Imperial College of Science, Technology and Medicine
Department of Computing

A Layered Approach to Improving Blockchain Systems Security

Daniel Perez Hernandez

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, April 2023

Abstract

During the past several years, blockchain systems have gained a lot of traction and adoption, with during peak periods, the total capitalisation of these systems exceeding 2 trillion. Given the permissionless nature of blockchain systems and their large scope in terms of software – e.g. distributed consensus, untrusted program execution – numerous attack vectors need to be studied, understood and protected against for blockchain systems to be able to deliver their promises of a safer financial system.

In this thesis, we study and contribute to improving the security of various parts of the blockchain stack, from the execution to the application layer.

We start with one of the lowest layers of the Ethereum blockchain stack, the EVM, and study the resource metering mechanism that is used to limit the total amount of resources that can be consumed by a smart contract. We discover inconsistencies in the metering mechanism and show and responsibly disclose that it would have been possible to execute transactions that would result in a denial of service attack on the Ethereum blockchain. Our findings were part of the motivation of Ethereum for changing some of its gas metering mechanisms.

We then broaden our analysis to other blockchain systems and study how different fee mechanisms affect the transactional throughput as well as the usage of the blockchain. We discover that low fees, which are in theory attractive to users, can lead to a lot of spam. We find that for two of the blockchain we analyse, EOS and Ripple, this type of spam leads to system outages where the blockchain is unable to process transactions. Finally, we find that a common motivation for spam transactions is to artificially inflate the activity of the application layer, through wash-trading for example.

In the last main chapter of this thesis, we move to the application layer and turn our focus on decentralised finance (DeFi) ecosystem, which is one of the most prevalent types of application implemented on top of blockchain systems. We start by giving formal definitions of the different types of security, namely technical and economic security. With that definition in mind, in the first part of this chapter, we study technical security exploits and develop an automated tool to detect on-chain exploits. We find that the majority of the exploits found through techniques such as program analysis are not exploited in practice, either because of the lack of feasibility of the exploit or because of the lack of economic incentive to do so. In the second part of this

chapter, we focus on economic security and study the liquidation mechanism that is used to protect the users of DeFi lending protocols. We highlight how the efficiency of the liquidations has increased over time, and how depegging events of stablecoin have caused very large amounts of liquidations because of the over-confidence in their stability.

Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-NonCommercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Acknowledgements

I would like to express my sincere gratitude to my PhD supervisor, Benjamin Livshits, for his invaluable guidance and support throughout my research. His insightful feedback, expertise, and encouragement have been instrumental in shaping my ideas and helping me overcome challenges along the way.

I would also like to thank Professor William Knottenbelt for his constructive comments and feedback on my work, which have contributed significantly to the quality of my research.

This thesis would also not have been possible without the support of the Ethereum Foundation, which provided me with financial support, allowing me to work on this exciting topic.

I am deeply grateful to my friends Lewis, Paul, and Sam, for their unwavering support and encouragement through my PhD journey. Their support and kindness have been a source of strength for me.

My parents have been my pillars of strength throughout my life, and I would like to express my heartfelt gratitude to them for their love, encouragement, and support.

Lastly, I would like to thank my wife, Ai Miyuki, for her love, patience, and support during this journey. Her encouragement, support, and positivity have been my driving force, and I am incredibly lucky to have her in my life.

Contents

Abstract **i**

Acknowledgements **v**

1 Introduction **1**

1.1 Motivation and Objectives 2

1.2 Contributions 3

1.2.1 Execution Layer 3

1.2.2 Transactional Layer 4

1.2.3 Application Layer 4

1.3 Statement of Originality 6

1.4 Publications 6

2 Background **10**

2.1 Blockchain fundamentals 10

2.1.1 Transactions and blocks 11

2.1.2 Consensus 12

2.1.3 Incentives and fees 13
2.2 Smart contracts 13
2.3 Applications 14
2.4 Examples of Attacks on Blockchain Systems 15

3 Virtual Machine Security 17

3.1 Introduction 18
3.2 Background 20
3.2.1 Metering in EVM 20
3.2.2 Gas Statistics 22
3.2.3 Previously Known Attacks 23
3.3 Case Studies in Metering 25
3.3.1 Experimental setup 25
3.3.2 Arithmetic Instructions 27
3.3.3 Gas and System Resources Consumption 28
3.3.4 High-Variance Instructions in the EVM 30
3.3.5 Memory Caches and EVM Costs 31
3.3.6 Summary 34
3.4 Attacking the Metering Model of EVM 35
3.4.1 Constructing Resource Exhaustion Attacks 35
3.4.2 Effectiveness of Attacks with Synthetic Contracts 40
3.4.3 Evaluation on Other Ethereum Clients 44

3.4.4 REA as a Form of DoS 46
3.4.5 Responsible Disclosure 48
3.5 Towards a Better Approach 49
3.6 Related Work 51
3.6.1 Gas Usage and Metering 51
3.6.2 Virtual Machines and Metering 53
3.6.3 Follow-up work 53
3.7 Conclusion 54

4 Transactional Level Security 55

4.1 Introduction 56
4.2 Background 58
4.2.1 Consensus Mechanisms 58
4.2.2 Account and Transaction Types 59
4.2.3 Expected Use Cases 61
4.3 Methodology 63
4.3.1 Definitions 63
4.3.2 Measurement Framework 64
4.3.3 Data Collection 67
4.4 Data Analysis 68
4.4.1 Transaction Overview 68
4.4.2 Transaction Patterns 71

4.4.3 Analysis Summary . . . 73
4.5 Case Studies . . . 75
4.5.1 Malicious Transactions on EOSIO . . . 75
4.5.2 Governance Transactions on Tezos . . . 77
4.5.3 Zero-value Transactions on XRPL . . . 79
4.6 Discussion . . . 85
4.6.1 Interpretation of the Throughput Values . . . 85
4.6.2 Revisiting Research Questions . . . 86
4.6.3 Transaction Fee Dilemma . . . 87
4.7 Related Work . . . 88
4.7.1 Previous work . . . 88
4.7.2 Follow-up work . . . 90
4.8 Conclusions . . . 90

5 Application Security 92

5.1 Technical and Economic Security . . . 92
5.1.1 Technical Security . . . 92
5.1.2 Economic security . . . 94
5.2 Technical Security: Smart contracts exploits in practice . . . 95
5.2.1 Introduction . . . 96
5.2.2 Background . . . 97
5.2.3 Dataset . . . 102

5.2.4 Methodology . . . 105
5.2.5 Analysis of Individual Vulnerabilities . . . 114
5.2.6 Limitations . . . 122
5.2.7 Discussion . . . 124
5.2.8 Related Work . . . 130
5.2.9 Conclusion . . . 134
5.3 Economic Security: Liquidations in lending protocols . . . 134
5.3.1 Introduction . . . 135
5.3.2 Background . . . 136
5.3.3 Protocols for Loanable Funds (PLF) . . . 139
5.3.4 Methodology . . . 142
5.3.5 Analysis . . . 148
5.3.6 Discussion . . . 156
5.3.7 Related Work . . . 160
5.3.8 Conclusion . . . 161

6 Conclusion **163**

Bibliography **164**

List of Tables

3.1 Fees for different types of transactions 22

3.2 Median gas price, gas used and transaction fee 22

3.3 Correlation scores between gas and system resources. 28

3.4 Instructions with the highest execution time variance. 30

3.5 Evaluation of different Ethereum clients 44

4.1 Distribution of action types per blockchain. 59

4.2 Datasets for each blockchain 63

4.3 EOSIO top applications 69

4.4 Tezos accounts with the highest number of sent transactions. 72

4.5 XRPL accounts with the highest number of transactions. 74

4.6 Rate (in XRP) of BTC IOUs on XRPL. 80

5.1 A summary of smart contract analysis tools presented in prior work. 100

5.2 Summary of the contracts in our dataset. 103

5.3 Agreement among tools for reentrancy analysis. 104

5.4 Mapping of the different vulnerabilities analyzed. 104

5.5 Datalog setup. 107

5.6 Top contracts victim of reentrancy attack 114

5.7 Top contracts affected by unhandled exceptions 116

5.8 Top contracts potentially victim of transaction ordering dependency attack . . . 118

5.9 Understanding the exploitation of potentially vulnerable contracts. 118

5.10 Top six most valuable contracts flagged as vulnerable by at least one tool. . . . 125

5.11 Events emitted by the Compound protocol 147

5.12 Monitored contracts 148

5.13 Top 10 suppliers and borrowers on Compound 149

5.14 Top 10 miners involved in liquidations 159

List of Figures

3.1 Correlation between gas and clock time when performing a resource exhaustion attack. 25

3.2 Comparing execution time and gas usage of arithmetic instructions. 27

3.3 Comparison of throughput with and without page cache 31

3.4 Measuring block execution speed with and without the effect of cache. 32

3.5 Evolution of the throughput as a function of the number of generations 41

3.6 Execution time as a function of the gas used by contracts within a block 42

4.1 On-chain throughput over time 70

4.2 Tezos Babylon on-chain amendment voting process. 77

4.3 XRPL throughput by transaction type 81

4.4 Value flow on the XRP ledger between October 1, 2019 and April 30, 2020 . . . 83

5.1 Diagram of a technical exploit. 93

5.2 Diagram of an economic exploit. 94

5.3 Overlap in the contracts analysed and flagged by examined tools 104

5.4 Correctly handled failed `send`. 108

5.5 Ether held in contracts: describing the distribution. 123

5.6 Steps of leveraging using a PLF 146

5.7 Number of active accounts and amount of funds on Compound over time. 148

5.8 Cumulative distribution of funds in USD on Compound 150

5.9 Collateral locked on Compound over time 152

5.10 Sensitivity analysis of the liquidatable collateral amount 153

5.11 Amount (in USD) of liquidated collateral from May 2019 to August 2020. . . . 154

5.12 Number of blocks elapsed until liquidation 155

List of Code Listings

3.1 Example gas cost of an EVM program 21

3.2 Bytecode snippet generated by our genetic algorithm. Instructions in bold involve some sort of IO operations. 43

4.1 Configuration file for our measurement framework 65

4.2 Main interfaces of our measurement framework 66

5.1 Sample execution trace information. 105

5.2 Failure handling in Solidity. 108

5.3 EVM instructions for failure handling. 108

Chapter 1

Introduction

Bitcoin launched in 2008 and was the first blockchain system to be deployed. It was the first time that the double-spending problem was solved in a production system. This opened the door to a new type of financial system, where the trust is not in a centralised entity, but in the system itself. Bitcoin did allow for some programmability, but it was limited to a simple scripting language. To allow for more complex programmability, and open the door to more complex financial applications, the Ethereum blockchain was launched, featuring an almost Turing-complete programming language that could be used in such a decentralised and trustless environment.

Since the launch of Bitcoin, Ethereum and other blockchains, the blockchain ecosystem has grown tremendously. The total capitalisation of blockchain systems has reached a peak of 2 trillion dollars in 2021, and the number of users has grown very rapidly. The amount of money that is stored in these systems is also very large, with the total amount of money locked in Decentralised Finance (DeFi) protocols exceeding 100 billion dollars at peak time.

With such strong financial incentives, it is not surprising that the blockchain ecosystem has attracted a lot of attention from attackers.

1.1 Motivation and Objectives

Due to the inherent complexity of distributed systems, the execution of untrusted code, and the game theoretical nature of the blockchain, the attack surface of blockchain systems is extremely large. The incentive of attacking such systems is also very high, as the attacker can potentially steal millions of dollars in a single attack. To make things even worse, the permissionless and pseudonymous nature of blockchains makes it very difficult to identify the attacker and hold them accountable for their actions.

There have been many attacks over the years, that have led to the loss of millions of dollars. Since user funds are usually more exposed at the application layer, this type of attack has received the most attention. However, many attacks have also been reported at the lower layers of the blockchain stack, such as the execution layer, the transactional layer, and the consensus layer. All in all, these attacks had a very negative impact on the adoption and trust in blockchain systems.

For blockchain systems to be able to deliver their promises of a safer financial system, their security needs to improve significantly. In that regard, it is important not only to study and understand the different attack vectors that exist in these systems but also to propose and implement improvements that will help make these systems more robust at every part of the stack.

In this thesis, we aim to contribute to improving the security of blockchain systems. On a high level, the goal is to improve the understanding of the current security landscape of blockchain systems and to propose improvements that can be deployed in practice and make blockchain systems more secure. More particularly, we focus on three main layers of the blockchain stack: the execution layer, the transactional layer, and the application layer. At each layer, we keep the same high-level goals and start by analysing and documenting the security landscape. Then, we propose and implement tools that can be used in practice to make blockchain systems more secure.

1.2 Contributions

In this thesis, we make contributions to improving the security of blockchain systems at the execution layer, the transactional layer, and the application layer. In the following, we provide a summary of the main contributions of this thesis.

1.2.1 Execution Layer

At the execution layer, we contribute to improving the security of the Ethereum Virtual Machine (EVM), and in particular, the gas metering mechanism.

To achieve this, we first create an instrumented version of the EVM that enables us to replay and analyse the execution of smart contracts. Using this tool, we analyse several months of transactions and uncover a number of discrepancies in the metering model, including significant inconsistencies in the pricing of instructions. Additionally, we demonstrate that there is very little correlation between the execution cost and the utilised resources, such as CPU and memory.

Based on these observations, we introduce a new type of DoS attack called the *Resource Exhaustion Attack*, which exploits these imperfections to generate low-throughput contracts. We design a genetic algorithm that generates contracts with a throughput on average 100 times slower than typical contracts, showing that all major Ethereum client implementations are vulnerable. If running on commodity hardware, these clients would be unable to stay in sync with the network when under attack.

Our research indicates that such an attack could be financially attractive not only for Ethereum competitors and speculators but also for Ethereum miners. We responsibly disclose this vulnerability to the Ethereum Foundation and receive a bug bounty reward of 5,000 USD. Finally, we discuss short-term and potential long-term solutions to defend against these attacks.

For this chapter, we have implemented an extension to the aleth Ethereum client[1] that allows

[1] `https://github.com/danhper/aleth`

us to precisely measure gas usage, as well as generate a Resource Exhaustion Attack.

1.2.2 Transactional Layer

In the next chapter, we focus on the transactional layer, and in particular, the transaction throughput of more blockchains with higher scalability. We analyze network traffic data of three high-scalability blockchains - EOSIO, Tezos, and XRP Ledger (XRPL) - over seven months. Our analysis reveals that a small fraction of transactions is used for value transfer purposes. Specifically, 96% of transactions on EOSIO were triggered by airdrops of a currently valueless token, 76% of throughput on Tezos was used for maintaining consensus, and over 94% of transactions on XRPL carried no economic value. This shows that a lot of the throughput of these blockchains is, in one way or another, artificial and does not represent actual adoption. We also identify a persisting airdrop on EOSIO as a DoS attack and detect a two-month-long spam attack on XRPL. We explore the different designs of the three blockchains and how they shape user behaviour. Through this analysis, we shed light on the utilization patterns of transactional throughput and provide insights into how the designs of these blockchains can affect user behaviour. Since this analysis was first concluded, other metrics have come up to help confirm the findings, such as the total value locked (TVL) in protocols running on Tezos or EOSIO being as low as respectively 60 million and 120 million at the time of writing, which is a major contrast to the 30 billion locked on Ethereum.

For this chapter, we have implemented a tool that allows us to fetch and analyse transactions from the three blockchains[2].

1.2.3 Application Layer

At the application layer, we focus on one of the most popular applications on the blockchain: Decentralised Finance (DeFi). We start by providing formal definitions of *technical security* and

[2] `https://github.com/danhper/blockchain-analyzer`

economic security. With that definition in mind, we make contributions to both the technical security and the economic security aspects.

Technical security

Most research focused on technical security has been focused on identifying vulnerable contracts. However, little has been done to understand the practical implications of these vulnerabilities. Here, we take a different approach by examining the extent to which these vulnerabilities are being exploited in practice.

To conduct this study, we surveyed over 20,000 vulnerable contracts that were reported by six recent academic projects. We develop a tool capable of automatically detecting the exploitation of these vulnerabilities in the Ethereum blockchain. The findings indicate that despite the large amounts at stake, only a small percentage of these contracts have been exploited since deployment.

We explain these results by demonstrating that the funds are highly concentrated in a small number of contracts that are not exploitable in practice. This suggests that while identifying and mitigating vulnerabilities in smart contracts is essential, it is equally important to focus on understanding the practical implications of these vulnerabilities.

For this part of the chapter, we have implemented a tool that allows us to detect the exploitation of vulnerabilities in the Ethereum blockchain[3].

Economic security

In the economic security section, we focus on one of the most common economic security mechanisms in DeFi: liquidations.

We present the first in-depth empirical analysis of liquidations on protocols for loanable funds (PLFs), focusing on Compound, one of the most widely used PLFs. Our study demonstrates

[3] `https://github.com/danhper/evm-analyzer`

the very thin margin with which some users interact with PLF, and that even small variations of only 3% in an asset's dollar price can result in over 10 million USD becoming liquidatable.

To further understand the implications of this, we investigate the efficiency of liquidators and find that their efficiency has improved significantly over time, with currently over 70% of liquidatable positions being immediately liquidated. Lastly, we discuss how a misconception of the stability of non-custodial stablecoins can foster a false sense of security, leading to an increased overall liquidation risk faced by PLF participants.

The findings of this study highlight the need for robust liquidation mechanisms to protect against potential losses and provide insights into the behaviour of PLF participants.

For this part of the chapter, we have implemented a tool that allows us to simulate the Compound protocol and to analyse users' behaviour of Compound[4].

1.3 Statement of Originality

I hereby declare that this thesis, entitled "A Layered Approach to Improving Blockchain Systems Security", represents my own original work, as well as joint work with co-authors of the included publications. No work included in this thesis has been submitted for any other degree or qualification, neither by myself nor my co-authors. All sources used in this research have been duly acknowledged and referenced. This thesis is the product of my independent research and represents a significant contribution to the field of blockchain security.

1.4 Publications

The majority of the research presented in this thesis relies on the following peer-reviewed publications.

[4] `https://github.com/merofinance/analyzer`

Chapter 3. Daniel Perez and Benjamin Livshits. "Broken Metre: Attacking Resource Metering in EVM". in: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020.* The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/broken-metre-attacking-resource-metering-in-evm/

Chapter 4. Daniel Perez, Jiahua Xu, and Benjamin Livshits. "Revisiting Transactional Statistics of High-scalability Blockchains". In: *ACM Internet Measurement Conference.* Vol. 16. 20. New York, NY, USA: ACM, Oct. 2020, pp. 535–550. ISBN: 9781450381383. URL: https://dl.acm.org/doi/10.1145/3419394.3423628

Chapter 5.

- Sam M. Werner et al. "SoK: Decentralized Finance (DeFi)". Jan. 2021. URL: http://arxiv.org/abs/2101.08778 to be included In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies.* AFT'22
- Daniel Perez and Benjamin Livshits. "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited". In: *30th USENIX Security Symposium (USENIX Security 21).* USENIX Association, Aug. 2021, pp. 1325–1341. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/perez
- Daniel Perez, Sam M. Werner, Jiahua Xu, and Benjamin Livshits. "Liquidations: DeFi on a Knife-Edge". In: *Financial Cryptography and Data Security.* Ed. by Nikita Borisov and Claudia Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 457–476. ISBN: 978-3-662-64331-0

During the course of this PhD, we also contributed to the following publications.

1. Sam M. Werner and Daniel Perez. "PoolSim: A Discrete-Event Mining Pool Simulation Framework". In: *Mathematical Research for Blockchain Economy.* Ed. by Panos Pardalos, Ilias Kotsireas, Yike Guo, and William Knottenbelt. Cham: Springer International Publishing, 2020, pp. 167–182. ISBN: 978-3-030-37110-4

2. Lewis Gudgeon, Daniel Perez, Dominik Harz, Benjamin Livshits, and Arthur Gervais. “The decentralized financial crisis”. In: *2020 crypto valley conference on blockchain technology (CVCBT)*. IEEE. 2020, pp. 1–15

3. Paul J Pritz, Daniel Perez, and Kin K Leung. “Fast-fourier-forecasting resource utilisation in distributed systems”. In: *2020 29th International conference on computer communications and networks (ICCCN)*. IEEE. 2020, pp. 1–9

4. Sam M. Werner, Paul J. Pritz, and Daniel Perez. “Step on the Gas? A Better Approach for Recommending the Ethereum Gas Price”. In: *Mathematical Research for Blockchain Economy*. Springer, Mar. 2020, pp. 161–177. URL: http://link.springer.com/10.1007/978-3-030-53356-4_10

5. Alexei Zamyatin, Zeta Avarikioti, Daniel Perez, and William J. Knottenbelt. “TxChain: Efficient Cryptocurrency Light Clients via Contingent Transaction Aggregation”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomarti. Cham: Springer International Publishing, 2020, pp. 269–286. ISBN: 978-3-030-66172-4

6. Lewis Gudgeon, Sam Werner, Daniel Perez, and William J. Knottenbelt. “DeFi Protocols for Loanable Funds: Interest Rates, Liquidity and Market Efficiency”. In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. AFT ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 92–112. ISBN: 9781450381390. DOI: 10.1145/3419614.3423254. URL: https://doi.org/10.1145/3419614.3423254

7. Toshiko Matsui and Daniel Perez. “Data-driven analysis of central bank digital currency (CBDC) projects drivers”. In: *Mathematical Research for Blockchain Economy: 3rd International Conference MARBLE 2022, Vilamoura, Portugal*. Springer. 2023, pp. 95–108

8. Daniel Perez and Lewis Gudgeon. “Dissimilar redundancy in DeFi”. In: *Mathematical Research for Blockchain Economy: 3rd International Conference MARBLE 2022, Vilamoura, Portugal*. Springer. 2023, pp. 109–125

9. Jiahua Xu, Daniel Perez, Yebo Feng, and Benjamin Livshits. "Auto-gov: Learning-based On-chain Governance for Decentralized Finance (DeFi)". In: *arXiv preprint arXiv:2302.09551* (2023). URL: `https://arxiv.org/abs/2302.09551`

Chapter 2

Background

In this chapter, we will provide background about blockchains, smart contracts, and their applications that will be useful for understanding the rest of the thesis. More specific background information, such as some more details about the Ethereum platform, will be provided in the relevant chapters.

2.1 Blockchain fundamentals

In its essence, a blockchain is an append-only, decentralized database that is replicated across multiple computer nodes. Most blockchain systems record activities in the form of "transactions". A transaction typically contains information about its sender, its receiver, as well as the action taken, such as the transfer of an asset. Newly created transactions are broadcast across the network where they get validated by the participants. Valid transactions are grouped into data structures called *blocks*, which are appended to the blockchain by referencing the most recent block. Blocks are immutable, and state changes in the blockchain require new blocks to be produced.

Network latency and asynchrony inherent in the distributed nature of blockchains lead to various challenges. In particular, a blockchain must be able to reach a consensus about the

current state when the majority of participating nodes behave honestly. In order to resolve the disagreement, a consensus protocol prescribing a set of rules is applied as part of the validation process.

To ensure consistency, there needs to be:

1. a set of rules to validate transactions ;
2. a set of rules to validate blocks ;
3. a mechanism to determine which chain of blocks represents the current state

In the following subsections, we will provide definitions and explanations about transactions, blocks and consensus mechanisms, and then describe how the above requirements are fulfilled in most existing blockchain systems.

2.1.1 Transactions and blocks

Transactions. The smallest unit of work in most blockchain systems, including Bitcoin and Ethereum, is a transaction. The exact content of a transaction varies between different systems. It typically contains information about the sender, the receiver, the amount of the asset being transferred, and information on the fees to be paid for the transaction to be processed. In the case of Ethereum, transactions can also contain arbitrary data, which can be used to invoke smart contracts, which we will cover in more depth further in this chapter. As part of the set of rules to fulfil Requirement 1 above, transactions are signed by the sender, and the signature is used to verify that the transaction is valid. The systems also verify that the sender has enough funds to cover the transaction, including its associated fees, and that the transaction is not a duplicate of a previously processed transaction. Finally, in the case of a smart contract interaction, the execution of the smart contract must also be successful.

Blocks. A block is a collection of transactions that are grouped and appended to the blockchain. Blocks typically contain some extra metadata, such as a hash of the previous

block, effectively linking them together, a timestamp, and information about the transactions included in the block. Each block is usually limited to a maximum number of transactions, and a block is considered valid if all transactions it contains are valid. However, this is not the only requirement for a block to be deemed valid since this requirement alone would be prone to double-spending. To fulfil Requirement 2 above, some consensus algorithm needs to be used. We will discuss the most common consensus algorithms in the next subsection.

2.1.2 Consensus

Proof-of-Work. The Proof-of-Work (PoW) consensus, introduced by Bitcoin requires the participant to solve a computationally expensive puzzle to create a new block. Although PoW can maintain consistency well, it is by nature very time- and energy-consuming, which limits its throughput. To preserve security while maintaining a sufficient degree of decentralization, scalability is often sacrificed [Xie+19]. Indeed, the rate of block creation for both Bitcoin is relatively slow—on average 10 minutes per block, respectively—and the only way to increase the throughput is to increase the size of a single block, allowing for more transactions per block.

Proof-of-Stake. Along with PoW, Proof-of-Stake (PoW), to which Ethereum has recently transitioned, is another consensus mechanism that solves the same issue without requiring a large amount of computational power. PoS requires its participants, called block proposers and validators, to stake a certain amount of their assets to be eligible to create a new block. The probability of a block proposer being selected to create a new block is proportional to the total amount of assets they have staked. The block proposer is then required to create a new block and broadcast it to the network. The validators then verify the block and vote on whether it is valid. If the block is invalid, the block proposer is penalised, by taking some of the assets it has staked, and the block is discarded. If a majority of the validators agree that the block is valid, it is appended to the blockchain.

Deciding on the valid chain. These consensus mechanisms alone are not enough to fulfil Requirement 3. In both PoW and PoS, there could be two competing versions of the blockchain

containing a different chain of blocks. For PoW, this could for example happen if two miners have found a solution to the puzzle at the same time. The most common solution to this problem is to treat the longest chain of blocks as the valid one. In the case of PoW, the longest chain is not exactly the one that contains the most blocks but the one that contains the most work, i.e., the one that requires the most computational effort to produce. In the case of PoS, the issue is more complex since it does not require any computational effort to produce a block. This means that in theory, someone could produce a large number of blocks to create the longest chain. To prevent this, multiple mechanisms exist but the one used by Ethereum is to force blocks to be produced within a fixed interval of time and to use some blocks as checkpoints that cannot be reverted.

2.1.3 Incentives and fees

For both PoW and PoS to work, the miners, or block proposers for PoS, need to be incentivised to participate in the consensus process. This is typically done by rewarding them with a certain amount of the asset when they produce a new block, and also giving them the transaction fees of the transactions included in the block. For Bitcoin, the incentive mechanism is straightforward: the miner receives a fixed block reward, of which the amount halves every four years, and the sum of all the transaction fees of transactions in the block. Ethereum used to work in the same way but has since changed to a more complex incentive mechanism. One of the main differences with the Bitcoin model is that the block reward is not fixed but instead varies with the total amount of assets staked in the system. Another notable difference is that part of the fees are not distributed to the block proposer but instead are burned, i.e., they are destroyed.

2.2 Smart contracts

The Ethereum [But14] platform was the first to allow its users to run “smart contracts” on its distributed infrastructure. Smart contracts are programs that define a set of rules for the governing of associated funds, typically written in a Turing-complete programming language,

usually Solidity [Dan17] in the case of Ethereum. Solidity is similar to JavaScript, yet some notable differences are that it is strongly typed and has built-in constructs to interact with the Ethereum platform. Programs written in Solidity are compiled into low-level untyped bytecode to be executed on the Ethereum platform by the Ethereum Virtual Machine (EVM) [Woo14]. It is important to note that it is also possible to write EVM contracts without using Solidity.

To execute a smart contract, a sender has to send a transaction to the contract and pay a fee which is derived from the contract's computational cost, measured in units of *gas*. Each executed instruction consumes an agreed-upon amount of gas [Woo14]. Consumed gas is credited to the miner of the block containing the transaction, while any unused gas is refunded to the sender. In order to avoid system failure stemming from never-terminating programs, transactions specify a gas limit for contract execution. An out-of-gas exception is thrown once this limit has been reached.

Smart contracts themselves have the capability to *call* another account present on the Ethereum blockchain. This functionality is overloaded, as it is used both to call a function in another contract and to send Ether (ETH), the underlying currency in Ethereum, to an account. A particularity of how this works in Ethereum is that calls from within a contract, also called *internal transactions*, do not create new transactions and are therefore not directly recorded on-chain. This means that looking at transactions without executing them does not provide enough information to follow the flow of Ether.

2.3 Applications

DAOs. One of the earliest applications of distributed ledger technology was the creation of Decentralized Autonomous Organizations (DAOs). A DAO is a decentralized organization that uses a blockchain to facilitate the management of its funds and decision-making process. It often uses at least a multi-signature wallet to manage its funds, which means that several members of the organisation are required to approve any transfer of funds. DAOs also often have more complex rules, such as delays to be able to perform some actions or recurrent payments, that can

be enforced by smart contracts. The blockchain executes the rules and decision-making process transparently, resulting in a system requiring less trust than its centralised counterparties. DAOs can serve a variety of functions, such as managing decentralized funds, investing in projects, creating decentralized social networks, or facilitating community governance.

DeFi. Decentralized Finance (DeFi) is a peer-to-peer financial system powered by a blockchain that is designed to operate without the need for traditional financial intermediaries such as banks, brokerages, or exchanges. In a DeFi system, financial transactions and services are facilitated by smart contracts that automatically execute transactions and enforce the terms of agreements. Examples of DeFi applications include decentralized exchanges (DEXs), where users can trade cryptocurrencies without the need for a centralized exchange, lending platforms that enable borrowers to access loans without going through a traditional bank, and stablecoins that are designed to maintain a stable value. We will discuss DeFi in depth in Chapter 5.

Others. There are many other applications of blockchain technology, such as the tokenisation of art, often through non-fungible tokens (NFT), decentralized identity management, decentralized storage, and decentralized social networks. Although these are important applications, they are not the focus of this thesis and will not be discussed in detail.

2.4 Examples of Attacks on Blockchain Systems

Over the years, blockchain systems have been the target of many attacks. These attacks have happened at different levels of the system, from the consensus protocol to the smart contracts. In this section, we provide an example attack for each of the layers that we cover in this thesis: execution layer, transactional layer, and application layer. Furthermore, for the application layer, we provide an example of a technical attack and an example of an economic attack. Formal definitions for these terms are provided in Chapter 5.

Shanghai attack. In September 2016, a DoS attack was performed on the Ethereum network. The attack was performed during a major Ethereum conference taking place in Shanghai,

hence the name of the attack. The gist of the attack was an attacker flooding Ethereum with transactions that were cheap to execute but would take the network participants a long time to process. This resulted in the network taking much longer than normal to produce new blocks and forced Ethereum to revisit its pricing mechanism for transactions. We give more in-depth details about this attack in Chapter 3.

Solana DDoS. While the Shanghai attack is a good example of a DoS attack that relies on abusing the resource pricing mechanism of a transaction, other blockchains have also been the target of DDoS attacks despite the resource pricing mechanism working as intended. This happens most often in blockchains that try to have very low transaction fees, such as Solana [22], where a single transaction costs less than a cent. In September 2021, an attacker performed a DDoS attack on Solana by sending over 400,000 transactions per second, which is vastly over the network's capacity. Because of the extremely low transaction price, the cost of the attack was very moderate for the attacker. The Solana network ended up having to roll back the network to be able to recover. We discuss similar types of attacks and discuss in-depth the trade-offs of low transaction fees in Chapter 4.

TheDAO hack. TheDAO exploit [SC17] is one of the most infamous exploits that occurred in an Ethereum smart contract. Attackers exploited a reentrancy vulnerability [ABC17] of the contract which allowed for the draining of the contract's funds. The attacker contract could call the function to withdraw funds in a re-entrant manner before its balance on TheDAO was reduced, making it indeed possible to freely drain funds. A total of over 3.5 million Ether were drained. Given the severity of the attack, the Ethereum community finally agreed on hard forking. We discuss this type of vulnerability in depth in Chapter 5.

Chapter 3

Virtual Machine Security

Blockchain systems, such as Ethereum, use an approach called “metering” to assign a cost to smart contract execution, an approach which is designed to incentivise miners to operate the network and protect it against DoS attacks. In the past, the imperfections of Ethereum metering allowed several DoS attacks which were countered through modification of the metering mechanism.

This chapter presents a new DoS attack on Ethereum which systematically exploits its metering mechanism. We first replay and analyse several months of transactions, during which we discover several discrepancies in the metering model, such as significant inconsistencies in the pricing of the instructions. We further demonstrate that there is very little correlation between the execution cost and the utilised resources, such as CPU and memory. Based on these observations, we present a new type of DoS attack we call *Resource Exhaustion Attack*, which uses these imperfections to generate low-throughput contracts. To do this, we design a genetic algorithm that generates contracts with a throughput on average 100 times slower than typical contracts. We then show that all major Ethereum client implementations are vulnerable and, if running on commodity hardware, would be unable to stay in sync with the network when under attack. We argue that such an attack could be financially attractive not only for Ethereum competitors and speculators but also for Ethereum miners. Finally, we discuss short-term and potential long-term fixes against such attacks. Our attack has been responsibly disclosed to

the Ethereum Foundation and awarded a bug bounty reward of 5,000 USD.

3.1 Introduction

Some blockchain systems support code execution, allowing arbitrary programs to take advantage of decentralised trust. Ethereum and its virtual machine, the Ethereum Virtual Machine (EVM), is probably the most widely used blockchain adopting this approach. However, allowing arbitrary programs from non-trusted users introduces many new challenges. One of these challenges is to prevent users from running code which could negatively impact the performance of the system. To tackle this challenge, Ethereum introduced the notion of "gas", which is a unit used to measure the execution cost of a program, referred to as a "smart contract" in this context. Gas-based metering is used to price the execution of smart contracts and must ensure that the throughput of the blockchain, in terms of gas per second, remains stable. Metering is therefore critical to keep the Ethereum blockchain safe against Denial of Service (DoS) attacks involving slow-running contracts. However, assigning costs to different instructions is a highly non-trivial task, and the costs originally assigned in the Ethereum yellow paper [Woo14], which were designed to maintain a throughput of 1 gas/μs, had many inconsistencies. As a consequence, several DoS attacks have been conducted on Ethereum [Butc; Butb], and the gas cost has also been reviewed several times [Buta; Swe19] to increase the cost of the under-priced instructions.

To the best of our knowledge, this was the first attempt to try to find and exploit such inconsistencies systematically. In this chapter, we design a new DoS attack which exploits inconsistencies in the gas metering mechanism by taking a systematic approach to finding these. We first replay and analyse several months of transactions to discover discrepancies in the gas cost. We then use the data and insight from our analysis to design a genetic algorithm capable of generating low-throughput contracts. We evaluate the contracts generated by our algorithm on all major Ethereum clients and find that they are all vulnerable to our attack.

Contributions. This chapter makes the following contributions:

1. **Exploration of metering in EVM**: We explore the history of executing 2.5 months worth of smart contracts on the Ethereum blockchain and identify several important edge cases that highlight inherent flaws in EVM metering; specifically, we identify i) EVM instructions for which the gas fee is too low compared to their resources consumption; and ii) cases of programs where the cache influences execution time by an order of magnitude.

2. **Resource Exhaustion Attacks (REA) contract generation strategy**: We present a code generation strategy able to produce REA attacks of arbitrary length. Some of the complexity comes from the need to produce well-formed EVM programs which minimise the throughput. We propose an approach which combines empirical data and a genetic algorithm in order to generate contracts with low throughput. We explore the efficacy of our strategy as a function of the throughput in terms of gas per second of the generated programs.

3. **Experimental evaluation**: We show that our REA can abuse imperfections in EVM's metering approach. Our genetic algorithm can generate programs with a throughput of 1.25M gas per second after a single generation. A minimum in our experiments is attained at generation 243 with a block using around 9.9M gas and taking about 93 seconds. We show that our method generates contracts on average more than 100 times slower than typical contracts. Finally, we evaluate our low-throughput contracts on the major Ethereum clients and show that they are all vulnerable. Using commodity hardware, nodes would be unable to stay in sync when under attack.

4. **Disclosure and fixes**: We responsibly disclosed our attack to the Ethereum Foundation and were awarded a bug bounty reward of 5,000 USD. We discussed with the developers about the ongoing efforts as well as some potential fixes, and present some of the short-term and long-term fixes in this chapter.

Chapter Organisation. The rest of the chapter is organised as follows. In Section 3.2, we provide background information about Ethereum and its metering scheme, as well as a few instances of how it has been exploited in the past. In Section 3.3, we present case studies based

on measurements that we obtained by re-executing the Ethereum main chain. In Section 3.4, we present our Resource Exhaustion Attacks (REA) and the results we obtained. In Section 3.5 we present short and long-term solutions to gas mispricing issues. Finally, we present related work in Section 3.6, and conclude in Section 3.7.

3.2 Background

In this section, we give an in-depth description of gas metering in EVM. We then provide insights into smart contract execution costs on the Ethereum main network. Then, we highlight some of the attacks which have been performed by abusing the gas mechanism.

3.2.1 Metering in EVM

As briefly outlined in Section 3.1, gas is a fundamental component of Ethereum, and generally applicable to permissioned and permissionless blockchain platforms that utilise a distributed virtual machine for contract code execution [Tez19a; Blo19]. Gas is the main protection against Denial of Service (DoS) attacks based on non-terminating or resource-intensive programs. It is also used to incentivise miners to process transactions by rewarding them with a fee computed based on the resource usage of the transaction.

Gas cost. In the EVM, each transaction has a cost which is computed and expressed as gas. The cost is split into two parts, a fixed *base cost* of $21,000$ gas, and a variable *execution cost* of the smart contract. Each instruction has a fixed gas cost which has been set by the designers of the EVM [Woo14], who classify the instructions into multiple tiers of gas cost: zero Tier (0 gas), base tier (2 gas), very low tier (3 gas), low tier (5 gas), high tier (10 gas) and special tier where the cost needs more complex rules. The gas cost for a transaction in the EVM is the sum of the cost of each instruction in the contract. For example, given the program in Code Listing 3.1, the gas cost will be computed as follow. `PUSH1` is in the Very Low Tier and therefore costs 3 gas. It is called 3 times in total and will therefore consume 9 gas. The arguments of

Code Listing 3.1: Example gas cost of an EVM program

```
PUSH1 0x02 ; very low tier (3 gas)
PUSH1 0x03 ; very low tier (3 gas)
MUL        ; low tier      (5 gas)
PUSH1 0x05 ; very low tier (3 gas)
SSTORE     ; special tier  (20k gas)
```

PUSH1 do not consume any extra gas. The MUL instruction is in the Low Tier and hence costs 5 gas. Finally, the SSTORE will store the result of 2×3 at location 5 in the storage. SSTORE is in the Special Tier and has slightly more complex pricing rules. Assuming the location in the storage was previously 0, the instruction allocates storage and will cost $20,000$ gas. Therefore, this program will cost a total of $20,014$ gas to execute. Given the current pricing for storage, the cost of the program is largely dominated by the storage operation.

It is important to note that, as the transaction has a base cost of 21,000 gas, it will cost a total of $21{,}000 + 20{,}014 = 41{,}014$ gas to execute the above transaction.

Ethereum Improvement Proposal (EIP) 150. Although the cost of each instruction was decided when first designing the EVM, the authors found that some costs were poorly aligned with actual resource consumption. Particularly, IO-heavy instructions tended to be too cheap, allowing for DOS attacks on the Ethereum [Butb] blockchain. As a fix, EIP 150 [Buta] was proposed and implemented, significantly increasing the gas consumption of instructions which require performing IO operations, such as SLOAD or EXTCODESIZE. This change revised the cost of under-priced instructions and prevented further DoS attacks such as the one seen in September 2016 [Butc]. This briefly highlights the potential risks rooted in mismatches between instructions and gas costs. While the above cases have been fixed, it is unclear whether all potential issues have been eradicated or not.

Gas price. Up to here, we have explained how the gas cost for executing a contract is computed. However, the gas cost is not the only element needed to compute the total execution cost of a contract. When a transaction is sent, the sender can choose a gas price, namely the amount of *wei* ($1\text{wei} = 10^{-18}$ ETH) that the sender is ready to pay per unit of gas. For conciseness, these amounts are often expressed in Gwei, where $1\text{Gwei} = 10^9\text{wei}$. Miners will

Table 3.1: Fees for different types of transactions. "Low" price is one of the lowest possible prices to have a transaction included while "High" is a price someone very eager to have his transaction included would pay.

Transaction type	Gas price Low (1Gwei)	 High (80Gwei)
Basic (21k gas)	$0.042	$3.36
Gas intensive (500k gas)	$1	$80

Table 3.2: Median gas price, gas used and transaction fee from block 8,652,096 (Sep-09-2019) to block 9,286,594 (Jan-15-2020).

Number of blocks:	613,475
Median gas price:	9.1 Gwei
Median gas used (by contracts):	53,787
Median transaction fee:	0.0008 ETH (1.6 USD)

usually prioritise transactions with high gas prices, as this will increase the final fee they receive for processing a transaction.

Transaction fee. The transaction fee is the total amount of wei that the sender of the transaction has to pay for the transaction. It is obtained by multiplying the gas price by the gas cost. The transaction fee is non-refundable: even if the transaction fails, it will be paid.

3.2.2 Gas Statistics

Now that we presented the key points about metering in the EVM, we provide concrete numbers about different aspects of the gas price and transaction fees. In particular, we show the total amount of transaction fees that a user would have to pay to have his transaction processed by the main Ethereum network.

To give a sense of the transaction fees, we show a variety of typical fees in Table 3.1. The fees are divided depending on their gas price and gas consumption. The *Low* gas price is close to the lowest price that can be paid to get the transaction accepted on the Ethereum blockchain. The *High* gas price refers to the price that people would pay when they are extremely eager to get their transaction included, for example when competing with other users to have a transaction

included first [Owo18]. The *basic* transaction type refers to transactions consuming only the base amount of gas, without executing any instruction. This is typically the cost to send Ether to a contract or another party. The *gas intensive* transaction type represents computationally expensive transactions, for example, verifying a zero-knowledge proof [Put18]. At the time of the analysis, in late 2019, the maximum amount of gas which can be used in a single block is 10,000,000, which means only 20 such transactions could be included in a single block.

In Table 3.2, we show the values of the gas price, gas used and transaction fee. In order to obtain results reflecting the current situation, we limit the analysis to recent blocks. We use all the transactions sent to contracts between September 30, 2019, and January 15, 2020. We find that the median gas price paid by a transaction's sender is around 9.1 Gwei, which is around 9 times more than the minimum possible fee. It is worth noting that when paying the minimum possible fee, the probability for the transaction to get included in the next block is relatively low and the transaction can therefore be delayed for several blocks: at the time of the analysis, about 40% of the last 200 blocks accepted a gas price of 1Gwei [Com19a]. This explains that users usually pay a higher fee to get their transactions included faster. The median for the gas consumed by contracts is around 50,000 gas, indicating that most transactions perform relatively simple computations. Indeed, with the basic fee being 21,000, a simple read followed by an allocation of storage would already result in 46,000 gas. Overall, the median fee paid per transaction is 0.0008 ETH which is around 1.6 USD.

3.2.3 Previously Known Attacks

The Ethereum network has been the victim of several Denial of Service (DoS) attacks due to instructions being underpriced. We present two considerable DoS attacks which were performed on the Ethereum network.

`EXTCODESIZE` attack. This attack is the 2016 Shanghai attack that we introduced in Chapter 2. It was a DoS attack performed on the Ethereum network by flooding it with transactions containing a very large number of `EXTCODESIZE` instructions [Butc]. `EXTCODESIZE` is an instruction

to retrieve the size in bytes of a given contract's code.

This attack happened because the EXTCODESIZE instruction was vastly underpriced. At the time of the attack, a single execution of this instruction cost 20 gas, meaning that one could perform around 1,500 instructions with less than $0.01. Although by itself, this issue might seem benign, EXTCODESIZE forces the client to search the contract on disk, resulting in IO heavy transactions. While replaying the Ethereum history on our hardware, the malicious transactions took around 20 to 80 seconds to execute, compared to a few milliseconds for the average transactions. We show the correlation between the clock time and the gas used by transactions during the period of the attack in Figure 3.1. Although this attack did not create any issue at the consensus layer, it reduced the rate of block creation by a factor of more than 2 times, with block creation time peaking to more than 30s [Eth19].

The Ethereum protocol was updated in EIP 150, with all the software running Ethereum, to increase the price of the EXTCODESIZE from 20 to 700 gas, making the aforementioned attack considerably more expensive to perform. Some performance improvements were also made at the implementation level, allowing clients to process IO-intensive instructions faster.

SUICIDE Attack. Shortly after the EXTCODESIZE attack, another DoS attack involving the SUICIDE instruction was performed [Butb]. The SUICIDE instruction kills a contract and sends all its remaining Ether to a given address. If this particular address does not exist, a new address would be newly created to receive the funds. Furthermore, at the time of the attack, calling SUICIDE did not cost any Ether. Given these two properties, an attacker could create and destroy a contract in the same transaction, creating a new contract each time at an extremely low fee. This quickly overused the memory of the nodes and particularly affected the Go implementation [Aut19] which was less memory efficient [But16].

A twofold fix was issued for this attack in EIP 150. First, and most importantly, SUICIDE would be charged the regular amount of gas for contract creation when it tried to send Ether to a non-existing address. Subsequently, the price of the SUICIDE instruction was increased from 0 to 5,000 gas. Again, these measures would make such an attack very expensive.

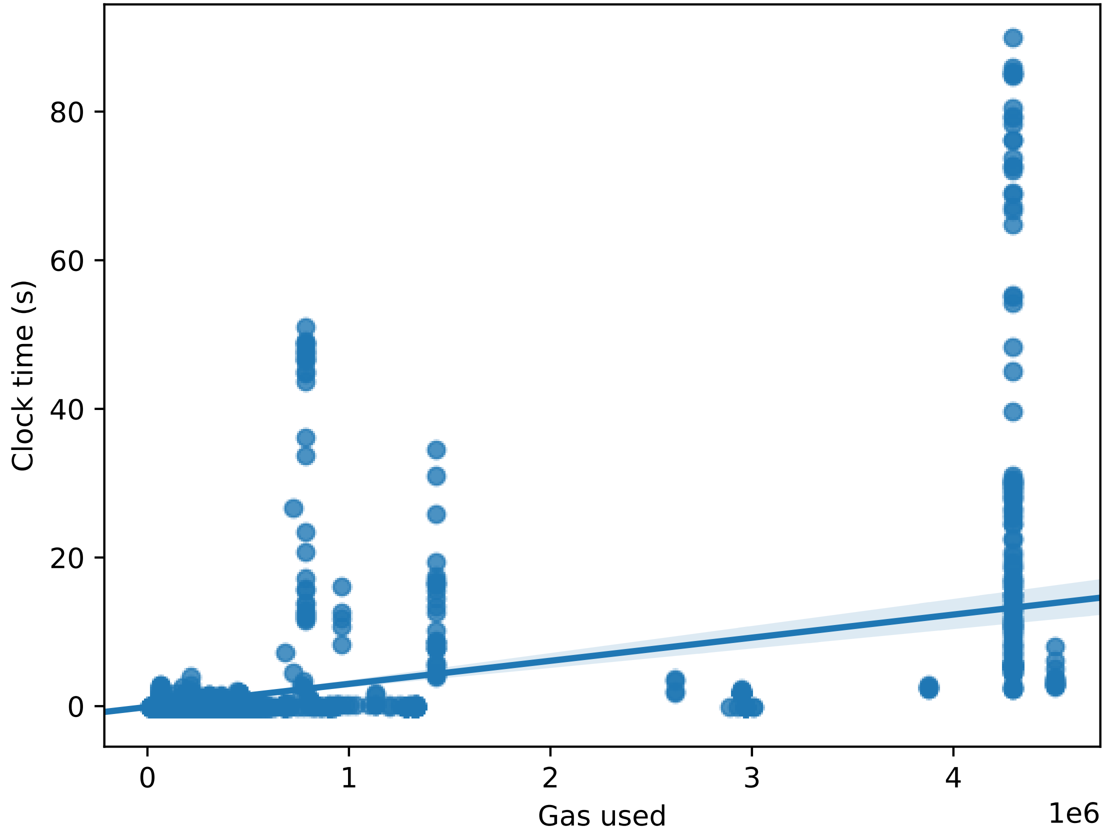


Figure 3.1: Correlation between gas and clock time when performing a resource exhaustion attack.

3.3 Case Studies in Metering

In this section, we instrument the C++ client of the Ethereum blockchain, called *aleth* [Eth], and report some interesting observations about gas dynamics in practice.

3.3.1 Experimental setup

Hardware. We run all of the experiments on a Google Cloud Platform (GCP) [Goo19] instance with 4 cores (8 threads) Intel Xeon at 2.20GHz, 8 GB of RAM and an SSD with a 400MB/s throughput. The machine runs Ubuntu 18.04 with the Linux kernel version 4.15.0. We selected this hardware because it is representative of what has been reported as sufficient to run a full Ethereum node [Pet18; Peg19; Pal19].

Software. To measure the speed of different instructions, we fork the Ethereum C++ client, *aleth*. Our fork integrates the changes to the upstream repository until Jun-26 2019. We choose the C++ client for two reasons: first, it is one of the two clients officially maintained by the Ethereum Foundation [19b] with geth [Aut19]; second, it is the only of the two without runtime or garbage collection, which makes measuring metrics such as memory usage more reliable.

We add compile options to the original C++ client to allow enabling particular measurements such as CPU or memory. Our measurement framework is open-sourced[1] and available under the same license as the rest of *aleth*.

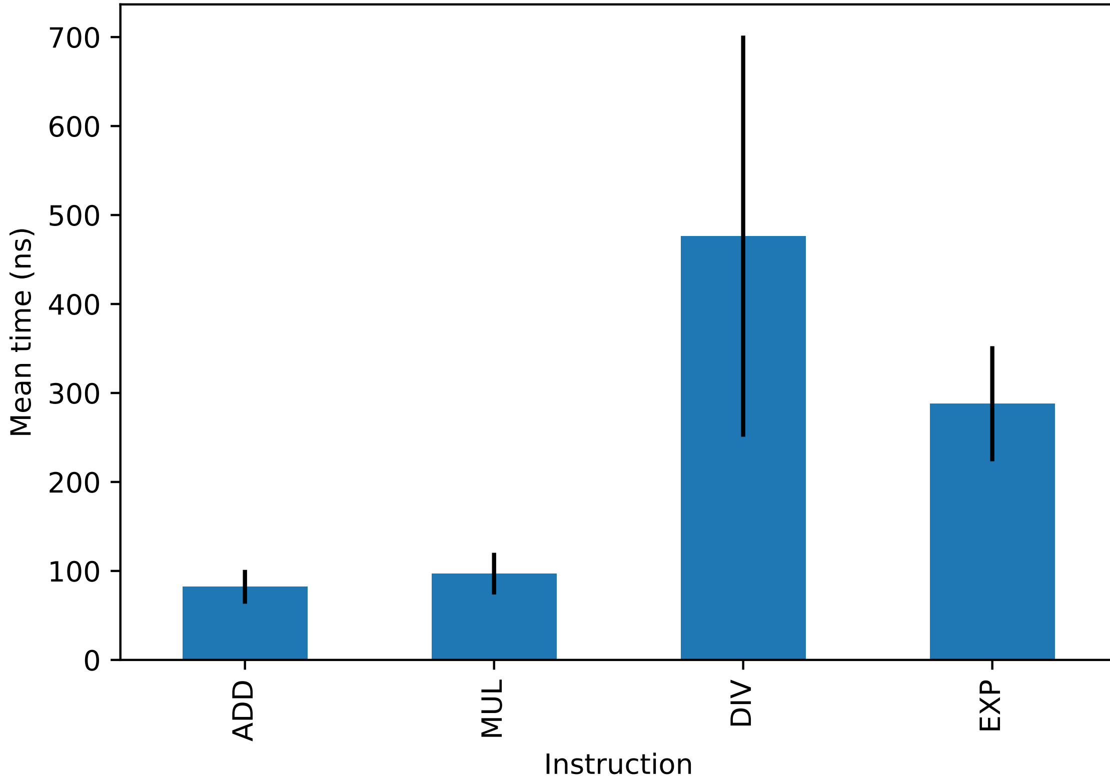
Measurements. For all our measurements, we only take into account the execution of the smart contracts and ignore the time taken in networking or other parts of the software. We use a nanosecond precision clock to measure time and measure both the time taken to execute a single smart contract and the time to execute a single instruction. To measure the memory usage of a single transaction, we override globally the **`new`** and **`delete`** operators and record all allocations and deallocations performed by the EVM execution within each transaction. We ensure that this is the only way used by the EVM to perform memory allocation.

Given the relatively large amount of time it takes to re-execute the blockchain, we only execute each measurement once when re-executing. We ensure that we always have enough data points, where enough is in the order of millions or more, so that some occasional imprecision in the measurements, which are inevitable in such experiments, does not skew the data.

In this section, the measurements are run between block 5,171,468 (Feb-28-2018) and block 5,587,480 (May-10-2018), except in Section 3.3.3 where we want to compare after and before EIP-150.

We note that these measurements are not representative of the current state of the Ethereum blockchain, as the network has evolved significantly since then. In particular, EIP-2929 was introduced in April 2021 [BS20], partly based on the findings from this chapter, and significantly increased the cost of accessing storage.

[1] `https://github.com/danhper/aleth/tree/measure-gas`



(a) Mean time for arithmetic instructions.

Instruction	**Gas cost**	**Count**	**Mean time** (ns)	**Throughput** (gas / μs)
ADD	3	453,069	82.20	36.50
MUL	5	62,818	96.96	51.57
DIV	5	107,972	476.23	10.50
EXP	~51	186,004	287.93	177.1

(b) Execution time and gas usage for arithmetic instructions.

Figure 3.2: Comparing execution time and gas usage of arithmetic instructions.

3.3.2 Arithmetic Instructions

In this experiment, we evaluate the correlation between gas cost and the execution time for simple instructions which include absolutely no IO access. We use simple arithmetic instructions for measurements, in particular the `ADD`, `MUL`, `DIV` and `EXP` instructions.

In Figure 3.2a, we show the mean time of execution for these instructions, including the standard deviation for each measurement. We contrast these results with the gas cost of the different

Table 3.3: Correlation scores between gas and system resources.

Phase	Resource	Pearson score
Pre EIP-150	Memory	0.545
	CPU	0.528
	Storage	0.775
	Storage/Memory	**0.845**
	Storage/Memory/CPU	0.759
Post EIP-150	Memory	0.755
	CPU	0.507
	Storage	0.907
	Storage/Memory	**0.938**
	Storage/Memory/CPU	0.893

instructions in Figure 3.2b. `EXP` is the only of these instructions with a variable cost depending on its arguments — the value of the exponent. We use the average gas cost in our measurements to compute the throughput. We see that although in practice `ADD` and `MUL` have similar execution time, the gas cost of `MUL` is 65% higher than the gas cost for `ADD`. On the other hand, `DIV`, which costs the same amount of gas as `MUL`, is around *5 times slower* on average. `EXP` costs on average *10 times* the price of `DIV` but executes 40% faster. Another point to note here is that `DIV` has a standard deviation much higher than the other three instructions. Although we were expecting that for such simple instructions, the execution time would reflect the gas cost, this does not appear to be the case in practice. We will show in the coming sections that IO-related operations tend to make things worse in this regard.

3.3.3 Gas and System Resources Consumption

In this section, we analyse the gas consumption of Ethereum smart contracts and try to correlate it with different system resources, such as memory, CPU and storage. As described in Section 3.2, EIP-150 influenced the price of many storage-related operations, which affected the gas cost of transactions. Therefore, we use a different set of transactions than for other case studies. We arbitrarily use block 1,400,000 to block 1,500,000 for measurements before EIP-150 and block 2,500,000 to 2,600,000 for measurements after EIP-150. We assume that

the sample of 100,000 blocks, which roughly corresponds to two weeks, is large enough to obtain reliable data.

We use our modified Ethereum client to perform the different measurements. To measure memory, we compute the difference between the total amount of memory allocated and the total amount of memory deallocated. For CPU, we use clock time measurements as a proxy for CPU usage. Finally, for storage usage, we count the number of EVM words (256 bits) of storage newly allocated per transaction.

We compute the Pearson correlation coefficient[2] [Bos12] between the different resources and the gas usage. We also compute multi-variate correlations between gas consumption and multiple resources. To compute the multi-variate correlation between multiple resources and gas usage, we first normalise the measurement vector of each targeted resource to have a mean of 0 and a standard deviation of 1. Then, we stack the vectors to obtain a matrix of m resources and n measurements and transform it into a single vector of n measurements using a principal component analysis [AW10]. The vector we obtain represents the aggregated usage of the different resources and can be correlated with the gas usage.

We present our results in Table 3.3. A first observation is that EIP-150 clearly emphasises the domination of storage in the price of contracts. We can see that storage alone has an extremely high correlation score, with a score of 0.907 after EIP-150. Memory usage is not as correlated as storage, but when combining both, they have the highest correlation score of 0.938. Finally, an important point is that CPU time seems completely uncorrelated with gas usage. Although it seems natural that CPU time by itself has a low correlation, as the gas cost is dominated by storage cost, adding the CPU time in the multi-variate correlation reduces the correlation. It is not enough to make any conclusion yet but gives a hint that as long as the storage is not explicitly touched, it could be possible for contracts to be both cheap and long to execute.

[2]Pearson score of 1 means perfect positive correlation, 0 means no correlation

Table 3.4: Instructions with the highest execution time variance.

Instruction	Mean time (μs)	Standard deviation	Measurements count
BLOCKHASH	768	578	240,000
BALANCE	762	449	8,625,000
SLOAD	514	402	148,687,000
EXTCODECOPY	403	361	23,000
EXTCODESIZE	221	245	16,834,000

3.3.4 High-Variance Instructions in the EVM

Here, we look at instructions which have a high variance in their execution time. We summarise the instructions which had the highest variance in Table 3.4. There are two main reasons why the execution time may vastly vary for the execution of the same instruction. First, many instructions take parameters. Depending on these, the time it takes to run the particular instructions can vary wildly. This is the case for an instruction such as `EXTCODECOPY`. The second reason is much more problematic and comes from the fact that some instructions may require to perform some IO access, which can be influenced by many different factors such as caching, either at the OS or at the application level. The instruction with the highest variance was `BLOCKHASH`. `BLOCKHASH` allows to retrieve the hash of a block and allows to look up up to 256 block before the current one. When it does so, depending on the implementation and the state of the cache, the EVM may need to perform an IO access when executing this instruction, which can result in vastly different execution times. The cost of `BLOCKHASH` being currently fixed and relatively cheap, 20 gas, it results in an instruction which is vastly under-priced. It is worth noting that in the particular case of `BLOCKHASH`, the issue has already been raised more than two years ago in EIP-210 [But19]. It discussed changing the price of `BLOCKHASH` to 800 gas but at the time of writing the proposal is still in draft status and was not included in the Constantinople fork[3] [Hud19] as it was originally planned to be. It has not been included in any further hard fork either.

[3] Hard fork which took place on Feb 28 2019 on the Ethereum main network

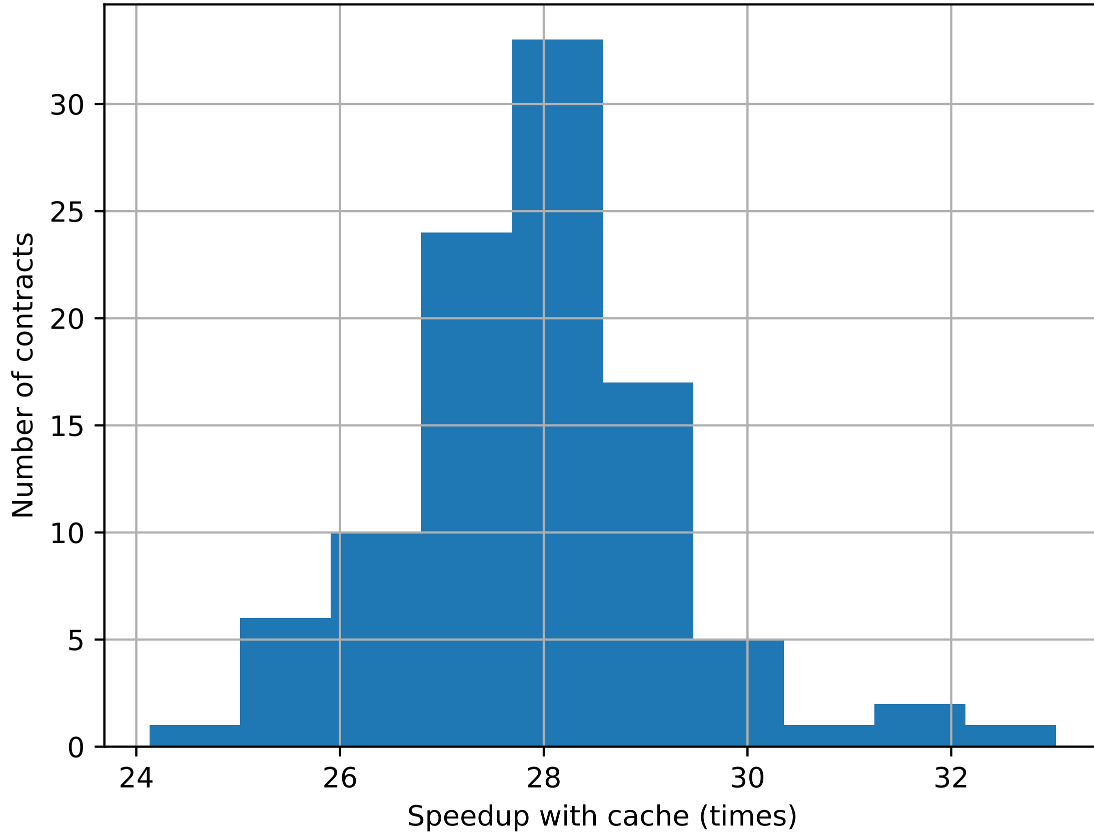


Figure 3.3: Comparing throughput with and without page cache: x axis is the relative speed improvement and y axis is the number of contracts.

3.3.5 Memory Caches and EVM Costs

Given the high variance in execution time for some instructions, we evaluate the effects caching may have on EVM execution speed. In particular, we evaluate both the speedup provided by the operating system page cache and the speedup across blocks provided by LevelDB LRU cache [GD11b]. In these experiments, we fix the block number at height 5,587,480.

Page cache. First, we evaluate how the operating system page cache influences the execution time by reducing the IO latency. We perform the experiment as follows:

1. Generate a contract

2. Run the code of the contract n times

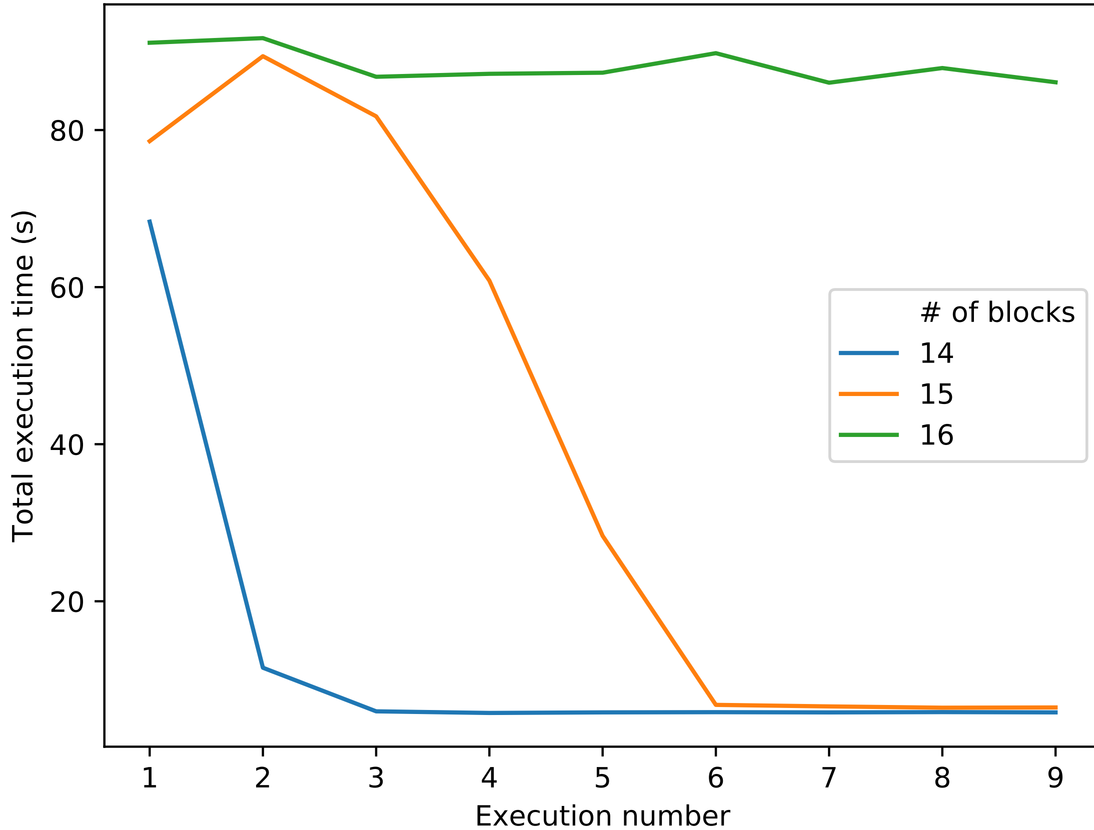


Figure 3.4: Measuring block execution speed with and without the effect of cache.

3. Run the code of the contract n times but drop the page cache between each run

We perform this for 100 different contracts and measure the execution time for the versions with and without cache. We generate relatively large contracts, which consume on average 800,000 gas each. Although the method is somewhat crude, it provides a good approximation of the extent to which the state of the page cache influences the execution time of a contract. In Figure 3.3, we show a distribution of the contracts throughput in terms of gas per second, with and without cache. We see that contracts execute between 24 and 33 times faster when using the page cache, with more than half of the contracts executing between 27 and 29 times faster. This vast difference in the execution speed is due to IO operations, which use LevelDB [GD11a], a key-value store database, under the hood. LevelDB keeps only a small part of its data in memory and therefore needs to perform disk access when the data was not found in memory. If the required part of the data was already in the page cache, no disk access will be required.

When keeping the page cache, all the items seen by the contract recently will already be available in the cache, eliminating the need for any disk access. On the other hand, if the caches are dropped, many IO-related operations will result in disk access, which explains the speedup. We notice that in the contracts with the highest speedup, `BLOCKHASH`, `BALANCE` and `SLOAD` are in the most frequent instructions. It is worth noting that if the generated contracts are small enough, most of the data will be in memory and dropping the page cache will have much less effect on the runtime. Indeed, when running the same experiment with contracts consuming on average 100,000 gas, only a 2 times average speedup has been observed.

Caching across blocks. In the next experiment, instead of measuring the cache impact by running a single contract multiple times, we evaluate how the cache impacts the execution time across blocks. In particular, we measure how many blocks need to be executed before the data cached during the previous execution of a contract gets evicted from the different caches. To do so, we perform the following experiment.

1. Generate n blocks, with different contracts in each
2. Execute sequentially all the blocks and measure the execution time
3. Repeat the previous step m times in the same process and record how the execution speed evolves

We set m to 10 and we try different values for n to see how many blocks are needed for the cache not to provide any further speedup. We use the first execution to warm up the node and use the 9 other executions for our measurements. We find that in our setup, assuming the blocks are full (i.e. close to the gas limit in terms of gas), 16 blocks are enough for the cache not to provide any more speedup. We plot the results for $n = 14$, $n = 15$ and $n = 16$ in Figure 3.4. When $n = 14$, we see that the second execution is much faster than the first one and that after the third execution, the execution time stabilises at around 6s to execute the 14 blocks. For $n = 15$, the execution time takes longer to decrease, but eventually also stabilises around the same value. It is slightly higher than when $n = 14$ because it has one more block to

execute. However, once we reach $n = 16$, we see that the execution time hardly decreases and stays stable at around 85s. We conclude that at this point, almost nothing that was cached during the previous execution of the block is still cached when re-executing the block.

This means that if a deployed contract function were re-executed more than 16 blocks after its initial execution, it would execute as slowly as the first time. This shows that not only the cache has a very high impact on execution time but also that the cached information is evicted relatively quickly.

3.3.6 Summary

In this section, we empirically analysed the gas cost and resource consumption of different instructions. To summarise:

- We see that even for simple instructions, the gas cost is not always consistent with resource usage. Indeed, even for instruction with very predictable speed, such as arithmetic operations, we observe that some instructions have a throughput 5 times slower than others.

- We notice that while most instructions have a relatively consistent execution speed, other instructions have large variations in the time it takes to execute. We find that these instructions involve some sort of IO operation.

- Finally, we demonstrate the effect that the page cache has on the execution speed of smart contracts and then show the typical number of blocks for which the page cache still provides speed up.

- Overall, we see that beyond some pricing issues, the metering scheme used by EVM does not allow to express the complexity inherent to IO operations.

3.4 Attacking the Metering Model of EVM

In light of the results we obtained in the previous sections, we hypothesise that it is possible to construct contracts which use a low amount of gas compared to the resources they use.

3.4.1 Constructing Resource Exhaustion Attacks

In particular, as we showed in Section 3.3, the gas consumption is dominated by the storage allocated but is not as much affected by other resources such as the clock time. Therefore, we decide to use the clock time as a target resource and look for contracts which minimise the throughput in terms of gas per second. We can formulate this as a search problem.

Problem formulation. We want to find a program which has the minimum possible throughput, where we define the throughput to be the amount of gas processed per second. Let $\mathbb{I}$ be the set of EVM instructions and P be the set of EVM programs. A program $p \in P$ is a sequence of instructions $I_1, \cdots, I_n$ where all $I_i \in \mathbb{I}$. Let $t : P \to \mathbb{R}$ be a function which takes a program as an input and outputs its execution time and $g : P \to \mathbb{N}$ be a function which takes a program as input and outputs its gas cost. We define our function to minimise $f : P \to \mathbb{R},\ f(p) = g(p)/t(p)$. Our goal is to find the program p_{slowest} such that

$$p_{\text{slowest}} = \arg\min_{p \in P}(f(p)) \tag{3.1}$$

The search space for a program of size n is $|\mathbb{I}|^n$. Given $|\mathbb{I}| \approx 100$, the search space is clearly too large to be explored entirely for any non-trivial program. Therefore, we cannot simply go over the space of possible programs and instead need to approximate the solution.

Although our problem resembles other program synthesis tasks [GPS+17], there is a notable difference. Program synthesis usually focuses on generating "meaningful" programs, either from specifications or examples. These tasks often do not have well-defined metrics allowing optimisation techniques (the genetic algorithm in our work). The task we solve is different

because we need to define "valid" but not "meaningful" programs and optimise for a well-defined metric: gas throughput.

Search strategy. With the problem formulated as a search problem, we now present our search strategy. We decide to design the search as a genetic algorithm [Whi94]. The reasons for this choice are as follows:

- we have a well-defined fitness function f
- we have promising initialisation values, which we will discuss below
- programs being a sequence of instructions, cross-over and mutations can be designed efficiently
- programs generated need to be syntactically correct but do not need to be semantically meaningful, making the cross-over and mutations more straightforward to design

We will now discuss in detail how we design the initialisation, cross-over and mutations of our genetic algorithm.

Program construction. Although our programs do not need to be semantically meaningful, they need to be executed successfully on the EVM, which means that they must fulfil some conditions. First, an instruction should never try to consume more values than the current number of elements on the stack. Second, instructions should not try to access random parts of the EVM memory, otherwise, the program could run out of gas straight away. Indeed, when an instruction reads or writes to a place in memory, the memory is "allocated" up to this position and a fee is taken for each allocated memory word. This means that if `MLOAD` would be called with 2^{100} as an argument, it would result in the cost of allocating 2^{100} words in memory, which would result in an out-of-gas exception.

Another potential issue would be to run into an infinite loop. However, we decide to explicitly exclude loops from our program generation algorithm for the following reason: adding loops is unlikely to make the generated programs slower. Indeed, if a piece of code is slow enough, our

genetic algorithm will tend to repeat it. The loop version could be faster if a value is already cached but have no reason to be slower.

We design the program construction so that all created programs will never fail because of either of the previous reasons. We first want to ensure that there are always enough elements on the stack to be able to execute an instruction. The instructions requiring the least number of elements on the stack are instructions such as `PUSH` or `BALANCE` which do not require any element, and the element requiring the most number of elements on the stack is `SWAP16` which requires 17 elements to be on the stack. We define the functions function $a : \mathbb{I} \rightarrow \mathbb{N}$ which returns the number of arguments consumed from the stack and $r : \mathbb{I} \rightarrow \mathbb{N}$ which returns the number of elements returned on the stack for an instruction I. We generate 18 sets of instructions using Equation 3.2.

$$\forall n \in [0, 17],\ \mathbb{I}_n = \{I \mid I \in \mathbb{I} \wedge a(I) \leq n\} \tag{3.2}$$

For example, the set $\mathbb{I}_3$ is composed of all the instructions which require 3 or fewer items on the stack. We will use these sets in Algorithm 1 to construct the initial programs but before, we need to define the functions we use to control memory access. For this purpose, we define two functions to handle these. First, $uses_memory : \mathbb{I} \rightarrow \{0, 1\}$ returns 1 only if the given instruction accesses memory in some way. Then, $prepare_stack : \mathbb{P} \times \mathbb{I} \rightarrow \mathbb{P}$ takes a program and an instruction and ensures that all the arguments of the instruction which influence which part of memory is accessed are below a relatively low value, that we arbitrarily set to 255. To ensure this, *prepare_stack* adds `POP` instruction for all arguments of I and adds the same number of `PUSH1` instructions with a random value below the desired value. For example, in the case of `MLOAD`, a `POP` followed by a `PUSH1` would be generated.

Using the sets $\mathbb{I}_n$, the *uses_memory* and *prepare_stack* functions, we use Algorithm 1 to generate the program. We assume that the *biased_sample* function returns a random element from the given set and will discuss how we instantiate it in the next section.

Initialisation. As the search space is very large, it is important to start with good initial values

Algorithm 1 Initial program construction

```
function GeneratePROGRAM(size)
    P ← ( )                                   ▷ Initial empty program
    s ← 0                                     ▷ Stack size
    for 1 to size do
        I ← biased_sample(I_s)
        if uses_memory(I) then
            P ← prepare_stack(P, I)
        end if
        P ← P · ( I )                          ▷ Append I to P
        s ← s + (r(I) − a(I))
    end for
    return P
end function
```

so that the genetic algorithm can search for promising solutions. For this purpose, we use the result we presented in Section 3.3, in particular, we use the throughput measured for each instruction. We define a function $throughput : \mathbb{I} \to \mathbb{R}$ which returns the measured throughput of a single instruction. When randomly choosing the instructions with *biased_sample*, we want to have a higher probability of picking an instruction with a low throughput but want to keep a high enough probability of picking a higher throughput instruction to make sure that these are not all discarded before the search begins. We define the weight of each instruction and then its probability with Equation 3.3 and Equation 3.4.

$$W(I \in \mathbb{I}) = \log\left(1 + \frac{1}{throughput(I)}\right) \tag{3.3}$$

$$P(I \in \mathbb{I}_n) = \frac{W(I)}{\sum_{I' \in \mathbb{I}_n} W(I')} \tag{3.4}$$

Given that the throughput can have order-of-magnitude differences among instructions, the log in Equation 3.3 is used to avoid assigning a probability too close to 0 to an instruction.

Cross-over. We now want to define a cross-over function over our search space, a function which takes as input two programs and returns two programs, i.e. $cross_over : \mathbb{P} \times \mathbb{P} \to \mathbb{P} \times \mathbb{P}$, where the output programs are combined from the input programs. To avoid enlarging the search space with invalid programs, we want to perform a cross-over such that the two

output programs are valid by construction. During program creation, we must ensure that each instruction of the output program will always have enough elements on the stack and that it will not try to read or write at random memory locations.

For the memory issue, we simply avoid modifying the program before an instruction manipulating memory or one of the `POP` or `PUSH1` added in the program construction phase. For the second issue, we make sure to always split the two programs at positions where they have the same number of elements on the stack.

We show how we perform the cross-over in Algorithm 2. In the CreateStackSizeIndex function, we create a mapping from a stack size to a set of program counters where the stack has this size. In the CrossOver function, we first create this mapping for both programs and randomly choose a stack size to split the program. We then randomly choose a location from each program with the selected stack size. Note that here, *sample* assigns the same probability to all elements in the set. Finally, we split each program in two at the chosen position, and cross the programs together.

Mutation. We use a straightforward mutation operator. We randomly choose an instruction I in the program, where I is not one of the `POP` or `PUSH1` instructions added to handle memory issues previously discussed. We generate a set M_I of replacement candidate instructions defined as follows.

$$M_I = \{I' \mid I' \in \mathbb{I}_{a(I)} \wedge r(I') = r(I)\} \tag{3.5}$$

In other words, the replacement must require at most the same number of elements on the stack and put back the same number as the replaced instruction. Then, we replace the instruction I with I', which we randomly sample from M_I. If I had `POP` or `PUSH1` associated with it to control memory, we remove them from the program. Finally, if I' uses memory, we add the necessary instructions before it.

Algorithm 2 Cross-over function

```
function CreateStackSizeMapping(P)
    S ← empty mapping
    pc ← 0
    s ← 0
    for I in P do
        if s ∉ S then
            S[s] ← {}
        end if
        S[s] ← S[s] ∪ {pc}
        s ← s + (r(I) − a(I))
        pc ← pc + 1
    end for
    return S
end function
function CrossOver(P1, P2)
    S1 ← CreateStackSizeMapping(P1)
    S2 ← CreateStackSizeMapping(P2)
    S ← S1 ∩ S2                                ▷ Intersection on keys
    s ← sample(S)
    i1 ← sample(S1[s])
    i2 ← sample(S2[s])
    P11, P12 ← split_at(P1, i1)
    P21, P22 ← split_at(P2, i2)
    P1' ← P11 · P22                            ▷ Concatenate
    P2' ← P21 · P12
    return P1', P2'
end function
```

3.4.2 Effectiveness of Attacks with Synthetic Contracts

We want to measure the effectiveness of our approach to produce Resource Exhaustion Attacks. To do so, we want to generate contracts and benchmark them while mimicking the behaviour of a regular full validating node as much as possible. To do so, we execute all the programs produced within every generation of our genetic algorithm, as if they were part of a single block. We use the following steps to run our genetic algorithm.

1. Clear the page cache;
2. Warm up caches by generating and executing randomly-generated contracts
3. Generate the initial set of programs;

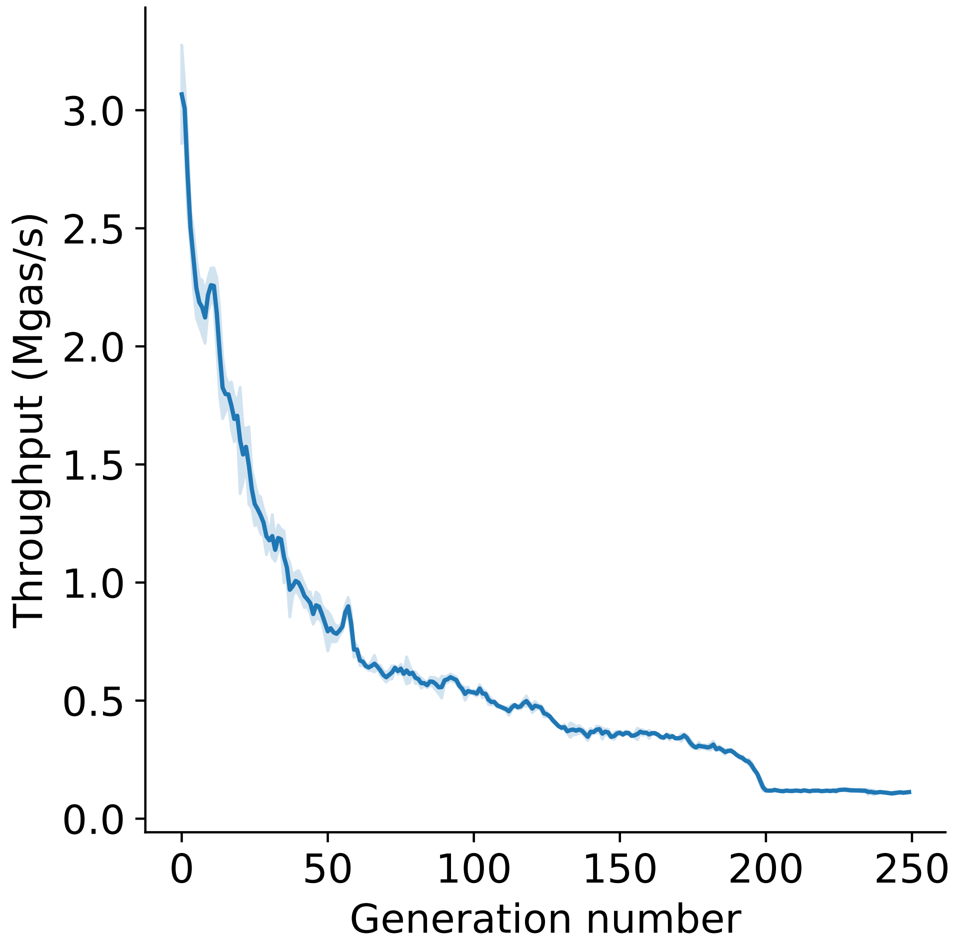


Figure 3.5: Evolution of the average contract throughput as a function of the number of generations.

4. Run the genetic algorithm for n generation.

An important point here is that when running the genetic algorithm, we only want to execute each program once, otherwise every IO access will already be cached and it will invalidate the results, as this is not what would happen when a regular validating node executes contracts. However, we of course do want to execute the measurements multiple times to be able to measure the execution time standard deviation. To work around these two requirements, we save all the programs generated while we run the experiment. Once the experiment has finished, we re-run all the programs in the same order. We combine these results to compute the mean and standard deviation of the execution time.

We note that generating a new generation takes on average less than 1 second but the time-

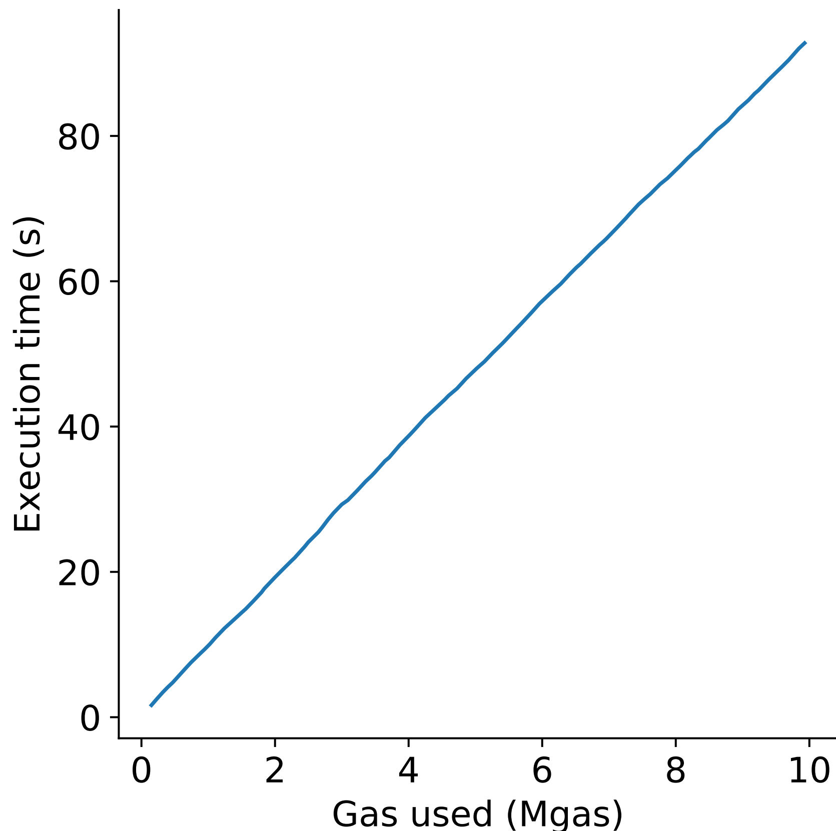


Figure 3.6: Execution time as a function of the total amount of gas used by contracts within a block.

consuming part of our algorithm is to compute the throughput of the generated programs. Indeed, we need to wait for the EVM to run the program, which can, as we show in this section, take more than 90 seconds for a single generation. Furthermore, parallelising this task could bias our measurements, which forces the algorithm to perform the evaluation serially.

Generated bytecode. Before discussing the results further, we show a small snippet of bytecode generated by our genetic algorithm in Code Listing 3.2. We highlight the instructions which involve IO operations in bold and show the instructions whose sole purpose is to keep the stack consistent in a smaller font. We can see that there is a large number of IO-related instructions, in particular, `BLOCKHASH` and `BALANCE` show up multiple times. Although the fee of `BALANCE` has been revised from 20 to 400 in EIP-150, this suggests that the instruction is still under-priced. In the snippet, we also see that the stack is cleared and replaced with small values before calling `CALLDATACOPY`. This corresponds to the *prepare_stack* function described in the program construction section: to avoid `CALLDATACOPY` reading very far away in memory,

Code Listing 3.2: Bytecode snippet generated by our genetic algorithm. Instructions in bold involve some sort of IO operations.

```
PUSH9 0x57c2b11309b96b4c59
BLOCKHASH
SLOAD
CALLDATALOAD
PUSH7 0xa29edb24d7b9a7
BALANCE
MSTORE8
PUSH11 0x518f6932049997fc5aab6d
PUSH14 0x50b9195e8f5acd66ad1b1b85a753
BALANCE
POP      ; prepare call to CALLDATACOPY
POP
POP
PUSH1 0xf7
PUSH1 0xf7
PUSH1 0xf7
CALLDATACOPY
PUSH7 0x421437ba67fe0e
ADDRESS
BLOCKHASH
```

which would make the program run out of gas, the arguments are replaced with small values. We note that our algorithm can generate programs of arbitrary length but in our experiments, we set it to create programs of around 4,000 instructions which consume between 100,000 and 150,000 gas.

Generating low-throughput contracts. We show how the throughput of the lowest-performing contract evolved with the number of generations in Figure 3.5. The line represents the mean of the measurements and the band represents the standard deviation of the measurements. The measurements are run 3 times. Except for one point in the first measurements, overall the standard deviation remains relatively low.

We can see that during the first generations, the throughput is around 1.25M gas per second, which is already fairly low given that the average throughput for a transaction on the same machine is around 20M gas per second. This shows that our initialisation is effective. The throughput decreases very quickly in the first few generations, and then steadily decreases down to around 110K gas per second, which is more than 180 slower than the average transaction.

Table 3.5: Evaluation of different clients when executing 10M gas worth of malicious transactions. What is presented is the mean across three measurements ± standard deviation. All the measurements are performed on our GCP except the "metal" which is done on our bare-metal server.

Client	**Throughput** Gas/s	**Time** second	**IO load** MB/s
Aleth	$107,349 \pm 606.6$	93.6 ± 0.53	9.12 ± 4.70
Parity	$210,746 \pm 7,672$	47.1 ± 1.61	10.0 ± 1.36
Parity (metal)	$542,702 \pm 9,487$	18.2 ± 0.23	17.2 ± 1.97
Geth	$131,053 \pm 4,207$	75.6 ± 2.42	6.57 ± 4.13
Geth (fixed)	$3,021,038 \pm 4.67e5$	3.33 ± 0.56	0.72 ± 0.11

After about 200 generations, the throughput plateaus.

Exploring the minimum. The minimum in our experiments is attained at generation 243. At this point, the block uses in total approximately 9.9M gas and takes around 93 seconds to execute, or throughput of about 107,000 gas per second. We show in Figure 3.6 how the execution time increases with the amount of gas consumed within the block. It is important to note that the execution time increases perfectly linearly with the gas used, which means that all transactions in the block have almost the same throughput. This implies that an attacker could easily tune the time he wants to delay the nodes depending on his budget. If a block full of malicious transactions were to be processed, given that an Ethereum block is produced roughly every 13 seconds, 7 new blocks would have been created by the time the node finishes validating the malicious one.

3.4.3 Evaluation on Other Ethereum Clients

We used aleth [Eth] to run our genetic algorithm and find low-throughput contracts. In this section, we show that the contracts crafted using our algorithm are also effective on the two most popular Ethereum clients: geth (v1.9.6) [Aut19] and Parity Ethereum (v2.5.9) [Par20]. We also show that the fix released in geth following discussions with the development team successfully resolves the issue. Our attack is mainly efficient on less powerful hardware, we include the measurements of Parity on a more powerful bare-metal machine with 4 cores (8 threads) at 2.7GHZ, 32GB of RAM and an SSD with 540MB/s throughput. To benchmark

the clients, we use the following procedure and repeat the measurements three times for each client.

1. Synchronise the client to test;
2. Start the client in a private network so that it does not execute anything else but our contracts;
3. Execute transactions on the client using the `eth_call` RPC endpoint;
   (a) Send transactions to warm up the client
   (b) Send enough malicious transactions to consume 10M gas
4. Measure the gas, time, IO, CPU and memory used during the execution of the malicious transactions.

We report our results in Table 3.5. Although we measured CPU, memory, and IO usage, most of the used time was related to IO operations and there was no significant increase in either CPU or memory usage during the attack. Therefore, we only report the IO measurements collected during the attack. We express the IO load in terms of MB/s, which we collect using Linux's `iotop` utility.

Before geth's fix, geth takes more than 75 seconds to execute 10M gas worth of malicious transactions. Parity Ethereum is the least vulnerable to our attack, but still takes on average about 47 seconds. Parity has on average a higher, but more constant IO load than geth. Large increases in the IO load tend to increase the IO wait time, which could explain why geth is vastly slower than Parity. Aleth is the slowest of the three clients. There could be two reasons for this: first, our algorithm is optimised on aleth, which makes it more likely to slow it down, second, aleth is less actively developed than the other two clients and might lack some optimisations.

The results of running Parity on a more powerful bare-metal server show that even such machines are relatively vulnerable to our attack. Indeed, Parity, which was the fastest of the three clients, still took more than 18 seconds to execute the transactions. An important point to

notice is that the IO throughput is considerably higher on our bare-metal server, which is most likely one of the main reasons for the speedup.

Finally, we ran our attack on an improved version of geth, which the Ethereum developers pointed us to as a result of our interactions with them. This version includes several optimisations to improve the storage access speed. We can see that these improvements drastically reduced the IO load of the client. With these improvements, geth executes the transactions more than 20 times faster, making the execution speed fast enough to counter such an attack. Our interaction with geth developers shows the effectiveness of responsible vulnerability disclosure, as discussed in Section 3.4.5.

3.4.4 REA as a Form of DoS

Malicious contracts crafted using our algorithm could easily be used to perform a DoS attacks on Ethereum. In this section, we will describe the threat model of such an attack, including the implications and feasibility of the attack.

Attack implications. As described in Section 3.2, there have already been several instances of DoS attacks against Ethereum [Butc; Butb]. There are several consequences to such attacks. The most direct one is a high increase in the block production time [Eth20b], which in the worst cases more than doubled, significantly decreasing the total throughput of the network. This decrease comes not only from miners who might take more time to validate blocks but also from full nodes who are supposed to relay validated blocks and might take vastly longer to do so. A further indirect consequence of such attacks is the loss of trust in the system, which can lead to a decrease in the price of Ethereum, at least for a short period of time [Che+17a].

Probable attacker. Although instances of irrational behaviours that likely did not profit the attacker have been seen on Ethereum [Bre+17], we assume that the attacker is rational and wants to profit from such an attack. In this context, there are several ways in which such an attack could be performed.

First, this attack could be beneficial to miners. A miner could use these malicious transactions

to perform a sort of selfish-mining [ES14]. Indeed, if the miner chooses to include a small number of malicious transactions in the blocks he mines, the propagation time per block is likely to increase and give the miner a head start on mining the next block. Given that the block arrival time in Ethereum is around 12 seconds, gaining a couple of seconds can be financially interesting for a miner. Furthermore, the only cost for a miner would be the opportunity cost of not including other transactions in the block, as he could include malicious transactions with a gas price of 0.

Another potential motivation for an attack could be to try to reduce the price of the ETH token and the trust in the Ethereum ecosystem. An attacker wanting to make a one-shot profit could spend some amount of money into performing such a DoS attack while taking a short position on ETH, waiting for the price to go down. Other blockchains competing with Ethereum could also potentially use such tactics to try to discredit the reliability of Ethereum.

Attack feasibility. To reason about the feasibility of this attack, we assume that given the same gas price, a malicious transaction has the same chance of being included in a block as any other transaction. We use the time we obtained in our experiments with geth, as it is the Ethereum client with the largest usage share [eth20].

To find a reasonable gas price, we analyse the gas price of all transactions and blocks from October 1, 2019 (block 8,653,171) to December 31, 2019 (block 9,193,265). We find that the median value of the minimum gas price in a block is around 1.1Gwei and that the average gas price is around 10Gwei with a standard deviation of 11Gwei. These values are in agreement with some other source of gas computation [Com19a]. Finally, we find that at least 2 million gas worth of transactions are included for less than 3Gwei in about 90% of the blocks, and choose this value as the gas price to compute the cost of an attack.

Given that our malicious transactions have a throughput of about 131,000 gas per second, using a price of 3 Gwei, it would cost roughly $131,000 \times 3 \times 10^9 = 3.93 \times 10^{14}$Wei $= 3.93 \times 10^{-4}$ ETH $\approx$ 0.786 USD to execute code for one second. Consequently, it would cost slightly more than 10.218 USD per block to prevent nodes from running on commodity hardware to keep up with the network. This is a very cheap price to pay and could indeed motivate the probable attackers

discussed earlier to execute such an attack.

It is worth noting that if an attacker wanted to fill a larger portion of the block with malicious transactions, he would need to increase the gas price. Indeed, to fill half of the block with malicious transactions would require paying around 15Gwei, or 5 times more per gas, than to fill only 20% of the block. This would result in a cost of $10,000,000 \times 50\% \times 1.5 \times 10^{10} = 0.075$ ETH ≈ 150 USD. Nevertheless, this remains a very low price to pay for an attacker with financial incentives such as the ones described earlier.

Attack limitations. The current requirements to run a full node on the Ethereum main net are low enough for most commodity hardware to be able to keep up without any issues. The documentation mentions that a full node requires only 16GB of RAM, 2TB of SSD and a 4-core CPU [23c]. However, there is very little information about the typical hardware setup of full nodes. Therefore, it is very difficult to accurately evaluate how many nodes would be affected by such an attack. Nevertheless, the attack was judged severe enough by the Ethereum developers to react very promptly (within less than 24 hours for the first reply and within four days for them to test the fix) after our disclosure.

3.4.5 Responsible Disclosure

Given that the attack is very easy and cheap to execute, and worked on all major clients, we went through a responsible disclosure process. The Ethereum Foundation has an official bug bounty program [Eth20a] to report vulnerabilities. With the help of colleagues[4], we wrote a report summarising our main findings, including a minimal script to execute our attack, and sent it to the bug bounty program on October 3, 2019. We received a reply the next day from the Ethereum Security Lead, acknowledging the issue and pointing us to some ongoing efforts to improve some of the inefficiencies exploited by our attack. The Ethereum foundation team also let us know that they would coordinate with Parity developers. After discussions about the ongoing efforts and some other potential solutions, we have confirmed that our report had

[4] Matthias Egli and Hubert Ritzdorf from PwC Switzerland

been awarded a reward of 5,000 USD on November 17, 2019. Finally, the official announcement was published on the bounty program website on January 7, 2020.

3.5 Towards a Better Approach

Gas metering and pricing is a difficult but fundamental problem in Ethereum and other blockchains which use a similar approach to price contract execution. Mispricing of gas instructions has been a concern for a long time and improvements have been included in several hard forks [Buta; Tan]. However, there remain issues in the current Ethereum pricing model, allowing attacks such as the one we presented in the previous sections. In this section, we will discuss short-term fixes which can be used to prevent DoS such as the one presented in this chapter, and then briefly present longer-term potential solutions which are still being actively researched.

The main attack vector presented in this chapter comes from the low speed of searching for an account which is not currently cached. One of the main issues is that the state of Ethereum gets larger with time. This means that operations accessing the state get more expensive with time in terms of resource usage.

Short-term fixes. Short-term fixes for slow IO-related issues can be categorised in the two following classes: increase in the gas cost of IO instructions, as seen in EIP-150 [Buta] and EIP-2200 [Tan], and improvements in the speed of Ethereum clients.

Increasing the cost of IO instructions improves the fairness of the gas costs yet is often not sufficient to protect against DoS attacks, albeit it does increase their cost. The attack we present in this chapter uses mainly instructions whose prices have increased in EIP-150 or EIP-2200 but remain relatively cheap to execute.

Improvements involve adding more layers of caches to reduce the number of IO accesses, which are typically the bottleneck. However, this requires keeping more data in memory and therefore creates a trade-off between memory consumption and execution speed. Regarding account

lookup, two cases must be considered: when the looked-up account exists and when it does not. Naively caching all the accounts could allow an attacker to easily evict existing accounts from the cache and is therefore dangerous. To check whether a particular account exists, a Bloom filter can be utilised as a first test. This eliminates the need for most of the IO accesses in case the queried address does not exist while keeping a relatively low memory footprint [Mit02]. The next case which needs to be handled is the fast lookup of existing accounts. The current attempt to do this keeps an on-disk dynamic snapshot of the accounts state [Szi19], which allows performing an on-disk look-up of an account in $\mathcal{O}(1)$, at the cost of increasing the storage usage of the node. This indeed solves the bottleneck of accessing account data but is very specific to this particular issue.

Long-term fixes. Long-term fixes are likely to only arrive in Ethereum 2.0, as most of them will require major and breaking changes. There have been several solutions discussed by the community and other researchers, which can mostly be categorised as either a) changing the gas pricing mechanism or b) changing the way clients store data.

Current proposals to change the gas mechanism involve making the pricing more dynamic than it is currently. Chen et al. [Che+17a] propose a mechanism where contracts using a single instruction in excess would be penalised. The threshold is set using historical data in order to penalise only contracts which diverge too much from regular usage. Although the approach has some advantages over the current pricing mechanism, it is unclear how well it would be able to prevent attacks taking this mechanism into account.

A promising and actively researched approach is the use of stateless clients and stateless validation. The key idea is that instead of forcing clients to store the whole state, the entity emitting transactions must send the transaction, the data needed by the transactions, and proof that this data is correct. The proof can be fairly trivially constructed as a Merkle proof, as the block headers hold a hash of the root of the state and the state can be represented as a Merkle tree. This allows such clients to verify all transactions without accessing IO resources at all, making execution and storage much cheaper, at the cost of an increased complexity when creating transactions and higher bandwidth usage.

Another active area of research which should help make things better in this direction is sharding [Al-+17]. Although sharding does not address the fundamental issue of gas pricing in the presence of IO operations, it does help to keep the state of the nodes smaller, as different shards will be responsible for storing the state of different parts of the network.

3.6 Related Work

There has been a great deal of attention focused on the correctness of smart contracts on blockchains, especially, the Ethereum blockchain. Some of the vulnerability types have to do with gas consumption, but not all. There has been relatively little attention given to the organisation of metering for blockchain systems. We will first highlight the work that focuses on metering at the smart contract level and then, we will present research focusing on metering at the virtual machine level — EVM in the case of Ethereum.

3.6.1 Gas Usage and Metering

Yang et al. [Yan+19] have recently empirically analysed the resource usage and gas usage of the EVM instructions. They provide an in-depth analysis of the time taken for each instruction both on commodity and professional hardware. Although our work was performed independently, the results we present in Section 3.3 seem to concur mostly with their findings.

Other related themes have also been covered in the literature. One of the large themes is the optimisation of gas usage for smart contracts. Another one is estimating, preferably statically, the gas consumption of smart contracts.

Gas Usage Optimisation

Gasper [Che+17b] is one of the first papers which has focused on finding gas-related anti-patterns for smart contracts. It identifies 7 gas-costly patterns, such as dead code or expensive

operations in loops, which could potentially be costly to the contract developer in terms of gas. Gasper builds a control flow graph from the EVM bytecode and uses symbolic execution backed by an SMT solver to explore the different paths that might be taken.

MadMax [Gre+18] is a static analysis tool to find gas-focused vulnerabilities. Its main difference with Gasper from a functionality point of view is that MadMax tries to find patterns which could cause out-of-gas exceptions and potentially lock the contract funds, rather than gas-intensive patterns. For example, it can detect loops iterating on an unbounded number of elements, such as the number of users, and which would therefore always run out of gas after a certain number of users. MadMax decompiles EVM contracts and encodes properties about them into Datalog to check for different patterns. It is performant enough to analyse all the contracts of the Ethereum blockchain in only 10 hours.

Gas Estimation

Marescotti et al. [Mar+18] propose two algorithms to compute the upper-bound gas consumption of smart contracts. It introduces a "gas consumption path" to encode the gas consumption of a program in its program path. It uses an SMT solver to find an environment resulting in a given path and computes its gas consumption. However, this work is not implemented with actual EVM code and is therefore not evaluated on real-world contracts.

Gastap [Alb+18] is a static analysis tool which allows to compute sound upper bounds for smart contracts. This ensures that if the gas limit given to the contract is higher than the computed upper bound, the contract is assured to terminate without out-of-gas exception. It transforms the EVM bytecode and models it in terms of equations representing the gas consumption of each instructions. It then solves these equations using the equation solver PUBS [Alb+08]. Gastap can compute a gas upper bound on almost all real-world contracts it is evaluated on.

3.6.2 Virtual Machines and Metering

Zheng et al. [Zhe+17] propose a performance analysis of several blockchain systems which leverage smart contracts. Although the analysis goes beyond smart contracts metering, with metrics such as network-related performance, it includes an analysis of smart contracts metering at the virtual machine level. Notably, it shows that some instructions, such as `DIV` and `SDIV`, consume the same amount of gas while their consumption of CPU resources is vastly different.

Chen et al. [Che+17a] propose an alternative gas cost mechanism for Ethereum. The gas cost mechanism is not meant to replace completely the current one, but rather to extend it in order to prevent DoS attacks caused by under-priced EVM instructions. The authors analyse the average number of execution of a single instruction in a contract, and model a gas cost mechanism to punish contracts which excessively execute a particular instruction. This gas mechanism allows normal contracts to almost not be affected by the price changes while mitigating spam attacks which have been seen on the Ethereum blockchain [Butc].

3.6.3 Follow-up work

The work we presented in this chapter has inspired some further research in the area of smart contract metering. We will present some of the work that has been done in relation to it.

In a study from Khan et al. [KSA21], 5,000 Solidity-based smart contract transactions were analysed to identify patterns that affect gas consumption. The researchers performed statistical analyses, including correlation and regression, to investigate the relationship between Solidity parameters, opcodes, and gas usage. They pinpointed factors that contribute to increases or decreases in gas consumption, with their regression analysis revealing that 87.8% of the variability in gas consumption is attributed to the examined parameters.

The research conducted by Li et al. [LWT21] uncovers vulnerabilities in transaction handling across all known Ethereum clients, such as Geth. They exploit these flaws to design a series of low-cost denial-of-service attacks called DETER, which can disable a remote Ethereum node's

txpool and disrupt critical downstream services, including mining, transaction propagation, and gas stations. DETER attacks are characterized by minimal or zero Ether cost and can potentially cause widespread disruption to the Ethereum network by targeting centralized services like mining pools and transaction relay services.

3.7 Conclusion

In this work, we presented a new DoS attack on Ethereum by exploiting the metering mechanism. We first re-executed the Ethereum blockchain for 2.5 months and showed some significant inconsistencies in the pricing of the EVM instructions. We further explored various other design weaknesses, such as gas costs for arithmetic EVM instructions and cache dependencies on the execution time. Additionally, we demonstrated that there is very little correlation between gas and resources such as CPU and memory. We found that the main reason for this is that the gas price is dominated by the amount of *storage* used.

Based on our observations, we presented a new attack called *Resource Exhaustion Attack* which systematically exploits these imperfections to generate low-throughput contracts. Our genetic algorithm can generate programs which exhibit a throughput of around 1.25M gas per second after a single generation. A minimum in our experiments is attained at generation 243 with the block using around 9.9M gas and taking around 93 seconds. We showed that we can generate contracts with throughput as low as 107,000 gas per second, or on average more than 100 times slower than typical contracts, and that all major Ethereum clients are vulnerable. We argued that several attackers such as speculators, Ethereum competitors or even miners could have financial incentives to perform such an attack. Finally, we discussed short-term and potential long-term fixes for gas mispricing. Our attack went through a responsible disclosure process and has been awarded a bug bounty reward of 5,000 USD by the Ethereum foundation.

Chapter 4

Transactional Level Security

Scalability has been a bottleneck for major blockchains such as Bitcoin and Ethereum. As we have seen in the previous chapter, the execution layer can be a major bottleneck for scalability and even potentially lead to DoS attacks. Some newer blockchains have improved scalability and allowed for much higher transactional throughput. However, there has been little effort to understand how their transactional throughput is being used. In this chapter, we examine recent network traffic of three major high-scalability blockchains—EOSIO, Tezos and XRP Ledger (XRPL)—over a period of seven months. Our analysis reveals that only a small fraction of the transactions are used for value transfer purposes. In particular, 96% of the transactions on EOSIO were triggered by the airdrop of a currently valueless token; on Tezos, 76% of throughput was used for maintaining consensus; and over 94% of transactions on XRPL carried no economic value. We also identify a persisting airdrop on EOSIO as a DoS attack and detect a two-month-long spam attack on XRPL. The chapter explores the different designs of the three blockchains and sheds light on how they could shape user behaviour.

4.1 Introduction

As the most widely-used cryptocurrency and the first application of a blockchain system, Bitcoin has been frequently criticized for its slow transactional throughput, making it hard to adopt as a payment method. Indeed, Bitcoin is only able to process around 10 transactions per second, significantly slower than the throughput offered by centralized payment providers such as Visa, which can process over 65,000 transactions per second [Vis20]. Many blockchains have since been designed and developed in order to improve scalability, the most valued of these in terms of market capitalization [Coi20] being EOSIO [blo18], Tezos [Goo14], and XRP Ledger (XRPL) [XRP19b].

Although many of these systems have existed for several years already, to the best of our knowledge, no in-depth evaluation of the actual usage of their transactional throughput has yet been performed, and it is unclear up to what point these blockchains have managed to generate economic activity. The knowledge of both the quantity and the quality of the realized throughput is crucial for the improvement of blockchain design, and ultimately a better utilization of blockchains. In this chapter, we analyse transactions of the three blockchains listed above and seek to find out:

RQ1 To what extent has the alleged throughput capacity been achieved in those three blockchains?

RQ2 Can we classify transactions by analysing their metadata and patterns?

RQ3 Who are the most active transaction initiators and what is the nature of the transaction they conducted?

RQ4 Can we reliably identify DoS and transactional spam attacks by analysing transaction patterns?

Contributions. We contribute to the body of literature on blockchain in the following ways:

1. We perform the first large-scale detailed analysis of transaction histories of three of the most widely-used high-throughput blockchains: EOSIO, Tezos, and XRPL.

2. We classify on-chain transactions and measure each category's respective share of the total throughput, in terms of the number of transactions and their economic volume.

3. We establish a measurement framework for assessing the quality of transactional throughput in blockchain systems.

4. We expose spamming behaviours that have inflated throughput statistics and caused network congestion.

5. We highlight the large gap between the alleged throughput capacity and the transactions with some economic value being performed on those three blockchains.

Our analysis serves as the first step towards a better understanding of the nature of user activities on high-scalability blockchains. On-chain monitoring tools can be built based on our framework to detect undesired or even malicious behaviour.

Summary of our findings. Despite the advertised high throughput and the seemingly commensurate transaction volume, a large portion of on-chain traffic, including payment-related transactions, does not result in actual value transfer. The nature and purpose of non-payment-related activities vary significantly across blockchains.

Specifically, we observe that the current throughput is only 34 TPS (transactions per second) for EOSIO, 0.43 TPS for Tezos and 15 TPS for XRPL. We show that 96% of the throughput on EOSIO was used for the airdrop of a valueless token, 76% of transactions on the Tezos blockchain were used to maintain consensus, and that over 94% of transactions on XRPL carried zero monetary value.

4.2 Background

In this section, we describe the structure of the three blockchain systems that we evaluate, highlighting their various design aspects. We then provide some typical use cases for each of these.

4.2.1 Consensus Mechanisms

In response to the scalability issues related to PoW, many blockchains have developed other mechanisms to ensure consensus, which allows higher rates of block creation.

Delegated Proof-of-Stake (DPoS) in EOSIO. EOSIO uses the Delegated Proof-of-Stake (DPoS) protocol which was first introduced in Bitshares [Bit18].

Users of EOSIO, stake `EOS` tokens to their favoured block producers (BPs) and can choose to remove their stake at any time. The 21 BPs with the highest stake are allowed to produce blocks whereas the rest are put on standby. Blocks are produced in rounds of 126 (6×21). The order of block production is scheduled prior to each round and must be agreed upon by at least 15 block producers [blo18].

Liquid Proof-of-Stake (LPoS) in Tezos. For its consensus mechanism, Tezos employs another variant of Delegated Proof-of-Stake: the Liquid Proof-of-Stake (LPoS) [Tez18]. Tezos' LPoS differs from EOSIO's DPoS in that with the former, the number of consensus participants—or "delegates"—changes dynamically [Tez18; Goo14]. This is because any node with a minimum amount of staked assets, arbitrarily defined to be 8,000 `XTZ` (about 8,000 USD at the time of writing [Coi20]), is allowed to become a delegate, who then has the chance to be selected as either a "baker" or an "endorser". Each block is produced ("baked") by one randomly selected baker, and verified ("endorsed") by 32 randomly selected endorsers [Tez18]. The endorsements are included in the following block.

XRP Ledger Consensus Protocol (XRP LCP) in XRPL. XRPL is a distributed payment network created by *Ripple Labs Inc.* in 2012 that uses the XRP ledger consensus proto-

col [CM18]. Each user sets up its own unique node list of validators (UNL) that it will listen to during the consensus process. The validators determine which transactions are to be added to the ledger. Consensus is reached if at least 90% of the validators in each one's UNL overlap. If this condition is not met, the consensus is not assured to converge and forks can arise [CM18].

Table 4.1: Distribution of action types per blockchain.

	EOSIO			**Tezos**			**XRPL**		
Category	Action name	#	%	Operation kind	#	%	Transaction type	#	%
P2P transactions	**Transfer**	8,479,573,653	**96.2**	Transaction	1,941,230	21.4	**Payment**	100,328,458	**36.9**
							EscrowFinish	677	0.0
Account actions	newaccount	289,680	0.0	Reveal	113,915	0.0	TrustSet	3,339,620	1.2
	bidname	244,248	0.0	Origination	3,159	1.3	AccountSet	150,401	0.1
	deposit	243,881	0.0	Activate	2,659	0.0	SignerListSet	13,707	0.0
	linkauth	148693	0.0				SetRegularKey	734	0.0
	updateauth	136,926	0.0				DepositPreauth	3	0.0
Other actions	delegatebw	684,449	0.0	**Endorsement**	6,957,612	**76.6**	**OfferCreate**	160,451,595	**59.1**
	undelegatebw	461,320	0.0	Delegation	56,336	0.6	OfferCancel	7,259,908	2.7
	buyrambytes	353,695	0.0	Reveal nonce	9,409	0.1	EscrowCreate	1,393	0.0
	rentcpu	187,878	0.0	Ballot	514	0.0	EscrowCancel	84	0.0
	voteproducer	137,713	0.0	Proposals	90	0.0	PaymentChannelClaim	172	0.0
	buyram	89,971	0.0	Double baking evidence	4	0.0	PaymentChannelCreate	33	0.0
	Others	332,799,590	3.8				EnableAmendment	12	0.0
Total		8,815,351,697	100.0		9,084,928	100.0		271,546,797	100.0

4.2.2 Account and Transaction Types

In this section, we describe the types of transactions that exist on the three blockchains.

EOSIO

EOSIO differentiates between system and regular accounts. The former are built-in accounts created when the blockchain was first instantiated, and are managed by currently active BPs, while the latter can be created by anyone. System accounts are further divided into privileged and unprivileged accounts. Privileged accounts, including `eosio`, `eosio.msig`, and `eosio.wrap`, can bypass authorization checks when executing a transaction [EOS19; Kau19] (see Section 4.2.1).

EOSIO system contracts, defined in `eosio.contracts` [EOS20c], are held by system accounts. One of the most commonly used system contracts is `eosio.token`, which is designed for creating

and transferring user-defined tokens [EOS19]. Regular accounts can freely design and deploy smart contracts.

Each smart contract on EOSIO has a set of actions. Actions included in non-system contracts are entirely user-defined, and users have a high degree of flexibility in terms of structuring and naming the actions. This makes the analysis of actions challenging, as it requires understanding their true functionality on a case-by-case basis. While many actions have a candid name that gives away their functionality (e.g. `payout` from contract `betdicegroup`), some are less expressive (e.g. `m` from user `pptqipaelyog`).

In Table 4.1, we show different types of existing actions. Since actions from non-system contracts have arbitrary designs, we only examine actions that belong to system accounts for the moment, as these are already known and are easier to classify. We make one exception to this and include the actions of *token* contracts, as they have a standardized interface [Lab19]. Overall, we can see that token transfers account alone for more than 96% of the transactions. The rest of the transactions are mostly user-defined and appear under "Others" in the table, while actions defined in system contracts only account for a very small percentage of the entire traffic volume.

Tezos

Tezos has two types of accounts: implicit and originated. Implicit accounts are similar to the type of accounts found in Ethereum, generated from a public-private key pair [Woo19]. These accounts can produce—or "bake"—blocks and receive stakes, but cannot be used as smart contracts. Bakers' accounts must be implicit, to be able to produce blocks. Originated accounts are created and managed by implicit accounts, but do not have their private key [Nom18d]. They can function as smart contracts and can delegate voting rights to bakers' implicit accounts [Nom18a].

"Transactions" on Tezos are termed "operations". Operations can be roughly classified into three types: consensus-related, governance-related and manager operations [Ami19].

Consensus-related operations, as the name indicates, ensure that all participating nodes agree on one specific version of data to be recorded on the blockchain. Governance-related operations are used to propose and select a new set of rules for the blockchain. However, these events are very rare and only involve bakers, which is why these operations only represent a low percentage of the total number of transactions. Operations mainly consist of delegations and peer-to-peer payment transactions. As shown in Table 4.1, `endorsement` operations account for a vast majority, 76%, of total operations. Endorsements are performed by bakers, and a block needs a minimum of 32 endorsements for it to be accepted [Nom18b].

XRPL

XRPL also uses an account-based system to keep track of asset holdings. Accounts are identified by addresses derived from a public and private key pair. There are a handful of "special addresses" that are not derived from a key pair. Those addresses either serve special purposes (e.g. acting as the `XRP` issuer) or exist purely for legacy reasons. Since a secret key is required to sign transactions, funds sent to any of these special addresses cannot be transferred out and are hence permanently lost [XRP19a].

XRPL has a large number of predefined transaction types. We show part of them in Table 4.1. The most common transaction types are `OfferCreate`, which is used to create a new order in a decentralized exchange (DEX) on the ledger, and `Payment`, which is used to transfer assets. There are also other types of transactions such as `OfferCancel` used to cancel a created order or `TrustSet` which is used to establish a "trustline" [XRP19b] with another account.

4.2.3 Expected Use Cases

In this section, we describe the primary intended use cases of the three blockchains and provide a rationale for the way they are being used, to better understand the dynamics of actual transactions evaluated in Section 4.4.

EOSIO. EOSIO was designed with the goal of high throughput and has a particularity compared to many other blockchains: there are no direct transaction fees. Resources such as CPU, RAM and bandwidth are rented beforehand, and there is no fixed or variable fee per transaction [blo18]. This makes it a very attractive platform for building decentralized applications with a potentially high number of micro-payments. Many games, especially those with a gambling nature, have been developed using EOSIO as a payment platform. EOSIO is also used for decentralized exchanges, where the absence of fees and the high throughput allow placing orders on-chain, unlike many decentralized exchanges on other platforms where only the settlement is performed on-chain [WB17].

Tezos. Tezos was one of the first blockchains to adopt on-chain governance. This means that participants can vote to dynamically amend the rules of the consensus. A major advantage of this approach is that the blockchain can keep running without the need for hard forks, as often observed for other blockchains [17a; 17b]. Another characteristic of Tezos is the use of a strongly typed programming language with well-defined semantics [Nom18c] for its smart contracts, which makes it easier to provide these for correctness. These properties make Tezos a very attractive blockchain for financial applications, such as the tokenization of assets [BD19].

XRPL. Similar to EOSIO, XRPL supports the issuance, circulation, and exchange of customized tokens. However, in contrast to EOSIO, XRPL uses the IOU ("I owe you") mechanism for payments. Specifically, any account on XRPL can issue an IOU with an arbitrary ticker — be it `USD` or `BTC`. Thus, if Alice pays Bob 10 `BTC` on XRPL, she is effectively sending an IOU of 10 `BTC`, which means "I (Alice) owe you (Bob) 10 `BTC`". Whether the `BTC` represents the market value of Bitcoin depends on Alice's ability to redeem her "debt" [XRP20c]. This feature contributes to the high throughput on XRPL, as the speed to transfer a specific currency is no more constrained by its original blockchain-related limitations: For example, the transfer of `BTC` on XRPL is not limited by the block production interval of the actual Bitcoin blockchain (typically 10 minutes to an hour to fully commit a block), and the transfer of `USD` is not limited to the speed of the automated clearing house (ACH) (around two days [Lov+13]).

Table 4.2: Characterizing the datasets for each blockchain. All measurements are performed from October 1, 2019 to April 30, 2020. Max throughput is the average TPS within a 6-hour interval that has the highest count of transactions. Storage size is computed with data saved as JSON Lines with one block per line and compressed using gzip level 6 of compression.

	Block index		**Count**	**Count**	**Storage**	**Throughput (TPS)**		
	from	to	**of blocks**	**of transactions**	**(.gzip, GB)**	Alleged	Max	Average
EOSIO	82,152,667	118,286,375	36,133,709	631,445,236	264	4,000 [KO19]	136	34
Tezos	630,709	932,530	301,822	7,890,133	1.4	40 [Arl18]	0.57	0.43
XRPL	50,399,027	55,152,991	4,753,965	271,546,797	130	65,000 [Rip20]	56	15

4.3 Methodology

In this section, we describe the methodology used to measure the transactional throughput of the selected blockchains.

4.3.1 Definitions

We first introduce important definitions used in the rest of this chapter.

Throughput-related definitions. When quantified, a throughput value is expressed in TPS (transactions per second).

Alleged Capacity The theoretical capacity that a blockchain claims to be able to achieve

Average Throughput Average throughput recorded on the network throughout the observation period

Maximum Throughput Maximum throughput recorded on the network during the observation period

Blockchain-related definitions. We unify the terms that we use across the systems analysed in this work. We sometimes diverge from the definition provided by a particular blockchain for terminological consistency.

Block Blocks are named as such on EOSIO and Tezos, while the equivalent on XRPL is termed a "ledger".

Transaction Transactions are named as such on EOSIO and XRPL but are called "operations" in Tezos.

Action Actions are entities included as part of the transaction and describe what the transaction should do. EOSIO and Tezos can have multiple actions per transaction. A single transaction containing multiple actions is only counted towards throughput once. Actions are called as such in EOSIO and are the "contents" of an "operation" on Tezos. XRPL does not feature this concept and each XRPL transaction can be thought of as a single action.

4.3.2 Measurement Framework

We implement an extensible and reusable measurement framework to facilitate future transaction analysis-related research. Our framework currently supports the three blockchains analysed in this work, Tezos, EOSIO and XRPL but can easily be extended to support other blockchains. The core of the software is implemented in Go and is designed to work well on a single machine with many cores. The framework frontend is provided as a cross-platform static binary command line tool.

While the framework is responsible for the heavy lifting and processing of gigabytes of data, we also provide a companion tool implemented in Python to generate plots and tables from the data generated by the framework.

Data fetching. The framework currently allows fetching data either using RPC over HTTP or websockets. Tezos and EOSIO both use the HTTP adapter to retrieve data while XRPL uses the websocket interface. The data is retrieved from publicly available archive nodes but the framework can be configured to use other nodes if necessary. The retrieved data is stored in a gzipped JSON Lines format where each line corresponds to a block. Blocks are stored in chunks of n blocks per file — where n can be configured — making parallel processing

straightforward. It took less than two days to fetch all the data presented in Table 4.2. We note that the average throughput values for XRPL and Tezos are, at the time of writing, still similar to what is presented in Table 4.2, while EOSIO's throughput has more than halved since then.

Code Listing 4.1: Configuration file for our measurement framework

```
{
  "Pattern": "/data/eos_blocks-*.jsonl.gz",
  "StartBlock": 82152667,
  "EndBlock": 118286375,
  "Processors": [{
      "Name": "TransactionsCount",
      "Type": "count-transactions"
    }, {
      "Name": "GroupedActionsOverTime",
      "Type": "group-actions-over-time",
      "Params": {
        "By": "receiver",
        "Duration": "6h"
      }
    }, {
      "Name": "ActionsByName",
      "Type": "group-actions",
      "Params": {
        "By": "name"
      }
    }
}
```

Data processing. The framework provides several processors which can mainly be used to

aggregate the data either over time or over certain properties such as the sender of a transaction. The framework is configured using a single JSON file, containing the configuration for the data to be processed as well as the specification of what type of statistics should be collected from the dataset. We show a sample configuration file in Code Listing 4.1. This configuration computes three statistics from block 82,152,667 to block 118,286,375, using the data contained in all the files matching `/data/eos_blocks-*.jsonl.gz`. The framework will compute the total number of transactions, the number of actions grouped using their receiver over a period of 6 hours, and finally the total number of actions grouped by their name. All the statistics described above can be used for all the blockchains but the framework also supports blockchain-specific statistics where needed. New statistics can easily be added to the framework by implementing a common interface.

Our framework was able to analyze the data and output all the statistics required for this chapter in less than 4 hours using a powerful 48-core machine.

Code Listing 4.2: Main interfaces of our measurement framework

```
type Blockchain interface {
    FetchData(filepath string,
              start, end uint64) error
    ParseBlock(rawLine []byte) (Block, error)
    EmptyBlock() Block
}

type Block interface {
    Number() uint64
    TransactionsCount() int
    Time() time.Time
    ListActions() []Action
}

type Action interface {
    Sender() string
    Receiver() string
    Name() string
}
```

Extending to other blockchains. The framework has been made as generic as possible to allow the integration of other blockchains when performing similar kinds of analysis. In

particular, the framework contains three main interfaces shown in Code Listing 4.2. The `FetchData` method can be implemented by reusing the HTTP or websocket adapters provided by the framework while the `Block` and `Action` interfaces typically involve defining the schema of the block or action of the blockchain implemented. In our implementation, adding a blockchain takes on average 105 new lines of Go code not including tests.

4.3.3 Data Collection

We collect historical data on the three blockchains from October 1, 2019 to April 30, 2020. We provide an overview of the characteristics of the data in Table 4.2. We note that the number of transactions is not the same as in Table 4.1 as here we count only a transaction once, while in the previous table, we counted all the actions included in a single transaction.

For all three of the blockchains, we first pinpoint the blocks which correspond to the start and end of our measurement period and use our framework to collect all the blocks included in this range. Each time, we use publicly available nodes or data providers to retrieve the necessary data.

EOSIO. EOSIO nodes provide an RPC API [EOS20d] which allows clients to retrieve the content of a single block, through the `get_block` endpoint [EOS20a]. EOSIO also has a list of block producers who usually provide a publicly accessible RPC endpoint. Out of 32 officially advertised endpoints, we shortlist 6 that have a generous rate limit with stable latency and throughput.

We collect data from block 82,152,667 to block 118,286,375, or a total of 36,133,709 blocks containing 631,445,236 transactions, representing more than 260GB of data.

Tezos. Similar to EOSIO, Tezos full nodes provide an RPC API and some bakers make it publicly available. We measure the latency and throughput of several nodes and select the one for which we obtained the best results [Ukr20]. We obtain 301,822 blocks containing 7,890,133 transactions, for a total size of approximately 1.4 GB of data.

XRPL. XRPL has both an RPC API and a websocket API with similar features. Although there are no official public endpoints for XRPL, a high-availability websocket endpoint is provided by the XRP community [Win20]. We use the `ledger` method of the websocket API to retrieve the data in the same way we did with EOSIO and Tezos.

In addition, we use the API provided by the ledger explorer XRP Scan [Tec20] to retrieve account information including username and parent account.[1] Since large XRP users such as exchanges often have multiple accounts, this account information can be used to identify and cluster accounts.

In total, we analyze 4,753,965 blocks covering seven months of data and containing a total of more than 150 million transactions. The total size of the compressed data is about 130 GB.

4.4 Data Analysis

In this section, we present summary statistics and high-level illustrations of the transactions contained in the datasets of the three different blockchains.

4.4.1 Transaction Overview

In Figure 4.1, we decompose the number of actions into different categories. XRPL and Tezos have well-defined action types, and we use the most commonly found ones to classify the throughput. EOSIO does not have pre-defined action types: contract creators can decide on arbitrary action types. To be able to classify the actions and understand where throughput on EOSIO is coming from, we manually label the top 100 contracts, representing more than 99% of the total throughput, by grouping them into different categories and assign one of the categories to each action.

EOSIO. Interestingly, there is a huge spike in the number of token actions from November 1, 2019, onward. We find that this is due to a new coin called `EIDOS` [enu19] giving away tokens.

[1] A parent account sends initial funds to activate a new account.

Table 4.3: EOSIO top applications as measured using the number of received transactions.

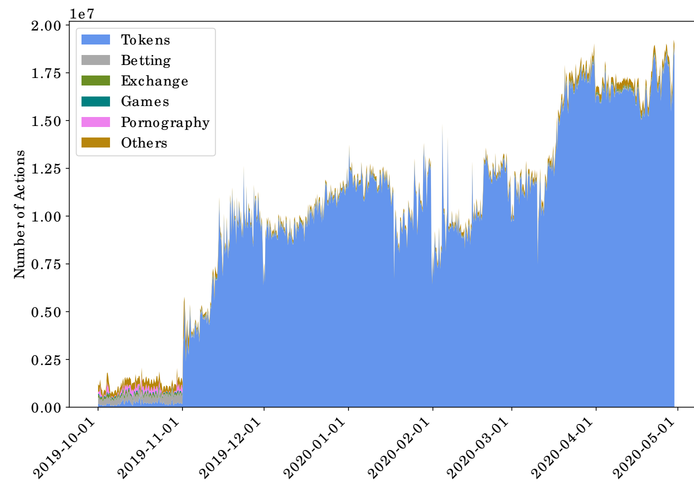
Receiver	Description	Tx Count	Actions Name	%
eosio.token	EOS token	8,430,707,864	transfer	100.0%
betdicetasks	Gambling game	32,804,674	removetask log	66.65% 15.71%
whaleextrust	Decentralized exchange	26,102,077	verifytrade2 verifytrade3 clearing	18.63% 17.52% 16.77%
pptqipaelyog	Unknown	24,109,437	m	93.00%
pornhashbaby	Pornography website	23,677,938	record	99.86%

We will describe this more extensively as a case study in Section 4.5.1. Before this peak, the number of actions on EOSIO was vastly dominated by games, in particular betting games.

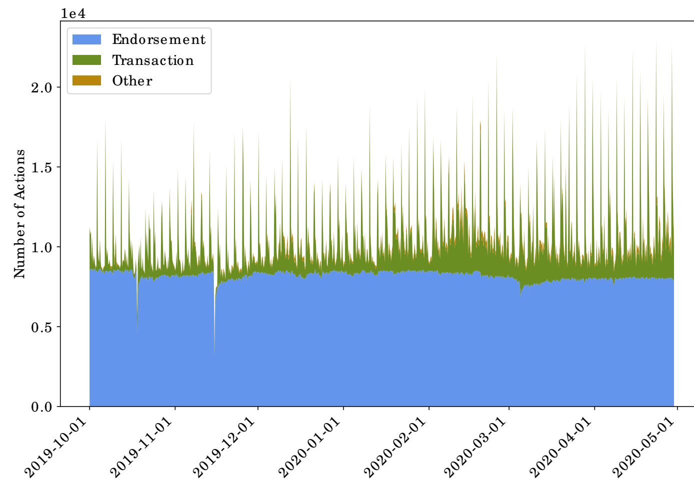
Tezos. Tezos has a high number of "endorsements"—76%, which are used as part of the consensus protocol, and only a small fraction of the throughput are actual actions. It is worth noting that the number of "endorsements" should be mostly constant regardless of the number of transactions and that if the number of transactions were to increase enough, the trend would reverse. We can also clearly see that Tezos has very regular spikes, with an interval of approximately two to three days each time. These appear to be payments from bakers to stakers [Dat20],[2] which can arguably be thought to be part of the consensus. We use the TzKT API[3] to find account names and find that roughly 53% of these "Transaction" actions are sent by bakers and 6% of them are sent by the Tezos faucet [Fou20]. Endorsements and actions sent by either bakers or the faucet sum up about 87% of the total number of actions.

XRPL. On XRPL, both successful and unsuccessful transactions are recorded. A successfully executed transaction executes the command—such as `Payment`, `OfferCreate`, `OfferCancel`—specified by its initiator, while the only consequence of an unsuccessful transaction is the deduction of transaction fees from the transaction initiator. Across the sample period, roughly one tenth of transactions are unsuccessful (Figure 4.1c), with the most frequently registered errors being `PATH_DRY` for `Payment` (insufficient liquidity, hence "dry", for specified payment path)

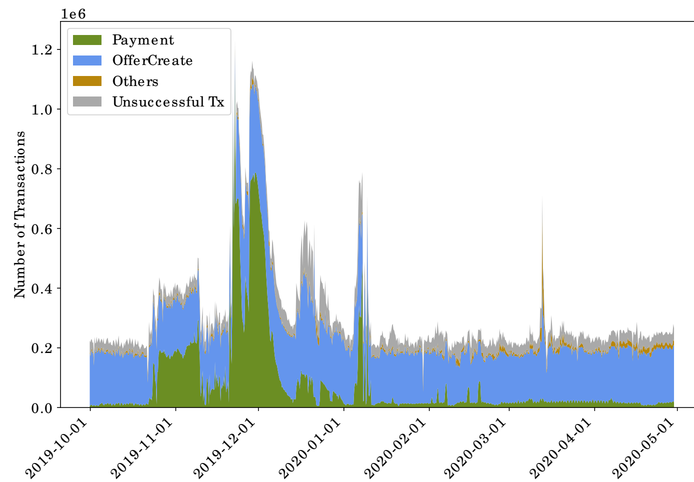
[2] https://twitter.com/CitezenB/status/1256147427905716224
[3] https://api.tzkt.io/



(a) EOSIO throughput over time



(b) Tezos throughput over time



(c) XRPL throughput over time

Figure 4.1: On-chain throughput over time, the y-axis represents transaction count per 6 hours.

and tecUNFUNDED_OFFER for OfferCreate (no funds for the currency promised to offer by the offer creator).

Successful transactions primarily consist of Payment (39.6%) and OfferCreate (59.1%) (Table 4.1). The number of OfferCreate transactions is generally constant across time, but the number of Payment has a very high variance, with some periods containing virtually no payments and others having significant spikes. In Section 4.5.3, we reveal why most transactions during these high-volume periods are economically meaningless.

Except for the two spam periods, we observe that OfferCreate is the most common transaction type. Nonetheless, OfferCreate transactions contribute little to the total volume on XRPL.[4] This is because an offer expires if not fulfilled (fully or partially) before the expiry time defined by the offer creator; it can also be cancelled by its creator or superseded by a new offer. In fact, 0.2% of OfferCreate transactions resulted in an actual token exchange deal during our observation period.

4.4.2 Transaction Patterns

To understand better what the major sources of traffic constitute, we analyze the top accounts on EOSIO, Tezos, and XRPL, and find various transaction patterns.

EOSIO. In Table 4.3, we show EOSIO accounts with the highest number of received actions. We can see that the eosio.token account, which is the account used to handle EOS token transfers, is by far the most used, and almost all calls to this account use the transfer action. Although EOS transfers are indeed a central part of the EOSIO ecosystem, more than 99.9% of the transfers shown are exclusively to and from this EIDOS account. The second account is a betting website where all the bets are performed transparently using EOSIO. However, around 80% of the actions—removetask and log—are bookkeeping, and the actual betting-related actions such as betrecord represent a very low percentage of the total number of

[4] On May 10, 2020, for example, Ripple reported that the 24-hour XRP ledger trade volume—enabled via OfferCreate transactions—only accounts for 1% of the total ledger volume, while the payment volume—enabled via Payment transactions—accounts for 99%.

Table 4.4: Tezos accounts with the highest number of sent transactions.

Sender	Sent count	Unique receivers	Avg. # of transactions per receiver
tz1VwmmesDxud2BJEyDKUTV5T5VEP8tGBKGD	106,477	23,649	4.50
tz1cNARmnRRrvZgspPr2rSTUWq5xtGTuKuHY	105,202	2,096	50.19
tz1Mzpyj3Ebut8oJ38uvzm9eaZQtSTryC3Kx	93,448	93,444	1.00
tz1SiPXX4MYGNJNDsRc7n8hkvUqFzg8xqF9m	57,841	19,382	2.98
tz1acsihTQWHEnxxNz7EEsBDLMTztoZQE9SW	42,683	1,436	29.72

actions. The third account is a decentralized exchange and is used to exchange different assets available on EOSIO. This exchange will be discussed in Section 4.5. We could not find information about the fourth account, but it is very actively sending EOS tokens to the EIDOS account. Finally, the last account was a pornography website which used EOSIO as a payment system. This account is still the fifth account with the highest number of received actions although the service was discontinued in November 2019 for financial reasons [Has19].

Tezos. As Tezos neither has account names nor actions in the transaction metadata, analysing the top receivers' accounts is less interesting, as it is very difficult to perform any type of attribution. However, we find interesting patterns from observing the top sending accounts. Most of the top senders in Tezos seem to follow a similar pattern: Sending a small number of transactions (between 5 and 50) to many different accounts. Another important thing to note is that all of these accounts are not contracts but regular accounts, which means that the transactions are automated by an off-chain program. After further investigation, we find that the top address is the Tezos Faucet [Fou20]. The other addresses appear to be bakers' payout addresses and the transactions are payouts to stakers [Lab20], corresponding to the peaks seen in Figure 4.1b. For completeness, we include the top senders and some statistics about them in Table 4.4.

XRPL. From October 1, 2019 to April 30, 2020, a total of 195 thousand accounts collectively conducted 272 million transactions, i.e. an average of 1.4 thousand transactions per account during the seven-month observation period.

The distribution of the number of transactions per account is highly skewed. Over one third (71

thousand) of the accounts have transacted only once during the entire observation period, whereas the 35 most active accounts are responsible for half of the total traffic. Table 4.5 lists of the top 10 accounts by the number of conducted transactions. With the exception of rKLpjpCoXgLQQYQyj13zgay73rsgmzNH13 and r96HghtYDxvpHNaru1xbCQPcsHZwqiaENE, all these accounts share suspiciously similar patterns:

1. more than 98% of their transactions are OfferCreate;
2. they are either descendants of an account from Huobi, a crypto exchange founded in China, or frequently transact with descendants from Huobi;
3. they have all transacted using CNY;
4. their payment transactions conspicuously use the same destination tag 104398, a field that—similar to a bank reference number—exchanges and gateways use to specify which client is the beneficiary of the payment [XRP20b].

The aforementioned similarities, in particular the last one, signal that those accounts are controlled by the same entity, presumably with a strong connection to Huobi. The frequent placement of offers might come from the massive client base of the entity.

Notably, the sixth most active account, r96HghtYDxvpHNaru1xbCQPcsHZwqiaENE, registered under the username chineseyuan only carried out *one* successful Payment transaction during the observation period, while the rest of the over four million transactions failed with a PATH_DRY error. Recall that failed transactions still occupy on-chain throughput. Therefore, it is evident that chineseyuan spammed the network.

4.4.3 Analysis Summary

Here, we highlight some of the observations about the data described above.

- Transactions on EOSIO can be roughly divided by the category of contracts they belong to. Before the arrival of the EIDOS token, approximately 50% of these are transactions

Table 4.5: XRPL accounts with the highest number of transactions.

Account	Type	Count	TotalCount	% of throughput
r4AZpDKVoBxVcYUJCWMcqZzyWsHTteC4ZE	OfferCreate Others Payment	21,790,612 291,687 132	22,082,431	8.13%
rQ3fNyLjbvcDaPNS4EAJY8aT9zR3uGk17c	OfferCreate Others Payment	21,716,850 140,088 46	21,856,984	8.05%
rh3VLyj1GbQjX7eA15BwUagEhSrPHmLkSR	OfferCreate Others Payment	21,510,597 31,295 37	21,541,929	7.93%
r4dgY6Mzob3NVq8CFYdEiPnXKboRScsXRu	OfferCreate Others Payment	21,474,131 29,841 163	21,504,135	7.92%
rKLpjpCoXgLQQYQyj13zgay73rsgmzNH13	Payment	4,493,754	4,493,754	1.65%
r96HghtYDxvpHNaru1xbCQPcsHZwqiaENE	Payment	4,488,127	4,488,127	1.65%
rBW8YPFaQ8WhHUy3WyKJG3mfnTGUkuw86q	OfferCreate Others	4,474,481 967	4,475,448	1.65%
rDzTZxa7NwD9vmNf5dvTbW4FQDNSRsfPv6	OfferCreate Others	4,472,749 1,043	4,473,792	1.65%
rV2XRbZtsGwvpRptf3WaNyfgnuBpt64ca	OfferCreate Others Payment	4,470,525 977 76	4,471,578	1.65%
rwchA2b36zu2r6CJfEMzPLQ1cmciKFcw9t	OfferCreate Others Payment	4,470,528 1,008 15	4,471,551	1.65%

to betting games. The rest was split between token transfers and various forms of entertainment, such as games not involving betting as well as payments to pornography websites. The launch of `EIDOS` increased the total number of transactions more than tenfold, resulting in 96% of the transactions being used for token transfers.

- The vast majority (76%) of transactions on Tezos are used by the `endorsement` operation to maintain consensus. This is because blocks typically contain 32 endorsements [Tez18] and the number of transactions on the network is still low. The rest of the throughput is mainly used by transactions to transfer assets between accounts.

- `OfferCreate` and `Payment` are the two most popular transaction types on XRPL, accounting for 59.1% and 36.9% of the total throughput, respectively. Between October 1, 2019 and October 8, 2019, before the systematic spamming periods, the fractions of `OfferCreate` and `Payment` are 79% and 18%, respectively. Overall, one-tenth of the transactions fail.

4.5 Case Studies

In this section, we present several case studies of how the transaction throughput on the three blockchains is used in practice, for both legitimate and less legitimate purposes.

4.5.1 Malicious Transactions on EOSIO

Exchange Wash-trading. We investigate WhaleEx, which claims to be the largest decentralized exchange (DEX) on EOSIO in terms of daily active users [Wha20]. As shown in Table 4.3, the most frequently-used action of the WhaleEx contract are `verifytrade2` and `verifytrade3`, with a combined total of 9,437,393 calls over the seven months observational period, which corresponds to approximately one action every two seconds. These actions are executed when a buy offer and a sell offer match each other and signal a settled trade.

Firstly, and most obviously, we notice that in more than 75% of the trades, the buyer and the seller are *the same*. This means that no asset is transferred at the end of the action. Furthermore, the transaction fees for both the buyer and the seller are 0, which means that such a transaction is achieving absolutely nothing else than *artificially* increasing the service statistics, i.e. wash-trading.

Further investigation reveals that accounts involved in the trades that are signalled by either `verifytrade2` or `verifytrade3` are highly concentrated: the top 5 accounts, as either a "seller" or a "buyer", are associated with over 78% of the trades. We compute the percentage of such transactions for the top 5 accounts and find that each of these accounts acts simultaneously as both seller and buyer in more than 88% of the transactions they are associated with. This means that the *vast majority* of transactions of the top 5 accounts represent wash-trading.

Next, we analyse the total amount of funds that have been moved, i.e. the difference between the total amount of cryptocurrency sent and received by the same account. For the most active account, we find that only one of the 4 currencies has a balance change of over 0.3%. The second most frequently used account has a similar transaction pattern, with only 2 out

of the 32 currencies traded showing a balance change larger than 0.6%. The rest of the top accounts all follow a very similar trend, with almost all the traded currencies having almost the same sent and received amounts.

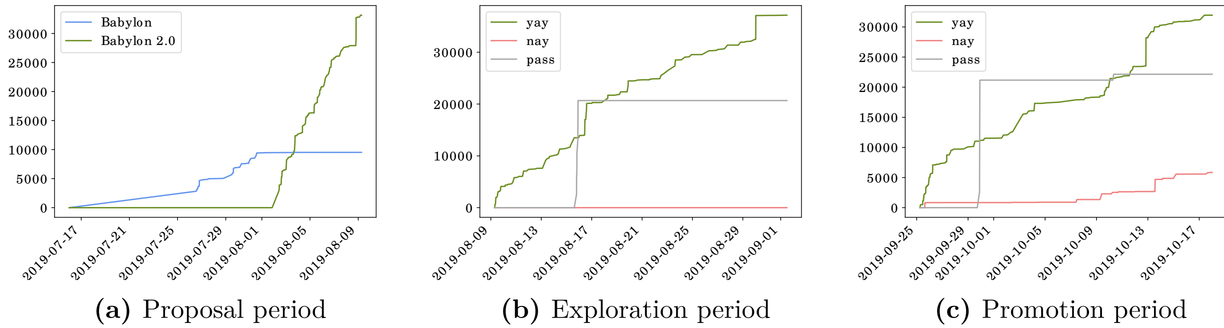
Boomerang transactions. As shown in Figure 4.1a, there was a very sharp increase in activity on EOSIO after November 1, 2019. After investigating, we find that this increase is due to the airdrop of a new coin called `EIDOS` [enu19].

The token distribution works as follows: Users send any amount of `EOS` to the EIDOS contract address, the EIDOS contract sends the `EOS` amount back to the sender and also sends 0.01% of the `EIDOS` tokens it holds. This creates a "boomerang" transaction for the `EOS` token and a transaction to send the `EIDOS` token. The tokens can then be traded for `USDT` (Tether) which can in turn be converted to other currencies. There are no transaction fees on EOSIO and users can execute transactions freely within the limits of their rented CPU capacity. Therefore, this scheme incentivises users with idle CPU resources on EOSIO to send transactions to this address, creating a large increase in the number of transactions.

Soon after the launch of this coin, the price of CPU usage on EOSIO spiked by 10,000% and the network entered a congestion mode. In normal mode, users can consume more CPU than they staked for, but when the network is in congestion mode, they can only consume the amount staked. Although this is how the network is supposed to behave, it is problematic if it lasts for a non-negligible period. For example, `EOS` is used for games where many users make a small number of transactions without staking CPU. When the network enters congestion mode for a long period, these users cannot continue to play unless they actively stake `EOS` for CPU. This has caused some services to threaten with their migration to another blockchain [Ear19].

The coin seems to be operated by an entity called Enumivo but there is very scarce information about what service it provides. Given the very hostile tone in communications[5], it is likely that the creator indeed intended to congest the EOSIO network. Furthermore, the entity behind the `EIDOS` token seems to be willing to launch a "sidechain" of EOSIO [Tea19].

[5] `https://twitter.com/enumivo/status/1193353931797057536`



(a) Proposal period **(b)** Exploration period **(c)** Promotion period

Figure 4.2: Tezos Babylon on-chain amendment voting process.

Summary. One of the major selling points of EOSIO is its absence of transaction fees for most users. Although this provides advantages for users, it can also result in spamming behaviours, as observed in this section. The fee-free transaction environment encourages market manipulation such as the WhaleEx wash-trading; moreover, it has also back-fired with the `EIDOS` token, as the network had to enter congestion mode and users have to stake an amount much higher than transaction fees in the Bitcoin network [Ear19].

4.5.2 Governance Transactions on Tezos

One of the main particularities of Tezos, compared to other blockchains, is its on-chain governance and self-amendment abilities. Given that only *bakers* are allowed to send such transactions and that they can only perform a limited number of actions within a certain time frame, governance-related transactions represent only a very small fraction of the total number of transactions: merely 604 within our observation period. However, given that this type of transaction is rather unique and has, to the best of our knowledge, not been researched before, we analyze how the different phases of the governance process are executed in practice.

Tezos voting periods. Tezos voting is divided into four periods, each lasting around 23 days [Goo14]. During the first period, the proposal period, bakers are allowed to propose an amendment in the form of source code to be deployed as the new protocol for Tezos. At the end of this period, the proposal with the highest number of bakers' votes is selected for the next period: The exploration period. During the exploration period, the bakers either choose

to approve, refuse or abstain from voting on the proposal. If the quorum and the minimum positive votes—both thresholds are dynamically adjusted based on past participation— are reached, the proposal enters the testing period. During the testing period, the proposal is deployed on a testing network, without affecting the main network. Finally, the last period is the promotion vote period, which works in the same way as the exploration period but if successful, the new protocol is deployed as the new main network.

Analyzing Tezos Voting. To investigate the entire voting process in Tezos, we collect extra data associated with a recent amendment called Babylon 2.0 [Cry19], which was proposed on August 2, 2019 and promoted to the main network on October 18, 2019. We show the evolution of the votes during the different voting phases in Figure 4.2.

During the proposal period, a first proposal, "Babylon", was submitted and slowly accumulated votes. During this phase, the authors of Babylon received feedback from involved parties and released an updated protocol, Babylon 2.0. Votes can be placed on multiple proposals which is why the number of previous votes on Babylon did not decrease. At the end of the vote, the participation was roughly 49%. It is worth noting that, although in practice any baker can propose an amendment to the network, from the creation of the Tezos blockchain up until the time of this writing, only Cryptium Labs and Nomadic Labs, who are both supported by the Tezos Foundation, have made successful proposals.

During the exploration period, participants can vote "yay" to support the proposal, "nay" to reject it, or "pass" to explicitly abstain from voting. No negative votes were cast during this period and the only abstention was from the Tezos Foundation, whose policy is to always abstain to leave the decision to the community. This phase had the participation of over 81%, significantly higher than for the previous round. This can be explained by the fact that explicit abstention counts as participation, while there is no way to explicitly abstain in the proposal phase.

Finally, after the testing period during which the proposal was deployed and tested on a testnet, the promotion period started. The trend was mostly similar to what was observed in the exploration period, but the number of votes against the proposals increased from 0 to 15%, as

some bakers encountered trouble during the testing period due to changes in the transaction format that led to breaking components [Obs19].

Improvement potential on voting mechanism. There are currently four periods in the Tezos voting system. First, participants can submit proposals, then they decide whether to try the elected proposal on a testing network and finally whether to amend the main network using the proposal. However, in every exploration period seen, proposals have always received more than 99% approval during the exploration period. With the only exception where more than 99% of rejections were received [Tez19b] during the exploration period, the participation during the proposal period was below 1%. This shows that proposals selected by a large enough number of participants are almost unanimously approved in the exploration period. Although the current voting scheme could be useful in the future, we believe this shows that in the current state of the network, the proposal and exploration periods could be merged. This would allow a reduction in the time until amendments ship to the main network without compromising the functionality or security of the network.

4.5.3 Zero-value Transactions on XRPL

Payments with zero-value tokens. As described in Section 4.2.3, XRPL offers autonomy in currency issuance. On the flip side, this means that it is easy to generate seemingly high-value, but in effect valueless and useless transactions. Currencies bearing the same ticker issued by different accounts can have drastically differing valuations due to the varying level of trust in their issuers and the redeemability of their IOU tokens, which has in the past caused confusion among less informed users.[6]

In fact, the only currency whose value is recognized outside of XRPL is its native currency `XRP`, which is also the only currency that cannot be transferred in the form of IOUs. Non-native currencies can be exchanged with each other or to `XRP` via decentralized exchanges (DEX) on the ledger. Therefore, a reliable way of evaluating a currency by a certain issuer is to look up its

[6] https://twitter.com/Lord_of_Crypto/status/965344062084497408

Table 4.6: Rate (in XRP) of BTC IOUs on XRPL.

(a) Rates (in XRP) of BTC IOUs issued by exemplary accounts in demonstration of the wide rate range. Each rate value is the average exchange rate of the issuer-specific BTC IOU tokens. Data retrieved through https://data.ripple.com/v2/exchange_rates/BTC+{issuer_address}/XRP?date=2020-01-01T00:00:00Z&period=30day [XRP20a].

Issuer name	Issuer account	Rate
Bitstamp	rvYAfWj5gh67oV6fW32ZzP3Aw4Eubs59B	36,050
Gatehub Fifth	rchGBxcD1A1C2tdxF6papQYZ8kjRKMYcL	35,817
BTC 2 Ripple	rMwjYedjc7qqtKYVLiAccJSmCwih4LnE2q	409
not registered	r3fFaoqaJN1wwN68fsMAt4QkRuXkEjB3W4	1
not registered	rpJZ5WyotdphojwMLxCr2prhULvG3Voe3X	0

(b) Rate (in XRP) of BTC IOUs issued by rKRNtZzfrkTwE4ggqXbmfgoy57RBJYS7TS at different time. In all the three exchange transactions, the account that buys the BTC IOU against XRP is rMyronEjVcAdqUvhzx4MaBDwBPSPCrDHYm

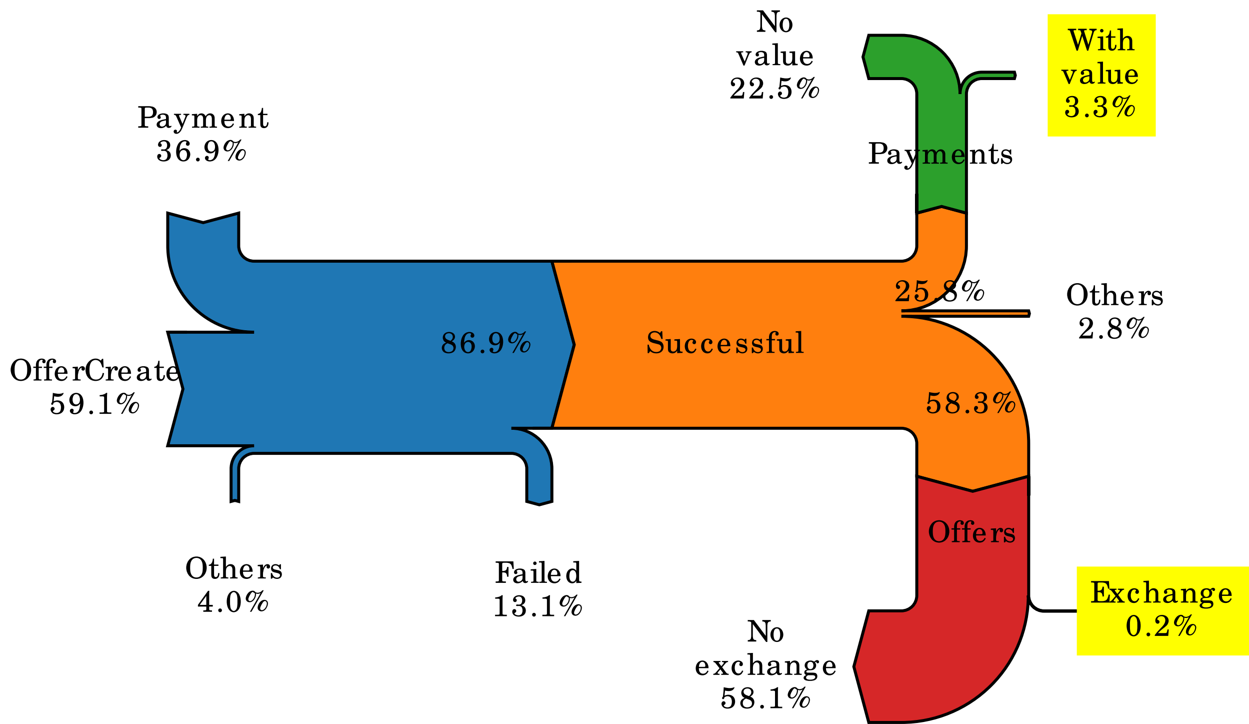
Date	Seller account of BTC IOU	Rate
2019-12-14	rHVsygEmrjSjafqFxn6dqJWHCdAPE74Zun	30,500
2020-01-09	rU6m5F9c1eWGKBdLMy1evRwk34HuVc18Wg	1
2020-01-09	rU6m5F9c1eWGKBdLMy1evRwk34HuVc18Wg	0.1

exchange rate against XRP. Normally, only IOU tokens issued by featured XRPL gateways are deemed valuable; in contrast, tokens issued by random accounts are most likely to be deemed worthless. For example, the value of BTC IOUs from various issuer accounts could range from 0 to 36,050 XRP (Table 4.6a).

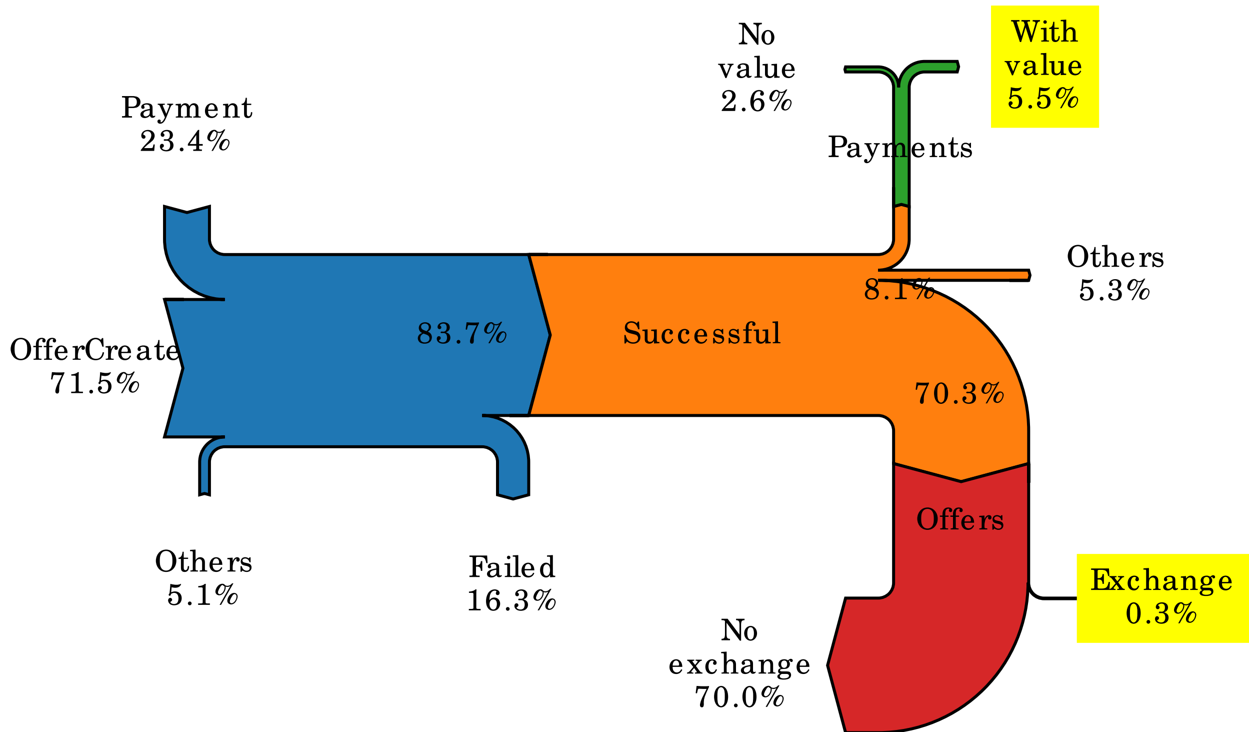
The ledger experienced two waves of abnormally high traffic in the form of Payment transactions in late 2019, the first between the end of October and the beginning of November, the second—at a higher level—between the end of November and the beginning of December (Figure 4.1c). The culprit behind the increased traffic is rpJZ5WyotdphojwMLxCr2prhULvG3Voe3X, an account activated on October 9, 2019 which itself managed to activate 5,020 new accounts within one week with a total of 1 million XRP (roughly 250,000 USD), only to have them perform meaningless transactions between each other, wasting money on transaction fees. The behaviour triggered a heated debate in the XRP community where a member claimed that the traffic imposed such a burden on their validator that it had to be disconnected [tul19].

Ripple suspected it to be "an attempt to spam the ledger" with little impact on the network.[7] However, large exchanges such as Binance suffered from temporary XRP withdrawal failures,

[7] https://twitter.com/nbougalis/status/1198670099160322048



(a) Observation period: October 1, 2019 to April 30, 2020.



(b) Observation period: February 1, 2020, to April 30, 2020, during which the throughput was not polluted by systematic `Payment` spams.

Figure 4.3: XRPL throughput by transaction type, success and value transferred. Highlighted transactions carry economic value.

which cited the XRP network congestion as the cause [Ato19]. It remains something of a mystery how such an expensive form of "spam" benefited its originators.

The payment transactions from the spam did not carry any value, since they involved transferring `BTC` IOU tokens unacceptable outside of the spammer's network.

To quantify true value-transferring `Payment` transactions, we retrieve the exchange rate with respect to `XRP` of all the issuer-specific tokens that were transferred between October 1, 2019 and April 30, 2020. Only 12.8% (3.3%/25.8%) of all successful `Payment` transactions involve tokens with a positive `XRP` rate (Figure 4.3a).

To obtain a picture of throughput usage uncontaminated by systematic spam, we re-examine the transaction data from February 1, 2020, to April 30, 2020. During this period, 67.9% successful `Payment` transactions led to value transfer (Figure 4.3b). Nevertheless, the value-carrying share of total throughput remains under 6%, since successful `Payment` transactions only account for a small fraction (8.1%) of the overall traffic and the majority (97.9%) of `OfferCreate` transactions eventually becomes void.

In Figure 4.4, we show the top senders and receivers of value-carrying `Payment` transactions, as well as the most popular currencies being transferred. To cluster accounts, we rely on usernames as the identifier, as one entity can have multiple addresses under a given user name (e.g. `Binance`, `Coinbase`). For accounts with no registered username, we use their parent's username, if available, plus the suffix "descendant" as their identifier.

As one might expect, `XRP` is by far the most used currency on the ledger in terms of payment volume: 125 billion `XRP` for seven months, or 586 million `XRP` per day.

The top 10 senders cover 53% of this volume, while the top 10 receivers are the beneficiaries of 50% of the volume. Payments from Ripple alone account for 7% (9 billion `XRP`) of the `XRP` volume, largely due to transactions associated with the monthly release of one billion `XRP` from escrows. While the `XRP` release itself is captured through `EscrowCreate` transactions, 90% of the released funds were unused and returned to escrows for future release [Tea20] through `Payment` transactions. All other top accounts presented are held either by exchanges, or, in

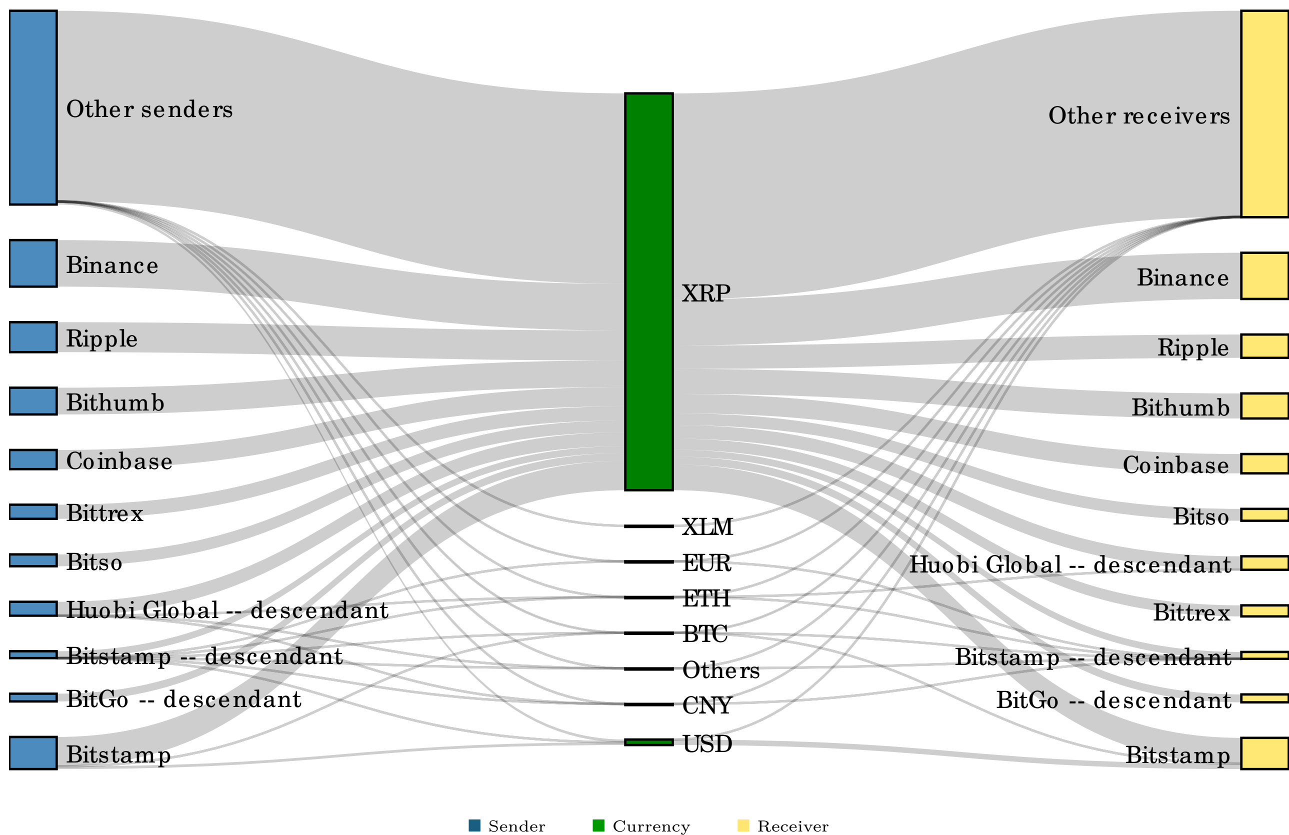


Figure 4.4: Value flow on the XRP ledger between October 1, 2019 and April 30, 2020. The bandwidth of each flow represents the magnitude of aggregate value transferred denominated in `XRP`. Only `Payment` transactions are included.

rare cases, by accounts that were opened by an exchange. `Binance` appears to be the most avid `XRP` user, sending 15.2 billion and receiving 14.5 billion `XRP` during the observation period.

The most popular IOU tokens for fiat currencies include `USD`, `EUR` and `CNY` (Figure 4.4). Specifically, 328 million `USD`, 8 million `EUR` and 19 million `CNY` issued had positive exchange rates against `XRP`. The average on-ledger exchange rates of those three fiat currency tokens, irrespective of their issuers, were 5.4 `XRP/USD`, 5.5 `XRP/EUR` and 0.7 `XRP/CNY`, largely in accordance with the off-ledger exchange rates.[8]

Fulfilled offers with zero-value tokens. We found a series of conspicuous payment transactions with the aggregate transfer of 360,222 `BTC` IOU, issued by `rKRNtZzfrkTwE4ggqXbmfgoy57RBJYS7TS`, an account activated by Liquid (liquid.com), from the issuer itself to `rMyronEjVcAdqUvhzx4MaBDwBPSPCrDHYm`,

[8]`https://finance.yahoo.com/`

an account activated by uphold (uphold.com). The BTC IOU token was exchanged at 30,500 XRP, resulting in a valuation of 11 billion XRP of those payments. We examine the legitimacy of the exchange rates in the next step.

The issuer is not the only factor behind the value of an IOU token. Even IOU tokens for the same currency from the same issuer can at times exhibit vastly different rates. Table 4.6b shows an example where the BTC IOU from the same issuer rKRNtZzfrkTwE4ggqXbmfgoy57RBJYS7TS was traded at 30,500 XRP in December 2019 but then declined to 0.1 XRP within a month.

The three exchange instances in Table 4.6a were OfferCreate transactions where the initiator intended to sell BTC ICO for XRP. We discover that all three offers were filled by the same account rMyronEjVcAdqUvhzx4MaBDwBPSPCrDHYm, who received the aforementioned BTC IOU tokens directly from the issuer's account. Additional evidence on social media reveals that the IOU issuer's account is held by someone named Myrone Bagalay.[9] It becomes obvious that the offer taker's address, starting with rMyronE, must belong to the same person.

By tracing the transaction history of the concerned accounts, we notice that the two offer creators' accounts received their initial BTC IOU tokens through payments from the offer taker. Furthermore, one offer creator's account, rU6m5F9c1eWGKBdLMy1evRwk34HuVc18Wg, was activated by the offer taker's account. Now we can safely assume that all the accounts involved are controlled by that Myrone Bagalay, who issued BTC IOU tokens and traded them at arbitrarily determined rates with himself.

What Myrone Bagalay did is completely legitimate within the confines of XRPL. One of the key features of the ledger is the flexibility to establish a closed economy with a limited number of mutually-trusting users who can exchange self-defined assets that are not necessarily acknowledged outside the system. However, this makes it challenging to gauge the true value transfer on XRPL since an IOU token's price—which we proxy by its exchange rate against XRP—can be easily inflated or deflated.

Additionally, privately-issued IOU tokens that are never exchanged on the ledger, while seem-

[9]See https://youtu.be/gVoyCEPvO30 and https://www.twipu.com/MyroneBagalay/tweet/1161288087386894341

ingly worthless, might be valuable to their transactors after all, should they reach an agreement on those tokens' value of the ledger. However, there is no easy way to assess such value, and we leave the analysis of IOUs to future work.

Summary. In summary, the throughput on XRPL during our observation period appeared to be fraught with zero-value transactions. We learned that both transaction volume and token value on XRPL are highly manipulable. One must thus fully understand the underlying measurement approach to correctly interpret the resultant statistics.

4.6 Discussion

In this section, we discuss the results from the previous sections and also answer the research questions presented in Section 4.1 in light of our results.

4.6.1 Interpretation of the Throughput Values

Overall, we observe that the throughput on EOSIO has been volatile since last November, the throughput on Tezos has been very stable over time, and the throughput on XRPL has been stable in general except during the spam episode. A common factor between all blockchains is that the current throughput is vastly lower than the alleged capacity even during their utilization peaks, and is on average several orders of magnitude lower. A similarity between EOSIO and XRPL is that the maximum throughput was reached due to DoS attacks on the network. Indeed, the maximum number of transactions on EOSIO is due to the `EIDOS` coin airdrop, while the peak on XRPL was due to the network being spammed with payments. However, while the spam on XRPL appeared to be anecdotal and lasted for roughly two months, the spam attack on EOSIO is persistent and has continued for over six months to date. This increased throughput has different implications for each network. While on XRPL the consequences of such a spam attack are limited, on EOSIO they forced the network to enter congestion mode, hindering normal usage of the network as transactions become too costly due to the elevated

threshold for staking.

Unlike XRPL and EOSIO, Tezos has not seen any spam attacks and the level of utilization has been consistent, and relatively low, over time. A majority of the throughput is used for consensus, with most of the spikes in the number of transactions due to baker payments, which are also related to consensus.

4.6.2 Revisiting Research Questions

We now return to the research questions posed in Section 4.1 and seek to understand better how the different blockchains are used in practice, by attempting to answer them based on the data analysis we perform above.

RQ1: used throughput capacity. Although the maximum throughput of all blockchains appears vastly lower than the alleged capacity, the situation is not as simple for EOSIO and XRPL. As previously discussed, EOSIO started to be congested because of an airdrop, preventing regular users to use the blockchain normally. During the attack against XRPL, there were several reports of the network being congested [Ato19; tul19], showing that although the claimed capacity was much higher, the *actual* capacity might have maxed. Nevertheless, it is yet unclear whether the congestion is mainly due to the suboptimal design of blockchain protocols or the physical constraint of participating nodes' infrastructure. On the other hand, Tezos has not yet come close to maximizing its actual capacity.

RQ2: classifying actions. We made a generalized categorization of transaction types. Some transaction types are common to all blockchains, such as peer-to-peer transactions and account-related transactions, while other types of transactions are inherent to the particularities of the underlying blockchain. While XRPL and Tezos contain easily identifiable action types, making them easy to classify, EOSIO does not have pre-defined action types and classifying actions requires knowledge of the account receiving the action.

RQ3: identifying active blockchain participants. EOSIO has named accounts which

makes it easy to identify participants. XRPL has optional names, which are registered by the most active players such as exchanges. Tezos endorsements are often created by bakers, who usually publicise their addresses and are identifiable. However, there is no easy way to identify participants in peer-to-peer transactions and doing so would require using de-anonymization techniques [BKP14; GPL19].

RQ4: detecting DoS and spam. The blockchains analysed are currently under-utilized and when spam occurs, their utilization level increases significantly, as seen in Figure 4.1. This makes DoS and spam attacks very easy to detect by simply looking at the transactions, as we saw for EOSIO and XRPL.

4.6.3 Transaction Fee Dilemma

Overall, we have seen that there is a dilemma between having lower transaction fees, which induces spam, or having higher transaction fees, which deters legitimate usage of the network.

On the one side, we have seen that both EOSIO and XRPL have chosen to go with extremely low transaction fees, which in both cases resulted in a very large amount of spam. On the other side of the spectrum, Ethereum, which has transaction fees based on supply and demand [Woo19] has seen a 10-times increase in the fees, mainly because of an increase in the utilization of decentralized finance protocols [Gud+20b], making it extremely difficult to use for regular users [Pec20].

There has been efforts on both sides to improve the current situation but, at the time of writing, no significant progress has been made. Despite fee structure changes having been proposed in XRPL [XRP19c], concerns are that a fee increase discourages the engagement of legitimate users. In EOSIO, despite the integration of a new rental market for CPU and RAM [EOS20b], the current fee structure remains problematic, as the network has now been congested for more than half a year, making it hard to use for regular users. On the Ethereum side of things, changes in the current pricing system to try to reduce the transaction fees have been proposed [But+19] but are still under discussion.

Overall, for a functional and sustainable blockchain system, it is crucial to find a balanced transaction fee mechanism that can make regular usage of the network affordable while DoS attacks remain expensive [PL20].

4.7 Related Work

4.7.1 Previous work

Existing literature on transactional patterns and graphs on blockchains has been largely focused on Bitcoin.

Ron et al. [RS13] are among the first to analyze transaction graphs of Bitcoin. Using on-chain transaction data with more than 3 million different addresses, the authors find that Mt. Gox was at the time by far the most used exchange, covering over 80% of the exchange-related traffic.

Kondor et al. [Kon+14] focus on the wealth distribution in Bitcoin and provided an overview of the evolution of various metrics. They find that the Gini coefficient of the balance distribution has increased quite rapidly and show that the wealth distribution in Bitcoin is converging to a power law.

McGinn et al. [McG+16] focus their work on visualizing Bitcoin transaction patterns. At this point, in 2016, Bitcoin already had more than 300 million addresses, indicating exponential growth over time. The authors propose a visualization which scales well enough to enable pattern searching. Roughly speaking, they present transactions, inputs and outputs as vertices while treating addresses as edges. The authors report that they were able to discover high-frequency transaction patterns such as automated laundering operations or denial-of-service attacks.

Ranshous et al. [Ran+17] extend previous work by using a directed hypergraph to model Bitcoin transactions. They model the transaction as a bipartite hypergraph where edges are in and out

amounts of transactions and the two types of vertices are transactions and addresses. Based on this hypergraph, they identify transaction patterns, such as "short thick band", a pattern where Bitcoins are received from an exchange, held for a while and sent back to an exchange. Finally, they used different features extracted from the hypergraph, such as the amount of Bitcoin received but also how many times the address appeared in a certain pattern, to train a classifier capable of predicting if a particular address belongs to an exchange.

Di Francesco Maesa et al. [DMR17] analyze Bitcoin user graphs to detect unusual behaviour. The authors find that discrepancies such as outliers in the in-degree distribution of nodes are often caused by artificial users' behaviour. They then introduce the notion of pseudo-spam transactions, which consist of transactions with a single input and multiple outputs where only one has a value higher than a Satoshi, the smallest amount that can be sent in a transaction. They find that approximately 0.5% of the total number of multi-input multi-output transactions followed such a pattern and that these were often chained.

Several other works also exist about the subject and very often try to leverage some machine learning techniques either to cluster or classify Bitcoin addresses. Monamo et al. [MMT16] attempted to detect anomalies on Bitcoin and show that their approach can partly cluster some fraudulent activity on the network. Toyoda et al. [TOM17] focus on classifying Ponzi schemes and related high-yield investment programs by applying supervised learning using features related to transaction patterns, such as the number of transactions an address is involved in, or its ratio of pay-in to pay-out.

More recently, a study of EOSIO decentralized applications (DApps) has been published [Hua+20]. The authors analyze the EOSIO blockchain from another angle: they look at the DApps activities and attempt to detect bots and fraudulent activities. The authors identified thousands of bot accounts as well as real-world attacks, 80 of which have been confirmed by DApp teams.

To the best of our knowledge, this was the first academic work to empirically analyze the transactions of Tezos and XRPL and the first to compare transactional throughput on these platforms.

4.7.2 Follow-up work

More research using similar methods to the one we present in this paper, or using similar datasets, has been published since our work has been published. We highlight a few of these here.

He et al. [He+21] perform a similar analysis to ours but focus on Bitcoin, Ethereum, and EOSIO blockchains. The authors analyse billions of transaction records collected from these blockchains over 10 years. They show that although the overall blockchain ecosystem shows promising growth over the last decade, a number of worrying “outliers”, such as attacks or spam, exist that have disrupted its evolution.

In another piece of work, He et al. [He+22] dig deeper into the EOSIO ecosystem. The authors collected all occurred attack events against EOSIO, and systematically studied their root causes, i.e., vulnerabilities lurked in all relying components for EOSIO, as well as the corresponding attacks and mitigations.

4.8 Conclusions

We investigate transaction patterns and value transfers on the three major high-throughput blockchains: EOSIO, Tezos, and XRPL. Using direct connections with the respective blockchains, we fetch transaction data between October 1, 2019 and April 30, 2020. With EOSIO and XRPL, the majority of the transactions exhibit characteristics resembling DoS attacks: on EOSIO, 95% of the transactions were triggered by the airdrop of a yet valueless token; on XRPL, over 94%—consistently in different observation periods—of the transactions carry no economic value. For Tezos, since transactions per block are largely outnumbered by mandatory endorsements, most of the throughput, 76% to be exact, is occupied for maintaining consensus.

Furthermore, through several case studies, we present prominent cases of how transactional throughput was used on different blockchains. Specifically, we show two cases of spam on EOSIO, on-chain governance-related transactions on Tezos, as well as payments and exchange

offers with zero-value tokens on XRPL.

The bottom line is: the three blockchains studied in this chapter demonstrate the capacity to support high levels of throughput; however, the massive potential of those blockchains has thus far not been fully realized for their intended purposes.

Chapter 5

Application Security

In this chapter, we go one level further up in the blockchain stack and focus on the security of applications built on top of the blockchain. To do so, we first introduce the concepts of technical and economic security. We then present an analysis of on-chain technical exploits, focusing on the most common classes of technical vulnerabilities, such as reentrancy. Finally, we explore economic security further by focusing on over-collateralization in lending protocols.

5.1 Technical and Economic Security

5.1.1 Technical Security

We define a security risk to be *technical* if an agent can atomically exploit a protocol. In a technical exploit, an attacker effectively finds a sequence of contract calls, whether in a single transaction or a bundle of transactions, that leads to a profit through a violation of a protocol's intended properties (as visualized in Fig. 5.1). Such exploits can be performed risk-free (and often in a sense "instantaneously") because the outcomes for the attacker are binary: either the attack is successful and the attacker profits or the transaction reverts (effectively the attack doesn't happen) and the attacker only loses some gas fees.

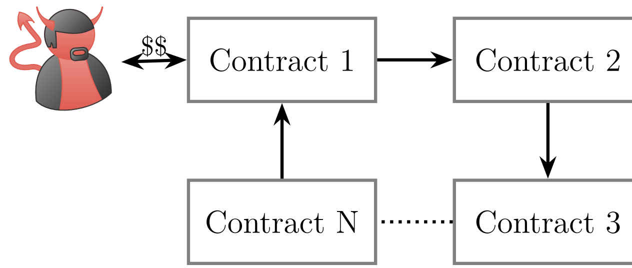


Figure 5.1: Diagram of a technical exploit.

In current blockchain implementations, this coincides with (1) manipulating an on-chain system within a single transaction, which is risk-free for anyone, and (2) manipulating transactions within the same block, which is risk-free for the miner generating that block or for an attacker who creates a bundle of transactions that are required to execute atomically in the order given. By exploiting technical structure, the underlying blockchain system allows no opportunity for markets or other agents to react in the course of such exploits (when such reaction is possible, we enter the domain of *economic* security problems in the next section). When there is competition to perform these exploits, they will factor into the game theory of blockchain mining (e.g., [Bia+19]) as part of MEV extraction (as discussed in [Dai+19]); however, attempting these exploits will be risk-free (minus potential gas fees) for any attacker. We identify three categories of attacks that fall within technical security risks of protocols: attacks exploiting smart contract vulnerabilities, attacks relying on the execution order of transactions in a block, as well as attacks which are executed within a single transaction. These security risks are often addressable with program analysis and formal models to specify protocols, although these problems can quickly become complex to formulate and computationally hard.

Technical Security

A protocol is technically secure if it is not possible for an attacker to atomically exploit the protocol at the expense of value held by the protocol or its users. Due to atomicity, these attacks can generate risk-free profit. A common property of technical exploits is that they occur within a single transaction or a bundle of transactions in a block.

5.1.2 Economic security

We define a security risk to be *economic* if an attacker can perform a strictly non-atomic exploit to realize a profit at the expense of value held by the protocol or its users. In an economic exploit, an attacker performs multiple actions at different places in the transaction sequence but doesn't control what happens between their actions in the sequence, which means that there is no guarantee that the final action is profitable (as visualized in Fig 5.2 and in comparison to the technical exploit in Fig 5.1). Economic security is effectively about an exploiting agent who tries to manipulate a market or incentive structure over some time period (even if short, it is not instantaneous). Compared to technical exploits, since economic exploits are non-atomic, they come with upfront tangible costs, a probability of attack failure and risk related to mis-estimating the market response. Thus they are not risk-free and commonly involve manipulations over many transactions or blocks.

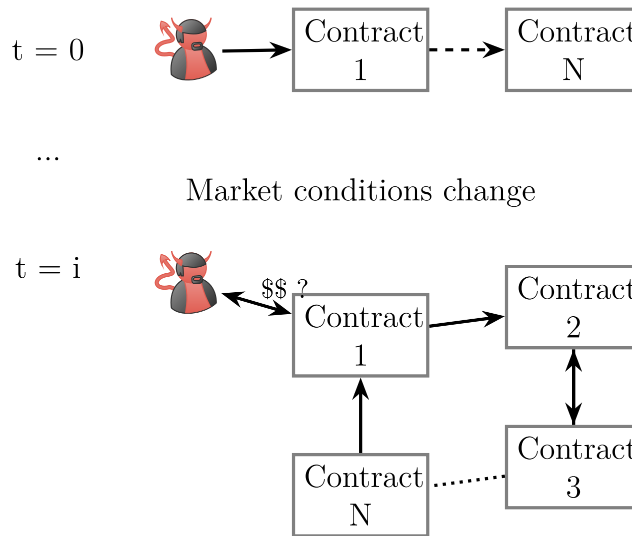


Figure 5.2: Diagram of an economic exploit.

In addition to comparing the structures in Figures 5.1 and 5.2, we provide a simple example to help illustrate the distinction between technical and economic security. Consider a protocol that uses an instantaneous Automated Market Maker (AMM) price as an oracle. An attacker can perform a (atomic) sandwich attack to steal assets, which amounts to a technical exploit. If instead the protocol used a time-weighted average AMM price as an oracle, then the attacker could manipulate this price over time (non-atomically) and may still be able to steal assets,

which would amount to an economic exploit.

Economic risks are inherently a problem of economic design and cannot be solved by technical means alone. To illustrate, while these attacks could be performed atomically (and risk-free) in a very poorly constructed system that allowed it, they are not solved, for example, just by adding a time delay that ensures they are not executed in the same block. Even if all technical issues are sorted, we are often left with remaining economic problems about how markets or other incentive structures could be manipulated over time to exploit protocols. From a practical perspective, progress on these economic problems inherently requires economic models of these market equilibria and the design of better protocol incentive structures. These models differ considerably from traditional security models and are sparsely studied. As a result, defensive measures for economic security risks are also not as well established.

In this way also, technical security must be a first bar: if a protocol is not technically secure, then it will break in the presence of rational agents. Economic security only makes sense if technical security is achieved. For instance, if a protocol's funds can be exploited because it is not technically secure, then in an economic sense no rational agents should participate.

Economic Security

A protocol is economically secure if it is economically infeasible (e.g., unprofitable) for an attacker to perform exploits that are strictly non-atomic at the expense of value held by the protocol or its users. As economic exploits are non-atomic (or else they are better described as technical), they are not risk-free.

5.2 Technical Security: Smart contracts exploits in practice

While most of the work related to technical security has focused on detecting *vulnerable* contracts, in this part of the chapter, we focus on finding out how many of these vulnerable contracts have actually been *exploited*. We survey the 23,327 vulnerable contracts reported by six recent academic projects and find that, despite the amounts at stake, only 1.98% of

them have been exploited since deployment. This corresponds to at most 8,487 ETH (~1.7 million USD[1]), or only 0.27% of the 3 million ETH (6000 million USD) at stake. We explain these results by demonstrating that the funds are very concentrated in a small number of contracts which are *not exploitable* in practice.

5.2.1 Introduction

When it comes to vulnerability research, especially as it pertains to software security, it is frequently difficult to estimate what fraction of discovered vulnerabilities are exploited in practice. However, public blockchains, with their immutability, ease of access, and what amounts to a replayable execution log for smart contracts present an excellent opportunity for such an investigation. In this work, we aim to contrast the vulnerabilities reported in smart contracts on the Ethereum [But14] blockchain with the actual exploitation of these contracts.

We collect the data shared with us by the authors of six recent papers [Luu+16a; Kal+18; Tsa+18; Gre+18; Nik+18; KR18] that focus on finding smart contract vulnerabilities. These academic datasets are significantly bigger in scale than reports we can find in the wild and because of the sheer number of affected contracts — 23,327 — represent an excellent study subject.

To make our approach more general, we express six different frequently reported vulnerability classes as Datalog queries computed over relations that represent the state of the Ethereum blockchain. The Datalog-based exploit discovery approach gives more scalability to our process; also, while others have used Datalog for static analysis formulation, we are not aware of it being used to capture the dynamic state of the blockchain over time.

We discover that the amount of smart contract exploitation which occurs in the wild is notably lower than what might be believed, given what is suggested by the sometimes sensational nature of some of the famous cryptocurrency exploits such as TheDAO [SC17] or the Parity wallet [Bre+17] bugs.

Contributions. Our contributions are:

- **Datalog formulation.** We propose a Datalog-based formulation for performing analysis over Ethereum Virtual Machine (EVM) execution traces. We use this highly scalable approach to analyze a total of more than 20 million transactions from the Ethereum blockchain to search for exploits. We highlight that our analyses run *automatically* using data extracted from the transactions and the Datalog rules that we define in this chapter.

- **Experimental evaluation of exploitation.** We analyze the vulnerabilities reported in six recently published studies and conclude that, although the number of *vulnerable* contracts and the amount of money at risk is very high, the amount of money actually *exploited* is several orders of magnitude lower.

  We discover that out of 23,327 vulnerable contracts worth a total of 3,124,433 ETH, 463 contracts may have been exploited for an amount of 8,487 ETH, which represents only 0.27% of the total amount at stake.

- **Proposed explanations.** We hypothesise that the main reason for these vast differences is that the amount of *exploitable* Ether is very low compared to the amount of Ether flagged *vulnerable*. Indeed, further analysis of the vulnerable contracts and the Ether they contain suggests that a large majority of Ether is held by only a small number of contracts and that the vulnerabilities reported on these contracts are either false positives or not exploitable in practice. We also confirm that the set of all contracts on the Ethereum blockchain has a similar distribution of wealth to our dataset.

To make many of the discussions in this chapter more concrete, we present a thorough investigation of the high-value contracts in Section 5.2.7.

5.2.2 Background

In this section, we first introduce different types of smart contract vulnerabilities. We then present some of the analysis tools available to prevent such vulnerabilities. Finally, we provide some definitions that will be used in the rest of the chapter.

Smart Contracts Vulnerabilities

In this subsection, we briefly review some of the most common vulnerability types that have been researched and reported for EVM-based smart contracts. We provide a two-letter abbreviation for each vulnerability which we shall use throughout the remainder of this section.

Reentrancy (RE). When a contract "calls" another account, it can choose the amount of gas it allows the called party to use. If the target account is a contract, it will be executed and can use the provided gas budget. If such a contract is malicious and the gas budget is high enough, it can try to call back from the caller — a re-entrant call. If the caller's implementation is not re-entrant, for example, because it did not update his internal state containing balances information, the attacker can use this vulnerability to drain funds out of the vulnerable contract [Luu+16a; Kal+18; Tsa+18]. This vulnerability was used in TheDAO exploit [SC17], essentially causing the Ethereum community to decide to roll back to a previous state using a hard-fork [Meh+19]. We provide more details about TheDAO exploit in Section 5.2.8

Unhandled Exceptions (UE). Some low-level operations in Solidity such as `send`, which is used to send Ether, do not throw an exception on failure, but rather report the status by returning a boolean. If this return value is unchecked, the caller continues its execution even if the payment failed, which can easily lead to inconsistencies [Bre+18; Luu+16a; Tik+18; Kal+18].

Locked Ether (LE). Ethereum smart contracts can, as any account on Ethereum, receive Ether. However, there as several reasons causing the received funds to get locked permanently into the contract.

One reason is that the contract may depend on another contract which has been destructed using the `SELFDESTRUCT` instruction of the EVM — i.e. its code has been removed and its funds transferred. If this was the only way for such a contract to send Ether, it will result in the funds being permanently locked. This is what happened in the Parity Wallet bug in November 2017, locking millions of USD worth of Ether [Bre+17]. We provide more details about the Parity Wallet bug in Section 5.2.8

There are also cases where the contract will *always* run out of gas when trying to send Ether which could result in locking the contract funds. More details about such issues can be found in [Gre+18].

Transaction Order Dependency (TO). In Ethereum, multiple transactions are included in a single block, which means that the state of a contract can be updated multiple times in the same block. If the order of two transactions calling the same smart contract changes the outcome, an attacker could exploit this property. For example, given a contract which expects participants to submit the solution to a puzzle in exchange for a reward, a malicious contract owner could reduce the amount of the reward when the transaction is submitted.

Integer Overflow (IO). Integer overflow and underflow are common types of bugs in many programming languages but in the context of Ethereum, they can have very severe consequences. For example, if a loop counter were to overflow, creating an infinite loop, the funds of a contract could become completely frozen. This can be exploited by an attacker if he has a way of incrementing the number of iterations of the loop, for example, by registering enough users to trigger an overflow.

Unrestricted Action (UA). Contracts often perform authorization, by checking the sender of the message, to restrict the type of action that a user can take. Typically, only the owner of a contract should be allowed to destroy the contract or set a new owner. This owner is usually set in the contract constructor but some contracts were found vulnerable because the owner was not initialized correctly, allowing, for example, an attacker to take ownership of the contract. A reason for such a bug could be a misnamed function in older versions of Solidity [Bre+18; KR18]. This issue was the root cause of the the Parity wallet bug [Tsa+18; Nik+18] which froze more than 500k Ether.

Such an issue can happen not only if the developer forgets to perform critical checks but also if an attacker can execute arbitrary code, for example by being able to control the address of a delegated call [KR18].

Table 5.1: A summary of smart contract analysis tools presented in prior work.

Name	Vulnerabilities						Report month	Citation
	RE	UE	LE	TO	IO	UA		
Oyente	✓	✓		✓	✓		2016-10	[Luu+16a]
ZEUS	✓	✓	✓	✓	✓		2018-02	[Kal+18]
Maian			✓			✓	2018-03	[Nik+18]
SmartCheck	✓	✓	✓		✓		2018-05	[Tik+18]
Securify	✓	✓	✓	✓		✓	2018-06	[Tsa+18]
ContractFuzzer	✓	✓					2018-09	[JLC18]
teEther						✓	2018-08	[KR18]
Vandal	✓	✓					2018-09	[Bre+18]
MadMax			✓		✓		2018-10	[Gre+18]

Analysis Tools

Smart contracts are generally designed to manipulate and *hold* funds denominated in Ether. This makes them very tempting attack targets, as a successful attack may allow the attacker to directly steal funds from the contract. Given the many common vulnerabilities in smart contracts, some of which we described in the previous section, a large number of tools have been developed to find them automatically [Luu+16a; Tsa+18; Con19c]. Most of these tools analyze either the contract source code or its compiled EVM bytecode and look for known security issues, such as reentrancy or transaction order dependency vulnerabilities. We present a summary of these different works in Table 5.1. The second and third columns respectively present the reported number of contracts analyzed and contracts flagged as vulnerable in each paper. The "vulnerabilities" columns show the type of vulnerabilities that each tool can check for. We present these vulnerabilities in Section 5.2.2 and give a more detailed description of these tools in Section 5.2.8.

Testing. Like any piece of software, smart contracts benefit from automated testing and some efforts have therefore been made to make the testing experience more straightforward. Truffle [Con19a] is a popular framework for developing smart contracts, which allows writing both unit and integration tests for smart contracts in JavaScript. One difficulty of testing on the

Ethereum platform is that the EVM does not have a single main entry point and bytecode is executed when fulfilling a transaction. Some tools, such as standalone EVM implementations [Con19b] have been developed to ease this process.

Auditing. As smart contracts can have a high monetary value, *auditing* contracts for vulnerabilities is a common industrial practice. Audits should preferably be performed while contracts are still in the testing phase but given the relatively high cost of auditing (usually around 30,000 to 40,000 USD [19g]) some companies choose to perform audits later in their development cycle. In addition to checking for common vulnerabilities and implementation issues such as gas-consuming operations, audits also usually check for divergences from specifications and other high-level logic errors, which are impossible for current automatic tools to detect.

Bounty programs. Another common practice for developers to improve the security of their smart contracts is to run bounty programs. While auditing is usually a one-time process, bounty programs remain ongoing throughout a contract's lifetime and allow community members to be rewarded for reporting vulnerabilities. Companies or projects running bounty programs can either choose to reward the contributors by paying them in a fiat currency such as US dollars, cryptograms — typically Bitcoin or Ether — or other crypto assets. Some bounty programs, such as the one run by the 0x project [19a], offer bounties as high as 100,000 USD for critical vulnerabilities.

Contract upgrades. In Ethereum, smart contracts are by nature immutable. Once a contract has been deployed on the blockchain, its code cannot be modified. This creates a challenge during the deployment of smart contracts, as upgrading the code requires working around this limitation. There are several approaches to deploying a new version of a smart contract [19c].

The first approach is to use a *registry contract* which returns the address of the latest version of a smart contract. When deploying a contract, the contract with the updated version of the code is deployed and the address of the latest version stored in the registry is updated. Although this leaves a lot of flexibility to the developers, it forces the users of the smart contracts to always query the registry before being able to interact with the contract. To avoid adding overhead to

the user of the contract, an alternative approach is to use a *facade contract*. In this approach, a contract with a fixed address is deployed but delegates all the calls to another contract, the address of which can be updated [But15]. The end-user of the contract can therefore always transact with the same contract, while the developers can update the behaviour of the contract by deploying a new contract and updating the facade to delegate to the newly deployed code.

There are two main drawbacks to this approach. One of the drawbacks of this approach is that developers cannot modify the contract interface, as the facade code does not change. The other is that there is a gas cost overhead, as the facade contract uses gas to call the backend contract.

Definitions

We give the definitions used in this section for the terms *vulnerable*, *exploitable* and *exploited*.

vulnerable: A contract is vulnerable if it has been flagged by a static analysis tool as such. As we will see later, this means that some contracts may be *vulnerable* because of a false-positive.

exploitable: A contract is exploitable if it is vulnerable and the vulnerability could be exploited by an external attacker. For example, if the "vulnerability" flagged by a tool is in a function which requires owning the contract, it would be *vulnerable* but not *exploitable*.

exploited: A contract is exploited if it received a transaction on Ethereum's main network which triggered one of its vulnerabilities. Therefore, a contract can be *vulnerable* or even *exploitable* without having been *exploited*.

5.2.3 Dataset

In this section, we analyze the vulnerable contracts reported by the following six academic papers: [Luu+16a], [Kal+18], [Nik+18], [Tsa+18], [Gre+18] and [KR18]. To collect information

Table 5.2: Summary of the contracts in our dataset.

Name	Contracts analyzed	Vulnerabilities found	Ether at stake at time of report
Oyente	19,366	7,527	1,287,032
Zeus	1,120	861	671,188
Maian	NA	2,691	15.59
Securify	29,694	9,185	724,306
MadMax	91,800	6,039	1,114,958
teEther	784,344	1,532	1.55

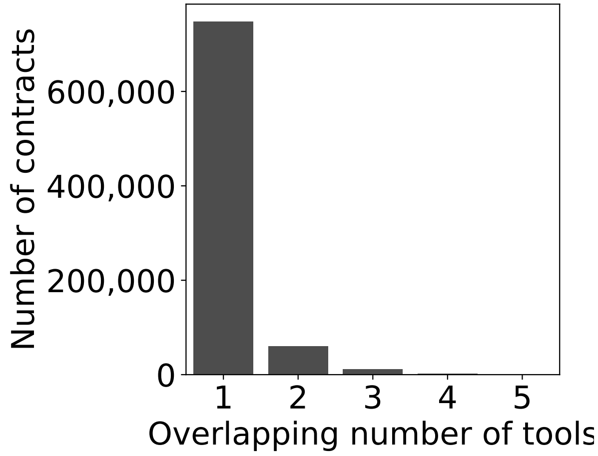
about the addresses analyzed and the vulnerabilities found, we reached out to the authors of the different papers.

Oyente [Luu+16a] data was publicly available [Luu+16b]. The authors of the other papers were kind enough to provide us with their dataset. We received all the replies within less than a week of contacting the authors.

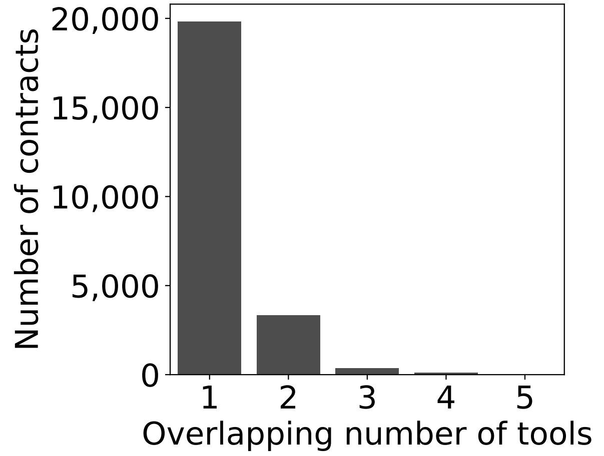
We also reached out to the authors of [Tik+18], [JLC18] and [Bre+18] but could not obtain their dataset, which is why we left these papers out of our analysis.

Our dataset is comprised of a total of 821,219 contracts, of which 23,327 contracts have been flagged as vulnerable to at least one of the six vulnerabilities described in Section 5.2.2. Although we received the data directly from the authors, the numbers of contracts analyzed usually did not match the data reported in the papers, which we show in Table 5.1. We believe the two main results for this are: authors improving their tools after the publication and authors, not including duplicated contracts in the data they provided us. Therefore, we present the numbers in our dataset, as well as the Ether at stake for vulnerable contracts in Table 5.2. The Ether at stake is computed by summing the balance of all the contracts flagged vulnerable. We use the balance at the time at which each paper was published rather than the current one, as it gives a better sense of the amount of Ether which could potentially have been exploited.

Taxonomy. Rather than reusing existing smart contract vulnerability taxonomies [ABC17] as-is, we adapt it to fit the vulnerabilities analysed by the tools in our dataset. We do not cover vulnerabilities not analyzed by at least two of the six tools. We settle on the six types of



(a) Overlapping contracts analysed.



(b) Overlapping vulnerabilities flagged.

Figure 5.3: Histograms that show the overlap in the contracts analysed and flagged by examined tools.

Table 5.3: Agreement among tools for reentrancy analysis.

Tools	Total	Agreed	Disagreed	% agreement
Oyente/Securify	774	185	589	23.9%
Oyente/Zeus	104	3	101	2.88%
Zeus/Securify	108	2	106	1.85%

vulnerabilities described in Section 5.2.2: reentrancy (RE), unhandled exception (UE), locked Ether (LE), transaction order dependency (TO), integer overflows (IO) and unrestricted actions (UA). As the papers we survey use different terms and slightly different definitions for each of these vulnerabilities, we map the relevant vulnerabilities to one of the six types of vulnerabilities we analyze. We show how we mapped these vulnerabilities in Table 5.4.

Overlapping vulnerabilities. In this subsection, we first check how much overlap there

Table 5.4: Mapping of the different vulnerabilities analyzed.

	Oyente	ZEUS	Securify	MadMax	Maian	teEther
RE	reentrancy	reentrancy	no writes after call	—	—	—
UE	callstack	unchecked send	handled exceptions	—	—	—
TO	concurrency	tx order dependency	tx ordering dependency	—	—	—
LE	—	failed send	Ether liquidity	unbounded op wallet griefing	greedy	—
IO	—	integer overflow	—	integer overflows	—	—
UA	—	integer overflow	—	integer overflows	prodigal	exploitable

Code Listing 5.1: Sample execution trace information.

```
[
    {"op": "EQ", "pc": 7,
    "depth": 1, "stack": ["2b", "a3"]},
    {"op": "ISZERO", "pc": 8, "depth": 1,
    "stack": ["00"]}
]
```

is between contracts in our dataset: how many contracts have been analyzed by multiple tools and how many contracts were flagged vulnerable by multiple tools. We note that most papers, except for [Luu+16a], are written around the same period. We find that 73,627 out of 821,219 contracts have been analyzed by at least two of the tools but only 13,751 by at least three tools. In Figure 5.3a, we show a histogram of how many different tools analyze a single contract. In Figure 5.3b, we show the number of tools which flag a single contract as vulnerable to any of the analyzed vulnerabilities. The overlap for both the analyzed and the vulnerable contracts is relatively small. We assume one of the reasons is that some tools work on Solidity code [Kal+18] while other tools work on EVM bytecode [Tsa+18; Luu+16a], making the population of contracts available different among tools.

We also find a lot of contradiction in the analysis of the different tools. We choose reentrancy to illustrate this point, as it is supported by three of the tools we analyze. In Table 5.3, we show the agreement between the three tools supporting reentrancy detection. The *Total* column shows the total number of contracts analyzed by both tools in the *Tools* column and flagged by at least one of them as vulnerable to reentrancy. Oyente and Securify agree on only 23% of the contracts, while Zeus does not seem to agree with any of the other tools. This reflects the difficulty of building static analysis tools targeted at the EVM. While we are not trying to evaluate the different tools' performance, this gives us yet another motivation to find out the impact of the reported vulnerabilities.

5.2.4 Methodology

In this section, we describe in detail the different analyses we perform in order to check for exploits of the vulnerabilities described in Section 5.2.2.

To check for potential exploits, we perform bytecode-level transaction analysis, whereby we look at the code executed by the contract when carrying out a particular transaction. We use this type of analysis to detect the six types of vulnerabilities presented in Section 5.2.2.

To perform our analyses, we first retrieve transaction data for all the contracts in our dataset. Next, to perform bytecode-level analysis, we extract the execution traces for the transactions which may have affected contracts of interest. We use the EVM's debug functionality, which gives us the ability to replay transactions and trace all the executed instructions. To speed up the data collection process, we patch the Go Ethereum client [19f], as opposed to relying on the Remote Procedure Call (RPC) functionality provided by the default Ethereum client. We show a truncated sample of the extracted traces in Code Listing 5.1 for illustration. The `op` key is the current instruction, `pc` is the program counter, `depth` is the current level of call nesting, and finally, `stack` contains the current state of the stack. We use single-byte values in the example, but the actual values are 32 bytes (256 bits).

The extracted traces contain a list of executed instructions, as well as the state of the stack at each instruction. To analyze the traces, we encode them into a Datalog representation; Datalog is a language implementing first-order logic with recursion [Imm99], which allows us to concisely express properties about the execution traces. We use the following domains to encode the information about the traces as Datalog facts, noting V as the set of program variables and A as the set of Ethereum addresses. We show an overview of the facts that we collect and the relations that we use to check for possible exploits in Table 5.5. We highlight that our analyses run *automatically* based on facts extracted from transactions traces and the rules that we define in subsequent sections.

Reentrancy

In the EVM, as transactions are executed independently, reentrancy issues can only occur *within* a single transaction. Therefore, for reentrancy to be exploited, there must be a call to an external contract which invokes, directly or indirectly, a re-entrant callback to the calling contract. Therefore, we start by looking for `CALL` instructions in the execution traces, while

Table 5.5: Datalog setup.

(a) Datalog facts.

Fact	**Description**
is_output($v_1 \in V,\ v_2 \in V$)	v_1 is an output of v_2
size($v \in V,\ n \in \mathbb{N}$)	v has n bits
is_signed($v \in V$)	v is signed
in_condition($v \in V$)	v is used in a condition
call($a_1 \in A,\ a_2 \in A,\ p \in \mathbb{N}$)	a_1 calls a_2 with p Ether
create($a_1 \in A,\ a_2 \in A,\ p \in \mathbb{N}$)	a_1 creates a_2 with p Ether
expected_result($v \in V,\ r \in \mathbb{Z}$)	v's expected result is r
actual_result($v \in V,\ r \in \mathbb{Z}$)	v's actual result is r
call_result($v \in V,\ n \in \mathbb{N}$)	v is the result of a call and has a value of n
call_entry($i \in \mathbb{N},\ a \in A$)	contract a is called when program counter is i
call_exit($i \in \mathbb{N}$)	program counter is i when exiting a call to a contract
tx_sstore($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$)	storage key k is written in transaction i of block b
tx_sload($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$)	storage key k is read in transaction i of block b
caller($v \in V, a \in A$)	v is the caller with address a
load_data($v \in V$)	v contains transaction call data
restricted_inst($v \in V$)	v is used by a restricted instruction
selfdestruct($v \in V$)	v is used in SELFDESTRUCT

(b) Datalog rule definitions.

Datalog rules
depends($v_1 \in V,\ v_2 \in V$) :- is_output($v_1,\ v_2$). depends($v_1,\ v_2$) :- is_output($v_1,\ v_3$), depends($v_3,\ v_2$).
call_flow($a_1 \in A,\ a_2 \in A,\ p \in \mathbb{Z}$) :- call($a_1,\ a_2,\ p$). call_flow($a_1 \in A,\ a_2 \in A,\ p \in \mathbb{Z}$) :- create($a_1,\ a_2,\ p$). call_flow($a_1,\ a_2,\ p$) :- call($a_1,\ a_3,\ p$), call_flow($a_3,\ a_2,\ _$).
inferred_size($v \in V,\ n \in \mathbb{N}$) :- size($v,\ n$). inferred_size($v,\ n$) :- depends($v,\ v_2$), size($v_2,\ n$).
inferred_signed($v \in V$) :- is_signed(v). inferred_signed(v) :- depends($v,\ v_2$), is_signed(v_2).
condition_flow($v \in V,\ v \in V$) :- in_condition(v). condition_flow($v_1,\ v_2$) :- depends($v_1,\ v_2$), in_condition(v_2).
depends_caller($v \in V$) :- caller($v_2,\ _$), depends(v, v_2).
depends_data($v \in V$) :- load_data($v_2,\ _$), depends(v, v_2).
caller_checked($v \in V$) :- caller($v_2, _$), condition_flow(v_2, v_3), $v_3 < v$.

(c) Datalog queries for detecting different vulnerability classes.

Vulnerability	**Query**
Reentrancy	call_flow($a_1,\ a_2,\ p_1$), call_flow($a_2,\ a_1,\ p_2$), $a_1 \neq a_2$
Unhandled Excep.	call_result($v,\ 0$), $\neg$condition_flow($v, _$)
Transaction Order Dependency	tx_sstore(b, t_1, i), tx_sload(b, t_2, i), $t_1 \neq t_2$
Locked Ether	call_entry(i_1, a), call_exit(i_2), $i_1 + 1 = i_2$
Integer Overflow	actual_result($v,\ r_1$), expected_result($v,\ r_2$), $r_1 \neq r_2$
Unrestricted Action	restricted_inst(v), depends_data(v), $\neg$depends_caller(v), $\neg$caller_checked(v) $\vee$ selfdestruct(v), $\neg$caller_checked(v)

Code Listing 5.2: Failure handling in Solidity.

```
if (!addr.send(100)) { throw; }
```

Code Listing 5.3: EVM instructions for failure handling.

```
; preparing call
(0x65) CALL
; call result pushed on the stack
(0x69) PUSH1 0x73
(0x71) JUMPI ; jump to 0x73 if call was successful
(0x72) REVERT
(0x73) JUMPDEST
```

Figure 5.4: Correctly handled failed `send`.

keeping track of the contract currently being executed.

When `CALL` is executed, the address of the contract to be called as well as the value to be sent can be retrieved by inspecting the values on the stack [Woo14]. Using this information, we can record `call`(a_1, a_2, p) facts described in Table 5.5a. We note that a contract can also create a new contract using `CREATE` and execute a reentrancy attack using it [Rod+19]. Therefore, we treat this instruction similarly as `CALL`. Using these, we then use the query shown in Table 5.5c to retrieve potentially malicious re-entrant calls.

Analysis correctness. Our analysis for re-entrant calls is sound but not complete. As the EVM executes each contract in a single thread, a re-entrant call must come from a recursive call. For example, given A, B, C and D being functions, a re-entrant call could be generated with a call path such as $A \rightarrow B \rightarrow C \rightarrow A$. Our tool searches for all mutually-recursive calls; it supports an arbitrarily-long calls path by using a recursive Datalog rule, making the analysis sound. However, we have no way of assessing if a re-entrant call is malicious or not, which can lead to false positives.

Unhandled Exceptions

When Solidity compiles contracts, methods to send Ether, such as `send`, are compiled into the EVM `CALL` instructions. We show an example of such a call and its instructions counterpart in Code Listing 5.3. If the address passed to `CALL` is an address, the EVM executes the code of

the contract, otherwise, it executes the necessary instructions to transfer Ether to the address. When the EVM is done executing, it pushes either 1 on the stack, if the `CALL` succeeded, or 0 otherwise.

To retrieve information about call results, we can therefore check for `CALL` instructions and use the value pushed on the stack after the call execution. The end of the call execution can be easily found by checking when the `depth` of the trace turns back to the value it had when the `CALL` instruction was executed; we save this information as `call_result`(v, n) facts. An important edge case to consider are calls to pre-compiled contracts, which are typically called by the compiler and do not require their return value to be checked, as they are results of a computation where 0 could be a valid value, but could result in false positives. As pre-compiled contracts have known addresses between 1 and 10, we choose to simply not record `call_result` facts for such calls.

As shown in Code Listing 5.3, the EVM uses the `JUMPI` instruction to perform conditional jumps. At the time of writing, this is the only instruction available to execute conditional control flow. We, therefore, mark all the values used as a condition in `JUMPI` as `in_condition`. We can then check for the unhandled exceptions by looking for call results, which never influence a condition using the query shown in Table 5.5c.

Analysis correctness. The analysis we perform to check for unhandled exceptions is complete but not sound. All failed calls in the execution of the program will be recorded, while we accumulate facts about the execution. We then use a recursive Datalog rule to check if the call result is used directly or indirectly in a condition. We could obtain false negatives if the call result is used in a condition but the condition is not enough to prevent an exploit. However, given that the most prevalent pattern for this vulnerability is the result of `send` not being used at all [Tsa+18], and when the result is used, it is typically done within a `require` or `assert` expression, we hypothesize that such false negatives should be very rare.

Locked Ether

Although there are several reasons for funds being locked in a contract, we focus on the case where the contract relies on an external contract which does not exist anymore, as this is the pattern which had the largest financial impact on Ethereum [Bre+17]. Such a case can occur when a contract uses another contract as a library to perform some actions on its behalf. To use a contract in this way, the `DELEGATECALL` instruction is used instead of `CALL`, as the latter does not preserve call data, such as the sender or the value.

The next important part is the behaviour of the EVM when trying to call a contract which does not exist anymore. When a contract is destructed, it is not completely removed per se, but its code is not accessible anymore to callers. When a contract tries to call a contract which has been destructed, the call is a no-op rather than a failure, which means that the next instruction will be executed and the call will be marked as successful. To find such patterns, we collect Datalog facts about all the values of the program counter before and after every `DELEGATECALL` instruction. In particular, we first mark the program counter value at which the call is executed — `call_entry`($i_1 \in \mathbb{N},\ a \in A$). Then, using the same approach as for unhandled exceptions, we skip the content of the call and mark the program counter value at which the call returns — `call_exit`($i_2 \in \mathbb{N}$).

If the called contract does not exist anymore, $i_1 + 1 = i_2$ must hold. Therefore, we can use the Datalog query shown in Table 5.5c to retrieve the destructed contracts address.

Analysis correctness. The approach we use to detect locked Ether is sound and complete for the class of locked funds vulnerability we focus on. All vulnerable contracts must have a `DELEGATECALL` instruction. If the issue is present and the call contract has indeed been destructed, it will always result in a no-op call. Our analysis records all of these calls and systematically checks for the program counter before and after the execution, making the analysis sound and complete.

Transaction Order Dependency

The first insight to check for exploitation of transaction ordering dependency is that at least two transactions to the same contract must be included in the same block for such an attack to be successful. Furthermore, as shown in [Luu+16a] or [Tsa+18], exploiting a transaction ordering dependency vulnerability requires manipulation of the contract's storage.

The EVM has only one instruction to read from the storage, `SLOAD`, and one instruction to write to the storage, `SSTORE`. In the EVM, the location of the storage to use for both of these instructions is passed as an argument and referred to as the storage *key*. This key is available on the stack at the time the instruction is called. We go through all the transactions of the contracts and each time we encounter one of these instructions, we record either `tx_sload`($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$) or `tx_sstore`($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$) where in each case b is the block number, i is the index of the transaction in the block and k is the storage key being accessed.

The essence of the rule to check for transaction order dependency issues is then to look for patterns where at least two transactions are included in the same block with one of the transactions writing a key in the storage and another transaction reading the same key. We show the actual rule in Table 5.5c.

Analysis correctness. Our approach used to detect transaction order dependencies is sound but not complete. With the definition we use, for a contract to have a transaction order dependency, it must have two transactions in the same block, which affect the same key in the storage. We check for all such cases, and therefore no false negatives can exist. However, finding if there is a transaction order dependency requires more knowledge about how the storage is used and our approach could therefore result in false positives.

Integer Overflow

The EVM is completely untyped and expresses everything in terms of 256-bit words. Therefore, types are handled entirely at the compilation level and there is no explicit information about

the original types in any execution traces.

To check for integer overflow, we accumulate facts over two passes. In the first pass, we try to recover the sign and size of the different values on the stack. To do so, we use known invariants about the Solidity compilation process. First, any value which is the result of an instruction such as `SIGNEXTEND` or `SDIV` can be marked to be signed with `is_signed`(v). Furthermore, `SIGNEXTEND` being the usual sign extension operation for two's complement, it is passed both the value to extend and the number of bits of the value. This allows us to retrieve the size of the signed value. We assume any value not explicitly marked as signed to be unsigned. To retrieve the size of unsigned values, we use another behaviour of the Solidity compiler.

To work around the lack of type in the EVM, the Solidity compiler inserts an `AND` instruction to "cast" unsigned integers to their correct value. For example, to emulate an `uint8`, the compiler inserts `AND value 0xff`. In the case of a "cast", the second operand m will always be of the form $m = 16^n - 1,\ n \in \mathbb{N},\ n = 2^p,\ p \in [1, 6]$. We use this observation to mark values with the according type: `uintN` where $N = n \times 4$. Variables size are stored as `size`($v,\ n$) facts.

During the second phase, we use the `inferred_signed`(v) and `inferred_size`($v,\ n$) rules shown in Table 5.5b to retrieve information about the current variable. When no information about the size can be inferred, we over-approximate it to 256 bits, the size of an EVM word. Using this information, we compute the expected value for all arithmetic instructions (e.g. `ADD`, `MUL`), as well as the actual result computed by the EVM and store them as Datalog facts. Finally, we use the query shown in Table 5.5c to find instructions which overflow.

Analysis correctness. Our analysis for integer overflow is neither sound nor complete. The types are inferred by using properties of the compiler using a heuristic which should work for most cases but can fail. For example, if a contract contains code which yields `AND value 0xff` but `value` is an `uint32`, our type inference algorithm would wrongly infer that this variable is an `uint8`. Such errors during type inference could cause both false positives and false negatives. However, this type of issue occurs only when the developer uses bit manipulation with a mask similar to what the Solidity compiler generates. We find that such a pattern is rare enough not to skew our data, and give an estimate of the possible number of contracts which could follow

such a pattern in Section 5.2.5.

Unrestricted Action

Unrestricted actions are a broad class of vulnerability, as they can include the ability to set an owner without being allowed to, destruct a contract without permission or yet execute arbitrary code. As one of our main goals is to check the exploitation of vulnerable contracts, we stay close to the definitions given by previous works [KR18] and focus on unrestricted Ether transfer using `CALL`, unrestricted writes using and `SSTORE`, and code injection using `DELEGATECALL` or `CALLCODE`.

First, we need to remind ourselves that the caller, unlike for example the call data, cannot be forged. Therefore, one of the main insights is that if an action is restricted depending on who is calling, there should be an execution trace before the restricted operation which conditionally jumps, depending on the caller. This is enough for `SELFDESTRUCT` but not for other instructions as it would flag a line such as `balances[msg.sender] = msg.value` to be vulnerable. To model this, we track whether the message sender influences the storage key or the address to call. Finally, for code injection, we check whether the passed data influences the address called by `DELEGATECALL` or `CALLCODE`.

Analysis correctness. Our analysis for unrestricted actions is neither sound nor complete. We take a relatively simple approach of checking whether the message sender influences a condition or not before executing a sensitive instruction. This can result in false negatives because the check could be performed inappropriately, for example not reverting the transaction when needed, making the analysis unsound. Furthermore, there might be some use cases where it is acceptable to allow any sender to write to the storage, but our analysis would flag such as vulnerable, resulting in false positives. We discuss the implications further in Section 5.2.5.

Table 5.6: RE: Top contracts victim of reentrancy attack and ETH amounts exploited

Contract address	Last transaction	Amount exploited
`0xd654bdd32fc99471455e86c2e7f7d7b6437e9179`	2016-06-10	5,885
`0x675e2c143295b8683b5aed421329c4df85f91b33`	2015-12-31	50.49
`0xcd3e727275bc2f511822dc9a26bd7b0bbf161784`	2017-03-25	10.34

5.2.5 Analysis of Individual Vulnerabilities

As described in Section 5.2.3, the combined amount of Ether contained within *all* the vulnerable contracts exceeds 3 million ETH, worth 6,000 million USD. In this section, we present the results for each vulnerability one by one; our results have been obtained using the methodology described in Section 5.2.4; the goal is to show how much of this money is actually at risk.

Methodology. For each vulnerability, we perform our analysis in two steps. First, we fetch the execution traces of all the transactions up to block 10,200,000 affecting the contracts in our dataset, either directly or through internal transactions. We then run our tool to automatically find the total amount of Ether at risk and report this number. This is the amount we use to later give a total upper bound across all vulnerabilities. In the second step, we manually analyze the contracts at risk to obtain more insight into the exploits and find interesting patterns. As analysing all the contracts manually would be impractical, for each vulnerability we manually analyze the contracts with the highest amount of Ether at risk to understand better the reasons behind the vulnerabilities. We then present interesting findings as short case studies.

Runtime performance. Our analysis runs in linear time and memory with respect to the number of instructions executed by a given transaction. The number of instructions varies widely between transactions, anywhere from a few hundred to a few hundred thousand, with an average of around 100,000. Our tool takes on average less than 10ms (stddev. 20ms) per transaction with a maximum of less than 2 seconds for the largest transactions, which is below the timeout of 5 seconds which we set for a single transaction.

RE: Reentrancy

There are 4,337 contracts flagged as vulnerable to reentrancy by [Luu+16a; Tsa+18; Kal+18], with a total of 457,073 transactions. After running the analysis described in Section 5.2.4 on all the transactions, we found a total of 116 contracts which contain re-entrant calls. To look for the monetary amount at risk, we compute the sum of the Ether sent between two contracts in transactions containing re-entrant calls. The total amount of Ether exploited using reentrancy is of 6,076 ETH, which is considerable, as it represents more than 12,000,000 USD.

Manual analysis. We manually analyze the top contracts in terms of funds lost and present them in Table 5.6. Interestingly, one of these three potential exploits has a substantial amount of Ether at stake: 5,881 ETH, which corresponds to around 11,800,000 USD. This address has already been detected as vulnerable by some recent work focusing on reentrancy [Rod+19]. It appears that the contract, which is part of the Maker DAO [19e] platform, was found vulnerable by the authors of the contract, who themselves performed an attack to confirm the risk [16].

Sanity checking. We use two different contracts for sanity checking. First, we look at TheDAO attack, which is the most famous instance of a reentrancy attack. Our tool detects the following reentrancy pattern: the malicious account calls TheDAO main account, TheDAO main account calls into the reward account and the reward account sends the reward to the malicious account, allowing it to perform the re-entrant call into TheDAO main account.

As another sanity check, we look at a contract called SpankChain [18], which is known to recently have been compromised by a reentrancy attack. We confirm that our approach successfully marks this contract as having been the victim of a reentrancy attack and correctly identifies the attacker contract.

Finally, we note that our tool finds all the reentrancy patterns presented by Sereum [Rod+19], including delegated and create-based reentrancy[2].

[2] https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns

Table 5.7: UE: Top contracts affected by unhandled exceptions and ETH amounts at risk

Contract address	Amount at risk
0x7011f3edc7fa43c81440f9f43a6458174113b162	56.70
0xb336a86e2feb1e87a328fcb7dd4d04de3df254d0	42.27
0xdcabd383a7c497069d0804070e4ba70ab6ecdd51	33.44
0xfd2487cc0e5dce97f08be1bc8ef1dce8d5988b4d	21.60
0x9e15f66b34edc3262796ef5e4d27139c931223f0	10.50

UE: Unhandled Exceptions

There are 11,427 contracts flagged vulnerable to unhandled exceptions by [Tsa+18; Luu+16a; Kal+18] for a total of more than 3.4 million transactions, which is *an order of magnitude* larger than what we found for reentrancy issues.

We find a total of 264 contracts where failed calls have not been checked for, which represents roughly 2% of the flagged contracts. The next goal is to find an upper bound on the amount of Ether at risk because of these unhandled exceptions. We define the upper bound on the money at risk to be the minimum value of the balance of the contract at the time of the unhandled exception and the total of Ether which have failed to be sent. We then sum the upper bound of all issues found to obtain a total upper bound. This gives us a total of 271.89 Ether at risk for unhandled exceptions.

Manual analysis. We manually analyze the top contracts and summarize their addresses and the amount at risk in Table 5.7. The Solidity code is available for three of these contracts. We confirm that in all cases the issue came from misuse of a low-level Solidity function such as `send`.

Investigation of the contract at

`0x7011f3edc7fa43c81440f9f43a6458174113b162`:

The contract `0x7011f3edc7fa43c81440f9f43a6458174113b162` has failed to send a total of 52.90 Ether and currently still holds a balance of 69.3 Ether at the time of writing. After investigation,

we find that the contract is an abandoned pyramid scheme [17e]. The contract has a total of 21 calls which failed, all trying to send 2.7 Ether, which appears to have been the reward of the pyramid scheme at this point in time. Unfortunately, the code of this contract was not available for further inspection but we conclude that there is a high chance that some of the users in the pyramid scheme did not correctly obtain their reward because of this issue.

LE: Locked Ether

There are 7,285 contracts flagged vulnerable to locked Ether by [Tsa+18], [Gre+18], [Nik+18] and [Kal+18]. The contracts hold a total value of more than 1.4 million ETH, which is worth more than 2,000 million USD. We analyze the transactions of the contracts that could potentially be locked by conducting the analysis described in the previous section. Our tool shows that *none* of the contracts are affected by the pattern we check for — i.e., dependency on a contract that had been destructed. We note that our tool currently only covers dependency on a destructed contract as a reason for locked Ether and patterns such as unbounded mass operation are not yet covered.

Parity wallet. Contracts affected by the Parity wallet [3] bug [Bre+17], which our tool should flag as locked Ether, were not flagged by the tools we analysed, and are therefore not present in our dataset. As this is one of the most famous cases of locked Ether, we test our tooling on the contracts affected by the Parity wallet bug. To find the contracts, we simply have to use the Datalog query for locked Ether in Table 5.5c and insert the value of the Parity wallet address as argument a. Our results for contracts affected by the Parity bug indeed match what others had found in the past [Gal17], with the contract at address `0x3bfc20f0b9afcace800d73d2191166ff16540258` having as much as 306,276 ETH locked.

[3] Parity wallet Address: `0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`

Table 5.8: TOD: Top contracts potentially victim of transaction ordering dependency attack.

Contract address	First issue	Balance
0x3da71558a40f63b960196cc0679847ff50fad22b	2016-09-06	13,818
0xd79b4c6791784184e2755b2fc1659eaab0f80456	2016-05-03	2,013
0xf45717552f12ef7cb65e95476f217ea008167ae3	2016-03-15	1,064

Table 5.9: Understanding the exploitation of potentially vulnerable contracts.

Vulnerable				Exploited contracts		Exploited Ether	
Vuln.	**Vuln. contracts**	**Total Ether at stake**	**Transactions analysed**	**Contracts exploited**	**% of contracts exploited**	**Exploited Ether**	**% of Ether exploited**
RE	4,337	1,518,067	457,073	116	2.68%	6,076	0.40%
UE	11,427	419,418	3,400,960	264	2.31%	271.9	0.068%
LE	7,285	1,416,086	10,660,066	0	0%	0	0%
TO	1,881	302,679	3,002,304	54	3.72%	297.2	0.091%
IO	2,492	602,980	1,295,913	62	2.49%	1,842	0.31%
UA	5,163	580,927	3,871,770	42	0.813%	0	0%
Total	23,327	3,124,433	20,241,730	463	1.98%	8,487	0.27%

TO: Transaction Order Dependency

There are 1,881 contracts flagged vulnerable to transaction ordering dependency by [Luu+16a] and [Kal+18]. We run the analysis described in Section 5.2.4 on their 3,002,304 transactions and obtain a total of 54 contracts potentially affected by transaction-order dependency. To estimate the amount of Ether at risk, we sum up the total value of Ether sent, including by internal transactions, during all the flagged transactions, resulting in a total of 297.2 ETH at risk of transaction-order dependency.

Manual analysis. For each contract, we find the block where transaction order dependency could have happened with the highest balance and report top with their balance at the time of the issue in Table 5.8. We manually investigated the contracts listed, they all had their source code available. We confirmed that in all the contracts, a user could read and write to the same storage location within a single block. We inspected further `0x3da71558a40f63b960196cc0679847ff50fad22b`, a contract called `WithDrawChildDAO` and found that the read was simply for users to check their balance, making the issue benign.

IO: Integer Overflow

There are 2,472 contracts flagged vulnerable to integer overflow, which accounts for a total of more than 1.2 million transactions. We run the approach we described in Section 5.2.4 to search for actual occurrences of integer overflows. It is worth noting that for integer overflow analysis we rely on the properties of the Solidity compiler. To ensure that the contracts we analyze were compiled using Solidity, we fetched all the available source codes for contracts flagged vulnerable to integer overflow from Etherscan [23b]. Out of 2,492 contracts, 945 had their source code available and all of them were written in Solidity.

Effects of our formulation. As mentioned in Section 5.2.4, some types of bit manipulation in Solidity contracts could result in our type inference heuristic failing. We use the source codes we collected here to verify to what extent this could affect our analysis. We find that bit manipulation by itself is already fairly rare in Solidity, with only 244 out of the 2,492 contracts we collected using any sort of bit manipulation. Furthermore, most of the contracts using bit manipulation were using it to manipulate a variable as a bit array, and only ever retrieved a single bit at a time. Such a pattern does not affect our analysis. We found only 33 contracts which used `0xFF` or similar values, which could actually affect our analysis. This represents about 1.3% of the number of contracts for which the source code was available.

We find a total of 62 contracts with transactions where an integer overflow might have occurred. To find the amount of Ether at stake, we analyze all the transactions which resulted in integer overflows. We approximate the amount by summing the total amount of Ether transferred in and out during a transaction containing an overflow. We find that the total Ether at stake is 1,842 ETH. This is most likely an over-approximation but we use this value as our upper bound.

Manual analysis. We inspect some of the results we obtained a little further to get a better sense of what kind of cases lead to overflows. We find that a very frequent cause of overflow is rather an underflow of unsigned values. We highlight one such case in the following investigation.

Investigation of the contract at

`0xdcabd383a7c497069d0804070e4ba70ab6ecdd51`:

This contract was flagged positive to both unhandled exceptions and integer overflow by our tool. After inspection, it seems that at block height 1,364,860, the owner tried to reduce the fees but the unsigned value of the fees overflowed and became a huge number. Because of this issue, the contract was then trying to send large amounts of Ether. This resulted in failed calls which happened not to be checked, hence the flag for unhandled exceptions.

Unrestricted Action

There is a total of 5,163 contracts flagged by [Tsa+18; Nik+18; KR18] as vulnerable to unrestricted actions for a total of 3,871,770 transactions. We use the approach described in Section 5.2.4 and find a total of 42 contracts having suffered unrestricted actions, which were all non-restricted self-destructs, but none of them held Ether at the time of the exploit.

Effects of our formulation. As mentioned in Section 5.2.4, this analysis is not sound, which means we need to be cautious about false positives. A contract could have a check on the message sender which is incorrect and be exploited but not be flagged as such. While we hypothesize that it is an edge case, it cannot be completely excluded. However, having an automation method for such a check requires knowing the intent of the programmer, for example through specifications, which is out-of-scope of this work. Therefore, we decide to inspect the contracts in our dataset in more detail to understand better the level of exploitation.

Manual analysis. The tool teEther flags *exploitable* contracts, as opposed to simply *vulnerable* contracts. Therefore, expect these contracts to be more likely to have been exploited and focus on these for our manual analysis. We fetch all the historical balances of teEther contracts and retrieve the maximum amount held at any point in time and find the total of these to equal

to 4,921 Ether. However, we find that 4,867 Ether belonged to 48 different contracts with the same bytecode, and all had the same transaction pattern, which we describe in the following investigation.

Investigation of the contract at
`0xac54413f686927054a56d35415ba49618634e105`:

All contracts with a high historical monetary value found by teEther share the same bytecode, creator and transaction pattern as this contract. The contracts are created by `0x15f889d2469d1be0e0699632d8d448f2178a7afe`, receive Ether from Kraken, an exchange, and send the same amount to `0xd1bf1706306c7b667c67ffb5c1f76cc7637685bd` a couple of blocks later. We could not find further information about these addresses. We decompile the contract to understand how the contracts were exploitable and find that during the few blocks they held money, exploiting the contract would have been as simple as sending a transaction with the address to which to transfer the funds as an argument. This is a very dangerous situation but because the Ether was usually sent within a minute to another address, an attacker would have needed to be very proactive and use advanced tooling to exploit the contract. This illustrates well how a contract can be *exploitable* but have little chance of being *exploited* in practice.

Sanity checking. As a sanity check, in addition to our test suite, we use one of the most famous instances of an unrestricted action: the destructed Parity wallet library contract at address `0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`. We analyze the transactions and successfully find an unrestricted store instruction in transaction `0x05f71e1b`, which was used to take control of the wallet.

Summary

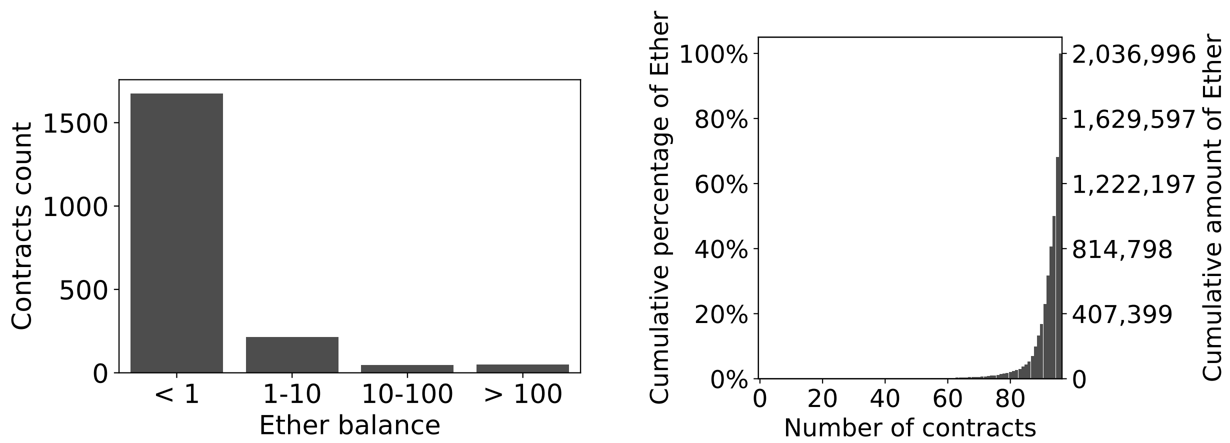
We summarize our findings, including the number of contracts originally flagged, the amount of Ether at stake, and the amount *actually exploited* in Table 5.9. The *Contracts exploited* column indicates the number of contracts that are flagged exploited and *% Contracts exploited* is the percentage of this number with respect to the number of contracts flagged vulnerable. The *Exploited Ether* column shows the maximum amount of Ether that could have been exploited and the next column shows the percentage this amount represents compared to the total amount at stake. The *Total* row accounts for contracts flagged with more than one vulnerability only once.

Overall, we find that the *number of contracts exploited* is non-negligible, with about 2% to 4% of vulnerable contracts exploited for 4 of the 6 vulnerabilities covered in our study. However, it is important to note that the percentage of Ether exploited is an order of magnitude lower, with at most 0.4% of the Ether at stake exploited for reentrancy. This indicates that exploited contracts are usually low-value. We will expand on this argument further in Section 5.2.7.

5.2.6 Limitations

In this section, we present the different limitations of our system and describe how we try to mitigate them.

Soundness vs Completeness. As for most tools such as this one, we are faced with the trade-off of soundness against completeness. Whenever possible we choose soundness over completeness — three out of six of our analyses are sound. When we cannot have a sound analysis, we are careful to only leave out cases which are unlikely to generate many false negatives. In other words, we try as much as possible to reduce the number of false negatives, even if this means increasing the number of false positives. Indeed, the main goal of our system is to provide us with an upper bound of the amount of potentially exploited Ether, which makes false negatives undesirable. Furthermore, we manually check the high-value contracts flagged as exploited, which assures us that even if false positives were flagged, they will not have an



(a) Ether held by contracts in our dataset with non-zero balance.

(b) Cumulative Ether held in the 96 contracts in our dataset containing at least 10 ETH.

Figure 5.5: Ether held in contracts: describing the distribution.

important influence on the final results. As an example of this trade-off, for reentrancy, we flag any contract which was called using a re-entrant call and lost funds in the process. However, in some cases, it could be an expected behaviour and the funds could have been transferred to an address belonging to the same entity.

Dataset. We only run our tool on the contracts included in our dataset, which means that we might be missing some exploits which occurred. Indeed, we did not have any contract affected by the Parity wallet bug nor had we the contract affected by TheDAO hack in the dataset. However, one of the main goals of this section is to quantify what fraction of vulnerabilities discovered by analysis tools is exploited in practice and our current approach allows us to do exactly this.

Other types of attacks. Our tool and analysis do not cover every existing attack on smart contracts. There are, for example, attacks targeting ERC-20 tokens [REC19], or yet some other types of DoS attacks, such as wallet griefing [Gre+18]. Furthermore, some detected “exploits” could be the results of Honeypots [TSS19] but our tool does not cover such cases. Although it would be interesting to also cover such cases, we had to decide the scope of the tool. Therefore, we focus on the vulnerabilities which have been the most covered in the literature, which we hypothesise is representative of how common the vulnerabilities are.

5.2.7 Discussion

Even considering the limitations of our system, it is clear that the exploitation of smart contracts is vastly lower than what could be expected. In this section, we present some of the factors we think might be impacting the actual exploitation of smart contracts.

We believe that a major reason for the difference between the number of vulnerable contracts reported and the number of contracts exploited is the distribution of Ether among contracts. Indeed, only about 2,000 out of the 23,327 contracts in our dataset contain Ether, and most of these contracts have a balance lower than 1 ETH. We show the balance distribution of the contracts containing Ether in our dataset in Figure 5.5a. Furthermore, the top 10 contracts hold about 95% of the total Ether. We show the cumulative distribution of Ether within the contracts containing more than 10 ETH in Figure 5.5b. This shows that, as long as the top contracts cannot be exploited, the total amount of Ether that is actually at stake will be nowhere close to the upper bound of "vulnerable" Ether.

To make sure this fact generalizes to the whole Ethereum blockchain and not only our dataset, we also fetch the balances for all existing contracts. This gives a total of 15,459,193 contracts. Out of these, we find that only 463,538 contracts have a non-zero balance, which is merely 3% of all the contracts. Out of the contracts with a non-zero balance, the top 10 contracts account for 54% of the total amount of Ether, the top 100 for 92% and the top 1000 for 99%. This shows that our dataset follows the same trend as the Ethereum blockchain in general: a very small amount of contracts hold most of the wealth.

Manual inspection of high-value contracts. We decide to manually inspect the top 6 contracts — i.e contracts with the highest balances at the time of writing — marked as vulnerable by any of the tools in our dataset. We focused on the top 6 because it happened to be the number of contracts which currently hold more than 100,000 ETH. These contracts hold a total of 1,695,240 ETH, or 83% of the total of 2,037,521 ETH currently held by all the contracts in our dataset.

Table 5.10: Top six most valuable contracts flagged as vulnerable by at least one tool.

Address	ETH balance	Deployed	Flagged Vulnerabilities
`0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae`	649,493	2015-08-08	Oyente: RE
`0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9`	369,023	2016-11-10	MadMax: LE, Zeus: IO
`0x851b7f3ab81bd8df354f0d7640efcd7288553419`	189,232	2017-04-18	MadMax: LE
`0x07ee55aa48bb72dcc6e9d78256648910de513eca`	182,524	2016-08-08	Securify: RE
`0xcafe1a77e84698c83ca8931f54a755176ef75f2c`	180,300	2017-06-04	MadMax: LE
`0xbf4ed7b27f1d666546e30d74d50d173d20bca754`	124,668	2016-07-16	Securify: TO, UE; Zeus: LE, IO

Investigation of the contract at

`0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae`:

The source code for this contract is not available on Etherscan. However, we discovered that this is the multi-signature wallet of the Ethereum foundation [15] and that its source code is available on GitHub [17c]. We inspect the code and find that the only calls taking place require the sender of the message to be an owner. This by itself is enough to prevent any re-entrant call, as the malicious contract would have to be an owner, which does not make sense. Furthermore, although the version of Oyente used in the paper reported the reentrancy, more recent versions of the tool did not report this vulnerability anymore. Therefore, we safely conclude that the reentrancy issue was a false alert.

We were able to inspect 4 of the 5 remaining contracts. The contract at address `0x07ee55aa48bb72dcc6e9d78256648910de513eca` is the only one for which we were unable to find any information. The second, third and fifth contracts in the list were also multi-signature wallets and exploitation would require a majority owner to be malicious. For example, for Ether to get locked, the owners would have to agree on adding enough extra owners so that all the loops over the owners result in an out-of-gas exception. The contract at address `0xbf4ed7b27f1d666546e30d74d50d173d20bca754` is a contract known as `WithDrawDAO` [17d]. We did not find any particular issue, but it does use a delegate pattern which explains the locked Ether vulnerability marked by Zeus.

Overall, all the contracts from Table 5.10 that we could analyze seemed quite secure and the vulnerabilities flagged were not exploitable. Although there are some very rare cases that we present in Section 5.2.8 where contracts with high Ether balances are being stolen, these remain exceptions. The facts we presented up to now help us confirm that the amount of Ether at risk on the Ethereum blockchain is nowhere as close as what is claimed [Kal+18; Gre+18]. We now present a thorough investigation of the high-value contracts.

Investigation of the contract at
`0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae`:

This contract has been flagged as being vulnerable to reentrancy by Oyente. For a contract to be a victim of a reentrancy attack, it must `CALL` another contract, sending it enough gas to be able to perform the re-entrant call. In Solidity terms, this means that the contract has to invoke `address.call` and not explicitly set the gas limit. By looking at the source code [17c], we find 2 such instances: one at line 352 in the `execute` function and another at line 369 in the `confirm function`. The `execute` is protected by the `onlyowner` modifier, which requires the caller to be an owner of the wallet. This means that for a re-entrant call to work, the malicious contract would need to be one of the owners of the wallet in order to work. The `confirm` function is protected by the `onlymanyowners` modifier, which requires at least `n` owners to agree on confirming a particular transaction before it is executed, where `n` is agreed upon at contract creation time. Furthermore, `confirm` will only invoke `address.call` on a transaction previously created in the `execute` function.

Investigation of the contract at
`0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9`:

This is the contract for the multi-signature wallet of the Golem project [19d] and uses a well-

known multi-signature implementation. We use the source code available on Etherscan to perform the audit. This contract is marked with two vulnerabilities, locked Ether by MadMax and integer overflow by Zeus.

We first focus on the locked Ether which is due to an unbounded mass operation [Gre+18]. An unbounded mass operation is flagged when a loop is bounded by a variable whose value could increase, for example, the length of an array. This is because if the number of iterations becomes too large the contract would run out of gas every time, which could indeed result in locked funds. All the loops except two of them are bound by the total number of owners. As owners can only be added if enough existing owners agree, running out of gas when looping on the number of owners cannot happen unless the owners agree. The two other loops are part of the `filterTransactions` that loops over the total number of transactions. However, this function is only used by two external getters, `getPendingTransactions` and `getExecutedTransactions` and could therefore not result in the Ether getting locked. In the worst case, these getters could become unusable, which would not be a security issue. Nevertheless, this is indeed an issue that should be fixed, most likely by limiting the maximum number of transactions that can be retrieved by `getPendingTransactions` and `getExecutedTransactions`.

Next, we look for possible integer overflows. All loops discussed above use an `uint` as a loop index. In Solidity, `uint` is a `uint256` which makes it impossible to overflow here, given that neither the number of owners nor transactions could ever reach such a number. The only other arithmetic operation performed is `owners.length - 1` in the function `removeOwner` at line 103. This function checks that the owner exists, which means that `owners.length` will always be at least 1 and `owners.length` can therefore never underflow.

Investigation of the contract at

`0x851b7f3ab81bd8df354f0d7640efcd7288553419`:

This contract is also a multi-sig wallet, this time owned by Gnosis Ltd.[4] We use the source code available on Etherscan to perform the audit. The contract looks very similar to the one used by `0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9` and has also been marked by MadMax as being vulnerable to locked Ether because of unbounded mass operations. Again, we look at all the loops in the contract and find that as in the previous contract, it loops exclusively on owners and transactions. As in the previous contract, we assume looping on the owners is safe and look at the loops over the transactions. This contract has two functions looping over transactions, `getTransactionCount` at line 303 and `getTransactionIds` at line 351. Both functions are getters which are never called from within the contract. Therefore, no Ether could ever be locked because of this. Unlike the previous contract, `getTransactionIds` allows to set the range of transactions to return, therefore making the function safe to unbounded mass operations. However, `getTransactionCount` does loop over all the transactions, and as before, could therefore become unusable at some point, although it is highly unlikely.

Investigation of the contract at

`0xcafe1a77e84698c83ca8931f54a755176ef75f2c`:

This contract is again a multi-sig wallet, this time owned by the Aragon project[5]. We also use the contract published on Etherscan for the audit. It appears that the source code for this contract is exactly the same as the one of `0x851b7f3ab81bd8df354f0d7640efcd7288553419` except that it is missing a contract called `MultiSigWalletWithDailyLimit`. This contract was also flagged as being at risk of unbounded mass operations by MadMax, the conclusions are therefore exactly the same as for the previous contract.

---

[4] https://gnosis.io/

[5] https://aragon.org/

Investigation of the contract at

`0xbf4ed7b27f1d666546e30d74d50d173d20bca754`:

This contract is the only one which is very different from the previous ones. It is the `WithdrawDAO` contract, which has been created for users to get their funds back after TheDAO incident [SC17]. We use the source code from Etherscan to audit the contract. This contract has been flagged with several vulnerabilities: Securify flagged it with transaction order dependency and unhandled exception, while Zeus flagged it with locked ether and integer overflow. The contract has two very short functions: `withdraw` which allows users to convert their TheDAO tokens back to Ether, and the `trusteeWithdraw` which allows sending funds which cannot be withdrawn by regular users to a trusted address. We first look at the transaction order dependency. As any user will only ever be able to receive the total amount of tokens he possesses, the order of the transaction should not be an issue in this contract. We then look at unhandled exceptions. There is indeed a call to `send` in the `trusteeWithdraw` which is not checked. Although it is not particularly an issue here, as this does not modify any other state, an error should probably be thrown if the call fails. We then look at locked ether. The contract is flagged with locked ether because of what Zeus classifies as a "failed send". This issue was flagged because if the call to `mainDAO.transferFrom` would always revert, then the call to `msg.sender.send` would never be reached, indeed preventing from reclaiming funds. However, in this context, `mainDAO` is a trusted contract and it is, therefore, safe to assume that `mainDAO.transferFrom` will not always fail. Finally, we look at the integer overflow issue. The only place where an overflow could occur is in `trusteeWithdraw` at line 23. This could indeed overflow without some assumptions on the different values. For this particular contract, the following assumptions are made.

```
this.balance+mainDAO.balanceOf(this)≥mainDAO.totalSupply()
mainDAO.totalSupply()>mainDAO.balanceOf(this)
```

As long as these assumptions hold, which was the case when the contract was deployed, this expression will never overflow. Indeed, if we note t the time before the first call to `trusteeWithdraw` and $t+1$ the time after the first call, we will have

```
this.balance_{t+1} = this.balance_t - (
  this.balance_t + mainDAO.balanceOf(this)
                 - mainDAO.totalSupply())
 = -mainDAO.balanceOf(this)+mainDAO.totalSupply()
```

which means that every subsequent call will compute the following.

```
this.balance_{t+1} + mainDAO.balanceOf(this) -
                          mainDAO.totalSupply()
 = -mainDAO.balanceOf(this)+mainDAO.totalSupply()+
   mainDAO.balanceOf(this) - mainDAO.totalSupply()
 = 0
```

This will always result in sending 0 and will therefore not cause any overflow. If some money is newly received by the contract, the amount received will be transferred the next time `trusteeWithdraw` is called.

5.2.8 Related Work

Previous work

There have been a lot of efforts in order to prevent exploits and to make smart contracts more secure in general. We will here present some of the tools and techniques which have been presented in the literature and, when relevant, describe how they compare to our work.

Analysis tools can roughly be divided into two categories: static analysis and dynamic analysis tools. Using the term "static" quite loosely, static analysis tools can be defined as tools which

catch bugs or vulnerabilities without the need to deploy the smart contracts. Runtime analysis tools try to detect these by executing the deployed contracts. Our tool fits into the second category.

Static analysis tools. Static analysis tools have been the main focus of research. This is understandable, given how critical it is to avoid vulnerabilities in a deployed contract. Most of these tools work by analysing either the bytecode or the high-level code of the contract and checking for known vulnerable patterns in these.

Oyente [Luu+16a] is one of the first tools which has been developed to analyze smart contracts. It uses symbolic execution in combination with the Z3 SMT solver [DB08] to check for the following vulnerabilities: transaction ordering dependency, reentrancy and unhandled exceptions.

ZEUS [Kal+18] is a static analysis tool which works on the Solidity smart contract and not on the bytecode, making it appropriate to assist development efforts rather than to analyze deployed contracts, for which Solidity code is typically not available. Zeus transpiles XACML-styled [SW13] policies to be enforced and the Solidity contract code into LLVM bitcode [LA04] and uses constrained Horn clauses [Bjø+12; McM07] over it to check that the policy is respected.

Securify [Tsa+18] is a static analysis tool which checks the security properties of the EVM bytecode of smart contracts. The security properties are encoded as patterns written in a Datalog-like [Ull84] domain-specific language and checked either for compliance or violation. Securify infers semantic facts from the contract and interprets the security patterns to check for their violation or compliance by querying the inferred facts. This approach has many similarities with ours, using Datalog to express vulnerability patterns. The major difference is that Securify works on the bytecode directly while our tool works on the execution traces.

MadMax [Gre+18] has similarities with Securify, as it also encodes properties of the smart contract into Datalog, but it focuses on vulnerabilities related to gas. It is the first tool to detect “unbounded mass operations”, where a loop is bounded by a dynamic property such as the number of users, causing the contract to always run out of gas passed a certain number

of users. MadMax is built on top of the decompiler implemented by Vandal [Bre+18] and is performant enough to analyze all the contracts of the Ethereum blockchain in only 10 hours.

Several other static analysis tools have been developed, some, such as SmartCheck [Tik+18], being quite generic and handling many classes of vulnerabilities, and others being more domain-specific, such as Osiris [TS+18] focusing on integer overflows, Maian [Nik+18] focusing on unrestricted actions or Gasper [Che+17b] focusing on costly gas patterns. More recently, ETHBMC [FAH20] was designed to also support inter-contract relations, cryptographic hash functions and memcopy-style operations.

Finally, there have also been some efforts to formally verify smart contracts. [Hir17] is one of the first efforts in this direction and defines the EVM using Lem [Mul+14], which allows generating definitions for theorem provers such as Coq [Bar+97]. [GMS18] presents a complete small-step semantics of EVM bytecode and formalizes it using the F* proof assistant [Swa+11]. A similar effort is made in [Hil+18] to give an executable formal specification of the EVM using the K Framework [RŞ10]. VerX [Per+19] is also a recent work allowing users to write properties about smart contracts which will be formally verified by the tool.

Dynamic analysis tools. Although dynamic analysis tools have been less studied than their static counterpart, some work has emerged in recent years.

One of the first works in this line is ContractFuzzer [JLC18]. As its name indicates, it uses fuzzing to find vulnerabilities in smart contracts and is capable of detecting a wide range of vulnerabilities such as reentrancy, locked Ether or unhandled exceptions. The tool generates inputs to the contract and checks using an instrumented EVM whether some vulnerabilities have been triggered. An important limitation of this fuzzing approach is that it requires the Application Binary Interface of the contract, which is typically not available for contracts deployed on the main Ethereum network.

Sereum [Rod+19] focuses on detecting reentrancy exploitation at runtime by integrating checks in a modified Go Ethereum client. The tool analyzes runtime traces and uses taint analysis to ensure that no variable accessing the contract storage is used in a re-entrant call. Although

there are some similarities with our tool, which also analyzes traces at runtime, Sereum focuses on reentrancy while our tool is more generic, notably because vulnerabilities pattern can easily be expressed using Datalog, and allows to analyse several more classes of vulnerabilities.

teEther [KR18] also works at runtime but is different from the previous works presented, as it does not try to protect contracts but rather to actively find an exploit for them. It first analyses the contract bytecode to look for critical execution paths. Critical paths are execution paths which may result in lost funds, for example by sending money to an arbitrary address or being destructed by anyone. To find these paths, it uses an approach close to Oyente [Luu+16a], first using symbolic execution and then the Z3 SMT solver [DB08] to solve path constraints.

Summary. Static analysis tools are typically designed to detect *vulnerable* contracts, while dynamic analysis tools are designed to detect *exploitable* contracts. The only exception is Sereum, which detects contracts *exploited* using reentrancy. Our work is, to the best of our knowledge, the first attempt to detect contracts *exploited* using a wide range of vulnerabilities. This is mostly orthogonal with other works and can support analysis tool development efforts by helping to understand what type of exploitation is happening in the wild.

Follow-up work

More related work has been published since the submission of the paper this part of the chapter is based on. We present some of these here.

TXSPECTOR [Zha+20], which was published soon after the first version of the paper this part of the chapter is based on, uses a very similar approach to ours to detect reentrancy, unchecked call and suicidal contracts. They also leverage a Datalog approach to detect vulnerabilities but first transform the transaction traces into a flow graph rather than adding facts about traces directly to the Datalog database. While this does add expressiveness, it makes the analysis significantly more complex, resulting in some analysis timing out on some transactions. Therefore, we believe that their approach could be complementary to ours and used to eliminate potential false positives of our approach.

Agarwal et al. [ATS21] have very similar goals to ours and try to identify malicious activity in contracts that are likely vulnerable to exploits. They use unsupervised machine learning techniques, in particular, clustering, to try to identify malicious activity in smart contracts. Although their approach is very different to ours, the authors confirm that their results were similar to ours and that only a very small portion of the potentially vulnerable contracts was exploited.

5.2.9 Conclusion

In this section, we surveyed the 23,327 vulnerable contracts reported by six recent academic projects. We proposed a Datalog-based formulation for performing analysis over EVM execution traces and used it to analyze a total of more than 20 million transactions executed by these contracts. We found that at most 463 out of 23,327 contracts have been subject to exploits but that at most 8,487 ETH (1.7 million USD), or only 0.27% of the 3 million ETH (6000 million USD) potentially at risk, was exploited. Finally, we found that a majority of Ether is held by only a small number of contracts and that the vulnerabilities reported on these are either false positives or not exploitable in practice, thus providing a reasonable explanation for our results.

5.3 Economic Security: Liquidations in lending protocols

One of the major defences against economic attacks in DeFi protocols is over-collateralization. Nonetheless, factors such as price volatility may undermine this mechanism. In order to protect protocols from suffering losses, under-collateralized positions can be *liquidated*. In this section, we present the first in-depth empirical analysis of liquidations on protocols for loanable funds (PLFs). We examine Compound, one of the most widely used PLFs, for a period starting from its conception to September 2020. We analyze participants' behaviour and risk appetite, in particular, to elucidate recent developments in the dynamics of the protocol. Furthermore, we

assess how this has changed with a modification in Compound's incentive structure and show that variations of only 3% in an asset's dollar price can result in over 10m USD becoming liquidatable. To further understand the implications of this, we investigate the efficiency of liquidators. We find that liquidators' efficiency has improved significantly over time, with currently over 70% of liquidatable positions being immediately liquidated. Lastly, we provide a discussion on how a false sense of security fostered by a misconception of the stability of non-custodial stablecoins, increases the overall liquidation risk faced by Compound participants.

5.3.1 Introduction

As explained earlier in this chapter, Decentralized Finance (DeFi) refers to a peer-to-peer, permissionless blockchain-based ecosystem that utilizes the integrity of smart contracts for the advancement and disintermediation of traditional financial primitives. One of the most prominent DeFi applications on the Ethereum blockchain [Woo14] is protocols for loanable funds (PLFs) [Gud+20a]. On PLFs, markets for loanable funds are established via smart contracts that facilitate borrowing and lending [Xu+23b]. In the absence of strong identities on Ethereum, creditor protection tends to be ensured through over-collateralization, whereby a borrower must provide collateral worth more than the value of the borrowed amount. In the case where the value of the collateral-to-borrow ratio drops below some liquidation threshold, a borrower defaults on his position and the supplied collateral is sold off at a discount to cover the debt in a process referred to as *liquidation*. However, little is known about the behaviour of agents towards liquidation risk on a PLF. Furthermore, despite liquidators playing a critical role in the DeFi ecosystem, the efficiency with which they liquidate positions has not yet been thoroughly analyzed.

In this section, we first lay out a framework for quantifying the state of a generic PLF and its markets over time. We subsequently instantiate this framework to all markets on Compound [LH18], one of the largest PLFs in terms of locked funds. We analyze how liquidation risk has changed over time, specifically after the launch of Compound's governance token. Furthermore, we seek to quantify this liquidation risk through a price sensitivity analysis. In a

discussion, we elaborate on how the interdependence of different DeFi protocols can result in agent behaviour undermining the assumptions of the protocols' incentive structures.

Contributions. This section makes the following contributions:

- We present an abstract framework to reason about the state of PLFs.
- We provide an open-source implementation[6] of the proposed framework for Compound, one of the largest PLFs in terms of total locked funds.
- We perform an empirical analysis on the historical data for Compound, from May 7, 2019 to September 6, 2020 and make the following observations:
    1. despite increases in the number of suppliers and borrowers, the total funds locked are mostly accounted for by a small subset of participants;
    2. the introduction of Compound's governance token had protocol-wide implications as liquidation risk increased as a consequence of the higher risk-seeking behaviour of participants;
    3. liquidators became significantly more efficient over time, liquidating over 70% of liquidatable positions instantly.
- Using our findings, we demonstrate how interactions between protocols' incentive structures can directly result in unexpected risks to participants.

5.3.2 Background

In this section, we introduce preliminary concepts about DeFi and its primitives necessary to the understanding of the rest of the chapter.

[6] `https://github.com/backdfund/analyzer`

Decentralised Finance

One of the major applications of blockchain systems, and Ethereum in particular is decentralized finance, often referred to as DeFi. DeFi is the development of financial systems on top of blockchains using smart contracts. DeFi systems have several major characteristics:

Non-custodial Users of DeFi systems should have control over their funds at all time

Permisionless DeFi systems should be available to everyone

Openly auditable Anyone has the ability to verify the state of a DeFi system at any point in time

Composable DeFi systems can be freely composed to interact with one another.

There are several primitive, in the form of smart contracts, that are often used when building DeFi systems.

Oracles

An oracle is a mechanism for importing off-chain data into the blockchain virtual machine so that it is readable by smart contracts. This includes off-chain asset prices, such as ETH/USD, as well as off-chain information needed to verify outcomes of prediction markets. Oracles are relied upon by various DeFi protocols (e.g. [LH19; AAV20b; Mak; Syn20; PK15; LH19]).

Oracle mechanisms differ by design and their risks, as discussed in [Kla+20; LS20]. A centralized oracle requires trust in the data provider and bears the risk that the provider behaves dishonestly should the reward from supplying manipulated data be more profitable than from behaving honestly. Decentralized oracles offer an alternative. As the correctness of off-chain data is not verifiable on-chain, decentralized oracles ten d to rely on incentives for accurate and honest reporting of off-chain data. However, they come with their own shortcomings.

Stablecoins

An alternative to volatile cryptoassets is given by stablecoins, which are priced against a peg and can be either custodial or non-custodial. For custodial stablecoins (e.g. `USDC` [Cir20]), tokens represent a claim of some off-chain reserve asset, such as fiat currency, which has been entrusted to a custodian. Non-custodial stablecoins (e.g. `DAI` [Mak]) seek to establish price stability via economic mechanisms specified by smart contracts. For a thorough discussion on stablecoin design, we direct the reader to [Kla+20].

Over-collateralization as Security

Collateralization is one of the primary devices to ensure economic security in a protocol. In general, collateral serves as a potential repercussion against misbehaving agents [Har+19] and allows creating protocols such as stablecoins, loanable funds, or decentralized cross-chain protocols. As asset prices evolve over time, these systems generally allow automated deleveraging: if an agent's level of collateralization (value of collateral / value of borrowing) falls below a protocol-defined threshold, an arbitrager in the system can reduce the agent's borrowing exposure in return for a portion of their collateral at a discounted valuation. This aims to keep the system fully collateralized.

Overcollateralization is not without risks, however. For instance, as explored in [Gud+20b; Kao+20b], times of financial crisis (wherein there are persistent negative shocks to collateral asset prices) can result in thin, illiquid markets, in which loans may become under-collateralized despite an automated deleveraging process. In such settings, it can become unprofitable for liquidators, a type of keeper, to initiate liquidations. Should this occur, rational agents will leave their debt unpaid as that results in a greater payoff.

5.3.3 Protocols for Loanable Funds (PLF)

In this section, we introduce several concepts of Protocols for Loanable Funds (PLFs) necessary for understanding how liquidations function in DeFi on Ethereum.

Supplying and borrowing in DeFi

In DeFi, asset supplying and borrowing is achieved via so-called *protocols for loanable funds* (PLFs) [Gud+20a], where smart contracts act as trustless intermediaries of loanable funds between suppliers and borrowers in markets of different assets. Unlike traditional peer-to-peer lending, deposits are pooled and instantly available to borrowers. On a DeFi protocol, the aggregate of tokens that the PLF smart contracts hold, which equals the difference between supplied funds and borrowed funds, is termed locked funds [23a].

Interest model

Borrowers are charged interest on the debt at a floating rate determined by a market's underlying interest rate model. A small fraction of the paid interest is allocated to a pool of reserves, which is set aside in case of market illiquidity, while the remainder is paid out to suppliers of loanable funds. Interest in a given market is generally accrued through market-specific, interest-bearing derivative tokens that appreciate against the underlying asset over time. Hence, a supplier of funds receives derivative tokens in exchange for supplied liquidity, representing his share in the total value of the liquidity pool for the underlying asset. The most prominent PLFs are Compound [Com19b] and Aave [AAV20a], with 2.01bn USD and 5.49bn USD in total funds locked respectively, at the time of writing [23a].

Collateralization

Given the pseudonymity of agents in Ethereum, borrow positions need to be overcollateralized to reduce the default risk. Thereby, the borrower of an asset is required to supply collateral,

where the total value of the supplied collateral exceeds the total value of the borrowed asset. Each asset is associated with a collateralization ratio, namely the minimum collateral-to-borrow ratio when the asset is used to collateralise a new borrow position. For example, in order to borrow 100 USD worth of `DAI` with `ETH` as collateral at a collateralization ratio of 125%, a borrower would have to lock 125 USD worth of `ETH` to collateralise the borrow position. Thus, the protocol limits monetary risk from defaulted borrow positions, as the underlying collateral of a defaulted position can be sold off to recover the debt. The inverse of the collateralization ratio is referred to as the *collateral factor*, which is the amount of a deposit that may be used as collateral. For example, if the collateralization ratio on a PLF for the market of `DAI` is 125%, the collateral factor would be 0.8, implying that for each $1 deposit of `DAI`, the supplier may borrow $0.8 worth of some other asset.

Liquidation

The process of selling a borrower's collateral to recover the debt value upon default is referred to as *liquidation*. A borrow position can be liquidated once the value of the collateral falls below some pre-determined liquidation threshold, i.e. the minimum acceptable collateral-to-borrow ratio. Any network participant may liquidate these positions by paying the debt asset to acquire the underlying collateral at a discount. Hence, liquidators are incentivised to actively monitor others' collateral-to-borrow ratios. Note that in practice, the amount of liquidatable collateral that a single liquidator can purchase may be capped.

Leveraging

In finance, leverage refers to borrowed funds being used as the funding source for additional, typically more risky capital. In DeFi, leverage is the fundamental component of PLFs, as a borrower is required to first take up the role of a supplier and deposit funds which are to be used as leverage for his borrow positions, as we have just seen. The typical aim of leveraging is to generate higher returns through increased exposure to a particular investment. For example, a borrower wanting to gain increased exposure to `ETH` may:

1. Supply `ETH` on a PLF.
2. Leverage the deposited `ETH` to borrow `DAI`.
3. Sell the purchased `DAI` for `ETH`.
4. Repeat steps 1 to 3 as desired.

This behaviour essentially enables users to construct so-called *leveraging spirals*, whereby a user repeatedly re-supplies borrowed funds in order to get increased exposure to some crypto assets. However, increased exposure comes at the cost of higher downside risk, i.e., the risk of the value of the leveraged asset or borrowed asset decreasing due to changing market conditions.

Use Cases of PLFs

We present the different incentives[7] an agent may have for borrowing from and/or supplying to a PLF:

Interest Suppliers of funds are incentivised by interests accruing on a per-block basis.

Leveraged long position To take on a long position of an asset refers to purchasing an asset with the expectation that it will appreciate in value. These positions can be taken on a PLF by leveraging the asset on which the long position shall be taken.

Leveraged short position A short position refers to borrowing funds from an asset, which one believes will depreciate. Consequently, the taker of a short position sells the borrowed asset, only to repurchase it and pay back the borrower once the price has fallen while profiting from the price change of the shorted asset. This can be achieved by taking on a leveraged borrow position of a stablecoin, where the locked collateral is the asset to short.

Liquidity mining As a means to attract liquidity, PLFs may distribute governance tokens to their liquidity providers. The way these tokens are distributed depends on the PLF. For

[7]Note that leverage on a PLF in DeFi may in part be motivated by tax benefits, as certain jurisdictions may not tax capital gains on borrowed funds. However, a detailed analysis of this lies outside the scope of this section.

instance, on Compound, the governance token COMP[8] is distributed among users across markets proportionally to the total dollar value of funds borrowed and supplied. This directly incentivises users to mine liquidity in a market through leveraging in order to receive a larger share of governance tokens. For example, a supplier of funds in market A can borrow against his position additional funds of A, at the cost of paying the difference between the earned and paid interest. The incentive for pursuing this behaviour exists if the reward (i.e. the governance token) exceeds the cost of borrowing.

Token utility An agent may be able to obtain a token from a PLF which has some desired utility. For example, in the case of governance tokens, the desired token utility could be the right to participate in protocol governance or a claim on protocol earnings.

5.3.4 Methodology

In this section, we describe our methodology for the different analyses we perform about leveraging in PLFs. To be able to quantify the extent of leveraged positions over time, we first introduce a state transition framework for tracking the supply and borrow positions across all markets on a given PLF. We then describe how we instantiate this framework on the Compound protocol using on-chain events data.

Definitions

Throughout the section, we use the following definitions in the context of PLFs:

Market A smart contract acting as the intermediary of loanable funds for a particular crypto asset, where users supply and borrow funds.

Supply Funds deposited to a market that can be loaned out to other users and used as collateral against depositors' borrow positions.

Borrow Funds loaned out to users of a market.

[8]Contract address: 0xc00e94cb662c3520282e6f5717214004a7f26888

Collateral Funds available to back a user's aggregate borrow positions.

Locked funds Funds remaining in the PLF smart contracts, equal to the difference between supplied and borrowed funds.

Supplier A user who deposits funds to a market.

Borrower A user who borrows funds from a market. Since a borrow position must be collateralized by deposited funds, a borrower must also be a supplier.

Liquidator A user who purchases a borrower's supply in a market when the borrower's collateral-to-borrow ratio falls below some threshold.

States on a PLF

In this section, we provide a formal definition of the state of a PLF. We denote $\mathfrak{P}_t$ as the global state of a PLF at time t. For brevity, in the following definitions, we assume that all the values are at a given time t. We define the global state of the PLF as

$$\mathfrak{P} = (\mathcal{M}, \Gamma, \mathcal{P}, \Lambda)$$

where $\mathcal{M}$ is the set of states of individual markets, Γ is the price the Oracle used, $\mathcal{P}$ is the set of states of individual participants and $\Lambda \in (0, 1)$ is the close factor of the protocol, which specifies the upper bound on the amount of collateral a liquidator may purchase.

We define the state of an individual market $m \in \mathcal{M}$ as

$$m = (\mathcal{I}, \mathcal{B}, \mathcal{S}, \mathcal{C})$$

where $\mathcal{I}$ is the market's interest rate model, $\mathcal{B}$ is the total borrows, $\mathcal{S}$ is the total supply of deposits, and $\mathcal{C}$ is the collateral factor.

$\mathcal{P}^m$ is the state of all participants in market m and the positions of a participant P in this

market is defined as

$$P^m = (B^m, S^m)$$

where B^m and S^m are respectively the total borrow positions and total supplied deposits of a market participant in market m.

For a given market m, the total deposits supplied $\mathcal{S}^m$ is thus given by:

$$\mathcal{S}^m = \sum_{P^m \in \mathcal{P}^m} S^m \tag{5.1}$$

Similarly, the market's total borrows $\mathcal{B}^m$ is given by:

$$\mathcal{B}^m = \sum_{P^m \in \mathcal{P}^m} B^m \tag{5.2}$$

The state of a participant P is liquidatable if the following holds:

$$\frac{\sum_{m \in \mathcal{M}} \left\{ [S^m \cdot \mathcal{C} + \mathcal{I}(S^m)] \cdot \Gamma(m) \cdot \mathcal{K}^m \right\}}{\sum_{m \in \mathcal{M}} \left\{ [B^m + \mathcal{I}(B^m)] \cdot \Gamma(m) \right\}} < 1 \tag{5.3}$$

where $\Gamma(m)$ returns the price of the underlying asset denominated in a predefined numéraire (e.g. USD), $\mathcal{I}(S^m)$ returns the interest earned with supply S^m, $\mathcal{I}(B^m)$ returns the interest accrued with borrow B^m, and $\mathcal{K}^m \leq 1$ denotes the liquidation threshold of market m. In Compound, liquidation threshold $\mathcal{K}^m$ is set to be constant at 100% protocol-wide, whereas with other protocols such as Aave, $\mathcal{K}^m$ is specific to the collateral asset from market m, and can be dynamically adjusted when the risk level of the asset changes.

The transition from a state of a market m from time t to $t+1$ is given by some state transition σ, such that $m_t \xrightarrow{\sigma} m_{t+1}$.

Leveraging Spirals on a PLF

Here we examine the workings of leveraging in DeFi using a PLF. We assume a speculator on some volatile asset B, holds initial capital α in B. In order to increase his exposure to B, the speculator may borrow a stable asset A against his α on a PLF at a collateralization ratio $\delta > 1$. For simplicity, we shall assume in this illustrative example that a speculator will leverage his position on the same PLF. Note that the cost of borrowing is given by some floating interest rate γ for the specific asset market. In return for his collateral, the borrower receives $\frac{\alpha}{\delta}$ in the volatile asset B. As the debt is denominated in units of a stable asset (e.g. `DAI`), the borrower has an upper limit on his net debt, remaining unaffected by any volatility in the value of asset A. In order to leverage his position, the debt denominated in A may be used to buy[9] additional units of asset B, which can subsequently be used to collateralise a new borrow position. This process is illustrated in Figure 5.6 and can be repeated numerous times, by which the total exposure to asset A, the underlying collateral to the total debt in asset A, increases at a decaying rate.

The total collateral $\mathcal{C}$ a borrower must post through a borrow position with a leverage factor k, a collateralization ratio δ and an initial capital amount α can be expressed as $\sum_{i=0}^{k} \frac{\alpha}{\delta^i}$. Hence, the total debt Π for the corresponding borrow position is:

$$\Pi = \left(\sum_{i=1}^{k} \frac{\alpha}{\delta^i} \right) \cdot (1 + \gamma) \tag{5.4}$$

where γ is the interest rate. Note that Equation (5.4) assumes a borrower uses the same collateralization ratio δ for his positions, as well as that all debt is taken out for the same asset on the same PLF and hence the floating interest rate is shared across all borrow positions.

[9] In practice this may be done via automated market makers [Xu+23c] (e.g. Uniswap [Uni20]) or via decentralized exchanges [dYd19].

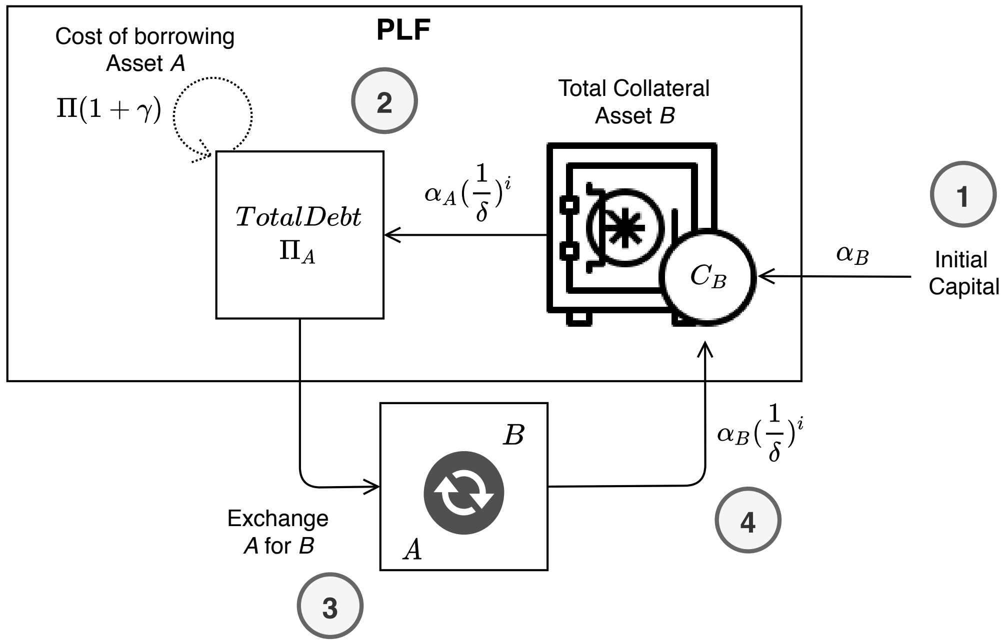


Figure 5.6: The steps of leveraging using a PLF. **1.** Initial capital α_B in asset B is deposited as collateral to borrow asset A. **2.** Interest accrues over the debt of the borrow position for asset B. **3.** The borrowed asset A is sold for asset B on the open market. **4.** The newly purchased units of asset B are locked as collateral for a new borrow position of asset A.

States and the Compound PLF

For our analysis, we apply our state transition framework to the Compound PLF. Therefore, we briefly present the workings of Compound in the context of our framework.

State Transitions. We initiate state transitions via events emitted from the Compound protocol smart contracts. We provide an overview of the state variables affected by Compound events in Table 5.11.

Funds Supplied. Every market on Compound has an associated “cToken", a token that continuously appreciates against the underlying asset as interest accrues. For every deposit in a market, a newly-minted amount of the market’s associated cToken is transferred to the depositor. Therefore, rather than tracking the total amount of the underlying asset supplied, we account for the total deposits of an asset supplied by a market participant in the market’s cTokens. Likewise, we account for the total supply of deposits in the market in cTokens.

Funds Borrowed. A borrower on Compound must use cTokens as collateral for his borrow position. The borrowing capacity equals the current value of the supply multiplied by the

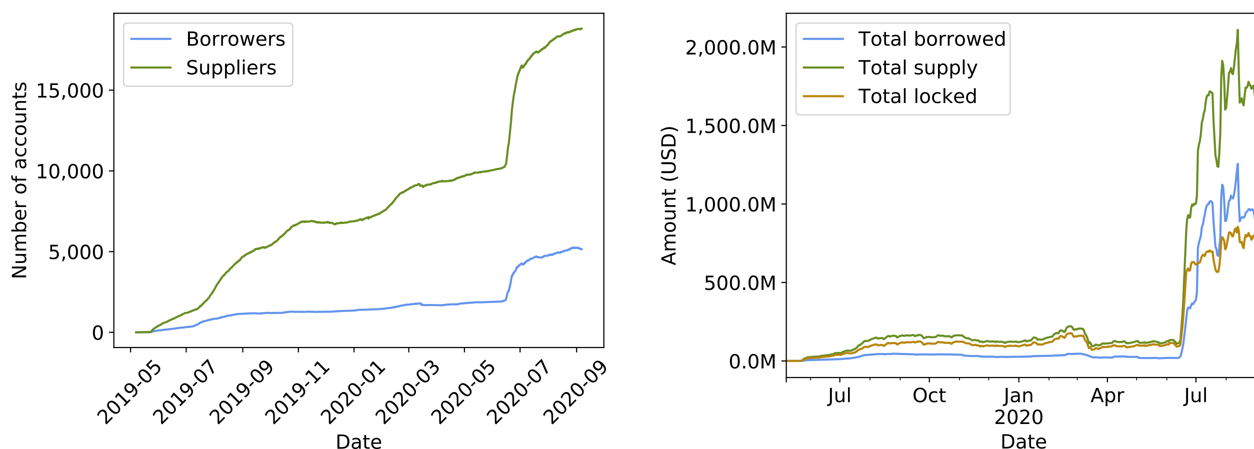
Table 5.11: The events emitted by the Compound protocol smart contracts used for initiating state transitions and the states affected by each event.

Event	Description	State variables affected
`Borrow`	A new borrow position is created.	$\mathcal{B}$
`Mint`	cTokens are minted for new deposits.	$\mathcal{S}$
`RepayBorrow`	A borrow position is partially/fully repaid.	$\mathcal{B}$
`LiquidateBorrow`	A borrow position is liquidated.	$\mathcal{B}$, $\mathcal{S}$
`Redeem`	cTokens are used to redeem deposits of the underlying asset.	$\mathcal{S}$
`NewCollateralFactor`	The collateral factor for the associated market is updated.	$\mathcal{C}$
`AccrueInterest`	Interest has accrued for the associated market and its borrow index is updated.	$\mathcal{B}$
`NewInterestRateModel`	The interest rate model for the associated market is updated.	$\mathcal{I}$
`NewInterestParams`	The parameters of the interest rate model for the associated market are updated.	$\mathcal{I}$
`NewCloseFactor`	The close factor is updated.	Λ

collateral factor for the asset. For example, given an exchange rate of 1 `DAI` = 50 `cDAI`, a collateral factor of 0.75 for `DAI` and a price of 1 `DAI` = 1 USD, a holder of 500 `cDAI` (10 `DAI`) would be permitted to borrow up to 7.5 USD worth of some other asset on Compound. Therefore, as funds are borrowed, an individual's total borrow position, as well as the respective market's total borrows are updated.

Interest. The accrual of interest is tracked per market via a borrow index, which corresponds to the total interest accrued in the market. The borrow index of a market is also used to determine and update the total debt of a borrower in the respective market. When funds are borrowed, the current borrow index for the market is stored with the borrow position. When additional funds are borrowed or repaid, the latest borrow index is used to compute the difference of accrued interest since the last borrow and added to the total debt.

Liquidation. A borrower on Compound is eligible for liquidation should his total supply of collateral, i.e. the value of the sum of the borrower's chosen holdings per market, weighted by each market's collateral factor, be less than the value of the borrower's aggregate debt



(a) Number of suppliers and borrowers. **(b)** Amount of funds supplied, borrowed and locked.

Figure 5.7: Number of active accounts and amount of funds on Compound over time.

Table 5.12: Monitored contracts

Name	Address
cBAT	0x6c8c6b02e7b2be14d4fa6022dfd6d75921d90e4e
cDAI	0x5d3a536e4d6dbd6114cc1ead35777bab948e3643
cETH	0x4ddc2d193948926d02f9b1fe9e1daa0718270ed5
cREP	0x158079ee67fce2f58472a96584a73c7ab9ac95c1
cSAI	0xf5dce57282a584d2746faf1593d3121fcac444dc
cUSDC	0x39aa39c021dfbae8fac545936693ac917d5e7563
cUSDT	0xf650c3d88d12db855b8bf7d11be6c55a4e07dcc9
cWBTC	0xc11b1268c1a384e55c48c2391d8d480264a3a7f4
cZRX	0xb3319f5d18bc0d84dd1b4825dcde5d5f7266d407
Comptroller	0x3d9819210a31b4961b30ef54be2aed79b9c9cd3b
Open Oracle Price Data	0x02557a5e05defeffd4cae6d83ea3d173b272c904
Uniswap Anchored View	0x9b8eb8b3d6e2e0db36f41455185fef7049a35cae

(Equation (5.3)). The maximum amount of debt a liquidator may pay back in exchange for collateral is specified by the close factor of a market.

5.3.5 Analysis

In this section, we present the results of the analysis performed with the framework outlined in Section 5.3.4. We analyze data from the Compound protocol [LH18] over a period ranging from May 7, 2019—when the first Compound markets were deployed on the Ethereum main network—to September 6, 2020. In Table 5.12, we provide a list of contracts we monitored in

Table 5.13: Top 10 suppliers and borrowers. Amounts are expressed in their USD equivalent. Addresses marked with ✓ are smart contract addresses, among which the one with the most supplied funds is a Curve pool address that aggregates funds from multiple parties.

(a) Top 10 users with the largest amount of funds supplied

Address		Amount	Description
0x554bd2947df1c8d8d38897bdc92b3b97692b2845		342,128,032	
0xa2b47e3d5c44877cca798226b7b8118f9bfb7a56	✓	40,284,236	Curve pool
0x04b0b0e460c9fc583d9c93bc9ae25b353390645e	✓	34,908,472	Instadapp smart wallet
0x25599dcbd434af9a17d52444f71c92987fa97cfc		34,530,570	
0x58485ea7106891bdd94c37ced30c6fdbc5293b16	✓	32,686,029	Multisig wallet
0x909b443761bbd7fbb876ecde71a37e1433f6af6f		29,308,425	
0xea61f3052753ea2c6a1c208583ad9b0394ed2f28	✓	28,854,366	DeFi Saver smart wallet
0x32b2d4ec46d76fc6dabfe958fb0e0bd8db740c84		27,928,637	
0xedcc13d25e23032b61d30c298334f92d7c0ba84e		27,709,153	
0x6d2af065ccb60c0f7e8ec5907c961c42a3447127		25,559,037	

(b) Top 10 users with the largest amount of funds borrowed

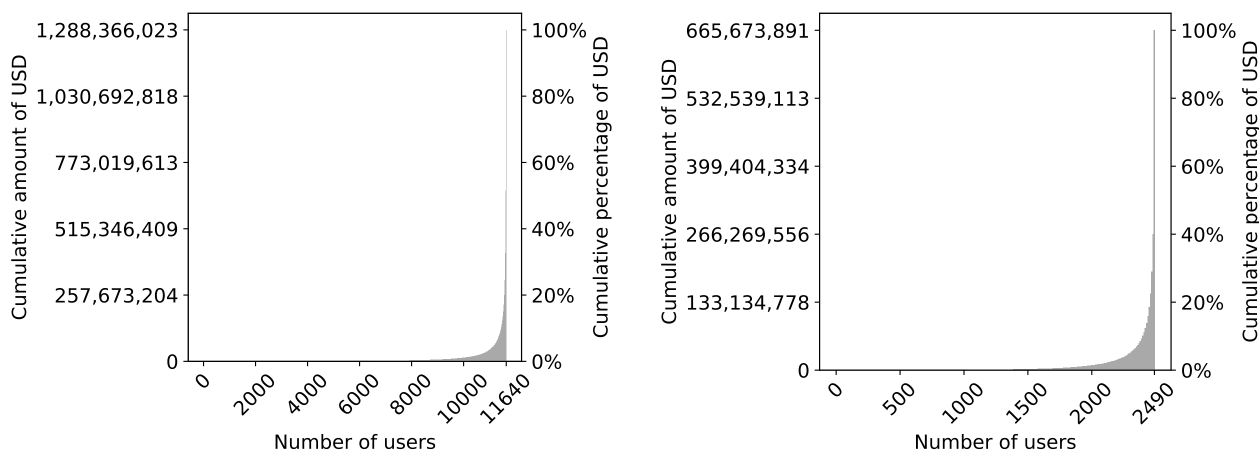
Address		Amount	Description
0x554bd2947df1c8d8d38897bdc92b3b97692b2845		247,143,532	
0x25599dcbd434af9a17d52444f71c92987fa97cfc		22,085,613	
0x909b443761bbd7fbb876ecde71a37e1433f6af6f		21,030,095	
0x58485ea7106891bdd94c37ced30c6fdbc5293b16	✓	20,149,687	Multisig wallet
0x32b2d4ec46d76fc6dabfe958fb0e0bd8db740c84		18,900,729	
0xea61f3052753ea2c6a1c208583ad9b0394ed2f28	✓	18,248,324	DeFi Saver smart wallet
0xedcc13d25e23032b61d30c298334f92d7c0ba84e		17,643,172	
0x6d2af065ccb60c0f7e8ec5907c961c42a3447127		12,015,576	
0x79dbd1baf124edd4205b2aba56c29bf3914c8ed0		11,632,820	
0x0c8a8dd439069690a5722d5fbb18359a68e279f1		10,009,553	

our analysis. When analysing a single market, we choose the market for DAI, as it is the largest by an order of magnitude.

Borrowers and Suppliers

We first examine the total number of borrowers and suppliers on Compound by considering any Ethereum account that, at any time within the observation period, either exhibited a non-zero cToken balance or borrowed funds for any Compound market. The change in the number of borrowers and suppliers over time is displayed in Figure 5.7a.

We see that the total number of suppliers always exceeds the total number of borrowers. This



(a) Distribution of supplied funds.

(b) Distribution of borrowed funds.

Figure 5.8: Cumulative distribution of funds in USD. Accounts are sorted from least to most wealthy and bucketed in bins of 10, i.e. a single bar represents the sum of 10 accounts.

is because on Compound, one can only borrow against funds he supplied as collateral, which automatically makes the borrower also a supplier. Interestingly, the number of suppliers has increased relative to the number of borrowers over time. There is a notable sudden jump in both the number of suppliers and borrowers in June 2020.

In terms of total deposits, a very similar trend is observable in Figure 5.7b, which shows that at the same time, the total supplied deposits increased, while the total borrows followed shortly after. Furthermore, the total funds borrowed exceeded the total funds locked for the first time in July 2020 and remained so until the end of the examined period. We discuss the reasons behind this in the next part of this section.

Despite the similarly increasing trend for the number of suppliers/borrowers and amount of supplied/borrowed funds, we can see in Figure 5.8 that the majority of funds are borrowed and supplied only by a small number of accounts. For instance, for the suppliers in Figure 5.8a, the top user and top 10 users supply 27.4% and 49% of total funds, respectively. For the borrowers shown in Figure 5.8b, the top user accounts for 37.1%, while the top 10 users account for 59.9% of total borrows. While one could think that this concentration comes from the fact that top accounts are pools receiving money from several participants, only one of the top 10 suppliers and none of the top 10 borrowers fit in this category. In Table 5.13, we show the top 10 suppliers and borrowers in terms of the total amount borrowed and supplied expressed in USD. We mark

the addresses that are smart contracts. Among those contracts, the one with the most supplied funds, `0xa2b47e3d5c44877cca798226b7b8118f9bfb7a56`, is the address of a Curve [20] pool that has funds coming from several independent parties. No other address among top suppliers and borrowers is a pool address.

Leveraging Spirals

As we have seen in Section 5.3.3, in PLFs, leveraging can be used either to gain more exposure to a particular currency or to gain some incentive provided by the protocol. To understand how leveraging can affect the total amounts borrowed and supplied on Compound, we use the methodology we defined in Section 5.3.4 to measure the existence of leveraging spirals on Compound.

We find that the top supplier deposited a total of 342 million USD and borrowed 247 million. However, after the inspection of leveraging spirals, we find that the user has provided only 16% of the funds, while the rest of the minted funds have been part of leveraging spirals, which means that the user provided a total of roughly 55 million USD to the protocol.

In total, we find a total of 2,141 accounts using this leveraging spiral technique for a total of over 600 million USD, or roughly half of the total amount of funds supplied to the protocol.

The `COMP` Governance Token

The sudden jumps exhibited in Figure 5.7a and Figure 5.7b can be explained by the launch of Compound's governance token, `COMP`, on June 15, 2020. The `COMP` governance token allows holders to participate in voting, create proposals, as well as delegate voting rights. In order to empower Compound stakeholders, new `COMP` is minted every block and distributed among borrowers and suppliers in each market.

Initially, `COMP` was allocated proportionally to the accrued interest per market. However, the `COMP` distribution model was modified via a governance vote on July 2, 2020, such that the

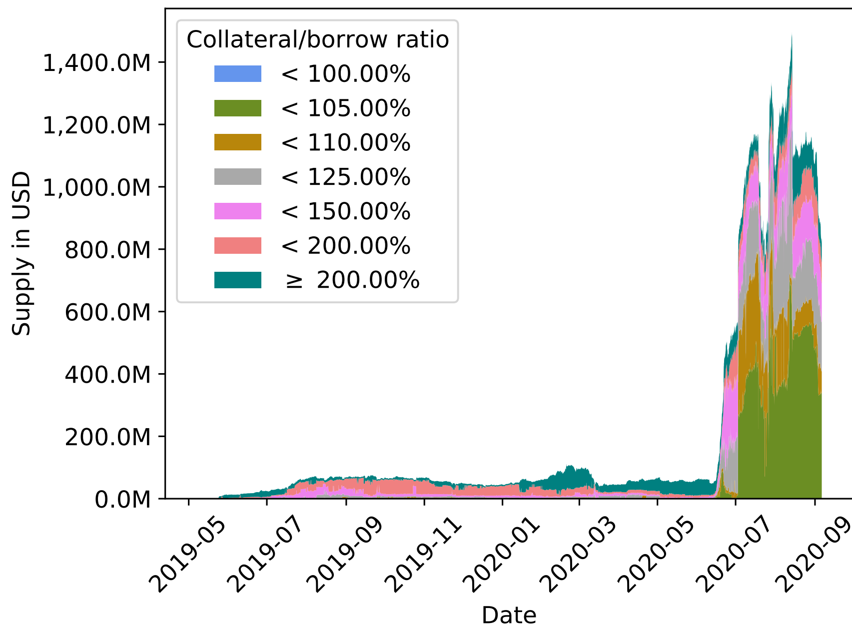


Figure 5.9: Collateral locked over time, showing how close the amounts are from being liquidated. Positions can be liquidated when the ratio drops below 100%.

borrowing interest rate was removed as a weighting mechanism in favour of distributing COMP per market on a borrowing demand basis, i.e. per USD borrowed. The distributed COMP per market is shared equally between a market's borrowers and suppliers, who receive COMP proportionally to their borrowed and supplied amounts, respectively. Hence, a Compound user is incentivized to increase his borrow position as long as the borrowing cost does not exceed the value of his COMP earnings. This presumably explains the drop in the degree of collateralization, as the total amount locked is seen surpassed by the total borrows after the COMP launch (Figure 5.7b), leading to elevated liquidation risk of borrow positions.

Liquidation Risk

Given the high increase in the number of total funds borrowed and supplied, as well as the decrease in liquidity relative to total borrows, we seek to identify and quantify any changes in liquidation risk on Compound since the launch of COMP. Figure 5.9 shows the total USD value of collateral on Compound and how close collateral amounts are to liquidation. In addition to the substantial increase in the total value of collateral on Compound since the launch of COMP, the risk-seeking behaviour of users has also changed. This can be seen by examining collateral-to-

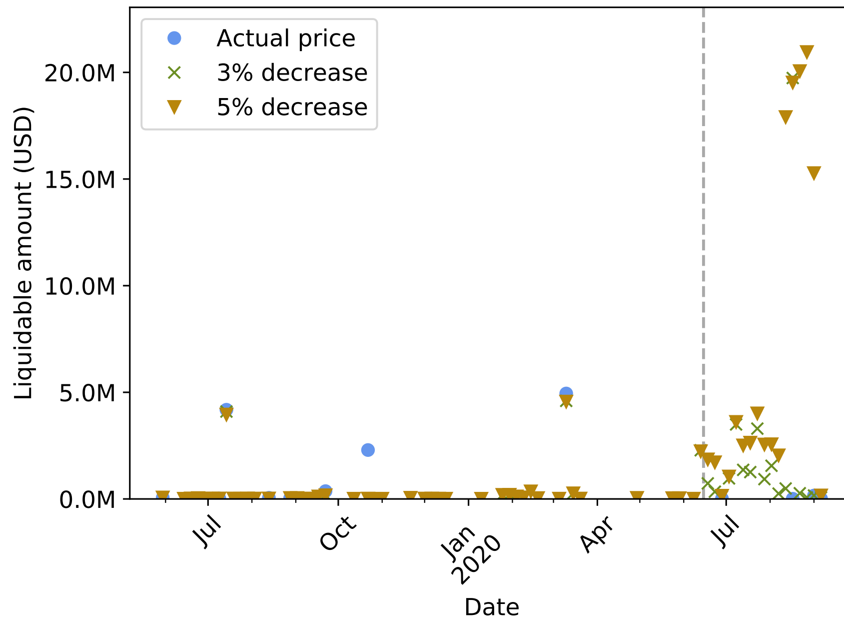


Figure 5.10: Sensitivity analysis of the liquidatable collateral amount given DAI price movement relative to its peg USD. COMP launch date is marked by the dashed vertical line.

borrow ratios, where since the beginning of July, 2020, a total of approximately 350m to 600m USD worth of collateral has been within a 5% price range of becoming liquidatable. However, it should be noted that the likelihood of the amount of this collateral becoming liquidatable highly depends on the price volatility of the collateral asset.

In order to examine how liquidation risk differs across markets, we measure for the largest market on Compound, namely DAI, the sensitivity of collateral becoming liquidatable given a decrease in the price of DAI. Figure 5.10 shows the amount of aggregate collateral liquidatable at the historic price, as well as at a 3% and 5% decrease relative to the historic price for DAI. We mark the date on which the COMP governance token launched with a dashed line. It can be seen that since the launch of COMP, 3% and 5% price decreases of DAI relative to its peg USD would have resulted in a substantially higher amount of liquidatable collateral. In particular, a 3% decrease would have turned collateral worth in excess of 10 million USD liquidatable.

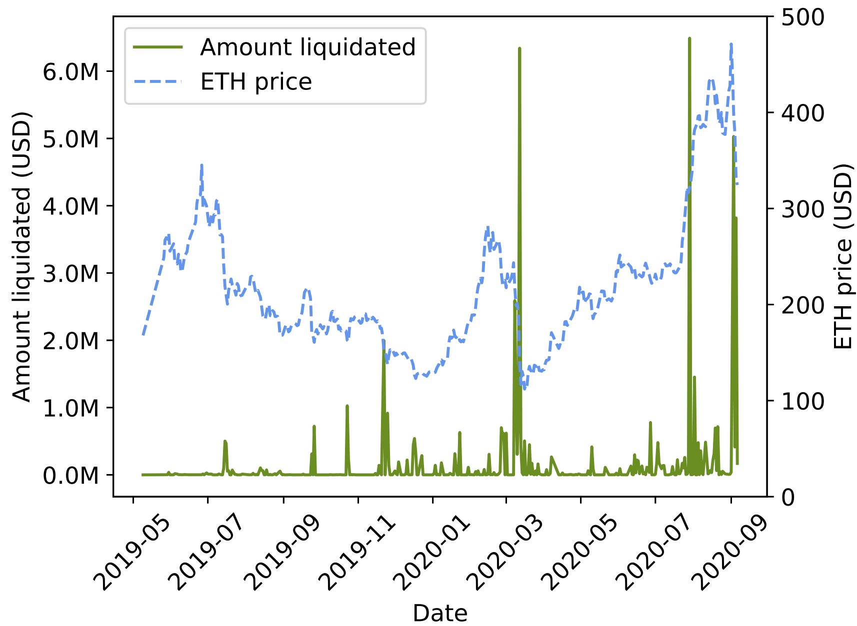


Figure 5.11: Amount (in USD) of liquidated collateral from May 2019 to August 2020.

Liquidations and Liquidators

In order to better understand the implications of the increased liquidation risk since the launch of COMP, we examine historical liquidations on Compound and subsequently measure the efficiency of liquidators.

Historical Liquidations. The increased risk-seeking behaviour suggested by the low collateral-to-borrow ratios presented in the previous section are in accordance with the trend of rising amount of liquidated collateral since the introduction of COMP. The total value of collateral liquidated on Compound over time is shown in Figure 5.11. It can be seen that the majority of this collateral was liquidated on a few occasions, perhaps most notably on Black Thursday (March 12, 2020), July 29, 2020 (DAI deviating from its peg significantly), and in early September 2020 (ETH price drop).

Liquidation Efficiency. We measure the efficiency of liquidators as the number of blocks elapsed since a borrow position has become liquidatable and the position actually being liquidated. The overall historical efficiency of liquidators is shown as a cumulative distribution function in Figure 5.12, from which it can be seen that approximately 60% of the total liqui-

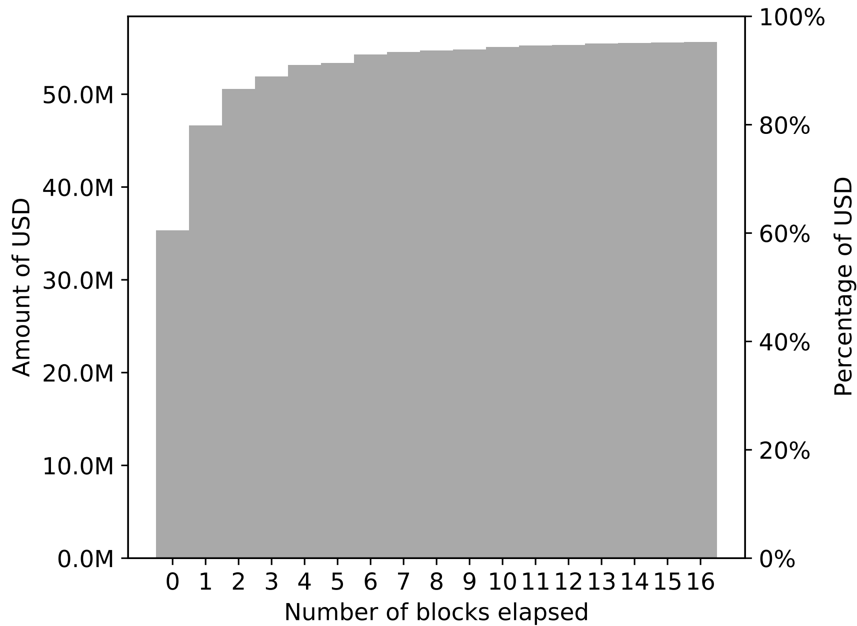


Figure 5.12: Number of blocks elapsed from the time a position can be liquidated to actual liquidation on Compound from May 7, 2019 to September 6, 2020, shown as a CDF.

dated collateral (35 million USD) was liquidated within the same block as it became liquidatable, suggesting that the majority of liquidations occur via bots in a highly efficient fashion. After 2 blocks have elapsed (on average half a minute), 85% of liquidatable collateral has been liquidated, and after 16 blocks this value amounts to 95%.

It is worth noting that liquidation efficiency has been skewed by the more recent liquidation activities which were of a much larger scale than when the protocol was first launched. Specifically, in 2019, only about 26% of the liquidations occurred in the block during which the position became liquidatable, compared to 70% in 2020. This resulted in some lost opportunities for liquidators as shown in Figure 5.10. The account `0xd062eeb318295a09d4262135ef0092979552afe6`, for instance, had more than 3,000,000 USD worth of `ETH` as collateral exposed at block 8,796,900 for the duration of a single block: the account was roughly 20 USD shy of the liquidation threshold but eventually escaped liquidation. If a liquidator had captured this opportunity, he could have bought half of this collateral (given the close factor of 0.5), at a 10% discount, resulting in a profit of 150,000 USD for a single transaction. It is clear that with such stakes, participants were incentivized to improve liquidation techniques, resulting in a high level of liquidation speed and scale.

Summary

In this section, we have analysed the Compound protocol with a focus on liquidations. We have found that despite the increase in the number of suppliers and borrowers over time, the total amount of funds supplied and borrowed remained extremely concentrated among a small set of participants.

We have also seen that the introduction of the `COMP` governance token has changed how users interact with the protocol and the amount of risk that they are willing to take. Users now borrow vastly more than before, with the total amount borrowed surpassing the total amount locked. Due to excessive borrowing without a sufficiently safe amount of supplied funds, borrow positions now face a higher liquidation risk, such that a crash of 3% in the price of `DAI` could result in an aggregate liquidation value of over 10 million USD.

Finally, we have shown that the liquidators have become more efficient with time, and are currently able to capture a majority of the liquidatable funds instantly.

5.3.6 Discussion

In this section, we enumerate several points that we deem important for the future development of PLFs and DeFi protocols. We first discuss the influence of governance tokens, by intention or not, on how users behave within a protocol. Subsequently, we discuss potential risks that lie in the use of governance tokens, and the contagion effect that user behaviour in a protocol can have on another protocol. Finally, we discuss how miner-extractable value [Dai+20] can potentially affect liquidation incentives in such protocols.

Governance Token Influence

As analysed in Section 5.3.5, the distribution of the `COMP` token has vastly changed the Compound landscape and user behaviour. Until the introduction of the token, borrowing was costly due to the payable interest, which implies a negative cash flow for the borrower. Therefore, a

borrower would only borrow if he could justify this negative cash flow with some application external to Compound. With the introduction of this token, borrowing could yield a positive cash flow due to the monetary value of the governance token. This creates a situation where both suppliers and borrowers end up with a positive cash flow, inducing users to maximize both their supply and borrow. This model is, however, only sustainable when the price of the `COMP` token remains sufficiently high to keep this cash flow positive for borrowers. This directly results in users taking increasingly higher risks in an attempt to gain larger monetary rewards, with liquidators ultimately profiting more from their operations.

Governance Token Risks

The increased use of governance tokens across DeFi protocols (e.g. `YFI` on Yearn Finance, `AAVE` on Aave, `UNI` on Uniswap) can be seen as a promising step towards achieving a higher degree of decentralization in terms of protocol governance. However, despite the increased usage of governance tokens, to the best of our knowledge, there is still a dearth of academic research examining the different governance models and specifically the relation between their security assumptions and the employed governance token. For instance, the option to aggregate governance tokens via flash loans [Wan+20] can pose a significant security risk to DeFi protocols should an attacker attempt to propose and execute malicious protocol updates. Furthermore, even in the case of flash loan-resistant governance models, the relationship between the financial value of a protocol's governance token and the economically secure regions of the protocol remains unexamined and serves as a further risk that designers of governance models have to take into account. Despite the existence of protective mechanisms against governance attacks on some protocols (e.g. multi-sig approvals or selected "guardians" that can halt the governance process), it remains questionable which of such mechanisms are indeed desirable from a decentralized governance perspective and whether there might be more suitable alternatives.

Contagion Effects

This behaviour also indirectly affected other protocols, in particular `DAI`. The price of `DAI` is aimed to be pegged to 1 USD resting on an arbitrage mechanism, whereby token holders are incentivized to buy or sell `DAI` as soon as the price moves below or above 1 USD, respectively. However, a rational user seeking to maximize profit will not sell his `DAI` if holding it somewhere else would yield higher profits. This was precisely what was happening with Compound, whose users locking their `DAI` received higher yields in the form of `COMP`, than from selling `DAI` at a premium, thereby resulting in upward price pressure [Cyr20]. Interestingly, `DAI` deviating from its peg also has a negative effect on Compound users. Indeed, as we saw in Section 5.3.5, many Compound users might have been overconfident about the price stability of `DAI` and thus only collateralise marginally above the threshold when they borrow `DAI`. This has resulted in large amounts being liquidated due to the actual, higher extent of the volatility in the `DAI` price.

Miner-Extractable Value

In the context of PLFs, liquidations can be seen as miner-extractable value. Indeed, it is easy for the miner to check whether a position is liquidatable or not after each processed transaction and to add a transaction to liquidate the position immediately after the transaction making it liquidatable. In our analysis of the Compound protocol, we have not found any sign of miners participating in liquidations, directly or indirectly. In Table 5.14, we show the 10 miners who mined the most blocks containing at least one liquidation. For each miner, we show the 5 liquidators who liquidated the most positions in blocks mined by the given miner. Overall, we see that for every miner, the liquidations are spread relatively evenly across the different liquidators. Although we only show the top 10 miners for space constraints, we noted that this was the case for all miners in our dataset. Although we found no correlation between miners and liquidators, this is a real risk that could make the role of liquidators, which is essential for protocol security, less interesting for those who are not collaborating with miners.

Table 5.14: Top 10 miners per number of blocks containing at least a liquidation event mined and top 5 liquidators for each miner per number of liquidations

Miner	Blocks count	Liquidators	Liquidations count
0x5A0b54D5dc17e0AadC383d2db43B0a0D3E029c4c	1281	0x6a0c50788E462f322959A2458687096994d66316	144
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	114
		0x0c31b6605686aa26df47eb45AF0e4aa6639A5fd6	91
		0xb00ba6778cF84100da676101e011B3d229458270	76
		0x268a1b7ECC1fE1FaB1eE32a7e61e3b7810BAD4a5	70
0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8	969	0x6a0c50788E462f322959A2458687096994d66316	88
		0xb00ba6778cF84100da676101e011B3d229458270	75
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	70
		0x0c31b6605686aa26df47eb45AF0e4aa6639A5fd6	52
		0x268a1b7ECC1fE1FaB1eE32a7e61e3b7810BAD4a5	50
0x829BD824B016326A401d083B33D092293333A830	310	0x6a0c50788E462f322959A2458687096994d66316	31
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	26
		0x402a75f3500CA1FbA17741Ec916F07a0c9DB195D	23
		0xb00ba6778cF84100da676101e011B3d229458270	18
		0x029720A9b3CE72f3e1D9C79257E1F19AfE20b6c9	17
0x52bc44d5378309EE2abF1539BF71dE1b7d7bE3b5	257	0x6a0c50788E462f322959A2458687096994d66316	22
		0x10aab4B0EF76AA2AC9b5909e671517a1171B050E	21
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	16
		0x402a75f3500CA1FbA17741Ec916F07a0c9DB195D	15
		0x0006e4548AED4502ec8c844567840Ce6eF1013f5	14
0x04668Ec2f57cC15c381b461B9fEDaB5D451c8F7F	185	0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	22
		0x5DAfafbd7AcD662C909a9601120cf1D9F277e8aE	14
		0x10aab4B0EF76AA2AC9b5909e671517a1171B050E	14
		0x268a1b7ECC1fE1FaB1eE32a7e61e3b7810BAD4a5	12
		0x6a0c50788E462f322959A2458687096994d66316	12
0xb2930B35844a230f00E51431aCAe96Fe543a0347	77	0x10aab4B0EF76AA2AC9b5909e671517a1171B050E	8
		0x0c31b6605686aa26df47eb45AF0e4aa6639A5fd6	8
		0x5DAfafbd7AcD662C909a9601120cf1D9F277e8aE	6
		0xf8E562f4F30c5DdA0978857067D6585265dA3437	6
		0xfDe817C7a0770f42fb80B93dd7A538291C871765	5
0xD224cA0c819e8E97ba0136B3b95ceFf503B79f53	73	0xb00ba6778cF84100da676101e011B3d229458270	13
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	8
		0x88886841CfCCBf54AdBbC0B6C9cBAceAbec42b8B	8
		0xfFA7370a03c2a91f5B1847a90750489d05f52Fa9	5
		0x492Ff1c96b398297FcAcd6E7E1E968d2b2fc7Da0	5
0x4C549990A7eF3FEA8784406c1EECc98bF4211fA5	68	0xb00ba6778cF84100da676101e011B3d229458270	12
		0x6a0c50788E462f322959A2458687096994d66316	9
		0x402a75f3500CA1FbA17741Ec916F07a0c9DB195D	5
		0x10aab4B0EF76AA2AC9b5909e671517a1171B050E	4
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	3
0xEEa5B82B61424dF8020f5feDD81767f2d0D25Bfb	55	0xb00ba6778cF84100da676101e011B3d229458270	7
		0x402a75f3500CA1FbA17741Ec916F07a0c9DB195D	6
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	5
		0x029720A9b3CE72f3e1D9C79257E1F19AfE20b6c9	5
		0x10aab4B0EF76AA2AC9b5909e671517a1171B050E	4
0x84A0d77c693aDAbE0ebc48F88b3fFFF010577051	46	0xb00ba6778cF84100da676101e011B3d229458270	7
		0x0c31b6605686aa26df47eb45AF0e4aa6639A5fd6	6
		0x6a0c50788E462f322959A2458687096994d66316	5
		0x5e32f33e261a90FF9fE94230387118945599268c	5
		0x8c863333c2E92f02e01F7A3c6d131E4d59f78990	5

5.3.7 Related Work

Previous work

In this section, we briefly discuss existing related work.

A thorough analysis of the Compound protocol with respect to market risks faced by participants was done by [Kao+20a]. The authors employ agent-based modelling and simulation to perform stress tests in order to show that Compound remains safe under high volatility scenarios and high levels of outstanding debt. Furthermore, the authors demonstrate the potential of Compound to scale to accommodate a larger borrow market while maintaining a low default probability. This differs from our work as we conduct a detailed empirical analysis of Compound, focusing on how agent behaviour under different incentive structures on Compound has affected the protocol's state with regard to liquidation risk.

A first in-depth analysis of PLFs is given by [Gud+20a]. The authors provide a taxonomy of interest rate models employed by PLFs, while also discussing market liquidity, efficiency and interconnectedness across PLFs. As part of their analysis, the authors examine the cumulative percentage of locked funds solely for the Compound markets `DAI`, `ETH`, and `USDC`.

In [BCL20], the authors provide a formal state transition model of PLFs[10] and prove fundamental behavioural properties of PLFs, which had previously only been presented informally in the literature. Additionally, the authors examine attack vectors and risks, such as utilization attacks and interest-bearing derivative token risks. This work differs from our work, as the authors of [BCL20] formalize the properties of PLFs through an abstract model, while we provide a thorough empirical analysis with a focus on liquidations and risks brought upon by governance tokens, such as for Compound and the `COMP` token.

In [KM19], the authors show how markets for stablecoins are exposed to deleveraging feedback effects, which can cause periods of illiquidity during a crisis.

The authors of [Gud+20b] demonstrate how various DeFi lending protocols are subject to

[10] Note that in [BCL20], PLFs are referred to as lending pools.

different attack vectors such as governance attacks and under-collateralization. In the context of the proposed governance attack, the lending protocol the authors focus on is Maker [Mak].

Follow-up work

In this section, we present work that has followed the paper on which this section is based.

Qin et al. [Qin+21] extend our analysis to include more DeFi protocols, including Aave, dYdX, and MakerDAO. The authors provide a way to compare the different liquidation mechanisms. They found that current liquidation designs effectively incentivize liquidators but often result in the sale of excessive amounts of discounted collateral at the expense of borrowers. The research also measures the risks faced by liquidation participants and quantifies the instability of existing lending protocols.

The research conducted by Kozhan et al. [KV21] dives deeper into the topic of liquidations in the context of collateralized debt positions and examines the relationship between peg volatility and collateral risk. Their findings reveal a negative covariation between DAI price and returns on risky collateral, even after accounting for safe-haven demand and the collateral liquidation impact. Moreover, the study shows that the incorporation of safer collateral types has contributed to enhanced peg stability.

5.3.8 Conclusion

In this section, we presented the first in-depth empirical analysis of liquidations on Compound, one of the largest PLFs in terms of total locked funds, from May 7, 2019 to September 6, 2020. We analysed agents' behaviour and in particular how much risk they are willing to take within the protocol. Furthermore, we assessed how this has changed with the launch of the Compound governance token `COMP`, where we found that agents take notably higher risks in anticipation of higher earnings. This resulted in variations as little as 3% in an asset's price being able to turn over 10 million USD worth of collateral liquidatable. In order to better understand the potential consequences, we then measured the efficiency of liquidators, namely how quickly

new liquidation opportunities are captured. Liquidators' efficiency was found to have improved significantly over time, reaching 70% of instant liquidations. Lastly, we demonstrated how overconfidence in the price stability of DAI, increased the overall liquidation risk faced by Compound users. Rather ironically, many users wishing to make the most of the new incentive scheme ended up causing higher volatility in DAI—a dominant asset of the platform, resulting in the liquidation of their assets. This is not Compound's misdoing, however, this highlights the to-date unknown dynamics of incentive structures across different DeFi protocols.

Chapter 6

Conclusion

In this chapter, we summarise chapter-by-chapter the achievements of our thesis.

Execution layer. At the execution layer, we contributed to improving the security of the Ethereum Virtual Machine (EVM), and in particular, the gas metering mechanism.

Our approach involved creating an instrumented version of the EVM that allowed us to replay and analyze the execution of smart contracts. By examining several months' worth of transactions, we identified several discrepancies in the metering model, including significant inconsistencies in the pricing of instructions. Furthermore, we discovered that there was a weak correlation between execution cost and the resources utilized, such as CPU and memory.

As a result, we introduced a new type of DoS attack, known as the "Resource Exhaustion Attack" that leveraged these weaknesses to generate low-throughput contracts. To demonstrate the vulnerability of major Ethereum client implementations, we designed a genetic algorithm that generated contracts with a throughput on average 100 times slower than typical contracts. Our findings indicated that if these clients were running on commodity hardware, they would be unable to remain synchronized with the network when subjected to this attack.

Transactional layer. Our focus at the transactional layer was on blockchains with higher scalability and their transactional throughput. To conduct our analysis, we examined transac-

tion data for three high-scalability blockchains: EOSIO, Tezos, and XRP Ledger (XRPL) over a period of seven months.

Our findings revealed that only a small portion of transactions was utilized for value transfer purposes. Specifically, 96% of transactions on EOSIO resulted from airdrops of a token without any current value. In the case of Tezos, 76% of throughput was utilized for maintaining consensus, while over 94% of transactions on XRPL had no economic value. We also identified a persisting airdrop on EOSIO that qualified as a DoS attack and detected a two-month-long spam attack on XRPL.

We also explored how the different designs of the three blockchains had impacted user behaviour. Through this analysis, we gained insights into utilization patterns of transactional throughput and how blockchain designs could influence user behaviour.

Application layer. At the application layer, we focused on the DeFi ecosystem, and first formalised the concepts of technical and economic security.

We then analysed the technical security of smart contracts deployed on Ethereum. We analysed 20 million transactions interacting with contracts flagged vulnerable by program analysis tools and found that at most 8,487 ETH (1.7 million USD) of the 3 million ETH (6000 million USD) potentially at risk was exploited, which represents a mere 0.27%. We investigated these contracts in more depth and found that most were not exploited in practice because of the lack of feasibility of the exploit or because of the lack of economic incentive to do so.

Finally, we studied the economic security of DeFi lending protocols and found that users often have very risky positions, with variations as small as 3% in an asset's price being able to turn over 10 million USD worth of collateral liquidatable. We also found that the efficiency of the liquidations has increased over time and that at the time of the analysis, over 70% of the liquidations were instant. Lastly, we also found that depegging events of stablecoin have caused very large amounts of liquidations because of the over-confidence in their stability.

Bibliography

[15] *contract with 11,901,464 ether? What does it do?* [accessed 12-February-2019]. 2015. URL: https://www.reddit.com/r/ethereum/comments/3gi0qn/contract_with_11901464_ether_what_does_it_do/.

[16] *Critical ether token wrapper vulnerability - eth tokens salvaged from potential attacks.* [accessed 9-February-2019]. 2016. URL: https://www.reddit.com/r/MakerDAO/comments/4niu10/critical_ether_token_wrapper_vulnerability_eth/.

[17a] *EIP 609: Hardfork Meta: Byzantium.* 2017. URL: https://eips.ethereum.org/EIPS/eip-609.

[17b] *EIP 779: Hardfork Meta: DAO Fork.* 2017. URL: https://eips.ethereum.org/EIPS/eip-779.

[17c] *Source code of the Ethereum Foundation Multisig wallet.* [accessed 12-February-2019]. 2017. URL: https://github.com/ethereum/dapp-bin/blob/master/wallet/wallet.sol.

[17d] *The DAO Refunds.* [accessed 12-February-2019]. 2017. URL: https://theethereum.wiki/w/index.php/The_DAO_Refunds.

[17e] *What's become of the ethereumpyramid?* [accessed 10-May-2019]. 2017. URL: https://www.reddit.com/r/ethtrader/comments/7eimrs/whats_become_of_the_ethereumpyramid/.

[18] *We Got Spanked: What We Know So Far*. [accessed 9-February-2019]. 2018. URL: https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe.

[19a] *0x Bug bounty*. [accessed 12-February-2019]. 2019. URL: https://0x.org/wiki#Deployed-Addresses.

[19b] *Ethereum - Github*. [accessed 8-September-2019]. 2019. URL: https://github.com/ethereum.

[19c] *Ethereum Smart Contract Best Practices*. [accessed 21-January-2019]. 2019. URL: https://consensys.github.io/smart-contract-best-practices/ (visited on 01/20/2019).

[19d] *golem — Computing Power. Shared*. [accessed 12-February-2019]. 2019. URL: https://golem.network/.

[19e] *MakerDAO*. [accessed 9-February-2019]. 2019. URL: https://makerdao.com/en/.

[19f] *Official Go implementation of the Ethereum protocol*. [accessed 21-January-2019]. 2019. URL: https://github.com/ethereum/go-ethereum.

[19g] *Solidified*. [accessed 11-February-2019]. 2019. URL: https://solidified.io/faq.

[20] *Curve.fi*. [accessed 20-08-2020]. 2020. URL: https://www.curve.fi/.

[22] *go-ethereum: Hardware requirements*. 2022. URL: https://solana.com/.

[23a] *DefiLlama - DeFi Dashboard*. [accessed 1-February-2023]. 2023. URL: https://defillama.com/.

[23b] *Etherscan — Ethereum (ETH) Blockchain Explorer*. [accessed 21-January-2019]. 2023. URL: https://etherscan.io.

[23c] *Web3 infrastructure for everyone*. 2023. URL: https://geth.ethereum.org/docs/getting-started/hardware-requirements.

[AAV20a] AAVE. *AAVE*. [accessed 17-08-2020]. 2020. URL: https://aave.com/.

[AAV20b] AAVE. *AAVE: Protocol Whitepaper V1.0*. [accessed 13-August-2020]. 2020. URL: https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf.

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on Ethereum smart contracts (SoK)". In: *POST*. 2017. ISBN: 9783662544549. DOI: 10.1007/978-3-662-54455-6_8.

[Al-+17] Mustafa Al-Bassam et al. "Chainspace: A sharded smart contracts platform". In: *arXiv preprint arXiv:1708.03778* (2017).

[Alb+08] Elvira Albert et al. "Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis". In: *Static Analysis*. Ed. by María Alpuente and Germán Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 221–237. ISBN: 978-3-540-69166-2.

[Alb+18] Elvira Albert et al. "GASTAP: A Gas Analyzer for Smart Contracts". In: *CoRR* abs/1811.1 (Nov. 2018). arXiv: 1811.10403. URL: http://arxiv.org/abs/1811.10403.

[Ami19] Amit Panghal. *The Lifecycle of an Operation in Tezos*. 2019. URL: https://medium.com/tqtezos/lifecycle-of-an-operation-in-tezos-248c51038ec2.

[Arl18] Jacob Arluck. *Liquid Proof-of-Stake*. 2018. URL: https://medium.com/tezos/liquid-proof-of-stake-aec2f7ef1da7.

[Ato19] AtoZ Markets. *Binance facing XRP withdrawal issues*. 2019. URL: https://atozmarkets.com/news/binance-facing-xrp-withdrawal-issues/.

[ATS21] Rachit Agarwal, Tanmay Thapliyal, and Sandeep K. Shukla. "Vulnerability and Transaction behavior based detection of Malicious Smart Contracts". In: *International Conference on Cryptography and Security Systems*. 2021.

[Aut19] The go-ethereum Authors. *Official Go implementation of the Ethereum protocol*. [accessed 25-August-2019]. 2019. URL: https://github.com/ethereum/go-ethereum/.

[AW10] Hervé Abdi and Lynne J Williams. "Principal component analysis". In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459.

[Bar+97] Bruno Barras et al. "The Coq proof assistant reference manual: Version 6.1". PhD thesis. Inria, 1997.

[BCL20] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. "SoK: Lending Pools in Decentralized Finance". In: *arXiv preprint arXiv:2012.13230* (2020).

[BD19] BTG Pactual and Dalma Capital. *BTG Pactual and Dalma Capital to utilize Tezos Blockchain for Security Token Offerings (STOs)*. 2019. URL: https://www.prnewswire.co.uk/news-releases/btg-pactual-and-dalma-capital-to-utilize-tezos-blockchain-for-security-token-offerings-stos--880726956.html.

[Bia+19] Bruno Biais et al. "The blockchain folk theorem". In: *The Review of Financial Studies* 32.5 (2019), pp. 1662–1715.

[Bit18] Bitshares. *Delegated Proof of Stake (DPOS)*. 2018. URL: https://github.com/bitshares/how.bitshares.works/blob/master/docs/technology/dpos.rst.

[Bjø+12] Nikolaj Bjørner et al. "Program verification as satisfiability modulo theories". In: *In SMT*. 2012.

[BKP14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. "Deanonymisation of Clients in Bitcoin P2P Network". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 15–29. ISBN: 9781450329576. DOI: 10.1145/2660267.2660379. URL: https://doi.org/10.1145/2660267.2660379.

[blo18] block.one. *EOS.IO Technical White Paper v2*. [accessed 30-June-2020]. 2018. URL: https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.

[Blo19] Block.one. *About EOSIO*. [accessed 30-June-2020]. 2019. URL: https://eos.io/about-us/.

[Bos12] Sarah Boslaugh. *Statistics in a nutshell: A desktop quick reference.* " O'Reilly Media, Inc.", 2012.

[Bre+17] Lorenz Breidenbach et al. *An In-Depth Look at the Parity Multisig Bug.* [accessed 2-February-2020]. 2017. URL: https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[Bre+18] Lexi Brent et al. *Vandal: A Scalable Security Analysis Framework for Smart Contracts.* Tech. rep. 2018. arXiv: arXiv:1809.03981v1. URL: https://github.com/usyd-blockchain/vandal.

[BS20] Vitalik Buterin and Martin Holst Swende. *EIP-2929: Gas cost increases for state access opcodes.* [accessed 30-June-2021]. 2020. URL: https://eips.ethereum.org/EIPS/eip-2929.

[Buta] Vitalik Buterin. *EIP 150: Gas cost changes for IO-heavy operations.* [accessed 5-June-2019]. URL: https://eips.ethereum.org/EIPS/eip-150.

[Butb] Vitalik Buterin. *Geth nodes under attack again.* [accessed 4-April-2019]. URL: https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/?st=itxh568s&sh=ee3628ea.

[Butc] Vitalik Buterin. *Transaction spam attack: Next Steps.* [accessed 4-April-2019]. URL: https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/.

[But+19] Vitalik Buterin et al. *EIP-1559: Fee market change for ETH 1.0 chain.* [accessed 15-January-2020]. 2019. URL: https://eips.ethereum.org/EIPS/eip-1559.

[But14] Vitalik Buterin. "A next-generation smart contract and decentralized application platform". In: *Ethereum* January (2014), pp. 1–36. URL: http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf.

[But15] Vitalik Buterin. *EIP 7.* [accessed 21-January-2019]. 2015. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-7.md.

[But16] Vitalik Buterin. *Geth nodes under attack again (Geth issue).* [accessed 5-September-2019]. 2016. URL: https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/d8ebsad/.

[But19] Vitalik Buterin. *EIP 210.* [accessed 20-July-2019]. 2019. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-210.md.

[Che+17a] Ting Chen et al. "An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks". In: *Information Security Practice and Experience.* Ed. by Joseph K. Liu and Pierangela Samarati. Cham: Springer International Publishing, 2017, pp. 3–24. ISBN: 978-3-319-72359-4.

[Che+17b] Ting Chen et al. "Under-optimized smart contracts devour your money". In: *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering* (2017), pp. 442–446. DOI: 10.1109/SANER.2017.7884650. arXiv: 1703.03994.

[Cir20] Circle. *USDC.* [accessed 20-08-2020]. 2020. URL: https://www.circle.com/en/usdc.

[CM18] Brad Chase and Ethan Macbrough. *Analysis of the XRP Ledger Consensus Protocol.* Tech. rep. Ripple Labs, Inc., Feb. 2018. URL: http://arxiv.org/abs/1802.07242.

[Coi20] CoinMarketCap. *Tezos (XTZ) price, charts, market cap, and other metrics.* 2020. URL: https://coinmarketcap.com/currencies/tezos/.

[Com19a] Concourse Open Community. *ETH Gas Station.* [accessed 9-September-2019]. 2019. URL: https://ethgasstation.info/calculatorTxV.php.

[Com19b] Compound. *Compound.* [accessed 17-08-2020]. 2019. URL: https://compound.finance/.

[Con19a] ConSensys. *A development framework for Ethereum.* [accessed 21-January-2019]. 2019. URL: https://github.com/trufflesuite/truffle.

[Con19b] ConSensys. *Personal blockchain for Ethereum development.* [accessed 21-January-2019]. 2019. URL: https://github.com/trufflesuite/ganache.

[Con19c] ConsenSys. *Mythril Classic*. [accessed 21-January-2019]. 2019. URL: https://github.com/ConsenSys/mythril-classic.

[Cry19] Cryptium Labs. *Babylon 2.0 (PsBABY5HQ)*. 2019. URL: https://www.tezosagora.org/proposal/5.

[Cyr20] Cyrus. *Upcoming COMP farming change could impact the Dai peg*. [accessed 27-August-2020]. 2020. URL: https://forum.makerdao.com/t/upcoming-comp-farming-change-could-impact-the-dai-peg/2965.

[Dai+19] Philip Daian et al. "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized ex changes". In: *arXiv preprint arXiv:1904.05234* (2019).

[Dai+20] Philip Daian et al. "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 910–927.

[Dan17] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. 1st. Berkely, CA, USA: Apress, 2017. ISBN: 978-1-4842-2534-9.

[Dat20] Blockwatch Data. *Cryptium Labs Payouts*. 2020. URL: https://tzstats.com/tz1cNARmnRRrvZgspPr2rSTUWq5xtGTuKuHY.

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[DMR17] Damiano Di Francesco Maesa, Andrea Marino, and Laura Ricci. "An analysis of the Bitcoin users graph: inferring unusual behaviours". In: *Complex Networks & Their Applications V*. Ed. by Hocine Cherifi et al. Cham: Springer International Publishing, 2017, pp. 749–760. ISBN: 978-3-319-50901-3.

[dYd19] dYdX. *dYdX*. [accessed 20-08-2020]. 2019. URL: https://dydx.exchange/.

[Ear19] EarnBet. *EOS 30 Day Notice*. 2019.

[enu19] enumivo. *Get Free EIDOS*. [accessed 30-June-2020]. 2019. URL: https://enumivo.org/get-free-eidos.

[EOS19] EOSIO. *About System Contracts*. [accessed 30-June-2020]. 2019. URL: https://eosio.github.io/eosio.contracts/latest/index.

[EOS20a] EOSDocs.io. *API endpoints*. [accessed 30-June-2020]. 2020. URL: https://www.eosdocs.io/resources/apiendpoints/.

[EOS20b] EOSIO. *EOSIO Resource Allocation Proposal*. [accessed 30-June-2020]. 2020. URL: https://eos.io/news/eosio-resource-allocation-proposal/.

[EOS20c] EOSIO. *eosio.contracts*. [accessed 30-June-2020]. 2020. URL: https://github.com/EOSIO/eosio.contracts/blob/master/README.md.

[EOS20d] EOSIO. *RPC API Guide*. [accessed 30-June-2020]. 2020. URL: https://developers.eos.io/eosio-nodeos/reference#get_account.

[ES14] Ittay Eyal and Emin Gün Sirer. "Majority is not enough: Bitcoin mining is vulnerable". In: *International conference on financial cryptography and data security*. Springer. 2014, pp. 436–454.

[Eth] Ethereum community. *cpp-ethereum*. [accessed 1-May-2019]. URL: http://www.ethdocs.org/en/latest/ethereum-clients/cpp-ethereum/.

[Eth19] Etherscan. *Ethereum Average Block TimeChart*. [accessed 9-September-2019]. 2019. URL: https://etherscan.io/chart/blocktime.

[Eth20a] Ethereum Foundation. *Ethereum Bounty Program*. [accessed 5-January-2020]. 2020. URL: https://bounty.ethereum.org/.

[Eth20b] Etherscan. *Ethereum Average Block Time Chart*. [accessed 10-January-2020]. 2020. URL: https://etherscan.io/chart/blocktime.

[eth20] ethernodes.org. *Ethereum Mainnet Statistics*. [accessed 10-January-2020]. 2020. URL: https://www.ethernodes.org/.

[FAH20] Joel Frank, Cornelius Aschermann, and Thorsten Holz. "ETHBMC: A Bounded Model Checker for Smart Contracts". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2757–2774. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/frank.

[Fou20] Tezos Foundation. *Tezos Foundation's Faucet*. [accessed 20-September-2020]. 2020. URL: https://faucet.tezos.com/.

[Gal17] Max Galka. *Multisig wallets affected by the Parity wallet bug*. [accessed 21-January-2019]. 2017. URL: https://github.com/elementus-io/parity-wallet-freeze.

[GD11a] Sanjay Ghemawat and Jeff Dean. *LevelDB*. [accessed 5-August-2019]. 2011. URL: https://github.com/google/leveldb.

[GD11b] Sanjay Ghemawat and Jeff Dean. *LevelDB documentation*. [accessed 5-August-2019]. 2011. URL: https://github.com/google/leveldb/blob/master/doc/index.md.

[GMS18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "A Semantic Framework for the Security Analysis of Ethereum Smart Contracts". In: *Principles of Security and Trust*. Ed. by Lujo Bauer and Ralf Küsters. Cham: Springer International Publishing, 2018, pp. 243–269. ISBN: 978-3-319-89722-6.

[Goo14] L M Goodman. *Tezos—a self-amending crypto-ledger White paper*. Tech. rep. 2014.

[Goo19] Google. *Google Compute Engine documentation*. [accessed 8-September-2019]. 2019. URL: https://cloud.google.com/compute/docs/.

[GPL19] A Gaihre, S Pandey, and H Liu. "Deanonymizing Cryptocurrency With Graph Learning: The Promises and Challenges". In: *2019 IEEE Conference on Communications and Network Security (CNS)*. June 2019, pp. 1–3. DOI: 10.1109/CNS.2019.8802640.

[GPS+17] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. "Program synthesis". In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119.

[Gre+18] Neville Grech et al. "MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 116:1–116:27. ISSN: 2475-1421. DOI: 10.1145/3276486. URL: http://doi.acm.org/10.1145/3276486.

[Gud+20a] Lewis Gudgeon et al. "DeFi Protocols for Loanable Funds: Interest Rates, Liquidity and Market Efficiency". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. AFT '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 92–112. ISBN: 9781450381390. DOI: 10.1145/3419614.3423254. URL: https://doi.org/10.1145/3419614.3423254.

[Gud+20b] Lewis Gudgeon et al. "The decentralized financial crisis". In: *2020 crypto valley conference on blockchain technology (CVCBT)*. IEEE. 2020, pp. 1–15.

[Har+19] Dominik Harz et al. "Balance: Dynamic adjustment of cryptocurrency deposits". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1485–1502.

[Has19] HashBaby. *Mark's HBS, from dapp "vulgar rights" to Defi "industry foundation"*. [accessed 30-September-2020]. 2019. URL: https://bit.ly/hashbaby-closing.

[He+21] Ningyu He et al. "Understanding the Evolution of Blockchain Ecosystems: A Longitudinal Measurement Study of Bitcoin, Ethereum, and EOSIO". In: *ArXiv* abs/2110.07534 (2021).

[He+22] Ningyu He et al. "A Survey on EOSIO Systems Security: Vulnerability, Attack, and Mitigation". In: *ArXiv* abs/2207.09227 (2022).

[Hil+18] Everett Hildenbrandt et al. "KEVM: A complete formal semantics of the ethereum virtual machine". In: *Proceedings - IEEE Computer Security Foundations Symposium*. 2018. ISBN: 9781538666807. DOI: 10.1109/CSF.2018.00022.

[Hir17] Yoichi Hirai. "Defining the Ethereum Virtual Machine for Interactive Theorem Provers". In: *Workshop on Trusted Smart Contracts*. 2017.

[Hua+20] Yuheng Huang et al. *Characterizing EOSIO Blockchain*. 2020. arXiv: 2002.05369 [cs.CR].

[Hud19] Hudson Jameson. *Ethereum Constantinople Upgrade Announcement*. [accessed 5-July-2019]. 2019. URL: https://blog.ethereum.org/2019/01/11/ethereum-constantinople-upgrade-announcement/.

[Imm99] Neil Immerman. *Descriptive Complexity*. 1999.

[JLC18] Bo Jiang, Ye Liu, and WK Chan. "Contractfuzzer: Fuzzing smart contracts for vulnerability detection". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. 2018, pp. 259–269.

[Kal+18] Sukrit Kalra et al. "ZEUS: Analyzing Safety of Smart Contracts". In: *25th Annual Network and Distributed System Security Symposium, {NDSS} 2018, San Diego, California, USA, February 18-21, 2018*. 2018. DOI: 10.14722/ndss.2018.23082. URL: http://dx.doi.org/10.14722/ndss.2018.23082.

[Kao+20a] H T Kao et al. "An Analysis of the Market Risk to Participants in the Compound Protocol". In: 2020. URL: https://scfab.github.io/2020/FAB2020_p5.pdf.

[Kao+20b] Hsien-Tang Kao et al. "An analysis of the market risk to participants in the compound protocol". In: *Third International Symposium on Foundations and Applications of Blockchains*. 2020.

[Kau19] Josh Kauffman. *What Are EOSIO System Accounts and What Do They Each Do?* 2019. URL: https://www.eoscanada.com/en/what-are-eosio-system-accounts-and-what-do-they-each-do.

[Kla+20] Ariah Klages-Mundt et al. "Stablecoins 2.0: Economic Foundations and Risk-based Models". In: *2nd ACM Conference on Advances in Financial Technologies (AFT '20)*. New York, 2020. DOI: 10.1145/3419614.3423261.

[KM19] Ariah Klages-Mundt and Andreea Minca. "(In) Stability for the Blockchain: Deleveraging Spirals and Stablecoin Attacks". In: *arXiv preprint arXiv:1906.02152* (2019).

[KO19] Melanie Kramer and Michael O'Sullivan. *Could EOSIO be the Foundation of a Blockchain Future*. 2019. URL: https://bywire.news/articles/could-eosio-be-the-foundation-of-a-blockchain-future.

[Kon+14] Dániel Kondor et al. "Do the Rich Get Richer? An Empirical Analysis of the Bitcoin Transaction Network". In: *PLOS ONE* 9.2 (2014), pp. 1–10. ISSN: 19326203. DOI: 10.1371/journal.pone.0086197. URL: https://doi.org/10.1371/journal.pone.0086197.

[KR18] Johannes Krupp and Christian Rossow. "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts". In: *USENIX Security* (2018).

[KSA21] Muhammad Milhan Afzal Khan, Hafiz Muhammad Awais Sarwar, and Muhammad Awais. "Gas consumption analysis of Ethereum blockchain transactions". In: *Concurrency and Computation: Practice and Experience* 34 (2021).

[KV21] Roman Kozhan and Ganesh Viswanath-Natraj. "Decentralized stablecoins and collateral risk". In: *WBS Finance Group Research Paper* (2021).

[LA04] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[Lab19] Obsidian Labs. *eosio.token*. [accessed 30-June-2020]. 2019. URL: https://docs.eosstudio.io/contracts/eosio-token.html.

[Lab20] Cryptium Labs. *backerei — Automated reward payment & account management for Tezos bakers*. [accessed 30-November-2020]. 2020. URL: https://github.com/cryptiumlabs/backerei.

[LH18] Robert Leshner and Geoffrey Hayes. *Compound : The Money Market Protocol*. Tech. rep. 2018, pp. 1–10.

[LH19] Robert Leshner and Geoffrey Hayes. *Compound: The money market protocol*. Tech. rep. Compound Finance, Tech. Rep, 2019.

[Lov+13] Robert Anthony Love et al. *Value tracking and reporting of automated clearing house transactions*. 2013. URL: https://patentimages.storage.googleapis.com/03/e3/98/0cfb0fe7ee16e9/US8543477.pdf.

[LS20] Bowen Liu and Pawel Szalachowski. *A First Look into DeFi Oracles*. 2020. arXiv: 2005.04377 [cs.CR].

[Luu+16a] Loi Luu et al. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM. 2016, pp. 254–269. DOI: 10.1145/2976749.2978309.

[Luu+16b] Loi Luu et al. *Oyente Benchmarks*. [accessed 19-November-2018]. 2016. URL: https://oyente.tech/benchmarks/.

[LWT21] Kai Li, Yibo Wang, and Yuzhe Tang. "DETER: Denial of Ethereum Txpool SERvices". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1645–1667. ISBN: 9781450384544. DOI: 10.1145/3460120.3485369. URL: https://doi.org/10.1145/3460120.3485369.

[Mak] Maker. *The Maker Protocol: MakerDAO's Multi-Collateral Dai (MCD) System*. [accessed 8-June-2020]. URL: https://makerdao.com/en/whitepaper/.

[Mar+18] Matteo Marescotti et al. "Computing Exact Worst-Case Gas Consumption for Smart Contracts". In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2018, pp. 450–465. ISBN: 978-3-030-03427-6.

[McG+16] Dan McGinn et al. "Visualizing dynamic bitcoin transaction patterns". In: *Big data* 4.2 (2016), pp. 109–119.

[McM07] Kenneth L McMillan. "Interpolants and symbolic model checking". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2007, pp. 89–90.

[Meh+19] Muhammad Izhar Mehar et al. "Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack". In: *Journal of Cases on Information Technology (JCIT)* 21.1 (2019), pp. 19–32.

[Mit02] Michael Mitzenmacher. "Compressed bloom filters". In: *IEEE/ACM Transactions on Networking (TON)* 10.5 (2002), pp. 604–612.

[MMT16] P Monamo, V Marivate, and B Twala. "Unsupervised learning for robust Bitcoin fraud detection". In: *2016 Information Security for South Africa (ISSA)*. Aug. 2016, pp. 129–134. DOI: 10.1109/ISSA.2016.7802939.

[MP23] Toshiko Matsui and Daniel Perez. "Data-driven analysis of central bank digital currency (CBDC) projects drivers". In: *Mathematical Research for Blockchain Economy: 3rd International Conference MARBLE 2022, Vilamoura, Portugal*. Springer. 2023, pp. 95–108.

[Mul+14] Dominic P Mulligan et al. "Lem: reusable engineering of real-world semantics". In: *ACM SIGPLAN Notices*. Vol. 49. 9. ACM. 2014, pp. 175–188.

[Nik+18] Ivica Nikolić et al. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: ACM, 2018, pp. 653–663. ISBN: 978-1-4503-6569-7. DOI: 10.1145/3274694.3274743. URL: http://doi.acm.org/10.1145/3274694.3274743.

[Nom18a] Nomadic Labs. *Delegating your coins*. 2018. URL: https://tezos.gitlab.io/introduction/howtorun.html#delegating-your-coins.

[Nom18b] Nomadic Labs. *Delegation*. [accessed 30-September-2020]. 2018. URL: http://tezos.gitlab.io/whitedoc/proof_of_stake.html#delegation.

[Nom18c] Nomadic Labs. *Michelson: the language of Smart Contracts in Tezos*. [accessed 30-September-2020]. 2018. URL: https://tezos.gitlab.io/whitedoc/michelson.html.

[Nom18d] Nomadic Labs. *Originated accounts and contracts*. [accessed 30-September-2020]. 2018. URL: https://tezos.gitlab.io/introduction/howtouse.html.

[Obs19] ObsidianSys. *Summary of Ledger Support for the Babylon Protocol.* [accessed 30-September-2020]. 2019. URL: https://www.reddit.com/r/tezos/comments/dj7g9y/summary_of_ledger_support_for_the_babylon_protocol/.

[Owo18] Kevin Owocki. *A Brief History Of Gas Prices on Ethereum.* [accessed 5-August-2019]. 2018. URL: https://gitcoin.co/blog/a-brief-history-of-gas-prices-on-ethereum/.

[Pal19] Palau, Albert. *Analyzing the hardware requirements to be an Ethereum full validated node.* [accessed 8-September-2019]. 2019. URL: https://medium.com/coinmonks/analyzing-the-hardware-requirements-to-be-an-ethereum-full-validated-node-dc064f167902.

[Par20] Parity Technologies. *Parity Ethereum.* [accessed 5-January-2020]. 2020. URL: https://www.parity.io/ethereum/.

[Pec20] Marcel Pechman. *Record Ethereum Network Use and Gas Fees Pose Risk to DeFi Expansion.* [accessed 30-September-2020]. 2020. URL: https://cointelegraph.com/news/record-ethereum-network-use-and-gas-fees-pose-risk-to-defi-expansion.

[Peg19] PegaSys. *Pantheon Ethereum Client System requirements.* [accessed 8-September-2019]. 2019. URL: http://docs.pantheon.pegasys.tech/en/latest/HowTo/Get-Started/System-Requirements/.

[Per+19] Anton Permenev et al. "Verx: Safety verification of smart contracts". In: *Security and Privacy* 2020 (2019).

[Per+21] Daniel Perez et al. "Liquidations: DeFi on a Knife-Edge". In: *Financial Cryptography and Data Security.* Ed. by Nikita Borisov and Claudia Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 457–476. ISBN: 978-3-662-64331-0.

[Pet18] Petrov, Andrev. *An economic incentive for running Ethereum full nodes.* [accessed 8-September-2019]. 2018. URL: https://medium.com/vipnode/an-economic-incentive-for-running-ethereum-full-nodes-ecc0c9ebe22.

[PG23] Daniel Perez and Lewis Gudgeon. "Dissimilar redundancy in DeFi". In: *Mathematical Research for Blockchain Economy: 3rd International Conference MARBLE 2022, Vilamoura, Portugal*. Springer. 2023, pp. 109–125.

[PK15] Jack Peterson and Joseph Krug. "Augur: a decentralized, open-source platform for prediction markets". In: *arXiv preprint arXiv:1501.01042* (2015).

[PL20] Daniel Perez and Benjamin Livshits. "Broken Metre: Attacking Resource Metering in EVM". In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/broken-metre-attacking-resource-metering-in-evm/.

[PL21] Daniel Perez and Benjamin Livshits. "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1325–1341. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/perez.

[PPL20] Paul J Pritz, Daniel Perez, and Kin K Leung. "Fast-fourier-forecasting resource utilisation in distributed systems". In: *2020 29th International conference on computer communications and networks (ICCCN)*. IEEE. 2020, pp. 1–9.

[Put18] Dani Putney. *The AZTEC Protocol: A Zero-Knowledge Privacy System On Ethereum*. [accessed 23-August-2019]. 2018. URL: https://www.ethnews.com/the-aztec-protocol-a-zero-knowledge-privacy-system-on-ethereum.

[PXL20] Daniel Perez, Jiahua Xu, and Benjamin Livshits. "Revisiting Transactional Statistics of High-scalability Blockchains". In: *ACM Internet Measurement Conference*. Vol. 16. 20. New York, NY, USA: ACM, Oct. 2020, pp. 535–550. ISBN: 9781450381383. URL: https://dl.acm.org/doi/10.1145/3419394.3423628.

[Qin+21] Kaihua Qin et al. "An Empirical Study of DeFi Liquidations: Incentives, Risks, and Instabilities". In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. Virtual Event: Association for Computing Machinery, 2021,

pp. 336–350. ISBN: 9781450391290. DOI: 10.1145/3487552.3487811. URL: https://doi.org/10.1145/3487552.3487811.

[Ran+17] Stephen Ranshous et al. “Exchange Pattern Mining in the Bitcoin Transaction Directed Hypergraph”. In: *Financial Cryptography and Data Security.* Ed. by Michael Brenner et al. Cham: Springer International Publishing, 2017, pp. 248–263. ISBN: 978-3-319-70278-0.

[REC19] R. Rahimian, S. Eskandari, and J. Clark. “Resolving the Multiple Withdrawal Attack on ERC20 Tokens”. In: *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW).* June 2019, pp. 320–329. DOI: 10.1109/EuroSPW.2019.00042.

[Rip20] Ripple. *Benefits of XRP in Payments.* 2020. URL: https://ripple.com/xrp/.

[Rod+19] Michael Rodler et al. “Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks”. In: *Proceedings of 26th Annual Network & Distributed System Security Symposium (NDSS).* Feb. 2019. URL: http://tubiblio.ulb.tu-darmstadt.de/111410/.

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. “An Overview of the K Semantic Framework”. In: *Journal of Logic and Algebraic Programming* 79.6 (2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[RS13] Dorit Ron and Adi Shamir. “Quantitative Analysis of the Full Bitcoin Transaction Graph”. In: *Financial Cryptography and Data Security.* Ed. by Ahmad-Reza Sadeghi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 6–24. ISBN: 978-3-642-39884-1.

[SC17] Us Securities and Exchange Commission. *Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO.* Tech. rep. 2017. URL: http://www.virtualschool.edu/mon/Economics/SmartContracts.html..

[SW13] Remon Sinnema and Erik Wilde. *eXtensible Access Control Markup Language (XACML) XML Media Type.* [accessed 21-January-2019]. 2013. URL: https://tools.ietf.org/html/rfc7061.

[Swa+11] Nikhil Swamy et al. "Secure distributed programming with value-dependent types". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 266–278. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034811. URL: https://www.microsoft.com/en-us/research/publication/secure-distributed-programming-with-value-dependent-types/.

[Swe19] Martin Holst Swende. *EIP 1184*. [accessed 15-January-2020]. 2019. URL: https://eips.ethereum.org/EIPS/eip-1884.

[Syn20] Synthetix. *Litepaper*. [accessed 6-December-2020]. 2020. URL: https://docs.synthetix.io/litepaper/.

[Szi19] Szilágyi, Péter. *Dynamic state snapshot*. [accessed 5-January-2020]. 2019. URL: https://github.com/ethereum/go-ethereum/pull/20152.

[Tan] Wei Tang. *EIP 2200: Structured Definitions for Net Gas*. [accessed 10-January-2019]. URL: https://eips.ethereum.org/EIPS/eip-2200.

[Tea19] EIDOS Team. *YAS Network*. [accessed 30-June-2020]. 2019. URL: https://enumivo.org/blog/2019/11/24/yas-network.

[Tea20] Team Ripple. *Q4 2019 XRP Markets Report*. [accessed 8-September-2020]. 2020. URL: https://ripple.com/insights/q4-2019-xrp-markets-report/.

[Tec20] Scrambled Egg Technologies. *XRP Scan | Ripple XRP ledger explorer*. [accessed 19-September-2020]. 2020. URL: https://xrpscan.com/.

[Tez18] Tezos. *Proof-of-stake in Tezos*. [accessed 20-September-2020]. 2018. URL: https://tezos.gitlab.io/whitedoc/proof_of_stake.html.

[Tez19a] Tezos. *About Tezos*. [accessed 4-June-2019]. 2019. URL: https://tezos.com/learn-about-tezos.

[Tez19b] Tezos Agora. *Tezos Brest A amendment*. [accessed 20-September-2020]. 2019. URL: https://www.tezosagora.org/period/15.

[Tik+18] S. Tikhomirov et al. “SmartCheck: Static Analysis of Ethereum Smart Contracts”. In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. May 2018, pp. 9–16.

[TOM17] K Toyoda, T Ohtsuki, and P T Mathiopoulos. “Identification of High Yielding Investment Programs in Bitcoin via Transactions Pattern Analysis”. In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. Dec. 2017, pp. 1–6. DOI: 10.1109/GLOCOM.2017.8254420.

[TS+18] Christof Ferreira Torres, Julian Schütte, et al. “Osiris: Hunting for integer bugs in ethereum smart contracts”. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM. 2018, pp. 664–676.

[Tsa+18] Petar Tsankov et al. “Securify: Practical Security Analysis of Smart Contracts”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18 July (2018), pp. 67–82. DOI: 10.1145/3243734.3243780. arXiv: 1806.01143. URL: http://doi.acm.org/10.1145/3243734.3243780%20http://arxiv.org/abs/1806.01143.

[TSS19] Christof Ferreira Torres, Mathis Steichen, and Radu State. “The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1591–1607. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira.

[tul19] tulo. *Spam? suspect BTC payments on XRPL*. [accessed 8-September-2020]. 2019. URL: https://www.xrpchat.com/topic/33284-spam-suspect-btc-payments-on-xrpl/?do=findComment&comment=793048.

[Ukr20] Tezos Ukraine. *Tezos Ukraine Infrastructure*. [accessed 20-September-2020]. 2020. URL: https://api.tezos.org.ua/.

[Ull84] Jeffrey D Ullman. *Principles of database systems*. Galgotia publications, 1984.

[Uni20] Uniswap. *Uniswap Whitepaper*. [accessed 26-August-2020]. 2020. URL: https://hackmd.io/@HaydenAdams/HJ9jLsfTz#%F0%9F%A6%84-Uniswap-Whitepaper.

[Vis20] Visa. *Visa Fact Sheet*. 2020. URL: https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf.

[Wan+20] Dabao Wang et al. "Towards understanding flash loan and its applications in defi ecosystem". In: (Oct. 2020). URL: http://arxiv.org/abs/2010.12252.

[WB17] Will Warren and Amir Bandeali. "0x: An open protocol for decentralized exchange on the Ethereum blockchain". 2017.

[Wer+21] Sam M. Werner et al. "SoK: Decentralized Finance (DeFi)". Jan. 2021. URL: http://arxiv.org/abs/2101.08778.

[Wha20] WhaleEx. *WhaleEx - #1 DEX*. [accessed 30-September-2020]. 2020. URL: https://www.whaleex.com/.

[Whi94] Darrell Whitley. "A genetic algorithm tutorial". In: *Statistics and computing* 4.2 (1994), pp. 65–85.

[Win20] Wietse Wind. *XRP ledger full history cluster*. [accessed 8-September-2020]. 2020. URL: https://rippled.xrptipbot.com/.

[Woo14] Gavin Wood. *Ethereum yellow paper*. [accessed 12-February-2019]. 2014. URL: http://gavwood.com/paper.pdf.

[Woo19] Gavin Wood. *Ethereum: a secure decentralised generalised transaction ledger*. Tech. rep. 2019, pp. 1–32. DOI: 10.1017/CBO9781107415324.004.

[WP20] Sam M. Werner and Daniel Perez. "PoolSim: A Discrete-Event Mining Pool Simulation Framework". In: *Mathematical Research for Blockchain Economy*. Ed. by Panos Pardalos et al. Cham: Springer International Publishing, 2020, pp. 167–182. ISBN: 978-3-030-37110-4.

[WPP20] Sam M. Werner, Paul J. Pritz, and Daniel Perez. "Step on the Gas? A Better Approach for Recommending the Ethereum Gas Price". In: *Mathematical Research for Blockchain Economy*. Springer, Mar. 2020, pp. 161–177. URL: http://link.springer.com/10.1007/978-3-030-53356-4_10.

[Xie+19] Junfeng Xie et al. "A Survey on the Scalability of Blockchain Systems". In: *IEEE Network* 33.5 (2019), pp. 166–173. ISSN: 1558156X. DOI: 10.1109/MNET.001.1800290.

[XRP19a] XRP Ledger. *Accounts*. [accessed 19-September-2020]. 2019. URL: https://xrpl.org/accounts.html#special-addresses.

[XRP19b] XRP Ledger. *Fast, Efficient Consensus Algorithm*. [accessed 19-September-2020]. 2019. URL: https://xrpl.org/xrp-ledger-overview.html#fast-efficient-consensus-algorithm.

[XRP19c] XRP Ledger Project. *Fee Voting*. [accessed 19-September-2020]. 2019. URL: https://xrpl.org/fee-voting.html.

[XRP20a] XRP Ledger. *Get Exchange Rates*. [accessed 19-September-2020]. 2020. URL: https://xrpl.org/data-api.html#get-exchange-rates.

[XRP20b] XRP Ledger. *Source and Destination Tags*. [accessed 19-September-2020]. 2020. URL: https://xrpl.org/source-and-destination-tags.html.

[XRP20c] XRP Ledger. *Trust Lines and Issuing*. [accessed 19-September-2020]. 2020. URL: https://xrpl.org/trust-lines-and-issuing.html.

[Xu+23a] Jiahua Xu et al. "Auto-gov: Learning-based On-chain Governance for Decentralized Finance (DeFi)". In: *arXiv preprint arXiv:2302.09551* (2023). URL: https://arxiv.org/abs/2302.09551.

[Xu+23b] Jiahua Xu et al. "SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols". In: *ACM Comput. Surv.* 55.11 (Feb. 2023). ISSN: 0360-0300. DOI: 10.1145/3570639. URL: https://doi.org/10.1145/3570639.

[Xu+23c] Jiahua Xu et al. "SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols". In: *ACM Comput. Surv.* 55.11 (Feb. 2023). ISSN: 0360-0300. DOI: 10.1145/3570639. URL: https://doi.org/10.1145/3570639.

[Yan+19] Renlord Yang et al. "Empirically Analyzing Ethereum's Gas Mechanism". In: *CoRR* abs/1905.0 (2019). arXiv: 1905.00553. URL: http://arxiv.org/abs/1905.00553.

[Zam+20] Alexei Zamyatin et al. "TxChain: Efficient Cryptocurrency Light Clients via Contingent Transaction Aggregation". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology.* Ed. by Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomarti. Cham: Springer International Publishing, 2020, pp. 269–286. ISBN: 978-3-030-66172-4.

[Zha+20] Mengya Zhang et al. "TXSPECTOR: Uncovering Attacks in Ethereum from Transactions". In: *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, Aug. 2020, pp. 2775–2792. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-mengya.

[Zhe+17] P Zheng et al. "A Detailed and Real-Time Performance Monitoring Framework for Blockchain Systems". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP).* May 2017, pp. 134–143.