions (each at a low cryptocurrency value) is broadcast to the Bitcoin network and saturates the limited blockchain transaction throughput [9]. The impact of the campaign includes the transaction-inclusion backlog (victimizing other concurrent transactions and causing delays), enlarged memory pool (which psychologically causes subsequent transaction senders to pay a higher fee than they should [10, 11]), the increased UTXO set (victimizing the miners who need to maintain the full set of UTXO for transaction validation), As the spam transactions are at a low fee and have a low priority for blockchain inclusion, they incur low monetary costs to the attacker. Unlike the Bitcoin spam attacks, DETER is the first work to deny Ethereum network services by exploiting previously unknown design flaws in Ethereum’s transaction handling. It also presents a measurement study on the DETER’s impacts on real Ethereum clients/networks.

Blockchain RPC attacks: There are research works exploiting or aiming at modern blockchains’ RPC services. The work [15] measures the prevalence of cryptocurrency stealing attacks exploiting Ethereum’s RPC to unlock accounts, via deploying honeypots. DoERS [73] ex-
exploits the free-of-charge RPC interface (eth_call) to cripple the Ethereum nodes run by an RPC service. DETER can break an RPC service but exploit the Geth design flaws, which are different from the vulnerabilities in the existing RPC attacks.

**In-band bribery attacks**\(^{[107, 108, 109, 126, 127]}\) refers to the attack in which an average attacker account sets up a smart contract with deposit to reward any miner who delays the inclusion of the transactions that modify a victim account’s state after a prefixed timeout. In this attack, the miner is assumed to be rational and can choose which arriving transactions to include in order to maximize her revenue, eventually.

In-band bribery attack can be low-cost and be mounted by the average user, instead of an institutional user \(^{[107, 108, 109]}\). This is similar to the DETER attacks. Other than that, the two attacks are different: First, their attack methods are different. While in-band bribery attack allows miners to see the victim transactions (but misuse rational miners’ incentive to exclude them temporarily), DETER prevents the miners from seeing victim transactions in the first place. Thus, DETER does not require miners to be rational. Second, the consequences of these two attacks are different. While in-band bribery temporally censors the inclusion of selected transactions sent from a few victim accounts (the number of victim accounts can not be big to keep the bribe or attacker’s cost low), DETER attacks are to evict all transactions submitted during the attack period and cause them to be permanently excluded from the blockchain.

### 5.9 Responsible Disclosure

We have disclosed the DETER vulnerabilities to the Ethereum developer community of Geth/Parity/Besu/Nethermind\(^8\)(through their bug bounty programs), as well as tested service providers (including the RPC services and mining pools). The bugs have been confirmed by all clients’ bug-bounty programs with attacks reproduced. Particularly, the DETER bugs are assessed to be of “high impact” by Ethereum Foundation (Geth) and “median impact” by OpenEthereum (Parity).

\(^8\)We also send bug reports to Aleth, but with no response.
5.10 Summary

This chapter presents the DETER attacks that deny a remote Ethereum node’s service by exploiting flawed transaction handling in txpool. DETER attacks are of low Ether cost. DETER attacks can be extended by discovering nodes in critical services and result in a global impact on an Ethereum network. We evaluate and verify the effectiveness and low cost of DETER attacks on local nodes running different Ethereum clients and testnets. We also propose non-trivial methods to detect DETER vulnerability on blackbox mainnet nodes and confirm the mainnet nodes’ discoverability and exploitability.
CHAPTER 6

CONCLUSION AND FUTURE RESEARCH

6.1 Summary

This thesis focuses on examining the DoS security of the Ethereum blockchain’s communication network. For each network service inside the communication network, the thesis examines their designs and implementations and discovers various vulnerabilities that lead to DoS attacks, including the DoERs attacks on the relay service and DETER attacks on the mempool. Besides, to obtain the P2P network topology in Ethereum, this thesis proposes a novel technique TopoShot that leverages the handling of replacement transactions to detect the connections between different Ethereum nodes. Through systematic evaluations and measurements, this thesis confirms that real-world network services in the Ethereum blockchain are vulnerable to both DoERs and DETER attacks, leading to a potential collapse of the Ethereum ecosystem. Besides, the results of uncovered Ethereum P2P network topologies reveal interesting network properties, showing that testnet networks are resilient to network partition attacks and critical services (e.g., relay services, mining pools) in the mainnet adopt a biased neighbor selection strategy. Finally, to mitigate the discovered vulnerabilities resulting in DoERs and DETER attacks, this thesis proposes practical mitigation solutions to harden the security of affected network services in Ethereum’s communication network.
6.2 Other Research

Apart from blockchain network security, I have worked on other research directions, including cost optimization on the blockchain and enhancing the security of cloud storage by leveraging blockchain as a trusted third party.

6.2.1 Cost Optimization on Smart Contract Data Feeds and Invocations

Today, cost optimization is an essential research topic on the blockchain because users are appealing to reduce the monetary cost when uploading data feeds to the blockchain or sending transactions to invoke smart contracts. My research papers present two novel cost-saving frameworks which optimize the cost at the data-feed layer [128] and transaction layer [129]. At the data-feed layer, my research develops a middleware to dynamically replicate data feeds between the blockchain and off-chain cloud storage, which can be used to support real applications in Decentralized Finance (DeFi). Evaluation results show it can save data-feed costs by 10% - 74%. At the transaction layer, my research develops a middleware to securely batch the smart-contract invocations into one transaction. Evaluation results show that it can save smart contract invocation costs by 14.6% - 59.1%.

6.2.2 Enhanced Cloud Security by Blockchain

Third-party cloud services have been found to deviate from their security promises and violate consistency requirements. To enhance the cloud’s security and prevent forking attacks, my research papers present two techniques to log the cloud storage’s activities and detect the security violations by repurposing public blockchains’ security hardness [130, 131].
6.3 Future Research

For future research, one possible direction is to design a better mitigation solution to defend against the DETER attacks this dissertation discovered in Ethereum's mempool. Although several heuristic strategies have been proposed to mitigate DETER attacks in Chapter 5, these strategies can overkill benign transactions and reduce the miner's revenue. However, preserving the revenue is critical for miners, which can incentivize them to join the blockchain network and behave honestly. Therefore, designing better mitigation that can precisely defend against the DETER attacks while not overkilling benign transactions requires more research effort. In addition, this thesis has not thoroughly examined the attack surfaces in blockchain network services. Another research direction can be systematically analyzing the design and implementations of blockchain network services with advanced techniques, e.g., model checker, program analysis, automatic bug discovery, etc.

From the system’s perspective, this thesis examines the security of the blockchain’s communication network, which positions at the network layer in the complex blockchain system. Other layers in the blockchain system also deserve more research efforts. For instance, at the application layer, detecting vulnerabilities in DeFi smart contracts is an important research task as real-world hacks keep evolving and result in significant financial loss to blockchain users. It is thus imperative to develop advanced tools or techniques that can automatically detect vulnerabilities in complicated smart contracts.
REFERENCES


[41] “Cryptokitties: Collect and breed digital cats!” https://www.cryptokitties.co/


[48] “getfunddetails in the melon protocol,” https://github.com/melonproject/protocol/blob/f5bc07870c0f6a88a1b0ec855752dbd9cc6a23f5/src/contracts/factory/FundRanking.sol#L8


[51] “Geth nodes under attack again (reddit),” https://www.reddit.com/r/ethereum/comments/55s085/-geth_nodes_under_attack_again_we_are_actively/

[53] “Cryptokitties on dapp radar (showing five smart contracts deployed on ethereum),” https://dappradar.com/app/3/cryptokitties


[58] “Faucet in gitcoin,” https://gitcoin.co/faucet


[61] “curl, command line tool and library for transferring data with urls,” https://curl.haxx.se/


[64] “Eth namespace: Geth provides “state override” extensions to the standard “eth_call”,” https://geth.ethereum.org/docs/rpc/ns-eth


[72] K. Li, Y. Wang, and Y. Tang, “Deter: Denial of ethereum txpool services.”


[77] “Ethereum & ipfs apis. develop now on web 3.0;” https://infura.io/


[127] P. McCorry, A. Hicks, and S. Meiklejohn, “Smart contracts for bribing miners,” in Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN,


VITA

Kai Li was born in Jan. 1993 in Anqing, Anhui, China. He received a Bachelor of Engineering degree from Nanjing University of Aeronautics and Astronautics (Nanjing, China) in June of 2015. From 2015 to 2017, he worked as a Software Engineer in ZTE. Since 2018, he has been studying for the Ph.D. in Electrical and Computer Engineering at Syracuse University. This dissertation was defended in April 2022 at Syracuse University.