Advances in Static Analysis and Formal Verification of Smart Contracts

by

Soroush Farokhnia

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

January 2026, Hong Kong

Advances in Static Analysis and Formal Verification of Smart Contracts

by

Soroush Farokhnia

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

ABSTRACT

Blockchains are a family of distributed consensus protocols that maintain a shared, append-only ledger without relying on a trusted central authority. Smart contracts are self-enforcing programs executed on blockchains. They support arbitrary complex logic. Their applications span multiple domains, including finance, healthcare, and supply chain management, in both the public and private sectors. They are currently in charge of billions of dollars' worth of digital assets. Given their role, any flaw can have significant financial consequences. Thus, many optimization and verification problems arise in this domain, yet existing approaches often produce suboptimal results, lack strong assurances, or do not scale. These limitations frequently compel developers to rely on manual audits, which are costly, error-prone, and can leave vulnerabilities undiscovered.

This thesis develops algorithmic foundations that bridge the gap between scalability and reliable guarantees for smart contracts. We achieve this by capturing structural properties of contracts using techniques from parameterized complexity theory, polyhedral geometry, and real algebraic geometry. We demonstrate how these techniques can significantly advance the state-of-the-art solutions across various domains, including gas estimation, compiler optimization, miner revenue maximization, and decentralized exchange routing. The thesis goes beyond core technical results, arguing that blockchain benefits from interdisciplinary perspectives that can uncover costly vulnerabilities otherwise overlooked.

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Soroush Farokhnia

16 January 2026

Advances in Static Analysis and Formal Verification of Smart Contracts

by

Soroush Farokhnia

This is to certify that I have examined the above PhD thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Amir Kafshdar Goharshady, Thesis Supervisor

Jiasi Shen, Thesis Supervisor

Xiaofang Zhou, Head of Department

Department of Computer Science and Engineering

16 January 2026

Dedication

To my mother,
who always placed our education above all else.

Acknowledgments

I was privileged to have Amir Goharshady as my PhD supervisor. His insights and guidance were instrumental in my growth as a researcher. He consistently supported me in navigating career choices, grant applications, teaching opportunities, and interviews. Our long discussions about the right and wrong paths, whether in research, career, or life itself, have always been invaluable to me.

I am grateful to my co-supervisor, Jiasi Shen, for her consistent support throughout my time at HKUST. Thanks are also due to S. Akshay and Supratik Chakraborty for hosting my visit to IIT Bombay and for sharing valuable perspective on both research and career development. I greatly value the mentorship of Đorđe Žikelić, which supported my professional development. Finally, I appreciate Dimitris Papadopoulos and Junxian He for agreeing to chair my thesis defense sessions.

During my PhD, I had the privilege of working alongside exceptionally talented collaborators. Sergei Novozhilov helped me navigate many complex research problems. Our stormy meetings with Harshit Motwani pushed me beyond my academic comfort zone. With Togzhan Barakbayeva, I built working solutions from our research ideas. We spent countless hours tackling new challenges, meeting tight deadlines, sharing working lunches, and making room for bold ideas. The conversations and work we shared influenced many chapters of this thesis.

I am also proud to be part of the ALPACAS group and appreciate the support from my colleagues: Jonas Ballweg, Xuran Cai, Zhuo Cai, Giovanna Kobus Conrado, Pavel Hudec, S. Hitarth, Kerim Kochekov, Chun Kit Lam, Pingjiang Li, Richard Magambo, Zhaorun Lin, Tian Shu, and Ahmed Zaher.

I owe a profound debt of gratitude to my parents, Marzieh Javameh and Saeid Farokhnia, for their unwavering support throughout my life. My brother Sepher also deserves recognition for his encouragement and our many discussions[1]. I must also express my deepest appreciation to my fiancée, Sanaz Safaei. This doctorate would not have been possible without her kind support and encouragement. It is only fitting that our next chapter begins as this one closes.

I am grateful to my teachers over the years, whose guidance has brought me

[1] Though I must note that fewer hours spent debating Middle-earth could have helped me write a couple more papers!

to this point. Mahdi Kazemi and Hossein Boomari introduced me to the world of computer science. I have repeatedly returned to Mohammadreza Shabanali's writings for direction in personal growth and in how I think and learn. In the early stages of my career, I benefited greatly from the advice of Behnam Shakibafar and Reza Sajjadi.

Finally, I am grateful to HKUST and the Ethereum Foundation, which financially supported my PhD, and to the Alexander von Humboldt Foundation, which awarded me a research fellowship.

List of Publications

The following is a list of all publications from the PhD period. Following the norms of theoretical computer science, authors are listed in alphabetical order.

1. T. Barakbayeva, S. Farokhnia, A.K. Goharshady, S. Novozhilov. *Boosting Gas Revenues of Ethereum Miners.* IEEE/ACM International Conference on Software Engineering, ICSE 2026.

2. S. Farokhnia, S. Novozhilov, S. Safaei, J. Shen. *Hermes: Scalable and Robust Structure-Aware Optimal Routing for Decentralized Exchanges.* IEEE International Conference on Blockchain, Blockchain 2025.

3. S. Akshay, S. Chakraborty, S. Farokhnia, A.K. Goharshady, H.J. Motwani, Đ. Žikelić. *LP-Based Weighted Model Integration over Non-Linear Real Arithmetic.* International Joint Conference on Artificial Intelligence, IJCAI 2025.

4. T. Barakbayeva, S. Farokhnia, A.K. Goharshady, P. Li, Z. Lin. *Improved Gas Optimization of Smart Contracts.* International Conference on Fundamentals of Software Engineering, FSEN 2025.

5. T. Barakbayeva, S. Farokhnia, A.K. Goharshady, M. Gufler, S. Novozhilov. *Pixiu: Optimal Block Production Revenues on Cardano.* IEEE International Conference on Blockchain, Blockchain 2024.

6. S. Farokhnia, A.K. Goharshady. *Options and Futures Imperil Bitcoin's Security.* IEEE International Conference on Blockchain, Blockchain 2024.

7. Z. Cai, S. Farokhnia, A.K. Goharshady, S. Hitarth. *Asparagus: Automated Synthesis of Parametric Gas Upper-bounds for Smart Contracts.* ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2023.

TABLE OF CONTENTS

Title Page i

Abstract ii

Authorization iii

Signature Page iv

Dedication v

Acknowledgments vi

List of Publications viii

Table of Contents ix

Chapter 1 Introduction 1

1.1 Prologue 1
1.2 Outline 4
1.3 Summary of Contributions 5
1.4 Awards 9

Chapter 2 Preliminaries 10

2.1 Blockchain and Smart Contracts 10
2.2 Bitcoin Architecture 12
2.3 Cardano Architecture 15
2.4 Ethereum Architecture 18
2.5 Sparsity Parameters for Graphs 21
2.6 Algebro-Geometric Methods 24

Chapter 3 Gas Upper-bounds for Smart Contracts 29

3.1 Introduction 30
3.2 Gas Upper-Bound as a Synthesis Problem 31
3.3 Automated Gas Analysis Framework 32
3.4 Experimental Results 50

Chapter 4 Gas Optimization for Smart Contract Compilers 55

4.1 Introduction 56
4.2 Gas Optimization Problem 56
4.3 Our Algorithm 58
4.4 Experimental Results 60

Chapter 5 Maximizing Miners' Revenues 62

5.1 Introduction 63
5.2 Cardano eUTXO Model 64
5.3 Block Optimization via Treedepth 67

5.4 Evaluation on Real Cardano Blocks 71
5.5 Ethereum Account-Based Model 74
5.6 A Randomized Framework 79
5.7 Runtime and Revenue Results on Ethereum 87

Chapter 6 Optimal Routing for Decentralized Exchanges 93
6.1 Introduction 94
6.2 Exchange Rates as a Routing Problem 94
6.3 Graph Routing via Treewidth 96
6.4 Experimental Results 104

Chapter 7 Derivatives and Bitcoin Security 108
7.1 Introduction 109
7.2 Overview of the Attack 109
7.3 Block-reverting as a Minority Miner 113
7.4 Cost of the Attack 116
7.5 Bitcoin Derivatives 119

Chapter 8 Discussion and Conclusion 124
8.1 Discussion 124
8.2 Conclusion 125

Bibliography 127

Chapter 1

Introduction

1.1 Prologue

Blockchain and Smart Contracts. This thesis focuses on designing automated algorithms for optimization and verification problems in smart contracts. The recent success of Bitcoin [1] has highlighted the potential of blockchains, encouraging researchers to extend the concept of transactions beyond the mere transfer of digital assets to the execution of arbitrary programs, known as smart contracts. Such architectures have become prevalent because centralized systems are vulnerable to fraud, censorship, and manipulation, as they depend on a single authority for control and decision-making. Thus, smart contracts, which can employ any complex logic, remove the need for intermediaries by decentralizing control, making interactions more transparent, tamper-resistant, and censorship-proof. Consequently, they have become instrumental in holding digital assets valued at billions of dollars. Currently, Ethereum, the 2nd largest cryptocurrency with a market cap of 359.79 billion USD, and Cardano, the 10th largest with a market cap exceeding 12.19 billion USD, both support the execution of arbitrarily complex smart contracts [2][1].

Motivation and Challenges. Vulnerabilities in smart contracts can have devastating consequences, making their security a critical concern. For example, the DAO attack in 2016 resulted in a loss of 50 million USD, and a recent Bybit hack in February 2025 allowed attackers to steal more than 1.5 billion USD. These losses could have been prevented if the contracts had been formally and rigorously verified. In addition to security, there are also many unsolved or open-ended optimization problems in blockchains. A quintessential example is that users incur execution fees, known as "gas", when interacting with contracts. Between 1 January 2021 and 1 January 2026,

[1] Prices and exchange rates in Chapters 1 and 2 reflect values as of 01 January 2026; all other chapters use data from when their underlying work was originally written.

Ethereum users spent more than 19.6 billion USD on gas fees. Although the price of ether affects transaction fees, Figure 1.1 shows that gas usage on Ethereum remains persistently high, and thus execution fees continue to impose a substantial burden on users. The Ethereum Foundation recognizes that these high fees present a substantial obstacle to broader adoption [3]. Thus, any vulnerability or inefficiency can have profound financial repercussions. Relying on unsound heuristics or suboptimal algorithms is unacceptable when billions of dollars are at stake. Rigorous, principled approaches are essential to ensure both the security and efficiency of blockchain systems.

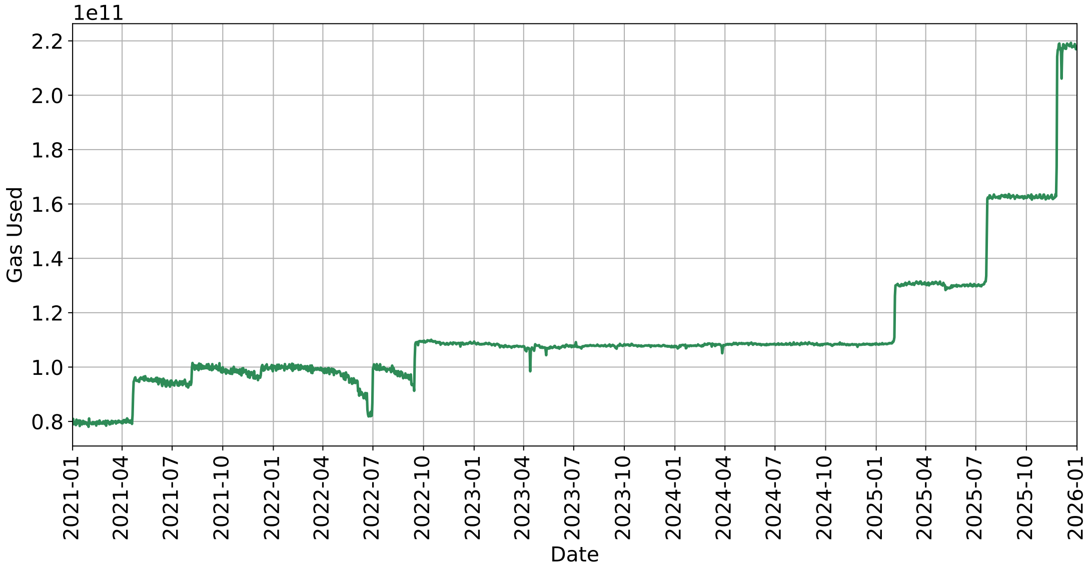


Figure 1.1: Gas usage on Ethereum from 1 January 2021 to 1 January 2026. During this period, users spent an average of 10.73 million USD on gas fees daily.

Real-World Applications of Smart Contracts. Smart contracts underpin a wide range of applications across both public and private sectors. In the public sector, they have been adopted in healthcare, crowdfunding, and supply chain management. For instance, the European Commission highlights properties such as trust, privacy, autonomy, inclusiveness, and transparency as key drivers for many projects launched to leverage decentralization for social and public good [4]. Another notable example is the Bloxberg blockchain, led by the Max Planck Digital Library, which supports scientific research through smart contract infrastructure [5]. In the private sector, smart contracts have reshaped various financial services such as lending, insurance, and asset management, which are often referred to as decentralized finance (DeFi). Stablecoins like Tether (USDT) and USD Coin (USDC), with market capitalizations of 186.8 billion USD and 75.25 billion USD, respectively, are implemented as Ethereum smart contracts [2]. A further illustration is Uniswap, a decentralized exchange (DEX) that allows users to trade digital assets without intermediaries and currently secures 3.2 billion USD [6]. Thus, the appeal of decentralization has drawn various professionals to blockchain. However, the cost of operations and potential vulnerabilities remain significant concerns for progress.

Given the rapid adoption of smart contracts, scalable and secure algorithms for their verification and optimization problems are essential. Many formal methods provide soundness (and, where applicable, completeness) but do not scale, while heuristics can leave contracts vulnerable or inefficient. This thesis shows that, by exploiting structural properties of smart contracts, we can achieve the best of both worlds: principled methods that scale in practice. Building on prior work, we study two complementary approaches to achieve this.

PARAMETERIZATION. The first method is to utilize techniques from parameterized complexity theory. Many optimization and verification tasks can be framed as graph problems over CFGs of programs. These problems are often intractable. When dealing with intractable problems, e.g., NP-hard graph problems, parameterized algorithms leverage the structural and sparsity properties of the underlying graphs. Unlike traditional complexity theory, which assesses runtime based solely on input size, parameterized complexity considers both the input size and an additional property of the instance (a "parameter"). For instance, *treewidth* is a well-known parameter that measures the tree-likeness of a graph, while *treedepth* formalizes the star-like structure of a graph. Many problems that are computationally intractable (e.g., NP-hard) on general graphs can be efficiently solved on graphs with small treewidth. The intuition is that, if a graph can be decomposed into small parts that are connected to each other in a tree-like manner, then one can treat the graph as if it were a tree and leverage tree-based algorithmic ideas such as bottom-up dynamic programming. It is well known that CFGs of structured programs are sparse and tree-like, i.e., they have bounded treewidth [7]. The same has also been established for smart contracts [8]. See Section 2.5 for more details.

ALGEBRO-GEOMETRIC METHODS. Our second approach uses tools from polyhedral and real algebraic geometry. Many of our problems can be formulated as solving systems of constraints. If we submit these constraints directly to a Non-linear Real Arithmetic (NRA) or SMT solver, the solver must handle quantifier alternation, which is very costly. In practice, first-order theories of the reals are slow and often fail even on small examples. To address this, we eliminate quantifier alternation by converting the original formula into an equivalent system of constraints if it admits certain structures. This produces a quantifier-free instance that is much easier for solvers to process. See Section 2.6 for more details.

1.2 Outline

This thesis is organized as follows. Chapter 2 presents common background on blockchains and smart contracts. It then gives protocol-specific details for Bitcoin (Section 2.2), Cardano (Section 2.3), and Ethereum (Section 2.4). The chapter also introduces the formal definitions and mathematical tools used throughout the thesis, covering parameterized complexity in Section 2.5 and algebro-geometric methods in Section 2.6.

The main body develops principled, scalable methods for two complementary classes of problems: (i) program-level analysis of smart contracts (Chapters 3 and 4); and (ii) interactions between users and contracts (Chapters 5 and 6). Chapter 7 further illustrates that blockchain is inherently interdisciplinary and that rigorous analysis benefits from perspectives beyond computer science. While not the main focus, this demonstrates that mathematically grounded, cross-disciplinary methods can yield novel insights. Each technical chapter presents a clear problem statement, algorithms, and empirical evaluation.

Concretely, the thesis contains the following chapters:

- Chapter 3 proposes a novel approach to automatically synthesize parametric gas-usage upper bounds for smart contracts, using tools and theorems from polyhedral and real algebraic geometry. To the best of our knowledge, this is the first identified use case of algebro-geometric methods in blockchain. Moreover, our approach is the first to successfully synthesize polynomial bounds.
- Chapter 4 studies the compiler-optimization problem of gas minimization. We present a dynamic-programming algorithm that builds on, and integrates with, the existing tool `syrup`, producing more gas-efficient contracts while retaining correctness guarantees.
- Chapter 5 addresses block-construction and transaction-selection problems from a miner's perspective for two major cryptocurrencies:
  - for Cardano (UTXO-based), we present a novel algorithm that exploits the sparsity of interrelations between Cardano transactions. We formulate this problem as a knapsack problem and derive an exact parameterized algorithm based on treewidth;
  - for Ethereum (account-based), we present a novel randomized approach to boost miners' transaction-fee revenues. We observe that a transaction's gas usage is often influenced by a small subset of other transactions, termed its

neighborhood. Our algorithm uses testing and decision trees to estimate neighborhoods, then derives gas-usage rules to predict each transaction's fee. The rules are encoded as an ILP instance and solved with an external ILP solver.

- Chapter 6 presents an extension of a treewidth-based algorithm for optimal routing on DEXs. By leveraging the structural properties of liquidity pools, the proposed approach bridges the gap between scalability and robustness. To our knowledge, this is the first application of parameterized algorithms in the context of DEXs.
- Chapter 7 refutes several common misconceptions regarding Bitcoin's security against block-reverting attacks. Specifically, we show that a successful block-reverting attack does not necessarily require (even close to) a majority of the hash power, and that Bitcoin derivatives, i.e., options and futures, imperil Bitcoin's security by creating an incentive for a block-reverting/majority attack.

Finally, Chapter 8 provides a discussion of broader impacts and summarizes the main findings and future directions of the thesis.

On Monetary Values and Market Data. Except in Chapters 1 and 2, the prices and numerical estimates presented throughout this thesis are based on data available at the time each chapter's underlying work was written. While these specific values may have changed since then, the fundamental arguments and methodologies remain sound.

1.3 Summary of Contributions

The main contributions of each chapter are outlined below:

- In Chapter 3, we consider the problem of finding polynomial gas-usage upper bounds for every function of a given smart contract. Our contributions are as follows:
  - *Theoretical Contributions.* We provide novel and fully automated algorithms, based on techniques from polyhedral and real algebraic geometry, to synthesize the tightest possible polynomial gas upper bounds for a given PTS. We show that our approach is not only sound, but also semi-complete, i.e., complete when the polynomial degree is sufficiently large. Our theoretical approach is language-independent and can be applied to

smart contracts written in any language and run on top of any blockchain protocol.

  - *Practical Contributions.* We provide extensive experimental results on 156,735 functions from 24,188 real-world smart contracts, showing that our approach is scalable and applicable to the vast majority of real-world Ethereum contracts (80.56%). Moreover, we compare our results with GASTAP, the only previous method that could synthesize parametric bounds. Our experiments demonstrate that Asparagus outperforms GASTAP both in terms of the number of contracts it can handle and the quality and tightness of the bounds.

- In Chapter 4, we study the compiler-optimization problem of gas minimization, and we make the following contributions:

  - *Theoretical Contributions.* We provide a simple dynamic-programming approach to enhance superoptimization techniques for reducing the gas usage of smart contracts.

  - *Practical Contributions.* We implement our approach and integrate it with `syrup 2.0`, the current state-of-the-art gas optimizer for Ethereum smart contracts. Over a benchmark set of 148 real-world and commonly called smart contracts on the Ethereum blockchain, we find that our approach more than doubles the benefits of `syrup`, increasing gas savings from 4.17% to 11.23%.

- In Chapter 5, we consider the problem of maximizing miners'/producers' revenues, i.e., a block with maximum total transaction fees, in Cardano and Ethereum. Our contributions are as follows:

  - For Cardano, we exploit the sparsity of interrelations between real-world Cardano transactions to find the optimal block to mine. Specifically:

    - *Theoretical Contributions.* Since the problem is NP-hard, we obtain an algorithm that has polynomial runtime for real-world instances of the problem. Formally, we consider a graph of interrelations between transactions and show that when this graph has bounded treedepth, i.e., when it is sparse and resembles a shallow tree, there is a polynomial-time algorithm for finding the optimal block to mine.

    - *Practical Contributions.* We experimentally show that the small-treedepth assumption holds for real-world Cardano instances. With the help of the Cardano Foundation, we conduct a 50-day-long experiment on the Cardano blockchain in which we run our algorithm on the same sets of transactions as those available to real-world Cardano block producers and compare the blocks produced by our approach with those actually added to Cardano. Our approach increases transaction-fee revenues by 1,357.82 USD/day (= 495,604.3 USD/year). Thus, producing optimal blocks that maximize transaction fees yields significant benefits in practice.

  - For Ethereum, we design and present a randomized algorithm to create a block that maximizes the miner's total tip revenue. Specifically:
    - *Theoretical Contributions.* We execute test cases (random permutations of transactions in a transaction pool) and profile their gas usage. Then, using decision trees, for every transaction in the pool, we identify a set of other transactions that can affect its gas usage; we call this set its *neighborhood.* Intuitively, we expect neighborhoods to be small because most transactions are independent; for example, two transactions that do not access the same smart contracts cannot affect each other's gas usage. We provide a probabilistic argument showing that, with high probability, our testing covers every possible permutation of each neighborhood. We cut, mix, and glue together parts of our test cases to create a block that maximizes the miner's total tip revenue. We model this step as an ILP instance and solve it using an external optimization suite. All steps of our algorithm are parallelizable and, except for the ILP solver, run in polynomial time. Because it relies on randomized sampling and ILP solvers, our algorithm is not guaranteed to always produce an optimal result.
    - *Practical Contributions.* We implement our algorithm and perform extensive experiments on 50,000 Ethereum blocks. For each block, we gather real-world transaction pool data and execute our algorithm. We then compare the tip revenues obtained by our framework with those of real-world miners. Our approach increases tip revenues by 73.45% on average per block, corresponding to roughly 24.1 USD per block and 63,357,892 USD per year at current exchange rates. We also compare our tool against the default Ethereum implementation and find that our approach increases tip revenues by 18.56% on average per block, corresponding to roughly 17.3 USD per block and 45,416,764 USD per year.

,000,000

- In Chapter 6, we analyze the structural properties of Uniswap exchange pools, with the following contributions:

  - *Theoretical Contributions.* We extend a parameterized algorithm that leverages the low treewidth of DEX graphs to find routes efficiently. The algorithm is specifically tailored for the dynamic, online environment of DEXs, where liquidity pools are constantly changing. Our formal complexity analysis shows that its theoretical runtime outperforms state-of-the-art algorithms.
  - *Practical Contributions.* We implement the algorithm in an open-source routing tool called Hermes and conduct a comprehensive experimental evaluation using real-world Uniswap transaction data covering 20,000 blocks. The results show that our treewidth-based approach achieves superior practical performance compared to existing methods. Hermes is the only method capable of processing token sets of size 100,000, with an average query time of 0.19 seconds. For instances where both tools yield results, Hermes reduces the average runtime from 2.81 seconds to 0.0002 seconds, an improvement of over four orders of magnitude.

- In Chapter 7, we refute several common misconceptions regarding Bitcoin's security against block-reverting attacks. In particular, we show that:
  - We show that an adversary with a small percentage of the hash power can still reliably create forks that are six blocks deep, thus fracturing the public's trust in Bitcoin and most probably causing a crash in its price;
  - We show that a majority attack on Bitcoin would cost only 6.77 billion USD, which is much lower than the community's expectation and only 0.48 percent of Bitcoin's market cap at the time of writing. Additionally, and more worryingly, an attack with only 30% of the total hash power, which would cost a mere 2.9 billion USD, will succeed with a probability of more than 95% within 34 days. Thus, such attacks are much more affordable than expected;
  - Finally, and most importantly, we show that it is possible for an attacker to actually benefit from the resulting crash in the value of Bitcoin, due to the vast derivatives (options and futures) market that is currently in existence. Put simply, an attacker can first short Bitcoin using widely-available derivative contracts and then intentionally perform an attack to crash the price and profit. Thus, there are real-world incentives for such attacks.

1.4 Awards

The research presented in Chapter 5 received an Academic Grant (ESP) from the Ethereum Foundation. Chapter 4 led to the award of a research fellowship from the Alexander von Humboldt Foundation. The work on Cardano block production (Sections 5.2–5.4) was conducted in collaboration with the Cardano Foundation, and the resulting research was invited for presentation at the Cardano Summit 2024. Chapters 5 and 3 each received a Research Travel Grant from the Hong Kong Research Grants Council to attend the conference and present the paper.

Chapter 2

Preliminaries

2.1 Blockchain and Smart Contracts

Blockchain. As pioneered by Bitcoin [1], most modern cryptocurrencies use a blockchain protocol. There are three fundamental objects: transactions, blocks, and the blockchain. Transactions are the basic units of record keeping, defining the cryptocurrency's history and order. In Bitcoin, transactions transfer money in the underlying cryptocurrency. Anyone can create and broadcast transactions, which must be validated, e.g. by checking digital signatures to prove ownership. Valid transactions are spread across the network using a peer-to-peer gossip protocol. Valid transactions are grouped into blocks of fixed maximum size, and each block contains a hash pointer to the previous block, forming a singly-linked list called the blockchain. Each node maintains a local copy of the blockchain, so its history consists of all transactions in its chain, which also provides their ordering.

Consensus mechanism. To ensure consensus, not every propagated transaction is finalized immediately; a consensus mechanism is required to make all nodes eventually agree on the blockchain's contents. Its purpose is to make adding new blocks subject to rules that prevent attackers from creating competing branches, or forks, which represent incompatible transaction histories. There are many consensus protocols [9, 10, 11, 12, 13], with the most prominent being proof of work, used by Bitcoin [1] and Ethereum Classic, and proof of stake [14, 15], used by Ethereum [16], Cardano [17], and Algorand [18]. In practice, different terms are used for miners; for example, Ethereum and most proof-of-stake currencies use *validators* or *block builders* to indicate they do not use proof of work. In this thesis, we use the term *miner* for all consensus mechanisms.

Mining. Every blockchain protocol requires a process to extend the chain by adding new blocks. This process is often called *mining* in proof-of-work blockchains such as Bitcoin [1] and entails finding a solution to a proof-of-work puzzle which is often based on inverting hash functions. While proof-of-stake blockchains [14, 15] and other alternative consensus mechanisms such as proof-of-space [19] do not require the expensive step of solving a hash puzzle, they nevertheless need rules for extending the chain. The nodes that take part in chain extension are known by various names such as validators, farmers or producers. For simplicity, in this thesis, we simply use the words *miner* and *producer* to refer to any node on the network who tries to add a new block to the blockchain according to the underlying consensus protocol.

Mining Steps. A miner has to first form a block by choosing a set of unmined transactions. Following Bitcoin [1], most blockchain protocols have a maximum size limit on the blocks, thus the miner has to strategically pick the transactions that are included in her block. The miner then proposes the block by publicly announcing it. The proposed block may or may not be adopted by the network, based on its rules of consensus. For example, in a proof-of-work cryptocurrency, only blocks that contain a valid solution to the hash puzzle may be accepted by the network.

Transaction Fees. Given that consensus heavily relies on miners and that mining is often costly, especially in proof-of-work settings, the protocol must provide rewards to incentivize mining. The miners are paid a fixed reward for every block they add to the blockchain. This is also how new units of currency are created. Additionally, to incentivize the miners to create non-empty blocks, a user who creates a transaction can decide on a transaction fee, which will be paid to the miner who adds it to the blockchain. It is well-known in the community that transactions with low fees are often ignored by the miners.

Smart Contracts. Bitcoin supports a limited scripting language for specifying conditions to spend a UTXO, such as requiring multiple signatures. In contrast, programmable blockchains, pioneered by Ethereum [16], allow arbitrary scripts in a Turing-complete language. The key idea is that blockchain consensus is independent of transaction type, enabling transactions beyond simple currency transfers. On Ethereum, a transaction can: (i) transfer money, (ii) deploy a *smart contract*—a program in Ethereum Virtual Machine (EVM) bytecode, or (iii) interact with existing smart contracts by calling their functions. A smart contract is a program added to the blockchain, making its code immutable. Each contract has dedicated storage and can hold currency. Once money is sent to a contract, it can only be recovered if the contract's code transfers it elsewhere. Since the protocol provides consensus on transaction history, it also extends to the state of every contract, as all nodes can execute the transactions.

Design Choices in Blockchains. While blockchains share fundamental principles, their differences, such as programming language and execution environment, can influence tasks such as smart contract optimization. Blockchains typically employ either imperative or functional programming languages. Imperative languages, such as Solidity used by Ethereum, execute instructions sequentially, while functional languages, like Plutus on Cardano, rely on nested function calls. Imperative languages offer straightforward control flow and are familiar to most developers, making them easier to learn and use for a wide range of applications. In contrast, functional languages emphasize immutability and stateless computation, which can enhance security and facilitate formal verification. In terms of execution environment, Ethereum adopts an account-based model with a global state, enabling complex contract interactions, whereas Cardano uses a UTXO-based model with local state, facilitating better parallelization [16, 20].

We next review Bitcoin, Cardano, and Ethereum, emphasizing aspects relevant to this thesis.

2.2 Bitcoin Architecture

Bitcoin. Bitcoin [1] was the first working protocol for a decentralized cryptocurrency and currently holds the largest market cap among all such currencies, amounting to more than 1.7 trillion USD at the time of writing [2].

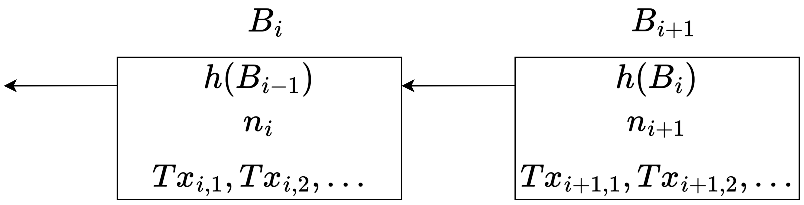

B_i
B_{i+1}
$h(B_{i-1})$
n_i
$Tx_{i,1}, Tx_{i,2}, \ldots$
$h(B_i)$
n_{i+1}
$Tx_{i+1,1}, Tx_{i+1,2}, \ldots$

Figure 2.1: A simplified view of the blockchain

Proof-of-Work [1, 21]. In Bitcoin, transactions are grouped into *blocks* of a fixed maximum size. The blocks are then chained together in a singly-linked list using hash pointers with each block containing the hash of its parent (previous) block. This linked list is aptly named the *blockchain.* The blockchain is subject to consensus, i.e. all honest nodes on the network should eventually agree on its contents. Thus, adding a new block to the end of the blockchain is a deliberately hard task, called *mining*, that requires the solution of a computationally-intensive hash inversion puzzle. This scheme is called *proof-of-work* and ensures that a miner's chance of adding the next block to the blockchain is proportional to the miner's computational power, i.e. how many hashes she can compute per unit of time.

Figure 2.1 shows an overview of this process. In this figure we have omitted implementation details that are not relevant to this thesis. Each block B_i contains the hash of the previous block B_{i-1}. This serves as a hash pointer in the linked list. It also contains a nonce n_i and a sequence of transactions $Tx_{i,1}, Tx_{i,2}, \ldots$. A miner who aims to add a new block B_{i+1} should first create the pointer to the previous block and populate a list of transactions that she intends to include. She should then choose the new nonce n_{i+1} such that the hash $h(B_{i+1})$ of her new block is below a certain predefined threshold[1]. Since the output of a hash function is unpredictable and an ideal cryptographic hash function can be modeled as a random oracle, the miner's only choice is to repeatedly try different nonces until she finds a valid block. Thus, her success probability is proportional to the number of hashes she can compute per unit of time.

Since mining is an expensive activity, due to both hardware and electricity costs, the miner should be financially incentivized to perform it. Bitcoin creates two incentives for the miner [22, 1, 23]: (a) a block reward (currently 6.25 BTC ≈ 549,016 USD, expected to halve in almost three weeks from now) is paid to each miner who successfully adds a new block, and (b) each transaction contains a transaction fee that is paid to the miner who adds it to the consensus chain.

UTXO. In Bitcoin [1], each transaction has multiple inputs and outputs: inputs are coins entering the transaction, outputs are coins leaving. Each input must reference an output from a previous transaction, ensuring only previously received coins can be spent. To prevent double-spending, each output can be spent only once. Spendable coins are called Unspent Transaction Outputs (UTXOs). Although UTXOs can be reconstructed from history, Bitcoin nodes maintain a current UTXO set for efficient transaction and block validation. Since transactions cannot create new coins, the sum of outputs must be less than the sum of inputs; the difference is paid to the miner as a transaction fee. Thus, each transaction has a fixed, known fee independent of other transactions.

Longest Chain Rule [1, 24]. In the event that two miners find a valid block at approximately the same time, a temporary *fork* happens in which there are two valid blockchains known to the network. In such a scenario, the Bitcoin protocol allows miners to try to extend either branch. However, as soon as a branch becomes longer than the other(s), the shorter branch(es) are dropped by everyone who honestly follows the protocol. Thus, the protocol mandates that the longest chain is always

[1] In Bitcoin, the threshold changes dynamically to ensure that a new block is mined roughly every 10 minutes.

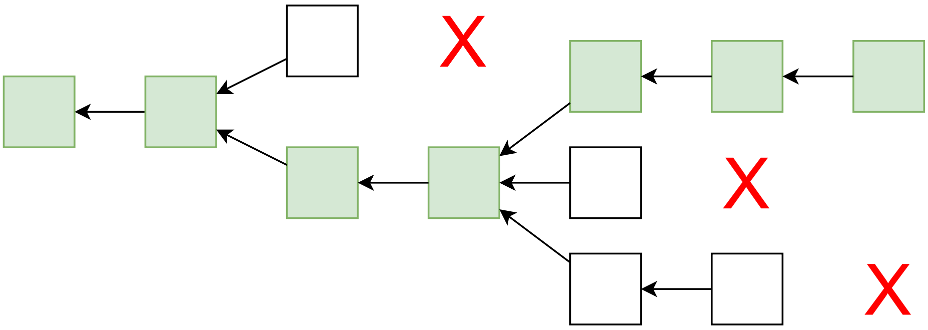


Figure 2.2: An illustration of the longest chain rule in Bitcoin

the consensus chain and that every node on the network must always consider the longest chain known to them as the authoritative blockchain[2]. This is illustrated in Figure 2.2. As long as a majority of the computational power on the network follows the protocol honestly, all honest participants are guaranteed to eventually reach a consensus about the blockchain.

DOUBLE-SPENDING [25, 1, 26, 27, 28]. Preventing double-spending is arguably the main contribution of the Bitcoin protocol. A *double-spending* attack is when a Bitcoin user tries to use the same coin (transaction output) in two different transactions. In such cases, the two transactions will be in conflict [22] and at most one of them can be added to the consensus chain. Specifically, if Tx_1 and Tx_2 both spend the same coin, then any proposed block that contains Tx_1 can only be valid if neither it nor any of its ancestors (previous blocks) contain Tx_2. Suppose ALICE is selling an item to BOB and BOB is paying the price by a Bitcoin transaction Tx_1 that transfers part of his money to ALICE. In this case, it is not enough for ALICE to see Tx_1, since BOB might have created a conflicting transaction Tx_2 that double-spends the same coin. Thus, ALICE should wait for Tx_1 to be added to the consensus chain. However, even this does not guarantee that the payment is finalized, since it is possible that the miner eventually creates a longer chain that contains Tx_2 and thus consensus switches from Tx_1 to Tx_2. In such cases, we would say that Tx_1 is *reverted*. In practice, this is unlikely to happen if Tx_1 is already in a block B_i and there are many blocks added after B_i. Such blocks are called *confirmation* blocks. The conventional wisdom and industrial standard practice is to wait for 6 confirmations before considering the transaction as irreversible, although some users take the risk of waiting for fewer confirmations [29].

[2] In practice, the length of a chain is not just the number of blocks in it, but rather the total difficulty of mining these blocks. However, this minor detail does not change any of the analyses in this thesis.

2.3 Cardano Architecture

Cardano [30]. Cardano is an open and decentralized blockchain platform that supports many cryptocurrencies and tokens. Its main currency, Ada, is currently one of the top 10 cryptocurrencies in terms of market cap and has a value of more than 12.19 billion USD [31]. Similar to Bitcoin, Cardano follows the (extended) UTXO model [32], in which every transaction has a set of inputs and outputs. Each input to a transaction should be an output of a previous transaction and no output may be spent (used as input) twice. It has two major advantages over Bitcoin: (i) support for arbitrarily complex smart contracts, i.e. transactions can invoke executions of programs in an expressive programming language, and (ii) a proof-of-stake consensus protocol, namely Ouroboros Praos [17], which avoids the costly mining process of Bitcoin's proof-of-work. In this section, we consider a simplified model of a Cardano transaction that precisely captures those aspects which are important to this thesis. For a more thorough treatment, see [30] or [33].

EUTXO. Most modern blockchains either follow the Unspent Transaction Output (UTXO) model, as in Bitcoin [1], or the account model, as in Ethereum [16]. To take advantage of the benefits of both models, Cardano proposes the Extended UTXO (EUTXO) model that allows having more expressive programs than simple scripts without adopting the account model. In UTXO, a transaction has a set of inputs and outputs. An input points to an output of a prior transaction that provides funds to be spent by this transaction. Moreover, every output can be used by at most one input. EUTXO follows the same policies except that there is one more output field called "datum" that carries contract-specific data. In addition, EUTXO allows the inclusion of arbitrary logic as scripts and the use of the data fields to decide if the transaction output can be spent. See [32, 33] for a more detailed treatment.

Proof-of-Stake [34, 14, 15]. In proof-of-stake protocols, a miner is randomly chosen to add the next block. Each miner's probability of being chosen is proportional to her stake in the currency, i.e. the number of coins she holds. Specifically, in Cardano's implementation of Ouroboros Praos [17], time is divided into epochs, each consisting of 432,000 slots. Each slot corresponds to one second. Thus, each epoch lasts for five days. In each slot, some miners/producers are randomly selected to propose blocks of transactions [35].

Stake Pools and Delegation [36]. Any user who holds Ada can take part in mining (block production) on Cardano. Users can also delegate their mining rights (stakes) to others, leading to stake pools which, as the name suggests, pool together stakes from many different users. The pool operator can then mine on behalf of all of its users and the probability that the pool is selected in each slot is proportional to the total stake of the members. The vast majority of Cardano blocks are produced by stake pools rather than individual stakeholders.

REWARDS. Producing blocks is a costly process. This is of course evident in proof-of-work blockchains, in which the miners have to solve hash puzzles requiring vast computational resources and electricity usage. Thus, Bitcoin rewards miners for every new block that they successfully add to the blockchain and also pays them transaction fees [1]. Even in proof-of-stake blockchains, there are costs associated with block production, e.g. the miner has to keep track of the whole history of the chain, constantly listen for new transactions, hold stake or convince others to delegate stakes to her, form valid blocks of transactions and announce/propose them on time. Therefore, it is necessary to incentivize block production by rewarding the miners. In Cardano, the block producers are rewarded in two ways [37]:

1. *Transaction Fees:* Every transaction contains a fixed fee. After each epoch, the fees of all transactions mined in that epoch are divided among the block producers.
2. *Monetary Expansion:* For each epoch, a fixed percentage of the remaining reserve of Ada is paid to all block producers of the epoch.

BLOCK CONSTRAINTS. In Cardano's EUTXO model, a block contains a sequence of valid transactions. The transactions in each block must satisfy the following requirements, otherwise the block is considered invalid and will be discarded by all nodes of the network.

- *Transaction Dependencies.* If a transaction t_2 uses an output of another transaction t_1 as one of its inputs, then t_2 depends on t_1. If the block producer decides to include t_2 in her new block, then she must also add t_1 in the same block and before t_2. See Figure 2.3 as an example.

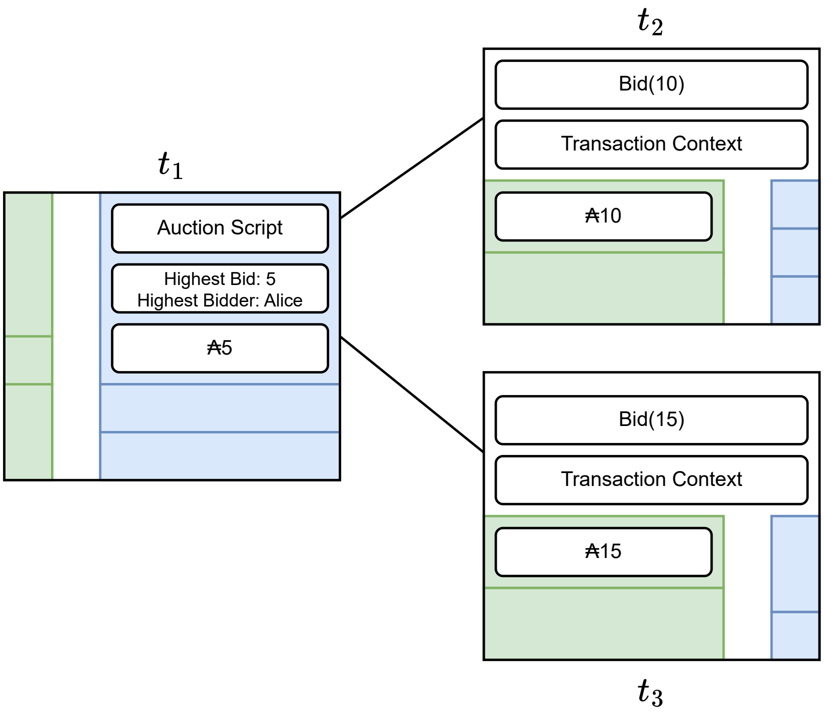


Figure 2.4: In an auction contract, Alice has bid 5 Ada in t_1. Both Bob and Carol are trying to bid right after Alice, hence using t_1's datum output as input to their transactions t_2 and t_3. Since every output can be used only once, t_2 and t_3 are in conflict.

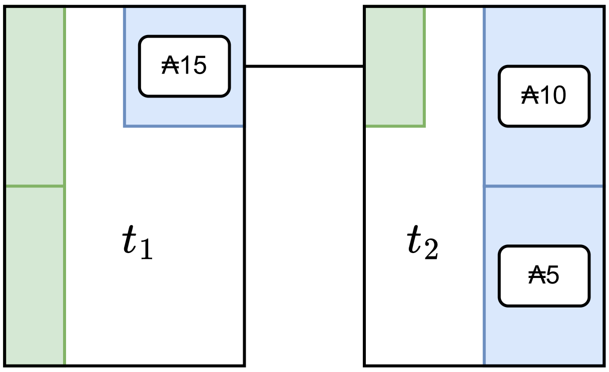


Figure 2.3: Alice pays 15 Ada to Bob in t_1. In t_2 Bob uses the same funds to pay 10 Ada to Carol and 5 Ada back to himself. The transaction t_2 has an input that is using an output of t_1. Thus, t_1 is a dependency of t_2.

- *Transaction Conflicts.* If two transactions t_2 and t_3 both spend the same output of a transaction t_1, since every output may be spent at most once, only one of the conflicting transactions may be selected and included in the new block. Thus, the block producer has a choice to include t_2, t_3 or neither in the new block, but she cannot include both. See Figure 2.4 as an example.
- *Block Size Limit.* Cardano imposes a block size limit, i.e. the total size of all

transactions included in a new block must not exceed 90112 bytes = 88 kilobytes. This is to ensure that the blocks are small enough to be efficiently propagated within the network with minimal latency. The block size limit may change in each epoch, but in practice it has been consistently kept at 88 kilobytes.

2.4 Ethereum Architecture

This section provides a quick introduction to Ethereum and its gas model, following [16, 38].

SMART CONTRACTS. Ethereum [16] allows arbitrary programs written in a Turing-complete language, which are called *Smart contracts.* Most other modern cryptocurrencies have followed Ethereum and enabled smart contracts. More specifically, a transaction in Ethereum can not only transfer money as in Bitcoin, but also perform other tasks:

- *Deployment of a Contract.* A transaction can contain the code of a program, i.e. a smart contract, and hence deploy it on the blockchain. Simply put, when this transaction is included in a block and then added to the blockchain, every node in the network has access to the smart contract's code, which is now part of the consensus. Moreover, the code is immutable since the blockchain is append-only.
- *Function Calls.* A transaction can "call" a desired function in a smart contract that was deployed by a previous transaction. To do so, the transaction has to provide a pointer to the smart contract, the name of the called function, and the parameters passed to it. A transaction record containing all this data will be added to the blockchain.

ACCOUNT MODEL. There is no concept of UTXO on Ethereum. Instead, every node on the Ethereum network keeps track of the so-called *world state* [16]. The world state consists of a number of *accounts.*

ACCOUNTS. In Ethereum, the currency unit is called an "ether" and uses the acronym ETH. An account is an entity with an ether (ETH) balance that can transfer money on Ethereum. It is either controlled by a user or a deployed smart contract.

The state of an account consists of the following fields:

- `nonce`: The number of transactions sent from the account.
- `balance`: The amount of ether owned by this account.

- `codeHash`: This is only for contract accounts and contains the hash of the contract's code.
- `storageRoot`: Each account has a storage, e.g. for storing records and global variables in a contract. The storage is encoded in a Merkle-Patricia tree. The `storageRoot` is a hash pointer to the root of this tree.

TRANSACTIONS. A user can publish an Ethereum transaction to transfer money, deploy a new contract and change the blockchain state.

BLOCKCHAIN STATE. In Ethereum, the complete blockchain state includes not only the sequence of blocks, but also implicitly contains the *world state* and the *machine state*. Most Ethereum nodes keep track of these explicitly. The world state represents the states of all accounts, which includes balances for everyone and the storage state of every smart contract. The machine state is the state of the Ethereum virtual machine during execution and contains global variables such as the block number. Transactions and blocks change the blockchain state, especially the world state.

EVM. The Ethereum Virtual Machine (EVM) deterministically and unambiguously describes the rules of executing transactions on Ethereum. EVM supports a set of operations (opcodes) to interact with input call data, stack, memory, storage and the blockchain world state.

SOLIDITY [39]. Solidity is an object-oriented, high-level Turing-complete language for implementing Ethereum smart contracts and the most widely-used smart contract language. Solidity supports all the standard data types and data structures such as arrays and mappings. The functions in Solidity contracts can call other functions from the same contract or external public functions by specifying the function signatures and contract addresses.

GAS. As the EVM language is Turing-complete, one can write and invoke contracts whose execution uses huge or even infinite time and memory. This is undesirable given that every transaction has to be run by all nodes on the network. Thus, intentionally executing long or resource-intensive contracts is a type of denial-of-service (DoS) attack. To combat this, Ethereum introduced the concept of *gas* [40]. Gas is a measure of the total resources required to run an invocation. A gas cost is assigned to every atomic operation in the EVM language, including memory usage or deploying code. The costs of atomic operations are part of the protocol and provided as a table in [41]. The initiator of a transaction has to pay a transaction fee proportional to the total gas cost of its execution. This effectively disincentivizes long and resource-hungry executions. Moreover, every block has a gas limit, which was originally set at 30,000,000 units of gas but has recently increased to 36,000,000. If the sum of gas usages of all transactions in a block exceeds this limit, the block is considered invalid and will not be added to the blockchain. This ensures that every node will have to perform only a limited amount of computation for each block.

Deposits and Out-of-gas Behavior. The user who initiates a transaction (function call to a smart contract) can specify the maximum amount of gas that the transaction is allowed to spend, as well as the price that the user is willing to pay per unit of gas[3]. They should then provide a deposit equal to the product of the two parameters. When the transaction is added to the blockchain, every node in the network executes it, while keeping track of the amount of gas that is being used. If the function call requires more than the allocated gas, then not only the deposit is confiscated but the call itself is reverted and canceled, i.e. all effects of this function call, be it on user's balances or the variables of the smart contracts, will be reverted as if this function call never existed. The only effect is that the initiator loses their deposit.

Gas Usage Formula. The Ethereum protocol fixes a specific gas cost for every one of the EVM opcodes using a complicated formula to address the different computational workloads of different opcodes. See [38, Page 14] for a complete table of gas costs. We follow these costs in our implementation but their specific values have no bearing on the theory. In summary:

- Most opcodes have fixed constant gas costs. For example, stack operation opcodes `PUSHx`, `POPx` and `SWAPx` have a gas cost of 3 units each.
- Some opcodes have a few different gas costs given different conditions. The `SSTORE` opcode is an example, which saves a word to storage. The gas cost is 20,000 if it is overwriting a zero with a non-zero value, and 5,000 otherwise. Moreover, if the operation clears a non-zero to a zero, then 15,000 units of gas are refunded, but this refund is made only if the current transaction uses more than 15,000 units of gas. Since these conditions are unnecessarily complicated and depend on the dynamic world state, and since they are not specified by the smart contract itself, we always replace them with their highest possible value to preserve soundness, e.g. we always assume that an `SSTORE` takes 20,000 units of gas.
- The gas costs of some operations depend on the size of a particular stack or array. For example, taking the hash of a piece of data that is n bytes long has a dynamic cost that is linearly dependent on n. To handle these cases, we always keep track of the sizes of the stacks/arrays as separate variables. Notably, while

[3] After the London upgrade in Ethereum, there is a minimum base price that the user must pay or else the transaction is invalid. However, this has no effect on our algorithms, since they all consider gas usage itself and are independent of the gas price.

these costs are not constants, they are always polynomials (but not necessarily linear) with respect to their input size n.

2.5 Sparsity Parameters for Graphs

2.5.1 Parameterized Complexity and FPT

This thesis uses parameterized algorithms, which leverage specific structural features of problem instances to achieve faster solutions. We present the essential definitions here; see [42, Chapter 2] for comprehensive technical discussions.

Parameterized Problem. We consider problems with multiple inputs, formalized as languages over two components. If a pair $\langle x, k\rangle$ belongs to such a language L, we call k the parameter. The parameter typically takes integer values, though it may represent other structures such as graphs or algebraic objects.

Fixed-Parameter Tractability (FPT) [42]. Formally, a parameterized language L is *fixed-parameter tractable* (FPT) if the following hold:

- There exists an algorithm Φ, a constant c, and a computable function f such that for all instances x, k, $\Phi(\langle x, k\rangle)$ runs in time $f(k) \cdot |x|^c$,
- $\Phi(\langle x, k\rangle) = 1$ if and only if $\langle x, k\rangle \in L$.

Intuitively, this means that for each fixed value of the parameter k, the problem can be solved in polynomial time with respect to the input size $|x|$, where the degree of the polynomial does not depend on k. The potentially expensive part of the computation is isolated in the function $f(k)$, which can grow quickly, but for small k the overall running time remains practical. This framework allows many NP-hard problems to become efficiently solvable when the parameter is small, even if the general problem is intractable.

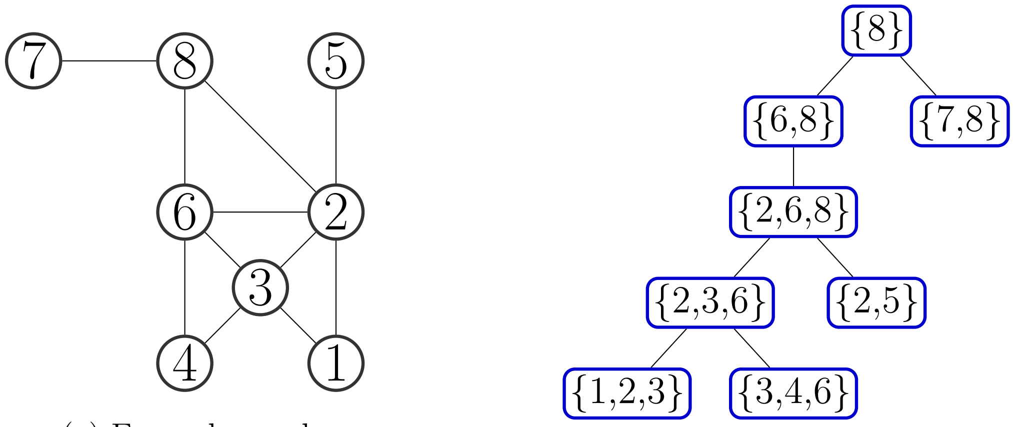
2.5.2 Treewidth

The treewidth of a graph $G = (V, E)$ is defined via a *tree decomposition*. A tree decomposition is a pair $(\mathcal{T}, \{X_i \mid i \in I\})$, where $\mathcal{T} = (I, F)$ is a tree and each X_i (called a bag) is a subset of V, satisfying:

1. The union of all bags equals V; i.e., $\bigcup_{i\in I} X_i = V$.
2. For every edge $(u, v) \in E$, there is at least one bag X_i containing both u and v.

3. For any vertex $v \in V$, the set of bags containing v forms a connected subtree in $\mathcal{T}$.

The *width* of a tree decomposition is $\max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G, denoted $tw(G)$, is the minimum width over all possible tree decompositions of G. Figure 2.5 shows a graph and one of its tree decompositions. A low treewidth indicates a structure amenable to efficient dynamic programming, a property that we exploit extensively in Chapter 6.



(a) Example graph

(b) Tree decomposition

Figure 2.5: A graph and one of its tree decompositions [43].

2.5.3 Pathwidth

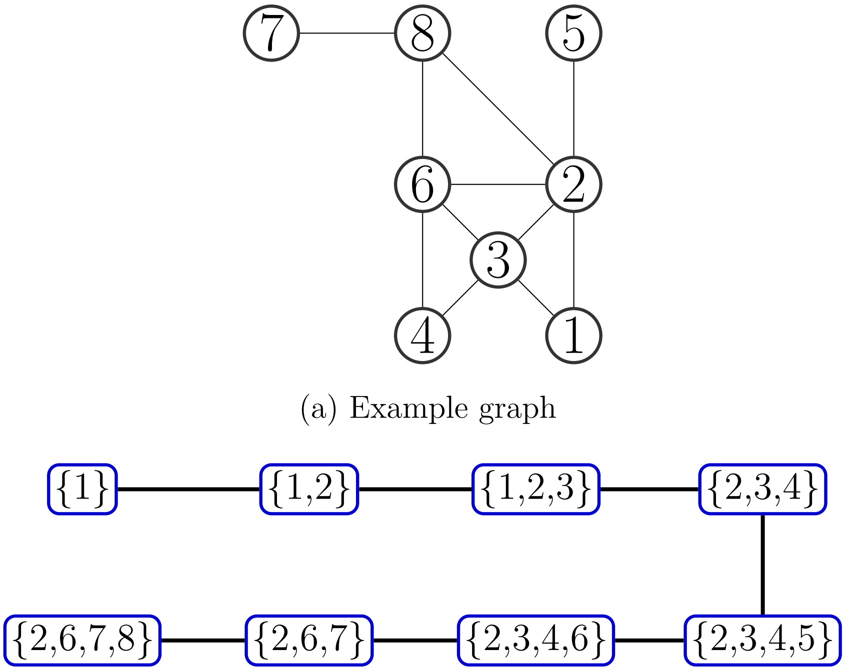
A *path decomposition* of a graph $G = (V, E)$ is a tree decomposition $(\mathcal{T}, \{X_i \mid i \in I\})$ where the tree $\mathcal{T}$ is a path. The *width* of a path decomposition is $\max_{i \in I} |X_i| - 1$, as in the case of treewidth [44, 45].

Intuitively, pathwidth measures how similar a graph is to a path: a lower pathwidth means the graph can be decomposed into a sequence of overlapping vertex sets (bags) with limited size, capturing the "path-like" structure of the graph. Pathwidth is always at least as large as treewidth, since every path decomposition is also a tree decomposition.

2.5.4 Treedepth Decomposition

For a graph $G = (V, E)$, a *treedepth decomposition* [46, 47, 48] is a rooted tree $T = (V, E_T)$ on the same set of vertices as G that satisfies the following requirement:

- For every undirected edge $\{u, v\} \in E$ or directed edge $(u, v) \in E$ of the original graph, either u is an ancestor of v in T or v is an ancestor of u in T.

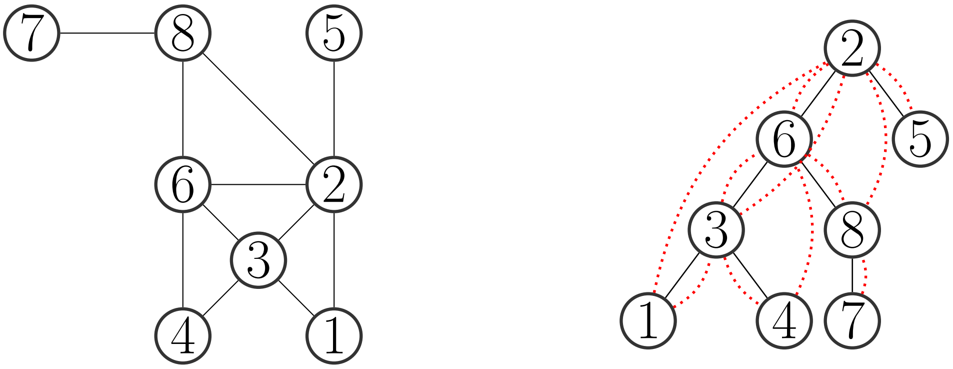


(a) Example graph

(b) Path decomposition

Figure 2.6: A graph and one of its path decompositions [43].

We say that a treedepth decomposition T is optimal if it has the smallest possible depth among all decompositions of G. This smallest depth is called the *treedepth* of G. Intuitively, treedepth is a measure of graph sparsity that captures how much a graph resembles a shallow tree. Figure 2.7 shows a graph and one of its treedepth decompositions.



(a) Example graph

(b) Treedepth decomposition

Figure 2.7: A graph and one of its treedepth decompositions [43].

For any small fixed d, there is an algorithm that decides whether an input graph has treedepth d in linear time and if so, outputs an optimal treedepth decomposition [49]. There are also well-optimized tools and libraries for computing treedepth decompositions [50].

2.6 Algebro-Geometric Methods

In this section, we outline Farkas' Lemma, Handelman's Theorem and Putinar's Positivstellensatz.

Notation. We first set down some basic notation:

1. Given a boolean formula $\phi := \bigwedge_i (f_i \geq 0)$ over the variables $\mathbb{V}$, we define

$$SAT(\phi) := \{\sigma : \mathbb{V} \to \mathbb{R} \mid \bigwedge_i \sigma \models f_i \geq 0\}$$

2. A polynomial $h \in \mathbb{R}[\mathbb{V}]$ is said to be a *sum of squares* (SOS) iff it can be written as a finite sum $h \equiv \sum_i g_i^2$ where $g_i \in \mathbb{R}[\mathbb{V}]$ are arbitrary polynomials.

2.6.1 Farkas Lemma

We start by stating Farkas' Lemma, which is a well-known result in polyhedral geometry and linear algebra.

Theorem 1 (Farkas' Lemma [51])**.** *Let $f_1, \dots f_r$ and g be any linear polynomials over the set of variables $\mathbb{V}$. The boolean implication*

$$f_1 \geq 0 \land f_2 \geq 0 \dots \land f_r \geq 0 \implies g \geq 0$$

holds for every valuation of $\mathbb{V}$ ***if and only if*** *there exist $\lambda_0, \lambda_1, \dots, \lambda_r$, such that*

$$g \equiv \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot f_i$$

where all λ_i's are non-negative reals. Moreover, $f_1 \geq 0 \land f_2 \geq 0 \dots \land f_r \geq 0$ is unsatisfiable ***if and only if*** *$-1 \equiv \lambda_0 + \sum_{i=1}^{r} \lambda_i f_i$ for some non-negative real λ_i's.*

Example 1. *Let $f_1 := -x-2\cdot y+z-5, f_2 := x+3\cdot y-2\cdot z-10, f_3 := 2\cdot x+4\cdot y+5\cdot z+5$ and $g := 3 \cdot y + 5 \cdot z - 15$. We can readily see that $g \equiv 2 \cdot f_1 + f_2 + f3$. Therefore, the implication $\phi := f_1 \geq 0 \land f_2 \geq 0 \implies g \geq 0$ holds. Note that Farkas' lemma is, in a sense, both sound and complete. If the entailment holds, then it guarantees that suitable λ_i's exist.*

Pseudocode. Algorithm 1 provides a more formal presentation of how Farkas' lemma is used to reduce entailments to a system of quadratic constraints.

Algorithm 1: ReduceWithFarkas

```
Input: An entailment constraint $f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g \geq 0$ over the set $\mathbb{V}$ of
       PTS variables
Result: A system $QS$ of quadratic constraints
Create fresh symbolic variables $\lambda_0, \lambda_1, \ldots, \lambda_r$;
$QS \leftarrow \lambda_0 \geq 0 \wedge \lambda_1 \geq 0 \wedge \ldots \wedge \lambda_r \geq 0$;
$eq \leftarrow \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot f_i - g$                ▷ Computed symbolically;
foreach monomial $m \in \mathbb{V} \cup \{1\}$ do
    $\texttt{coeff}_m \leftarrow$ the coefficient of $m$ in $eq$;
    $QS \leftarrow QS \wedge \texttt{coeff}_m = 0$;
end
return $QS$;
```

2.6.2 Handelman's Theorem

For our next theorem, we first need the concept of a monoid.

Monoid. Let $F := \{f_1, f_2, \ldots, f_r\}$ be a set of polynomials over $\mathbb{V}$. We define the monoid of F as the set:

$$\text{monoid}(F) := \{f_1^{k_1} \cdot f_2^{k_2} \cdots f_r^{k_r} \mid k_1, \ldots, k_r \in \mathbb{N}\}. \tag{2.1}$$

In other words, the monoid includes all polynomials that can be obtained as a multiplication of polynomials in F. We also define

$$\text{monoid}_d(F) := \{f \mid f \in \text{monoid}(F) \wedge \deg(f) \leq d\}.$$

Theorem 2 (Handelman's Theorem [52])**.** *Let $f_1, \ldots f_r$ be any linear polynomials and g be an arbitrary polynomial over $\mathbb{V}$ such that SAT$(f_1 \geq 0 \wedge f_2 \geq 0 \wedge \ldots \wedge f_r \geq 0)$ is a non-empty compact set. The boolean implication*

$$f_1 \geq 0 \wedge f_2 \geq 0 \wedge \ldots \wedge f_r \geq 0 \implies g \geq 0$$

holds over real numbers ***if and only if*** *there exist λ_i, m_i such that $g \equiv \sum_{i=1}^{r} \lambda_i \cdot m_i$, in which each λ_i is a non-negative real number and each $m_i \in monoid(\{f_1, \ldots, f_r\})$.*

Example 2. *Let $f_1 := -x - 2 \cdot y - 5$, $f_2 := 2 \cdot x + 3 \cdot y - 2$, and $g := 2 \cdot y - 2 \cdot x \cdot y - 3 \cdot y^2$. We only use the monoid of degree 2 in this example: $monoid_2(\{f_1, f_2\}) = \{1, f_1, f_2, f_1^2, f_2^2, f_1 \cdot f_2\}$. It can be verified that $g \equiv 2 \cdot f_1 \cdot f_2 + f_2^2 + 12 \cdot f_2$. Therefore, by Handelman's Theorem, the implication $\phi := f_1 \geq 0 \wedge f_2 \geq 0 \implies g \geq 0$ holds for every possible real valuation of x and y.*

Pseudocode. Algorithm 2 shows how Handelman's theorem reduces entailments to a system of quadratic constraints.

Algorithm 2: ReduceWithHandelman

```
Input: An entailment constraint $f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g \geq 0$ over the set $\mathbb{V}$ of
       PTS variables
Input: A degree bound $d$
Result: A set of quadratic inequalities
$M_d \leftarrow monoid_d(f_1, \ldots, f_r)$;
$r \leftarrow |M_d|$;
Create fresh symbolic variables $\lambda_0, \lambda_1, \ldots, \lambda_r$;
$QS \leftarrow \lambda_0 \geq 0 \wedge \lambda_1 \geq 0 \wedge \ldots \wedge \lambda_r \geq 0$;
$eq \leftarrow \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot M_d[i] - g$                ▷ Computed symbolically;
foreach monomial $m$ of $\mathbb{R}[\mathbb{V}]$ do
    $\texttt{coeff}_m \leftarrow$ the coefficient of $m$ in $eq$;
    $QS \leftarrow QS \wedge \texttt{coeff}_m = 0$;
end
return $QS$;
```

2.6.3 Putinar's Positivstellensatz

Finally, our strongest hammer is Putinar's Positivstellensatz. Positivstellensatz is German for "positive locus theorem" and denotes a theorem that characterizes positive polynomials over semi-algebraic sets.

Theorem 3 (Putinar's Positivstellensatz [53])**.** *Let $f_1, \ldots f_r$ and g be polynomials of any degree over $\mathbb{V}$ such that $SAT(f_i)$ is compact for at least one f_i. The boolean implication*

$$f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g > 0$$

*holds over real numbers **if and only if** g can be written as*

$$g \equiv h_0 + \sum_{i=1}^{r} h_i \cdot f_i \tag{2.2}$$

in which each h_i is a sum of squares.

Example 3. *As an example, let*

$$f_1 := x + y - 1 \qquad f_2 := -x^2 - y^2 + 10$$

$$f_3 := -2 \cdot x^3 - 4 \cdot x \cdot y^2 + 10 \qquad g := 2 \cdot x^2 \cdot y - 4 \cdot x \cdot y^2 + 10$$

We can choose the sum of squares $h_0 = 2 \cdot x^2, h_1 = 2 \cdot x^2, h_2 = 0$ and $h_3 = 1$, and write $g \equiv h_0 + h_1 \cdot f_1 + h_2 \cdot f_2 + h_3 \cdot f_3$. Therefore, the boolean implication $\phi := f_1 \geq 0 \wedge f_2 \geq 0 \wedge f_3 \geq 0 \implies g \geq 0$ holds.

SOS Templates. Recall that, in order to apply Putinar's Positivstellensatz, we needed to multiply each polynomial f_i with a sum of squares h_i. In our algorithm, we used a template for h_i. We now discuss how such a template can be generated automatically. Given a degree bound $2 \cdot m$, we can use Theorems 4 and 5 below to generate a general symbolic polynomial template for a SOS of degree $2 \cdot m$.

Theorem 4 ([54]). *Let V_m be the vector of all monomials of degree at most m over the variables $\mathbb{V} = \{x_1, \dots, x_n\}$. A polynomial $h \in \mathbb{R}[\mathbb{V}]$ of degree $2 \cdot m$ is a sum of squares **if and only if** there exists a symmetric positive semi-definite matrix Q such that $h = V_m^T \cdot Q \cdot V_m$.*

Theorem 5 (Cholesky Decomposition, [55]). *A square symmetric matrix Q is positive semi-definite **if and only if** there exists a lower-triangular matrix L with non-negative diagonal entries such that $Q = L \cdot L^T$.*

Hence, it suffices to generate a template for the lower-triangular matrix L, introducing new template variables for each of the entries. We can then simply compute a template for h by setting

$$h := V_m^T \cdot L \cdot L^T \cdot V_m.$$

Pseudocode. Based on the theorems above, Algorithm 3 provides a more formal presentation of how Putinar's Positivstellensatz reduces entailments to a system of quadratic constraints.

Algorithm 3: ReduceWithPutinar

```
Input: An entailment constraint $f_1 \geq 0 \wedge f_2 \geq 0 \dots \wedge f_r \geq 0 \implies g > 0$ over the set $\mathbb{V}$ of PTS variables
Input: An even degree bound $d$
Result: A system $QS$ of quadratic constraints
$V_{d/2} \leftarrow monoid_d(\mathbb{V})$        ▷ All monomials of degree at most $d/2$ over $\mathbb{V}$;
$w \leftarrow |V_{d/2}|$;
$QS \leftarrow$ true;
foreach $0 \leq i \leq r$ do
    $L \leftarrow$ a $w \times w$ lower-triangular matrix whose every non-zero entry at position $(j, k)$ is a fresh symbolic variable $\lambda_{i,j,k}$;
    foreach $1 \leq j \leq w$ do
        $QS \leftarrow QS \wedge \lambda_{i,j,j} \geq 0$        ▷ Non-negative diagonal entries;
    end
    $h_i \leftarrow V_{d/2}^T \cdot L \cdot L^T \cdot V_{d/2}$        ▷ Computed symbolically;
end
$eq \leftarrow h_0 + \sum_{i=1}^{r} h_i \cdot f_i - g$        ▷ Computed symbolically;
foreach monomial $m$ of $\mathbb{R}[\mathbb{V}]$ do
    coeff$_m$ $\leftarrow$ the coefficient of $m$ in $eq$;
    $QS \leftarrow QS \wedge$ coeff$_m = 0$;
end
return $QS$;
```

2.6.4 Summary of the Mathematical Tools

The following table summarizes how our algorithm applies the three theorems above in order to reduce a constraint of the form $f_1 \geq 0 \dots \wedge f_r \geq 0 \implies g \geq 0$ to a system of quadratic constraints.

Theorem	$F := f_1, \ldots, f_r$	g	Equality Used
Farkas' Lemma	Linear	Linear	$g \equiv \lambda_0 + \sum_{i=1}^{r} \lambda_i \cdot f_i$ where each λ_i is a non-negative real
Handelman's Theorem	Linear	Non-linear	$g \equiv \sum_i \lambda_i \cdot m_i$ where $m_i \in \text{monoid}_d(F)$
Putinar's Positivstellensatz	Non-linear	Arbitrary	$g \equiv h_0 + \sum_i h_i \cdot f_i$ where each h_i is an SOS

Table 2.1: Summary of the theorems used to reduce an entailment to quadratic constraints

We remark that Putinar's Positivstellensatz is more general than Farkas' Lemma and Handelman's theorem. Putinar can handle antecedents and consequents that are both polynomial, whereas Handelman requires linear antecedents and Farkas additionally requires linear consequents. In cases where more than one theorem is applicable, the quality of the bounds will not be affected and they can synthesize the exact same set of bounds. In other words, no theorem is tighter. This is because they are all complete. In practice, we prefer to first use Farkas if possible, and switch to Handelman only if Farkas fails, and then to Putinar if Handelman fails. Our tool makes heuristic choices to decide which entailments should be kept in Handelman and which escalated to Putinar. The preference for Farkas over Handelman over Putinar is because there is a tradeoff between generality and the number of new variables introduced by each approach. Farkas introduces the fewest number of extra variables, namely the λ_i's in Theorem 1, whereas Handelman has to generate a coefficient for each monoid element, i.e. λ_i's in Theorem 2, and Putinar needs to set up a template in which we have a new variable for each non-zero entry of the lower-triangular matrix L of Theorem 5.

Chapter 3

Gas Upper-bounds for Smart Contracts

This chapter originally appeard in the following publication:

- Z. Cai, S. Farokhnia, A.K. Goharshady, S. Hitarth. *Asparagus: Automated Synthesis of Parametric Gas Upper-bounds for Smart Contracts.* ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2023.

3.1 Introduction

Background. Modern programmable blockchains have built-in support for smart contracts, i.e. programs that are stored on the blockchain and whose state is subject to consensus. After a smart contract is deployed on the blockchain, anyone on the network can interact with it and call its functions by creating transactions. The blockchain protocol is then used to reach a consensus about the order of the transactions and, as a direct corollary, the state of every smart contract. Reaching such consensus necessarily requires every node on the network to execute all function calls. Thus, an attacker can perform DoS by creating expensive transactions and function calls that use considerable or even possibly infinite time and space. To avoid this, following Ethereum, virtually all programmable blockchains have introduced the concept of "gas". A fixed hard-coded gas cost is assigned to every atomic operation and the user who calls a function has to pay for its total gas usage. This technique ensures that the protocol is not vulnerable to DoS attacks, but it has also had significant unintended consequences. Out-of-gas errors, i.e. when a user misunderestimates the gas usage of their function call and does not allocate enough gas, are a major source of security vulnerabilities in Ethereum.

Our Contributions. We focus on the well-studied problem of automatically finding upper-bounds on the gas usage of a smart contract. This is a classical problem in the blockchain community and has also been extensively studied by researchers in programming languages and verification. In this chapter, we provide a novel approach using theorems from polyhedral geometry and real algebraic geometry, namely Farkas' Lemma, Handelman's Theorem, and Putinar's Positivstellensatz, to automatically synthesize linear and polynomial parametric bounds for the gas usage of smart contracts. Our approach is the first to provide completeness guarantees for the synthesis of such parametric upper-bounds. Moreover, our theoretical results are independent of the underlying consensus protocol and can be applied to smart contracts written in any language and run on any blockchain. As a proof of concept, we also provide a tool, called "Asparagus" that implements our algorithms for Ethereum contracts written in Solidity. Finally, we provide extensive experimental results over 24,188 real-world smart contracts that are currently deployed on the Ethereum blockchain. We compare Asparagus against Gastap, which is the only previous tool that could provide parametric bounds, and show that our method significantly outperforms it, both in terms of applicability and the tightness of the resulting bounds. More specifically, our approach can handle 80.56% of the functions (126,269 out of 156,735) in comparison with GASTAP's 58.62%. Additionally, even on the benchmarks where both approaches successfully synthesize a bound, our bound is tighter in 97.85% of the cases.

3.2 Gas Upper-Bound as a Synthesis Problem

Our Focus. In this chapter, we focus on the problem of automated synthesis of parametric bounds on the gas usage of a smart contract. This problem is well-studied in the literature. There are two main motivations for automated bound synthesis: (i) ensuring that a contract can safely call libraries and functions in other contracts, and (ii) avoiding out-of-gas vulnerabilities, where wrong estimations of the gas behavior of a smart contract can lead to catastrophic real-world consequences. In the simplest case, if a user mistakenly underestimates the amount of gas that is needed for the execution of their desired function call and chooses a maximum gas value that is too small, then they would lose their entire deposit in the process above. This happens even if the estimate was just one unit short of the required gas. While this is irritating, out-of-gas errors can lead to much more severe losses, as further described below. Moreover, the decision on the allocated gas is not always taken by the final user, but often by the programmer of a smart contract.

Problem Formulation. In this chapter, we focus on the problem of automated synthesis of parametric upper-bounds for a given smart contract $\mathscr{C}$. Our goal is to find a parametric expression B_f for every function f of $\mathscr{C}$, such that B_f, which can potentially depend on the parameters passed to f, the variables stored in $\mathscr{C}$, and the gas usage of other functions called by f, serves as a guaranteed upper-bound on the gas usage of any call to f. Of course, we would like each B_f to be as tight as possible.

Related Works. Due to the importance of avoiding out-of-gas vulnerabilities, this is a well-studied problem and there are several previous tools that attempt to solve it. Notably, `solc`, the standard compiler of Solidity, which is the most commonly used programming language for Ethereum smart contracts, already includes a static analyzer that tries to find such upper-bounds and issues a warning to the programmer if it cannot prove that a function has small gas usage. However, these tools are only able to find constant bounds on the gas usage and would simply report $+\infty$ as their bound if no such constant can be found. To the best of our knowledge, the only exception is GASTAP [56], which was the only previous tool to provide parametric bounds. We provide an extensive experimental comparison with GASTAP in Section 3.4.

Undecidability and Limitations. The general case of our problem, where $\mathscr{C}$ can be any contract written in a Turing-complete language, is clearly undecidable since B_f is finite if and only if f terminates and thus halting is a special case of our problem. Therefore, we apply some abstractions and allow only numerical variables in the smart contracts. Specifically, we abstract away the arrays and instead just keep track of their size. These abstractions are sound, but not necessarily complete. Moreover, we focus on synthesizing polynomial upper-bounds B_f. See Section 2.4 for more formal treatment. Thus, our approach can only find polynomial bounds over polynomial transition systems. This excludes non-polynomial smart contracts. It also excludes contracts written in functional programming languages such as Scilla [57], which may contain higher-order functions.

MOTIVATION FOR POLYNOMIAL BOUNDS. In our experimental result, 653 (2.7%) of the benchmarks required quadratic polynomial bounds. These are usually contracts that compute hashes or allocate memory within loops. Thus, no constant or linear bounds exist in these cases and non-linear bounds are required. Polynomial bounds are also desirable for the following additional reasons:

- Certain EVM operations, such as `MSTORE`, incur memory expansion costs which are *quadratic* in size of the used memory. To handle contracts that use a non-constant amount of memory, one has to go beyond linear bounds and introduce polynomials.

- At the time of writing, the gas price is too high, leading many smart contract programmers to avoid loops altogether. This is a major issue that is artificially keeping the contracts simple and not allowing them to exploit the promise of a Turing-complete language. Indeed, the vast majority of functions our experiments had constant gas usage. We expect that with the recent switch to proof-of-stake and proposals for relaxations of the block gas limit, the gas price would decrease significantly, allowing smart contract programmers to write more complicated functionality. Thus, our expectation is to see more contracts with non-linear gas usage in the future.

- Our approach outperforms both `solc` and GASTAP even in the cases when the gas bound is constant. Additionally, our bounds are tighter even when both approaches synthesize constant or linear bounds. These gains are due to the extra expressivity of our polynomial templates.

3.3 Automated Gas Analysis Framework

In this section, we first describe the translation pipeline from smart-contract bytecode to a polynomial transition system (PTS). We then present our synthesis procedure for deriving parametric gas bounds by constructing remaining-gas polynomials over the resulting PTS.

3.3.1 Intermediate Representations and Polynomial Transition Systems

GASTAP [56]. GASTAP is a tool to process EVM bytecode into an intermediate rule-based representation (RBR) and estimate gas bounds of function calls. The part of the GASTAP project which obtains the RBR is called EthIR and is publicly available [58]. Our algorithm works on a polynomial transition system (PTS, defined further below). However, the translation from RBR to PTS is much simpler than a direct translation from EVM bytecode. So, in practice, we first use [58] to translate EVM to RBR and then convert the RBR to PTS.

EVM to CFG. GASTAP firstly converts an EVM bytecode to a control flow graph (CFG). The EVM language uses a local stack, a volatile memory, and a persistent storage that is part of the blockchain state. Given the compiled smart contract, GASTAP groups the code into EVM basic blocks, which are maximal sequences of straight-line code. Each EVM block ends with either a `JUMP/JUMPI` operation where program execution jumps to a specified location, or an ending instruction like `RETURN` or `REVERT`. Each EVM block has an address according to the location where it appears in the bytecode. When a block jumps, static analysis is applied to parse the jump destination and form an edge to the destination block. As a result, a stack-sensitive control flow graph is created. This CFG is then used to translate the program into RBR and PTS formats below. See [58] for detailed syntax and semantics of RBR.

Example 4. *Figure 3.1 shows two Ethereum smart contracts, written in Solidity, which will serve as our running examples. Figure 3.2 shows part of the RBR representation of these contracts, as well as the edges in their CFGs.*

Polynomial Update Functions, Assertions and Transitions. Let $\mathbb{V} = \{v_1, \dots, v_k\}$ denote a finite set of variables. We denote the set of all polynomial expressions with real coefficients over the variables $\mathbb{V}$ by $\mathbb{R}[\mathbb{V}]$. A polynomial *update function* $U : \mathbb{V} \to \mathbb{R}[\mathbb{V}]$ maps each variable in $\mathbb{V}$ to a polynomial over $\mathbb{V}$. A *polynomial assertion* G is a logical formula formed by a boolean combination of the assertions of the form $f \geq 0$ and $f > 0$, where $f \in \mathbb{R}[\mathbb{V}]$. A *polynomial transition* $\tau = (U, G)$ is a pair of an update function and a guard. With a slight misuse of notation, we sometimes write τ_U instead of U and τ_G to denote G.

Valuations and Notation. A *valuation* $\sigma : \mathbb{V} \to \mathbb{R}$ assigns a real value to each variable in $\mathbb{V}$. A polynomial f can be evaluated at any valuation σ naturally by

```
contract VotingContract {

 struct Proposal {bytes32 name;  uint
    voteCount;}
 Proposal[] public proposals;

 function winningProposal() public
    returns (uint winningProposal){
  uint winningVoteCount = 0;
  for (uint p=0;p<proposals.length;p++){
   if (proposals[p].voteCount>
      winningVoteCount) {
    winningVoteCount = proposals[p].
       voteCount;
    winningProposal = p;
   }}}}
```

```
contract NestedLoop{

function main(uint a,
    uint b) public
    returns(uint)
 {
  uint count = 0;
  for (uint i=0;i<a;++i)
   for (uint j=0;j<b;++j
      )
    ++count;
  return count;
 }
}
```

Figure 3.1: Two Example Contracts

substituting the value of each variable in f and computing it. We will denote this value by $f(\sigma)$. Let U be an update function, then $U(\sigma)$ denotes a valuation in which $U(\sigma)(v) = U(v)(\sigma)$ for each v. If F is a polynomial or polynomial assertion, we define $U(F)$ as the result of substituting each occurrence of a variable v in F by the polynomial $U(v)$.

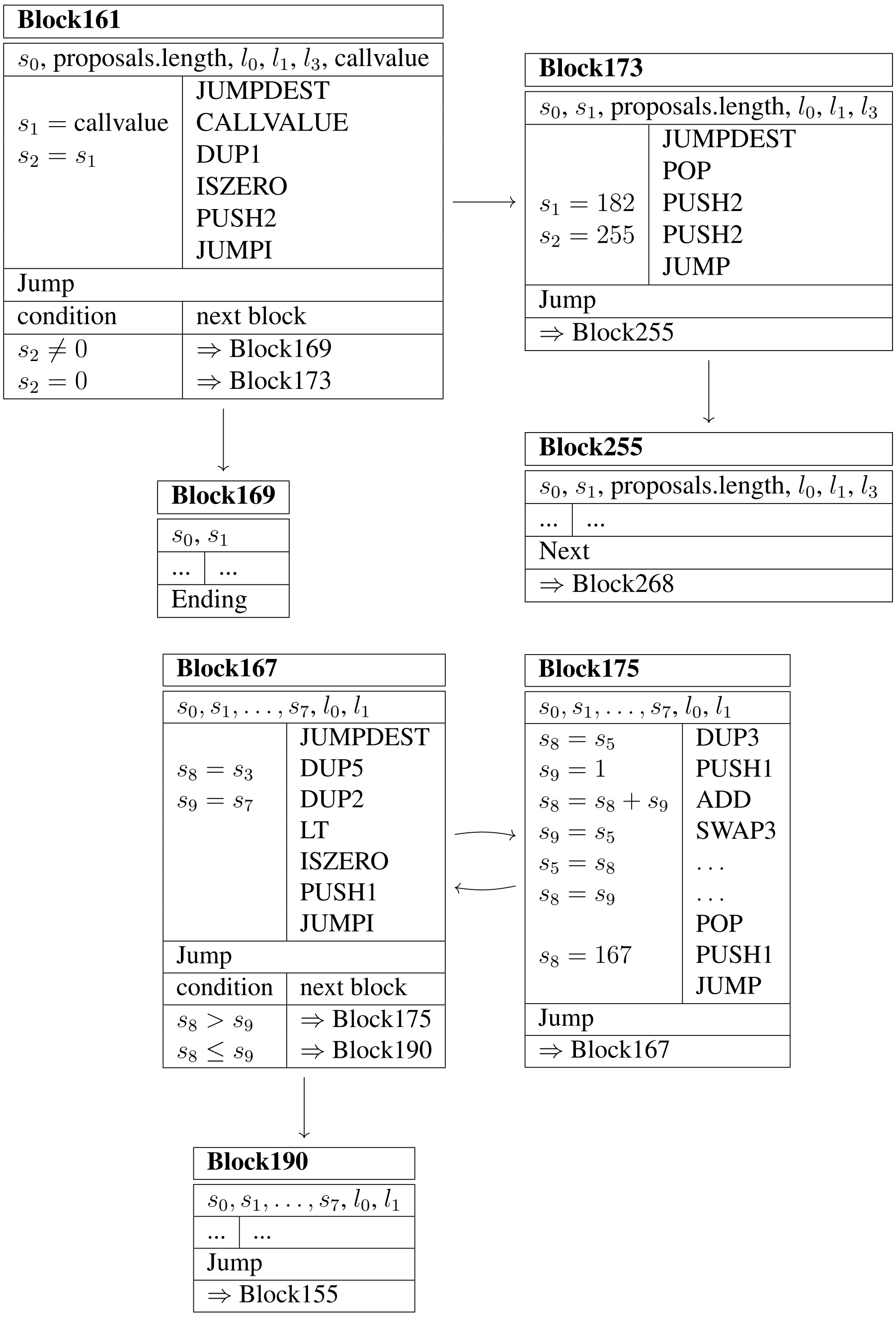
Composition of Transitions. Given two polynomial transitions $\tau_1 = (U_1, G_1)$ and $\tau_2 = (U_2, G_2)$, we denote their composition by $\tau_1 \circ \tau_2$ and define it as (U, G) where $U(v) = U_1(U_2(v))$, and $G = G_1 \wedge U_1(G_2)$.

PTS. A *Polynomial Transition System* (PTS) T is a tuple $(\mathbb{V}, \mathbb{L}, l_0, l_f, \delta)$ where $\mathbb{V}$ is a set of variables, $\mathbb{L}$ is a set of locations, l_0 and l_f are the initial and final locations, respectively, and δ is a partial function that maps each pair of locations (l, l') to a polynomial transition (U, G).

Example 5. *Figure 3.3 provides a simplified PTS representation for each of our example contracts of Figure 3.1. In practice, our tool automatically converts the CFG provided by [58] to an RBR. See Section 3.3.2 for details. However, the resulting PTSs are usually not human-readable, so Figure 3.3 is not a direct output of our tool, but a simplified version.*

Runs. Given an initial valuation $\sigma : \mathbb{V} \to \mathbb{R}$, a *run* R of the PTS T is an alternating sequence of locations and valuations $R := \sigma_0, l_0, \sigma_1, l_1, \sigma_2, l_2, \sigma_3 \ldots$ satisfying the following constraints:

1. $\sigma_i = \tau_U(\sigma_{i-1})$ for every $i \geq 1$, where $\tau := \delta(l_{i-1}, l_i)$.



Block161	
s_0, proposals.length, l_0, l_1, l_3, callvalue	
	JUMPDEST
s_1 = callvalue	CALLVALUE
$s_2 = s_1$	DUP1
	ISZERO
	PUSH2
	JUMPI
Jump	
condition	next block
$s_2 \neq 0$	⇒ Block169
$s_2 = 0$	⇒ Block173

Block173	
s_0, s_1, proposals.length, l_0, l_1, l_3	
	JUMPDEST
	POP
$s_1 = 182$	PUSH2
$s_2 = 255$	PUSH2
	JUMP
Jump	
⇒ Block255	

Block169	
s_0, s_1	
...	...
Ending	

Block255	
s_0, s_1, proposals.length, l_0, l_1, l_3	
...	...
Next	
⇒ Block268	

Block167	
$s_0, s_1, \ldots, s_7, l_0, l_1$	
	JUMPDEST
$s_8 = s_3$	DUP5
$s_9 = s_7$	DUP2
	LT
	ISZERO
	PUSH1
	JUMPI
Jump	
condition	next block
$s_8 > s_9$	⇒ Block175
$s_8 \leq s_9$	⇒ Block190

Block175	
$s_0, s_1, \ldots, s_7, l_0, l_1$	
$s_8 = s_5$	DUP3
$s_9 = 1$	PUSH1
$s_8 = s_8 + s_9$	ADD
$s_9 = s_5$	SWAP3
$s_5 = s_8$	...
$s_8 = s_9$	...
	POP
$s_8 = 167$	PUSH1
	JUMP
Jump	
⇒ Block167	

Block190	
$s_0, s_1, \ldots, s_7, l_0, l_1$	
...	...
Jump	
⇒ Block155	

Figure 3.2: Part of the Rule-based Representations (RBR) of *VotingContract* (top) and *NestedLoop* (bottom)

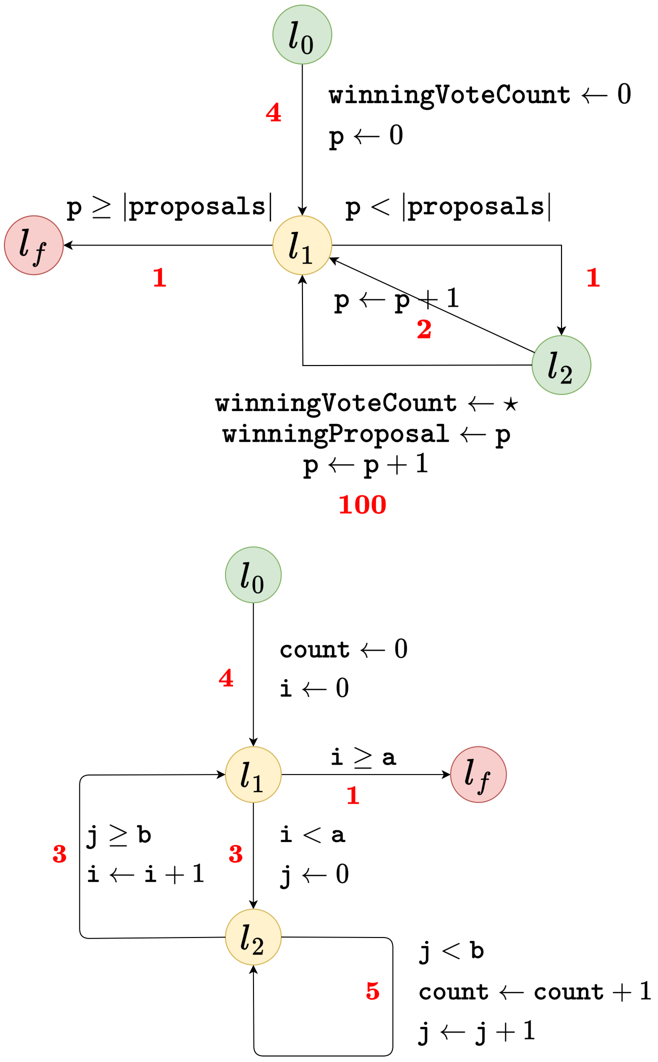


Figure 3.3: Simplified PTS Representations T_1 (top) and T_2 (bottom) of Running Examples. The numbers in red denote the cost of each transition. In these examples, the costs are constant, but our approach supports arbitrary polynomial costs. Cutpoints are shown in yellow.

2. $\sigma_i \models \tau_G$ for every $i \geq 1$. Here, we again have $\tau := \delta(l_{i-1}, l_i)$.

Example 6. *Consider the initial valuation* $\sigma_0 = \{\boldsymbol{a} \mapsto 0, \boldsymbol{b} \mapsto 0, \boldsymbol{count} \mapsto -1, \boldsymbol{i} \mapsto 10, \boldsymbol{j} \mapsto 0\}$ *in the PTS of Figure 3.3 (bottom). An example run starting from this valuation is:*

$$\{\boldsymbol{a} \mapsto 0, \boldsymbol{b} \mapsto 0, \boldsymbol{count} \mapsto -1, \boldsymbol{i} \mapsto 10, \boldsymbol{j} \mapsto 0\}, l_0$$
$$\{\boldsymbol{a} \mapsto 0, \boldsymbol{b} \mapsto 0, \boldsymbol{count} \mapsto 0, \boldsymbol{i} \mapsto 0, \boldsymbol{j} \mapsto 0\}, l_1$$
$$\{\boldsymbol{a} \mapsto 0, \boldsymbol{b} \mapsto 0, \boldsymbol{count} \mapsto 0, \boldsymbol{i} \mapsto 0, \boldsymbol{j} \mapsto 0\}, l_f.$$

Cutsets. Given a PTS T, a *cutset* of T is a set of locations C such that (i) C contains l_0 and l_f, and (ii) every cycle in the underlying graph of T passes through some location in the set C. In other words, removing C from T will reduce it to a directed acyclic graph. Each location in C is called a *cutpoint*.

Example 7. *In Figure 3.3 (top), the set* $\{l_0, l_1, l_f\}$ *is a cutset. Similarly, in Figure 3.3 (bottom),* $\{l_0, l_2, l_f\}$ *is a cutset. It is easy to verify that removing these cutsets would eliminate all cycles. Note that the edge labeled* ***5*** *is a cycle on its own, thus* l_2 *has to be included in every cutset.*

Basic Paths. Given a cutset C, a path $\Pi := l_1, \tau_1, \ldots, l_{m-1}, \tau_{m-1}, l_m$ in the underlying graph of T is said to be *basic* if it starts and ends at cutpoints and does not pass through any cutpoints in between. It is straightforward to observe that the number of all possible basic paths in a PTS is finite as there could be only a finite number of basic paths between each pair of cutpoints. We extend our transition functions to paths and define $\tau(\Pi) := \bigcirc_{i=0}^{m}(\tau_i)$.

Example 8. *Using the cutsets from the previous example, the path* l_1, ***1***, l_2, ***2***, l_1 *is a basic path of the top PTS and* l_2, ***5***, l_2 *and* l_0, ***4***, l_1, ***3***, l_2 *are examples of basic paths in the bottom PTS.*

Invariant Maps. Given a cutset C of the PTS T, we call a map $\mathbb{I}$ that maps each cutpoint to a polynomial assertion an *invariant map* if for any run $R = \sigma_0, l_1, \sigma_1, l_2, \sigma_2, \ldots$ of T that starts from some cutpoint l_1 and initial valuation $\sigma_0 \models \mathbb{I}(l_1)$, we have $\sigma_i \models \mathbb{I}(l_i)$ whenever $l_i \in C$. In other words, if the initial valuation satisfies the invariant at the start cutpoint, then whenever the run R reaches a cutpoint l_i, the valuation of the variables satisfies the invariant at l_i.

Inductive Invariants. An invariant map $\mathbb{I}$ over cutset C is said to be an *inductive invariant map* if for every pair of locations $l, l' \in C$ and every basic path $\Pi := l, \tau_0, l_1 \ldots l_{m-1}, \tau_m, l'$ from l to l' we have $\mathbb{I}(l) \wedge \tau_G \implies \tau_U(\mathbb{I}(l'))$, where $\tau = \bigcirc_{i=0}^{m}(\tau_i)$. In other words, if a valuation satisfies the invariant at the cutpoint l and also satisfies the transition conditions τ_G, then the updated valuation obtained using τ_U satisfies the invariant at the cutpoint l'.

Invariant Generation. Invariant generation and more specifically, the automated synthesis of linear/polynomial inductive invariants, is an orthogonal and well-studied problem with practically efficient tools such as [59, 60]. As such, in the sequel, we assume that every PTS comes with invariants generated using one of these tools.

Abstractions in the Translation from RBR to PTS. We translate a smart contract from the RBR format to a PTS in the standard manner, i.e. creating one location for every line or every basic block of code and following the operations and guards as in the control flow graph. See the next section for an example. However, this process necessarily leads to some imprecision:

- All variables in a PTS are real-valued. Hence, integer variables are converted to real.
- Some operations are inherently not applicable to real variables. We handle these by relying on non-determinism. For example, if we have `c := a%b`. In the PTS, the variable `c` will get a non-deterministic value and the invariant $0 \leq \mathtt{c} \leq \mathtt{b} - 1$ is added. We handle integer division similarly.
- The real variables in a PTS are unbounded, whereas the variable types available in real-world smart contracts are bounded. We add these bounds, e.g. $-2^{31} \leq \mathtt{x} \leq 2^{31} - 1$ for a 32-bit integer `x`, to the invariants.
- There is no support for arrays, stacks or strings in a PTS. Thus, we replace each array with a variable that keeps track of its size. We handle stacks and strings similarly.
- Some contracts have calls to external functions that were not available to us. The results of these calls are also handled by non-determinism. Moreover, for each such external function, we define a new variable that models its total gas cost and use it in our parametric bounds.

Note that the points above do not affect the soundness of our approach and hence all of our obtained bounds are correct. Moreover, the gas cost we assign to every PTS transition is the actual gas cost of the same transition in the original contract and is not at all affected by the translation to PTS. Our experimental results in Section 3.4 demonstrate the applicability of our approach to real-world smart contracts. Hence, while it is theoretically possible to write adversarial smart contracts to which our approach is not applicable, the vast majority of real-world smart contracts can be modeled as polynomial transition systems.

3.3.2 Synthesizing Parametric Gas Bounds

Now, we provide an overview of our approach and illustrate it on the two running examples of Figure 3.1. To increase readability, we leave out some of the mathematical details which was presented in Section 2.6. The central idea behind our algorithm is to synthesize a *remaining gas polynomial* at every line of a cutset which models an upper-bound on the possible gas usage if the program starts executing from that line. This is a straightforward extension of the classical concept of ranking functions. We then write a set of entailments between polynomial inequalities which model the requirements of our remaining gas polynomial. Finally, we use tools from polyhedral and real algebraic geometry to translate these entailments into quadratic constraints which are in turn passed to an external solver.

Our algorithm starts with a smart contract and translates it into a set of quadratic constraints using the steps shown in Figure 3.4. Our central theoretical contribution is the preservation of both soundness and completeness from the PTS all the way to the final solution. In other words, imprecisions can only be introduced in the translations from the smart contract to RBR or from RBR to PTS.

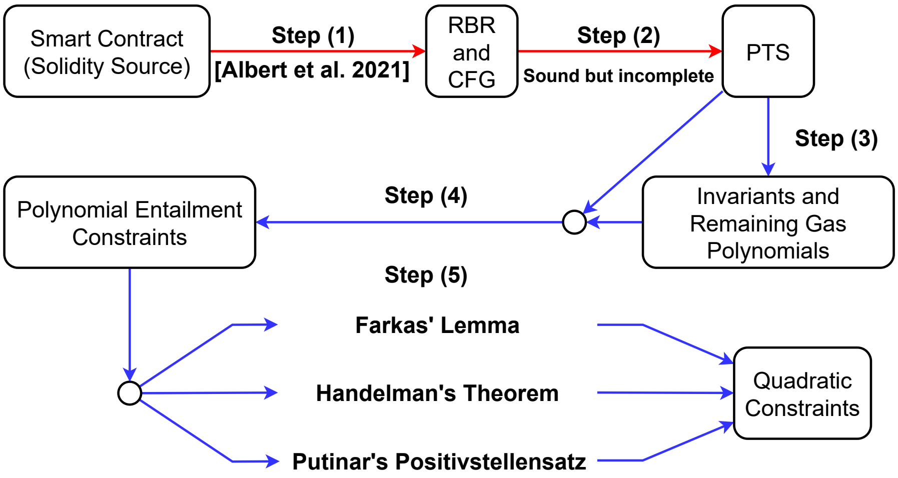


Figure 3.4: The Steps of Our Algorithm. Sound and Semi-complete steps are shown in blue, whereas sound steps which might break completeness are in red. See Theorem 6 for a more detailed treatment of soundness and completeness.

Our Algorithm. Our algorithm consists of the following six steps:

1. **Converting the Smart Contract to RBR.** To enable the static analysis of the smart contract, we use the intermediate representation of [56]. We compile the smart contract using the standard `solc` compiler to EVM bytecode, then we construct the control-flow graph of the bytecode and convert it to the RBR format, which explicitly represents the local and state variables, the operand stack, and blockchain data. This increases the number of variables that we use

in RBR but is essential to facilitate further static analyses. For this step, we rely entirely on the standard compiler and [56].

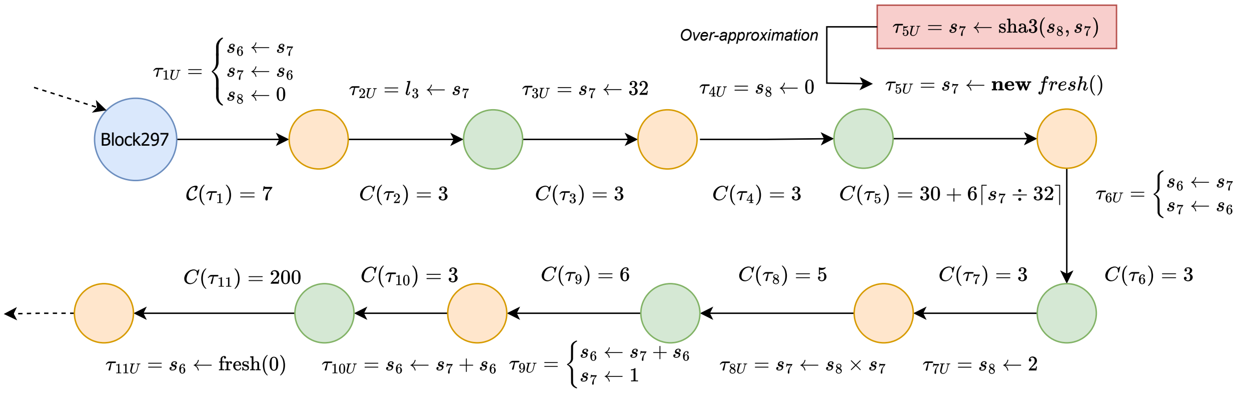
Example 9. *Figure 3.2 shows parts of the RBRs and CFGs of the two running examples as a graph wherein each vertex is a basic block and each edge is a transition. This is the output of the first step of our algorithm.*

2. **Transforming the RBR to Polynomial Transition System.** We parse the RBR file and create a corresponding polynomial transition system (PTS) $T = (\mathbb{V}, \mathbb{L}, l_0, l_f, \delta)$ as follows:

   (a) **Soundness-preserving Over-approximations.** Since the PTS can only have polynomial operations like addition, subtraction, and multiplication, we must perform a few modifications which preserve the soundness of our approach. As an example, if we encounter a non-polynomial expression such as `x := a%b` in the RBR, we replace it with `x := y` where `y` is a fresh variable with the invariant $0 \leq \texttt{y} \leq b - 1$. In other words, we are over-approximating non-polynomial operations by non-determinism. The soundness directly follows because our subsequent analysis is then guaranteed to work for all values of `y`, and hence it will also work for the actual value of `x`.

   (b) **Establishing a Gas Consumption Polynomial.** We associate a *gas consumption polynomial* $\mathcal{C}(\tau) \in \mathbb{R}[\mathbb{V}]$ to each transition τ in the PTS T. The polynomial $\mathcal{C}(\tau)$, when evaluated at a valuation σ, provides the amount of gas that will be consumed on taking the transition τ starting from the valuation σ. In practice, this is a direct encoding of the Ethereum gas table and does not add any imprecision. This is because the gas costs of all EVM opcodes are polynomial expressions. We note that the gas usage assigned to each transition is not necessarily constant and can be any polynomial. Handling polynomial costs is necessary for a faithful modeling of operations such as `SSTORE` which have a quadratic gas usage.

   (c) **Gas Consumption Polynomial of a Path.** Let $\Pi := l_1, \tau_1, l_2, \tau_2, l_3$ be a path in T. The total gas consumed on taking the path Π can be obtained by adding the consumption polynomial of the first transition to the consumption polynomial of the second transition, in which the values of all variables are updated according to the first transition. Formally, we define the consumption $\mathcal{C}(\Pi) = \mathcal{C}(\tau_1) + \tau_{1U}(\mathcal{C}(\tau_2))$. It is straightforward to extend this definition to paths of arbitrary length.



$\tau_{1U} = \begin{cases} s_6 \leftarrow s_7 \\ s_7 \leftarrow s_6 \\ s_8 \leftarrow 0 \end{cases}$
Block297
$\tau_{2U} = l_3 \leftarrow s_7$
$\tau_{3U} = s_7 \leftarrow 32$
$\tau_{4U} = s_8 \leftarrow 0$
Over-approximation
$\tau_{5U} = s_7 \leftarrow \text{sha3}(s_8, s_7)$
$\tau_{5U} = s_7 \leftarrow \textbf{new}\ \textit{fresh}()$
$\mathcal{C}(\tau_1) = 7$
$C(\tau_2) = 3$
$C(\tau_3) = 3$
$C(\tau_4) = 3$
$C(\tau_5) = 30 + 6\lceil s_7 \div 32\rceil$
$\tau_{6U} = \begin{cases} s_6 \leftarrow s_7 \\ s_7 \leftarrow s_6 \end{cases}$
$C(\tau_{11}) = 200$
$C(\tau_{10}) = 3$
$C(\tau_9) = 6$
$C(\tau_8) = 5$
$C(\tau_7) = 3$
$C(\tau_6) = 3$
$\tau_{11U} = s_6 \leftarrow \text{fresh}(0)$
$\tau_{10U} = s_6 \leftarrow s_7 + s_6$
$\tau_{9U} = \begin{cases} s_6 \leftarrow s_7 + s_6 \\ s_7 \leftarrow 1 \end{cases}$
$\tau_{8U} = s_7 \leftarrow s_8 \times s_7$
$\tau_{7U} = s_8 \leftarrow 2$

Figure 3.5: Part of the PTS for *VotingContract*

Example 10. *Figure 3.5 shows part of the transition system obtained for VotingContract . This contains not only a renamed version of every variable in the original contract, but also new operations which are not readily apparent in the Solidity code, such as keeping track of the function call stacks and their sizes. As another example, an array in Solidity is compiled to a hash-based map in EVM bytecode. Thus, accessing an element of the array requires a call to a hash function. Since this hash function is not a polynomial, we abstract it away and replace it with non-determinism in our PTS, thus over-approximating the behavior of the original contract. Also note that at this point every transition* τ *has a well-defined cost* $\mathcal{C}(\tau)$*.*

As shown in Figure 3.5, the polynomial transition system corresponding to a contract can become quite complicated due to the internal implementation details of Solidity. Thus, to make our illustration human-readable, we consider a simplified PTS for each of our contracts, as shown in Figure 3.3. Note that this is only for illustration and our algorithm works on the original PTS above. In the PTS for VotingContract , we have a variable |`proposals`| *which keeps track of the length of the array.* l_1 *corresponds to the* `for` *loops and* l_2 *to the* `if`*. Note that* l_2 *has two outgoing edges, which are chosen non-deterministically. This is because the branching in the original contract depends on the values saved in the array. However, these values are abstracted away and only the array's length is saved in the PTS. Thus, the transition becomes non-deterministic. Similarly, the assignment to* `winningVoteCount` *is based on a value in the array and is hence replaced by non-determinism.*

3. **Adding Symbolic Invariants and Remaining Gas Polynomials.** Let C be a cutset of T and $\mathbb{I}$ be an invariant map for C. Note that $\mathbb{I}$ can have both concrete and symbolic polynomials, i.e. polynomial templates with unknown

coefficients, as invariants. Our tool can use invariants generated by [59, 60] or generate its own if the given invariants are not strong enough.

(a) **Remaining Gas Polynomial.** We associate a so-called *Remaining Gas Polynomial* B_l to each cutpoint $l \in C$. The idea is that we want to ensure that for every basic path Π starting and ending at the cutpoints l and l', respectively, if we start with an amount of gas that is at least B_l, then the total gas *left* after running through the path is at least $B_{l'}$. The mapping B is said to be a remaining gas polynomial map over cutset C. Note that we do not know the B_l's. Thus, our algorithm simply creates a symbolic template for each of them, i.e. a symbolic polynomial in which the coefficients are unknown real values that have to be synthesized in the future. Our algorithm can create similar templates for the invariants, if they are not provided, and synthesize them similarly[1].

Recall that whenever we reach the location l, the invariant $\mathbb{I}(l)$ must hold. Therefore, we can formalize our requirements on B_l and $B_{l'}$ by taking any basic path Π from l to l' and requiring:

$$\mathbb{I}(l) \wedge \tau_G \implies B_l - \mathcal{C}(\Pi) \geq \tau_U(B_{l'}) \tag{3.1}$$

where $\tau := \tau(\Pi)$ is the composition of all transitions of Π.

(b) **Remaining Gas at the Final Location.** The remaining gas polynomial of the final location l_f is always set to 0. In other words, we ensure that if we reach the final location l_f, no matter what path we took to reach there from the initial location l_0, if we start with gas at least B_{l_0} evaluated at initial valuation, then we will always be left with non-negative gas on termination.

Example 11. *For the first example, we are looking for a linear bound. Thus, at every location $l_i \neq l_f$ of the PTS T_1, our algorithm computes the following remaining gas polynomial:*

$$B_{l_i} := t_{i,0} + t_{i,1} \cdot \texttt{winningVoteCount} + t_{i,2} \cdot \texttt{p} + t_{i,3} \cdot |\texttt{proposals}| + t_{i,4} \cdot \texttt{winningProposal}.$$

At l_f we have $B_{l_f} = 0$. Here, the $t_{i,j}$'s are new unknowns, which are called template variables*. Our goal is to find concrete values for the template variables such that the B_{l_i}'s satisfy the requirements of the remaining gas polynomials as in* (3.1)*.*

[1] To generate the templates, the algorithm needs to know the maximum degree d that might be used for the polynomials. In practice, we simply try different values of d, starting from 1 and going up, until the algorithm succeeds.

For the PTS T_2, our goal is to obtain a quadratic bound, thus our algorithm symbolically computes the following template expression at every location $l_i \neq l_f$:

$$B_{l_i} := t_{i,0} + t_{i,1} \cdot \texttt{count} + t_{i,2} \cdot \texttt{a} + t_{i,3} \cdot \texttt{b} + t_{i,4} \cdot \texttt{i} + t_{i,5} \cdot \texttt{j} +$$

$$t_{i,6} \cdot \texttt{count}^2 + t_{i,7} \cdot \texttt{count} \cdot \texttt{a} + t_{i,8} \cdot \texttt{count} \cdot \texttt{b} + t_{i,9} \cdot \texttt{count} \cdot \texttt{i} + t_{i,10} \cdot \texttt{count} \cdot \texttt{j} +$$

$$t_{i,11} \cdot \texttt{a}^2 + t_{i,12} \cdot \texttt{a} \cdot \texttt{b} + t_{i,13} \cdot \texttt{a} \cdot \texttt{i} + t_{i,14} \cdot \texttt{a} \cdot \texttt{j} +$$

$$t_{i,15} \cdot \texttt{b}^2 + t_{i,16} \cdot \texttt{b} \cdot \texttt{i} + t_{i,17} \cdot \texttt{b} \cdot \texttt{j} +$$

$$t_{i,18} \cdot \texttt{i}^2 + t_{i,19} \cdot \texttt{i} \cdot \texttt{j} + t_{i,20} \cdot \texttt{j}^2$$

In other words, the template for the remaining gas polynomial at every location includes all the possible monomials of degree at most 2 over the variables. Moreover, each monomial appears with an unknown coefficient $t_{i,j}$ that should be synthesized.

4. **Reduction to Entailment Constraints.** In this step, we reduce our main problem of synthesizing the remaining gas polynomial for each cutpoint to solving polynomial entailment constraints of the following standard form:

$$f_1 \bowtie_1 0 \wedge f_2 \bowtie_2 0 \wedge \ldots \wedge f_r \bowtie_r 0 \implies g \bowtie_0 0,$$

in which g and every f_i are polynomials over the PTS variables, whose *coefficients* might be symbolic and contain template variables. Moreover, we have $\bowtie_i \in \{\geq, >\}$.

Recall that $\mathbb{I}$ is a symbolic inductive invariant map and B a symbolic remaining gas polynomial map over the cutset C. The polynomials in both $\mathbb{I}$ and B are over the PTS variables. However, their coefficients are not concrete real numbers, but rather symbolic expressions over the template variables. This is why we call them symbolic polynomials.

Let P be the set of all basic paths in the PTS T. Let Φ be the set of constraints generated until this point of the algorithm. For each basic path $\Pi \in P$ going from l to l', and having transition effect τ, we add the following constraint to Φ :

$$\mathbb{I}(l) \wedge \tau_G \implies \tau_U(\mathbb{I}(l')) \wedge B_l - \tau_U(B_{l'}) \geq \mathcal{C}(\Pi).$$

The first conjunct $\tau_U(\mathbb{I}(l'))$ of the conclusion makes sure that the $\mathbb{I}$ is an inductive invariant map, and the second conjunct $B_l - \tau_U(B_{l'}) \geq \mathcal{C}(\Pi)$ of the conclusion ensures that B is a valid remaining gas polynomial map.

Example 12. *We give the constraints for one basic path. Other basic paths are handled similarly. In T_1, consider the basic path that starts at l_1, goes to l_2 with* *a cost of* 1 *and then back to* l_1 *with a cost of* 2. *This basic path can only be taken if* $\mathtt{p} < |\mathtt{proposals}|$. *Moreover, it increases the value of* $\mathtt{p}$ *by* 1. *Thus, our algorithm computes the following constraint:*

$$\mathbb{I}(l_1) \ \wedge \ \mathtt{p} < |\mathtt{proposals}| \implies \mathbb{I}(l_1)[\mathtt{p} \leftarrow \mathtt{p}+1] \ \wedge \ B_{l_1} - B_{l_1}[\mathtt{p} \leftarrow \mathtt{p}+1] \geq 3.$$

Suppose that the invariant at l_1 *is* $\mathbb{I}(l_1) := \mathtt{p} \geq 0 \ \wedge \ \mathtt{p} \leq |\mathtt{proposals}| + 1$. *The algorithm expands the constraint above and obtains:*

$$\mathtt{p} \geq 0 \ \wedge \ \mathtt{p} \leq |\mathtt{proposals}| + 1 \ \wedge \ \mathtt{p} < |\mathtt{proposals}| \implies$$

$$\mathtt{p\ +\ 1} \geq 0 \ \wedge \ \mathtt{p+1} \leq |\mathtt{proposals}| + 1 \ \wedge$$

$$t_{1,0} + t_{1,1} \cdot \mathtt{winningVoteCount} + t_{1,2} \cdot \mathtt{p} + t_{1,3} \cdot |\mathtt{proposals}| + t_{1,4} \cdot \mathtt{winningProposal} -$$

$$t_{1,0} - t_{1,1} \cdot \mathtt{winningVoteCount} - t_{1,2} \cdot \mathtt{(p\ +\ 1)} - t_{1,3} \cdot |\mathtt{proposals}| - t_{1,4} \cdot \mathtt{winningProposal} \geq 3$$

The algorithm simplifies the constraint using standard algebraic operations and obtains:

$$\mathtt{p} \geq 0 \ \wedge \ \mathtt{p} < |\mathtt{proposals}| \implies$$

$$\mathtt{p} + 1 \geq 0 \ \wedge \ \mathtt{p} + 1 \leq |\mathtt{proposals}| + 1 \ \wedge \ -t_{1,2} \geq 3.$$

5. **Reducing Entailment Constraints to Quadratic Constraints.** In the last step, the problem was reduced to solving a system of entailment constraints of the following form:

$$\textstyle\bigwedge_i f_i \geq 0 \implies g \geq 0.$$

Note that our approach can also handle strict inequalities. However, we focus on non-strict inequalities in the presentation. In the constraint above g and each f_i are symbolic polynomials over the variables $\mathbb{V}$ whose coefficients are symbolic expressions over the set $\mathbb{T}$ of template variables. Our goal is to find concrete values for each template variable $t \in \mathbb{T}$, such that the constraints above hold for every valuation over $\mathbb{V}$. So, this is a formula in the first-order theory of the reals and hence decidable.

Unfortunately, if we pass our constraints directly to a Non-linear Real Arithmetic (NRA) solver, or an SMT solver, it will have to deal with the notoriously difficult problem of quantifier elimination, since our formula has a quantifier alternation. Solvers for the first-order theory of the reals are famously inefficient and cannot handle even toy examples.

Our main algorithmic breakthrough in this step is to employ certain theorems from polyhedral and real algebraic geometry (Detailed in Section 2.6) to eliminate this quantifier alternation and reduce the problem to a set of quadratic constraints. This reduced instance does not involve any quantifier alternation and is usually much easier to solve using NRA solvers. Moreover, we will obtain a set of quadratic constraints over the variables in $\mathbb{T}$ only and all variables in $\mathbb{V}$ will be eliminated from the formula.

We now show the reduction for a single entailment constraint. Since our constraints are composed conjunctively, our algorithm applies the same reduction to every one of them and then conjunctively combines the quadratic constraints corresponding to each entailment constraint. We consider several cases:

(a) **Linear.** If both the premise and the conclusion of an entailment constraint consist only of linear inequalities over the variables, then we employ Farkas' Lemma to reduce the problem to quadratic constraints (Theorem 1).

Example 13. *We illustrate how we use Farkas' Lemma to handle the following constraint:*

$$t_0 \cdot x + y - 2 \geq 0 \wedge 2 \cdot x - t_1 \cdot y + 1 \geq 0 \implies 4 \cdot x - y - 3 \geq 0$$

where $t_0, t_1 \in \mathbb{T}$ are unknown template variables. Suppose we can somehow achieve

$$4 \cdot x - y - 3 \equiv \lambda_0 + \lambda_1 \cdot (t_0 \cdot x + y - 2) + \lambda_2 \cdot (2 \cdot x - t_1 \cdot y + 1)$$

with $\lambda_0, \lambda_1, \lambda_2$ being non-negative real numbers. This equality essentially guarantees that whenever $t_0 \cdot x + y - 2$ and $2 \cdot x - t_1 \cdot y + 1$ are non-negative, $4 \cdot x - y - 3$ will also be non-negative, since we are writing the conclusion as a linear combination of the premises and only using non-negative multipliers λ_i. We can simplify the RHS:

$$4 \cdot x - y - 3 \equiv (\lambda_1 \cdot t_0 + 2 \cdot \lambda_2) \cdot x + (\lambda_1 - \lambda_2 \cdot t_1) \cdot y + (\lambda_0 - 2 \cdot \lambda_1 + \lambda_2).$$

However, this equality should hold for all values of x and y. Hence, this equivalence will be true iff the corresponding coefficients on both sides are equal, i.e., $\lambda_1 \cdot t_0 + 2 \cdot \lambda_2 = 4$, $\lambda_1 - \lambda_2 \cdot t_1 = -1$, and $\lambda_0 - 2 \cdot \lambda_1 + \lambda_2 = -3$. These are precisely our quadratic constraints. Note that these constraints do not involve any of the PTS variables x and y, and there is also no quantifier alternation. Any solution to this set of quadratic constraints will give us a model for t_0 and t_1 that satisfies the original entailment.

Passing this system of quadratic constraints to an external solver, one possible solution could be $t_0 = 1, t_1 = 3, \lambda_0 = 0, \lambda_1 = 2, \lambda_2 = 1$. Plugging *these values back, we obtain the following valid constraint which holds for all values of x and y:*

$$\boxed{1} \cdot x + y - 2 \geq 0 \wedge 2 \cdot x - \boxed{3} \cdot y + 1 \geq 0 \implies 4 \cdot x - y - 3 \geq 0$$

(b) **Linear Premise and Non-linear Conclusion.** If the premise is linear but the conclusion is non-linear, then we use Handelman's Theorem (Theorem 2) to reduce the entailment constraint to quadratic constraints.

Example 14. *Consider the following constraint: $t_0 \cdot x - 2 \geq 0 \wedge x - t_1 \cdot y \geq 0 \implies x^2 - 8 \cdot y + 4 \geq 0$, where $t_0, t_1 \in \mathbb{T}$ are unknown constants.*

We cannot apply the same idea as in Farkas' Lemma since no linear combination of $t_0 \cdot x - 2$ and $x - t_1 \cdot y$ can generate a non-linear polynomial like $x^2 - 8 \cdot y + 4$. The intuition behind applying Handelman's Theorem is that as we know $t_0 \cdot x - 2$ is non-negative, then we can also assume that every power of this polynomial is also non-negative. More specifically, we can assume that $t_0^2 \cdot x^2 - 4 \cdot t_0 \cdot x + 4$ is also non-negative. We can also multiply non-negative powers of our assumed-to-be-non-negative premises together. Suppose we can write $x^2 - 8 \cdot y + 4 \equiv \lambda_0 + \lambda_1 \cdot (t_0 \cdot x - 2) + \lambda_2 \cdot (x - t_1 \cdot y) + \lambda_3 \cdot (t_0^2 \cdot x^2 - 4 \cdot t_0 \cdot x + 4)$ with each λ_i being a non-negative real as in the previous case. Using a similar argument as before, this will guarantee the validity of the constraint. In practice, our algorithm generates all possible products of the LHS inequalities up to a user-defined degree d. However, for brevity, we only included some of these products in this example.

Simplifying and equating the coefficients on both sides, we get $\lambda_3 \cdot t_0^2 = 1$ (coefficients of x^2), $\lambda_1 \cdot t_0 + \lambda_2 - 4 \cdot \lambda_3 \cdot t_0 = 0$ (coefficients of x), $-\lambda_2 \cdot t_1 = -8$ (coefficients of y), and $\lambda_0 - 2 \cdot \lambda_1 + 4 \cdot \lambda_3 = 4$ (constant terms). One possible solution to this system of quadratic constraints is $t_0 = 1, t_1 = 2$ and $\lambda_0 = 0, \lambda_1 = 0, \lambda_2 = 4, \lambda_3 = 1$. This leads to the following valid entailment, which holds for all values of x and y:

$$\boxed{1} \cdot x - 2 \geq 0 \wedge x - \boxed{2} \cdot y \geq 0 \implies x^2 - 8 \cdot y + 4 \geq 0$$

(c) **Non-linear Premise.** Finally, in the most general case, if both the premise and the conclusion might be non-linear, we use Putinar's Positivstellensatz (Theorem 3) to reduce the entailment to quadratic constraints.

Example 15. *Consider the following constraint:*

$$t_0 \cdot x^3 \geq 0 \implies x^9 + y^2 \geq 0$$

We can not apply Farkas' Lemma for the same reason as the previous example, nor we can apply Handleman's Theorem because no power of $t_0 \cdot x^3$ will generate a polynomial containing y. Intuitively, it is easy to observe that setting $t_0 = 1$ gives us a valid concrete entailment

$$\boxed{1} \cdot x^3 \geq 0 \implies x^9 + y^2 \geq 0$$

because we can write $x^9 + y^2 \equiv x^3 \cdot (x^3)^2 + y^2$. Note that $(x^3)^2$ and y^2 are both sums of squares, therefore they are always non-negative. As x^3 is also assumed to be non-negative in the premise, the conclusion should also be non-negative whenever the premise is non-negative.

When we apply Putinar's Positivstellensatz, we simply multiply each polynomial in the premise with a symbolic sum of squares of sufficiently high degree and sum them together. In other words, the multipliers λ_i are no longer non-negative scalar variables, but instead polynomials that are guaranteed to be sum-of-squares. We can write this guarantee as a system of quadratic constraints, too. Then we equate the coefficients of both sides, just as in the previous cases, to obtain a system of quadratic constraints. Solving this system gives a valid concrete constraint.

6. **Solving the Quadratic Constraints.** The previous step has already reduced the problem to a system quadratic constraints over template variables $\mathbb{T}$ and newly-introduced multipliers λ_i. We pass this system to an external NRA solver. If the NRA solver can find a solution to the system, then we substitute the value of the template variables in $\mathbb{T}$ in our symbolic remaining gas polynomial map B to obtain a concrete parametric bound on the gas usage. Finally, we return the concrete gas polynomial at the start location as the answer.

Example 16. *Table 3.1 shows one solution of the quadratic system for each of T_1 and T_2. Thus, our approach is able to prove that the total gas cost is at most $101 \cdot |\mathtt{proposals}| + 106$ for T_1 and $5 \cdot \mathtt{a} \cdot \mathtt{b} + 11 \cdot \mathtt{a} + 5 \cdot \mathtt{b} + 16$ for T_2. Note that the solutions are not unique, but any solution of the system leads to a valid bound on the gas usage of the PTS.*

Node	Remaining Gas Polynomial
l_0	$101 \cdot \lvert\mathtt{proposals}\rvert + 106$
l_1	$101 \cdot \lvert\mathtt{proposals}\rvert - 101 \cdot \mathtt{p} + 102$
l_2	$101 \cdot \lvert\mathtt{proposals}\rvert - 101 \cdot \mathtt{p} + 101$
l_f	0

Node	Remaining Gas Polynomial
l_0	$5 \cdot \mathtt{a} \cdot \mathtt{b} + 11 \cdot \mathtt{a} + 5 \cdot \mathtt{b} + 16$
l_1	$5 \cdot \mathtt{a} \cdot \mathtt{b} + 11 \cdot \mathtt{a} - 5 \cdot \mathtt{b} \cdot \mathtt{i} + 5 \cdot \mathtt{b} - 11 \cdot \mathtt{i} + 12$
l_2	$5 \cdot \mathtt{a} \cdot \mathtt{b} + 11 \cdot \mathtt{a} - 5 \cdot \mathtt{b} \cdot \mathtt{i} + 5 \cdot \mathtt{b} - 5 \cdot \mathtt{j} - 11 \cdot \mathtt{i} + 9$
l_f	0

Table 3.1: Synthesized solution for T_1 (left) and T_2 (right)

Objective Function. Our algorithm encodes all the requirements on the template variables $t_{i,j}$ first as entailment constraints and then as a system of quadratic constraints. Thus, every solution of this system, when plugged back into the template for B_{l_0} leads to a valid bound on the gas usage of the PTS. In our experiments, we do not use an objective function, but one can generally specify any objective function based on the unknown template variables $t_{i,j}$. This would lead to a Quadratically-Constained Quadratic Programming (QCQP) instance.

Pseudocode. Algorithm 4 provides a pseudocode of our approach.

Algorithm 4: Asparagus

```
Input: A smart contract $A$
Input: Start block $S$, degree bound $d$ for the polynomials
Result: Remaining Gas Polynomial $B_{l_0}$ at the initial location corresponding to $S$
$T \leftarrow$ parsePTS($A$)    ▷ Generate the PTS $T$ for $A$ (Steps 1 and 2) – Not guaranteed to preserve completeness.;
$C \leftarrow$ getCutset($T$);
$I \leftarrow$ TemplateInvariantMap($C, T, d$)    ▷ Step 3;
$B \leftarrow$ TemplateRemainingGasMap($C, T, d$);
$QS \leftarrow$ true;
foreach Basic Path $\Pi$ in $T$ w.r.t. $C$ do
    $(\phi \Rightarrow \psi) \leftarrow$ GetConstraintPair($\Pi, T$)    ▷ Step 4;
    if $\phi, \psi$ are both linear then
        $QS_c \leftarrow$ ReduceWithFarkas($\phi, \psi$)    ▷ Step 5a ;
    end
    else if $\phi$ is linear then
        $QS_c \leftarrow$ ReduceWithHandelman($\phi, \psi, d$)    ▷ Step 5b;
    end
    else
        $QS_c \leftarrow$ ReduceWithPutinar($\phi, \psi, d$)    ▷ Step 5c;
    end
    $QS \leftarrow QS \wedge QS_c$
end
Model $\leftarrow$ NRA_Solver($QS$)    ▷ Step 6;
if Model is UNSAT then
    return Failed
end
return ConcretePolynomial($B_S$, Model)    ▷ The concrete remaining gas polynomial;
```

3.3.3 Soundness, Completeness and Complexity

We end this section by stating that our algorithm is both sound and semi-complete.

Theorem 6. *Given a PTS T over variables with bounded values, a cutset C of T, a polynomial inductive invariant $\mathbb{I}$, and a template for a remaining gas polynomial B_l at each cutpoint $l \in C$, the algorithm of Section 3.3.2 has the following desired properties:*

- Soundness*: Every solution to the system of quadratic constraints in Step 6 leads to valid remaining gas polynomials when plugged back into the template $\{B_l\}_{l \in \mathbb{L}}$. We say that a set of remaining gas polynomials are valid if they satisfy Equation* (3.1)*.*

- Semi-completeness*: For every valid set $\{B_l^*\}_{l\in\mathbb{L}}$ of remaining gas polynomials, if the degree d chosen by the user is large enough, i.e. if polynomials of large enough degree are used in the templates, then there exists a solution of the system of quadratic constraints in Step 6 that when plugged into the templates $\{B_l\}_{l\in\mathbb{L}}$ leads to $\{B_l^*\}_{l\in\mathbb{L}}$.*

Proof. The entailment constraints generated in Step 4 precisely model the requirements of Equation (3.1). Thus, this step is sound. In Step 5, each of the three possible reductions are sound. In case 5a, the RHS is written as a linear combination of the LHS polynomials with non-negative coefficients λ_i. If this is successful, then it is trivial that the entailment holds, since a linear combination of non-negative inequalities with non-negative coefficients is guaranteed to be non-negative. In case 5b, we are first considering products of the non-negative inequalities on the LHS. Such products are of course non-negative. We then write the RHS as a linear combination of these products with non-negative coefficients. Thus, the same argument as in the previous case applies. Finally, case 5c is also similar to case 5a, except that the coefficients which were non-negative real constants are now replaced by sum-of-square polynomials. However, the same argument stands since sum-of-square polynomials are always non-negative. One can essentially repeat the same argument for corner cases involving strict inequalities. See [61, Chapter 7] for details.

For completeness, note that Step 4 is a precise encoding of Equation (3.1) and thus complete. Step 5a is complete due to Farkas' lemma (Theorem 1) which guarantees that whenever the entailment constraint holds, the RHS can be written as a linear combination of the LHS with non-negative coefficients λ_i. Similarly, Step 5b is complete due to Theorem 2 guaranteeing that if the entailment holds, then the RHS can be written as a combination of products of the LHS. Finally, the completeness of Step 5c is due to Theorem 3 in which the validity of the entailment $f_1 \geq 0 \wedge f_2 \geq 0 \ldots \wedge f_r \geq 0 \implies g > 0$ guarantees $g \equiv h_0 + \sum_{i=1}^{r} h_i \cdot f_i$, where each h_i is a sum-of-squares. Of course, the products generated in Step 5b and the sums-of-squares in Step 5c should have a high enough degree, since the theorems do not have a degree bound. This is why we call this semi-completeness, i.e. completeness as long as the degree d chosen for the templates is large enough. Finally, we note that both Theorem 2 and Theorem 3 have a compactness requirement. This holds naturally in our case since the variables in a smart contract are always bounded. Assuming that the largest possible value for a variable x is m, we can add the invariant $-m \leq x \leq m$ to every line of the program. This puts our valuations inside a bounded hypercube. Finally, by the Heine-Borel theorem, a subset of $\mathbb{R}^n$ is compact if and only if it is closed and bounded. Thus, the hypercube is compact. □

Theorem 7. *Given the same inputs as in the previous theorem, the runtime of our* *reduction to a system of quadratic constraints (Steps 4 and 5) is polynomial in the size n of the PTS and depends exponentially on the number of variables and the degree bound d.*

Proof. Step 4 generates one entailment constaint for each transition in the PTS. Thus, its runtime is $O(n)$. For each entailment constraint, we have the following analysis:

- Step 5a (Farkas' lemma) generates at most $|\mathbb{V}|$ fresh variables λ_i and the same number of quadratic equalities.
- Step 5b creates all monomials of degree up to d as in Equation (2.1). The number of such monomials is $\binom{d+|\mathbb{V}|}{d}$. There is at most one quadratic equality generated per monomial.
- Step 5c creates a constant number of sum-of-square polynomials h_i as in Equation (2.2). To generate a template for each such polynomial, we need to first generate all monomials of degree at most $d/2$ as in Theorem 4 and then form the L matrix of Theorem 5 which has $O(\binom{d/2+|\mathbb{V}|}{d/2}^2)$ entries. We will have at most one quadratic equality for each monomial.

The time spent in generating each quadratic equality is bounded by a polynomial in its size. Thus, the total runtime is polynomial in the size n of the PTS but depends exponentially on the number of variables in $\mathbb{V}$ and the degree bound d. □

3.4 Experimental Results

Implementation. We implemented our algorithm of Section 3.3 for Ethereum smart contracts and named our automated tool ***Asparagus***[2]. Our implementation is in Python 3 and can obtain linear and polynomial gas usage upper-bounds for public functions of any given contract written in Solidity. We used Slither [62] for parsing, `solc` 0.4.25 for compilation [39], Z3 [63] and Mathsat [64] to solve the final systems of quadratic constraints, and EthIR [58] to generate RBR intermediate representation. The implementation is open-source and dedicated to the public domain with a CC0 (no rights reserved) license. It is available as an archived artifact attached to this article.

Benchmarks and Experimental Setting. We compare Asparagus with GASTAP [56], which is the only previous tool able to generate parametric bounds, as well as the `solc` compiler's built-in static analyzer [39]. As benchmarks, we took the dataset provided by GASTAP. Since both GASTAP and Asparagus use EthIR in their pipeline, we excluded any benchmark on which EthIR failed to generate RBR. Surprisingly, such

[2] Automated synthesis of parametric gas upper-bounds for smart contracts

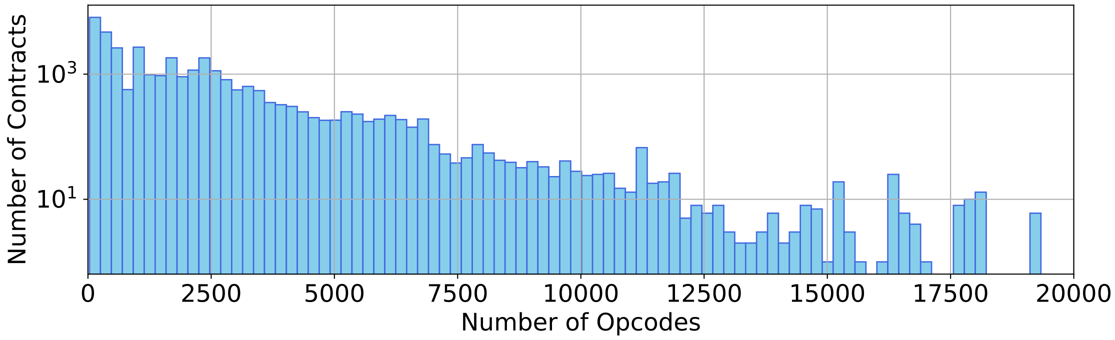


Figure 3.6: Distribution of Contract Lengths among the Benchmarks.

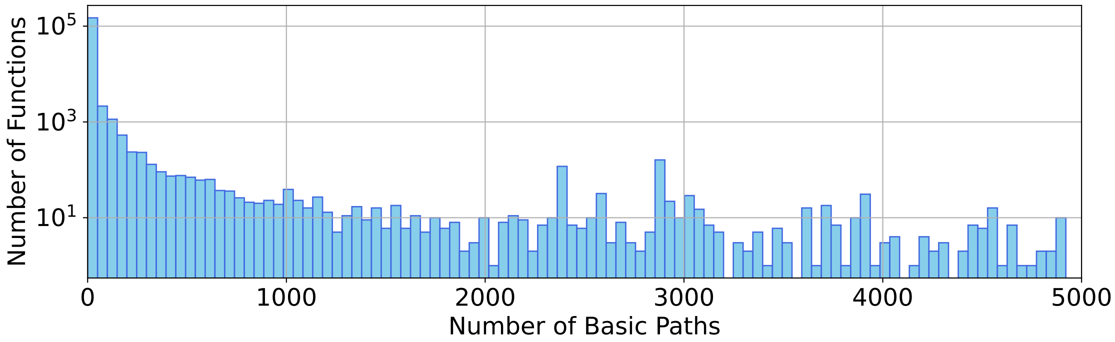


Figure 3.7: Number of Basic Paths in the Analyzed Functions.

benchmarks existed and we could not replicate and verify the experimental claims of GASTAP [56]. We suspect this might have to do with EthIR updates that were aimed at supporting newer versions of Solidity, and have likely reduced its applicability significantly in comparison to what GASTAP reports. Irrespective of the cause, our results have significant mismatches with GASTAP's claims. In total, we report experimental results on 24,188 contracts, containing 156,735 functions. Figure 3.6 shows the distribution of contract lengths in our benchmark suite and Figure 3.7 reports the number of basic paths in the analyzed functions.

Machine and Runtimes. All computations were performed on an Intel Xeon Gold 5115 CPU (2.40GHz, using 16 cores) running Ubuntu 20.04 and 64 GB of RAM. Our average runtime on each function was 5.18 seconds, leading to a total runtime of almost 225.8 hours for 156,735 functions. This demonstrates that our approach is scalable and easily applicable to real-world contracts.

Comparing Bounds. In cases where the bounds are constants, we can simply compare them. We say a parametric bound A is tighter than another bound B if for any initial state σ that satisfies the precondition, i.e. the invariant at the beginning point of the function, we have $A(\sigma) \leq B(\sigma)$ and additionally, $A(\sigma)$ is strictly less in

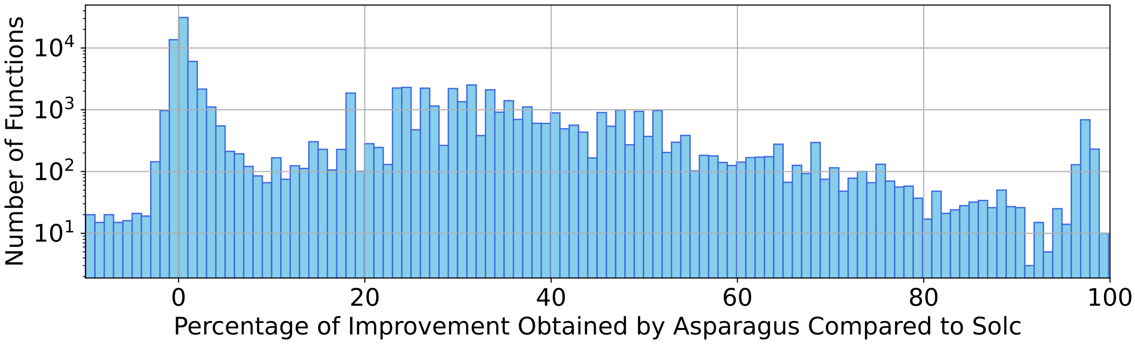


Figure 3.8: Comparsion of the Constant Bounds obtained by Asparagus and `solc`.

at least one such state. To compare bounds, we again use the theorems of Farkas, Handelman and Putinar to (i) check whether $\mathbb{I}(l_0) \Rightarrow A \leq B$ holds, and (ii) synthesize a particular state that satisfies $\mathbb{I}(l_0) \wedge A < B$. If this fails, we report that the bounds are incomparable.

Comparison with `solc`. We first compared Asparagus and `solc` on the entire dataset of 156,735 functions. Our tool successfully synthesized gas usage upper-bounds for **80.56%** of the instances compared to a success rate of **64.15%** for `solc`. Note that `solc` can only synthesize constant bounds and hence fails on all benchmarks that require parametric bounds. Table 3.2 provides a comparison of success rates and coverage of the two tools. In cases where both approaches could successfully find a bound, our bound was better in 81,425 cases. Figure 3.8 provides a comparison of the *constant* bounds obtained by the two tools.

	Asparagus		`solc`	
	#	%	#	%
Constant Gas Bound	124,987	–	100,550	–
Parametric Gas Bound	1,282	–	0	–
Solved	126,269	80.56%	100,550	64.15%
Failed	30,466	19.43%	56,185	35.84%

Table 3.2: Number and percentage of benchmark functions solved by Asparagus and `solc` on 156,735 functions.

Comparison with GASTAP. We also compared Asparagus with GASTAP. Unfortunately, we did not have the desired degree of access to GASTAP in order to perform a complete comparison due to the following reasons:

- GASTAP did not publish its gas upper-bound results for the dataset.
- GASTAP is a closed-source and proprietary piece of software that can only be

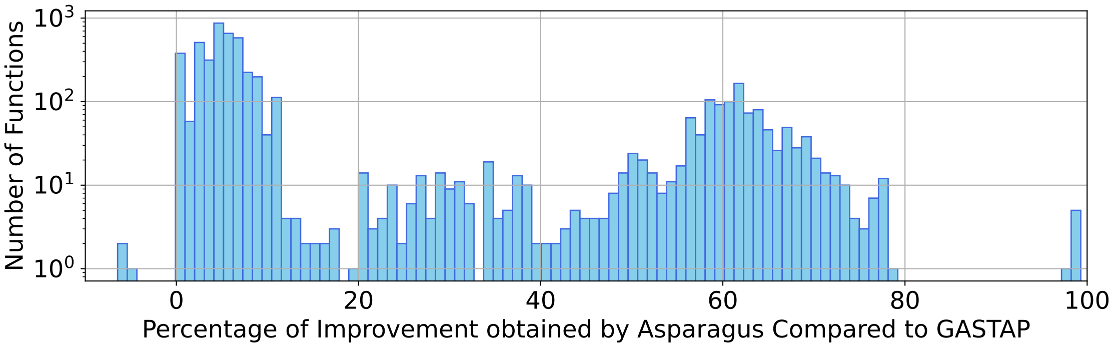


Figure 3.9: Comparsion of the Constant Bounds obtained by Asparagus and GASTAP.

accessed by an online web interface[3] with no API. To perform the comparison, we had to reverse-engineer their http requests and even then, most of our requests failed, even after sending them a dozen times each. We manually checked some of the failed requests on their online interface, and confirmed that the failures are due to GASTAP.

Due to these limitations, we could successfully run GASTAP on only 1,682 contracts, consisting of 10,186 functions. We thus report a comparison on these 10,186 functions. On these benchmarks, Asparagus successfully synthesized a gas upper-bound for **86.94%** of the functions compared to **58.62%** for GASTAP and **61.61%** for `solc`, demonstrating Asparagus' effectiveness. Table 3.3 provides a more detailed comparison and Figure 3.9 compares the *constant* bounds obtained by the two tools.

	Asparagus		GASTAP		`solc`	
	#	%	#	%	#	%
Constant Gas Bound	8,404	–	5,846	–	6,276	–
Parametric Gas Bound	452	–	126	–	0	–
Solved	8,856	86.94%	5,972	58.62%	6,276	61.61%
Failed	1,330	13.05%	4,214	41.37%	3,910	38.38%

Table 3.3: Number and percentage of benchmark functions solved by Asparagus, GASTAP and `solc` on 10,186 functions.

Notably, we compared the bounds in cases where both Asparagus and GASTAP were successful (5,916 functions). Asparagus' synthesized upper-bounds were strictly tighter in 5,789 cases, representing **97.85%**. GASTAP provided tighter bounds for only 45 functions, representing **0.76%** of the instances. In other cases, the bounds were either equal or incomparable.

Non-linear Bounds. The vast majority of real-world contracts have constant or linear gas usage. In total, only 653 of our benchmarks required a polynomial bound

[3] `costa.fdi.ucm.es/gastap`

and an application of the theorems of Handelman and Putinar. The rest of the instances were solved by Farkas' Lemma. Our implementation supports polynomials of any degree. We have experimented with nested loops requiring up to hexic bounds and successfully synthesized results on hand-crafted examples. However, none of the real-world benchmarks in our dataset needed bounds of degree higher than 2.

Summary. Asparagus is able to handle significantly more real-world benchmarks than both `solc` and GASTAP. Since GASTAP was the only previous tool that could generate parametric bounds, Asparagus is now the most widely-applicable tool for such bounds to the best of our knowledge. Moreover, we consistently generate tighter bounds than both GASTAP and `solc`.

Contracts beyond Asparagus' Capability. As mentioned above, Asparagus successfully handled 80.56% of the cases in our benchmark suite (See Table 3.3). This is considerably more than `solc` and GASTAP. The remaining benchmarks were beyond the capability of our tool, due to non-polynomial operations and bounds. Our PTS formalism can handle polynomial assignments, guards, templates and bounds. Thus, we replace non-polynomial behavior, such as the integer mod operation, with non-determinism, cf. Steps 1 and 2 of the algorithm. This can lead to failures in synthesizing bounds. Moreover, in some cases, no polynomial bound exists at all. Our completeness result guarantees that our approach will succeed if (i) the contract can be faithfully modeled as a PTS with polynomial assignments, invariants and templates, and (ii) the PTS has a polynomial gas usage bound. In practice, all real-world contracts on which Asparagus failed were violating condition (i) and had non-polynomial inherently-integer operations such as gcd or mod.

Chapter 4

Gas Optimization for Smart Contract Compilers

This chapter originally appeard in the following publication:

- T. Barakbayeva, S. Farokhnia, A.K. Goharshady, P. Li, Z. Lin. *Improved Gas Optimization of Smart Contracts.* International Conference on Fundamentals of Software Engineering, FSEN 2025.

4.1 Introduction

Background. Smart contracts are often written in high-level programming languages such as Solidity and then compiled to bytecode before being deployed on the blockchain. Thus, a natural compiler optimization problem arising in this context is to produce efficient bytecode that minimizes the total gas usage. A leading approach in this direction is superoptimization, which considers every basic block of the smart contract separately and tries to rewrite it as an equivalent block that uses as little gas as possible. The current state-of-the-art tool is `syrup 2.0`, which encodes gas superoptimization as `Max-SMT` and then relies on SMT-solvers to synthesize an equivalent contract with optimized gas usage.

Our Contributions. In this chapter, we make two observations: First, the performance of `Max-SMT` declines significantly as block sizes increase. Thus, although `syrup` is able to find an optimal rewriting for a small block with a dozen bytecode operations, its output on blocks with hundreds or thousands of operations, when given any realistic timeout, is far from optimal. Second, optimizations that can be applied to basic blocks are often local and compositional, i.e. they rewrite several small and disjoint parts of the block. Such locality is lost to `Max-SMT` solvers, mainly because it is unpredictable and there are no clear ways on how one should cut blocks of bytecode into smaller sub-blocks. To ameliorate these issues, we present a simple dynamic programming algorithm that tries every possible division of a block into sub-blocks, recursively calling `syrup` as a black box on each sub-block. Surprisingly, this simple idea leads to highly significant improvements in the gas usage, more than doubling the savings obtained by `syrup`, and reducing the gas usage of real-world smart contracts by 11.23 percent.

4.2 Gas Optimization Problem

Gas Optimization. Analyzing and reducing gas costs are central research problems in the blockchain community. Out-of-gas errors are the source of many vulnerabilities and thus there are several tools focused on finding upper-bounds on the gas usage of smart contracts [65, 66, 67]. In 2023, on Ethereum alone, gas costs were more than 4 billion USD [68]. Given this high cost, and the fact that gas usage is defined for low-level bytecode operations whereas programmers write their contracts in high-level languages such as Solidity, it is crucial that the compiler optimizes for gas usage. Indeed, the standard Solidity compiler `solc` has a flag `-optimize` which enables heuristics for optimizing the gas usage of the resulting bytecode. The Solidity language documentation [69] also talks about *gas-hungry* patterns and instructs programmers to avoid them in their code. There are many works on layer-two protocols which aim to minimize the amount of on-chain computation, i.e. gas-consuming calls to smart contracts, by moving most of the protocol off-chain [70] or delaying and avoiding the execution as far as possible [71].

Superoptimization. The current state-of-the-art in gas optimization by compilers is the work [72] which provides a tool called `syrup 2.0` that supports both Solidity source code and EVM bytecode as its input and outputs gas-optimized bytecode. Their approach is based on the concept of superoptimization [73]. Basically, the idea is to break the bytecode program down into its basic blocks, i.e. maximal straight-line subprograms that do not contain branching or jumps. Then, each basic block B is optimized separately by exhaustively trying all possible rewritings B' that are equivalent to B and taking the one with the smallest gas usage. Superoptimization is a well-known technique that has been implemented in mainstream tools and compilers such as LLVM [74, 75] usually with the goal of reducing runtime or memory usage. However, exhaustive search is far from scalable and can only be applied to toy programs with tiny basic blocks. For example, a `C++` basic block containing the single operation `x*=2` can easily be rewritten as `x«1`, reducing its execution time, but as the size of the block grows there will be a combinatorial explosion in the number of possible rewritings. Instead, [72] encodes the problem as `Max-SMT` and passes it to modern SMT-solvers. This encoding, and the rewrite rules for obtaining equivalent basic blocks, are far from trivial. Indeed, [72] provides several different encodings and experimentally finds the best combinations. The reduction to `Max-SMT` is the key to `syrup`'s scalability and enables huge savings in real-world gas costs of Ethereum smart contracts.

Example 17. *Consider a simple basic block in Solidity that performs the operation $y = x^\wedge x$. Here, x and y are integers and $^\wedge$ is the bitwise exclusive or operation. Our goal is to compile this basic block to EVM bytecode. A naive compiler that applies no optimization, such as* `solc` *with all optimizations turned off, would provide the bytecode in Figure 4.1 (left). In EVM bytecode,* `PUSH` *adds a new item to the top of the stack,* `POP` *removes the top item,* `SLOAD` *loads a word from storage,* `DUP` *duplicates an item on the stack and* `SWAP` *swaps items in the stack. As expected,* `XOR` *performs the bitwise exclusive or operation. Each of these operations has a gas cost, which is fixed by the Ethereum Yellowpaper [16, Appendix H]. In this case, the naive compilation will lead to a total gas cost of 1,420 for this basic block. In contrast, a compiler that realizes $x^\wedge x = 0$ and thus rewrites $y = x^\wedge x$ as $y = 0$ would not even need to perform the* `XOR` *operation. This leads to a much more gas-efficient bytecode such as the one in Figure 4.1 (center). This is the output of* `syrup` *and uses only 720 units of gas,*

```
PUSH1 0
SLOAD
PUSH1 0
SLOAD
XOR
PUSH1 1
DUP2
SWAP1
```

```
PUSH1 0
SLOAD
PUSH1 1
PUSH1 0
SWAP2
POP
SWAP1
SWAP1
```

```
PUSH1 0
SLOAD
PUSH1 1
```

Figure 4.1: Three compilations of $y = x^\wedge x$ to EVM bytecode: naive (left), optimized (center), and further optimized (right).

almost halving the execution cost. Finally, it is possible to further optimize even this bytecode, obtaining the basic block in Figure 4.1 (right), which uses 706 units of gas. Note that all three basic blocks of Figure 4.1 are semantically equivalent and differ only in their gas usage.

Our Fucus. In this chapter, we use `syrup` as a black box and design a simple and elegant dynamic programming algorithm for optimizing the gas usage of Ethereum smart contracts. Our algorithm is also a flavor of superoptimization, i.e. it optimizes each basic block separately. We report significant improvements in the gas usage of the resulting smart contracts, not only in comparison with the unoptimized version, but also against `syrup` itself.

4.3 Our Algorithm

In this section, we present our simple dynamic programming algorithm. Our approach is based on the two intuitive observations below.

Observation 1: Scalability. We observe that `syrup` works really well on small basic blocks and often finds the optimal rewriting. However, this is no longer the case when the basic block increases in size. For basic blocks with more than a hundred bytecode operations, `syrup` rarely finds the optimal rewriting and often produces extremely suboptimal results, even when given generous time limits of several hours. This is not surprising since the problem is reduced to `Max-SMT` and SMT-solvers are simply not scalable enough to handle large basic blocks.

Observation 2: Locality and Compositionality. Our second observation is that gas-optimizing changes to basic blocks are often local and compositional. For example, a block with thousands of operations will probably be optimizable by hundreds of different local rewritings which are quite independent of each other.

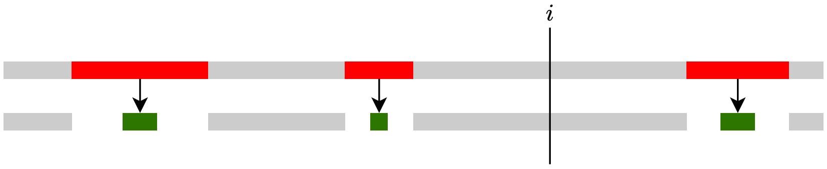


Figure 4.2: A basic block B (top) in which some portions (red) can be optimized to use less gas (green).

More formally, let $\mathsf{B} = \langle \mathsf{op}_1, \mathsf{op}_2, \dots, \mathsf{op}_n \rangle$ be a basic block consisting of n EVM operations, $g(\mathsf{B}) = \sum_{i=1}^{n} g(\mathsf{op}_i)$ be its gas usage, and $B^* = \mathsf{Optimized}(\mathsf{B})$ be the optimal rewriting of B, i.e. the equivalent basic block that uses minimal gas. Additionally, let $\mathsf{B}[i \dots j]$ be the sub-block of B from op_i to op_j. We conjecture that in almost all cases, there is an index i such that

$$g(\mathsf{Optimized}(\mathsf{B})) = g(\mathsf{Optimized}(\mathsf{B}[1 \dots i])) + g(\mathsf{Optimized}(\mathsf{B}[i+1 \dots n])).$$

In other words, B can be divided in two parts and each part can be (recursively) optimized separately. This is shown in Figure 4.2. This intuition leads to two challenges: (i) how to identify when this kind of compositionality is present, and (ii) how to find the correct index i for dividing B in two parts. Our algorithm sidesteps both of these difficulties by simply brute-forcing all possibilities.

Our Algorithm. We use `syrup` as a black box in our algorithm. Let $\mathsf{B} = \langle \mathsf{op}_1, \mathsf{op}_2, \dots, \mathsf{op}_n \rangle$ be a basic block and $\mathsf{syr}(\mathsf{B})$ be the gas-optimized block obtained by applying `syrup` to B. Instead of applying `syrup` directly to B, we can first divide B in two parts $\mathsf{B}[1 \dots i]$ and $\mathsf{B}[i+1 \dots n]$ and then optimize each part separately. We simply try this for all possible i, considering further sub-divisions recursively. Formally, let $\mathsf{leastGas}(\mathsf{B})$ be the minimum amount of gas usage that we can obtain by rewriting B to an equivalent basic block. We have:

$$\mathsf{leastGas}(\mathsf{B}) = \min \left\{ g(\mathsf{syr}(\mathsf{B})), \min_{i=1}^{n-1} \mathsf{leastGas}(\mathsf{B}[1 \dots i]) + \mathsf{leastGas}(\mathsf{B}[i+1 \dots n]) \right\}.$$

This formula leads itself to dynamic programming and tracing the dynamic programming steps can also help us find an equivalent block B^* with $g(\mathsf{B}^*) = \mathsf{leastGas}(B)$. More specifically, for every sub-block $\mathsf{B}[i \dots j]$, we can find the best optimization. This is shown in Algorithm 5. Note that our algorithm does not guarantee that the resulting block will be globally optimal, but only that it will use no more gas than `syrup`'s output. As we will see in Section 4.4, the improvement is quite substantial in practice. Finally, we note that our algorithm can easily be parallelized at lines 5 and 9.

Algorithm 5: Our Algorithm: Dynamic Programming using `syrup` as a Blackbox

```
Input: A basic block $\mathsf{B} = \langle \mathsf{op}_1, \dots, \mathsf{op}_n \rangle$ of $n$ EVM bytecode operations
Result: A gas-optimized block $\mathsf{B}^*$ which is equivalent to $\mathsf{B}$
int $\mathsf{leastGas}[n][n]$;
// $\mathsf{leastGas}[i][j]$ holds $\mathsf{leastGas}(\mathsf{B}[i \dots j])$
block $\mathsf{bestBlock}[n][n]$;
// $\mathsf{bestBlock}[i][j]$ holds the best rewriting for $\mathsf{B}[i \dots j]$
foreach $i \leq j$ do
    $\mathsf{bestBlock}[i][j] \leftarrow \mathsf{syr}(\mathsf{B}[i \dots j])$;
    // Start with syrup's output as the base case
    $\mathsf{leastGas}[i][j] \leftarrow g(\mathsf{bestBlock}[i][j])$;
end
for $1 \leq l \leq n$ do
    // $l$ is the length of our sub-block
    for $1 \leq a \leq n - l + 1$ do
        // $a$ is the starting index of our sub-block
        $b \leftarrow l + a - 1$;
        // $b$ is the end index of our sub-block
        for $a \leq i \leq b$ do
            // Try breaking the sub-block $\mathsf{B}[a \dots b]$ at index $i$
            if $\mathit{leastGas}[a][i] + \mathit{leastGas}[i+1][b] < \mathit{leastGas}[a][b]$ then
                $\mathsf{leastGas}[a][b] = \mathsf{leastGas}[a][i] + \mathsf{leastGas}[i+1][b]$;
                $\mathsf{bestBlock}[a][b] = \mathsf{bestBlock}[a][i] \cdot \mathsf{bestBlock}[i+1][b]$;
                // The operator $\cdot$ represents concatenation of basic blocks
            end
        end
    end
end
return $B^* = \mathit{bestBlock}[1][n]$;
```

4.4 Experimental Results

In this section, we provide an experimental comparison between our algorithm and `syrup` over real-world Ethereum smart contracts. We implemented our algorithm in Python and integrated it with `syrup 2.0` [72]. All results were obtained on an Intel Xeon Gold 5317 machine (3.0 GHz, 12 cores, 18M cache) with 128 GB of RAM running Ubuntu 20.04.6 LTS. We performed gas optimization experiments on the benchmark smart contracts from [72], which contain some of the most widely-used real-world contracts on the Ethereum blockchain.

Standard Benchmarks. In our first experiment, we considered the 148 smart contracts of the benchmark suite of [72]. According to [72], this suite contains a random sampling of the most commonly-called smart contracts on the Ethereum blockchain. 128 of these contracts are provided in the EVM bytecode format and match the exact versions deployed on the blockchain. These deployed bytecodes were most likely obtained by the developers compiling high-level Solidity code using a version of `solc` that was available at the time of their deployment. The other 20 are provided as Solidity source code. We compiled them to bytecode using `solc`. Overall, this benchmark set consists of 22,136 basic blocks.

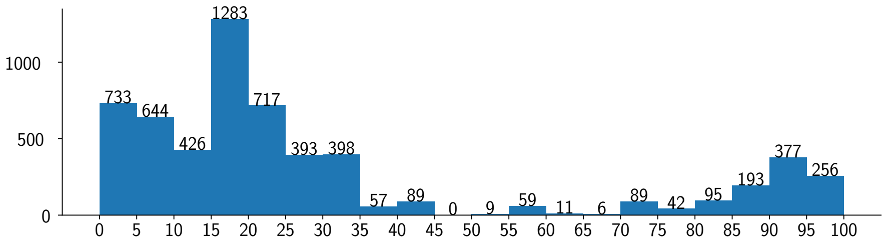


Figure 4.3: Histogram of gas improvements obtained by our approach over the unoptimized smart contracts. The x axis is the percentage of improvement (bin size = 5%) and the y axis is number of basic blocks.

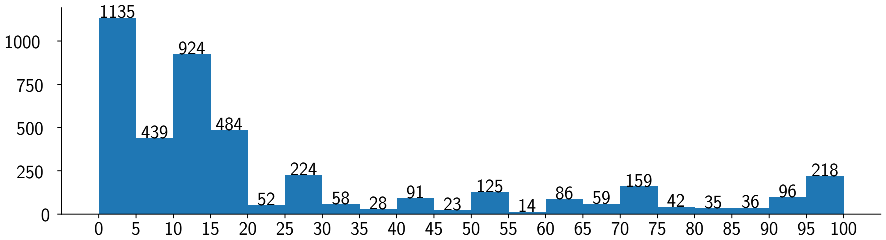


Figure 4.4: Histogram of gas improvements obtained by our approach over `syrup 2.0`. The x axis is the percentage of improvement (bin size = 5%) and the y axis is number of basic blocks.

Gas Optimization Results. The total gas usage of unoptimized contracts over all basic blocks was 1,972,141 units of gas. This was reduced to 1,889,918 units of gas when applying `syrup 2.0`. It was further reduced to 1,750,576 units by our dynamic programming algorithm. Thus, `syrup` provides an overall gas saving of 4.17 percent over the baseline, whereas our approach obtains a saving of 11.23 percent (7.37 percent over `syrup`). More specifically, `syrup` succeeded in improving the gas usage in 4,678 basic blocks, with an average improvement of 21.40 percent over these 4,678 blocks. Our approach improved the gas usage in 5,877 basic blocks, with an average improvement of 30.52 percent over these blocks. In comparison to `syrup`, our approach improved the gas usage in 4,328 basic blocks with the average improvement being 24.82 percent. Figures 4.3–4.4 visualize the data as histograms.

Real-world Blockchain Benchmarks. In a second experiment, we considered real-world Ethereum contracts that were called in blocks number 21190000 to 21200000 of the Ethereum blockchain. These blocks correspond to the timeframe from Nov-15-2024 02:06:11 UTC to Nov-16-2024 11:36:47 UTC. We considered the top 1000 most-commonly called contracts in these blocks. Of these, `syrup` could handle 656 contracts. Some contracts use a newer version of the EVM bytecode format and thus neither `syrup` nor our approach is applicable to them. Over these contracts, our approach saved a total of 124,163,227 units of gas, corresponding to 2.29 ETH.

Chapter 5

Maximizing Miners' Revenues

This chapter originally appeard in the following publications:

- T. Barakbayeva, S. Farokhnia, A.K. Goharshady, S. Novozhilov. *Boosting Gas Revenues of Ethereum Miners.* IEEE/ACM International Conference on Software Engineering, ICSE 2026.
-
- T. Barakbayeva, S. Farokhnia, A.K. Goharshady, M. Gufler, S. Novozhilov. *Pixiu: Optimal Block Production Revenues on Cardano.* IEEE International Conference on Blockchain, Blockchain 2024.

5.1 Introduction

Background. In modern cryptocurrencies, a blockchain is a linked-list of blocks, which in turn contain a sequence of transactions. Anyone on the network can create and broadcast a transaction, but the transaction is only considered finalized when it is added to the blockchain. The blockchain is subject to consensus, i.e. all nodes on the network will eventually agree on its contents. The consensus mechanism varies by cryptocurrency. For example, Bitcoin uses proof of work, whereas Ethereum runs on proof of stake. Irrespective of this, all modern cryptocurrencies heavily rely on miners. These are nodes who actively partake in running the consensus mechanism and extending the blockchain by adding new blocks. To incentivize the miners to add new transactions to the blockchain, each transaction includes a fee, which is paid to the miner who includes it in her block. This setting creates a natural optimization problem from the point-of-view of miners: Given a set of new transactions which are not yet added to the blockchain, how can we form an optimal block that maximizes the fees and thus the miner's revenue?

In this chapter, we address this optimization problem for two mainstream cryptocurrencies:

1. **Cardano:** A blockchain protocol based on proof-of-stake and an extended UTXO model which also supports arbitrary smart contracts. Its primary currency, Ada, is currently one of the global top ten cryptocurrencies with a market cap of more than 16 billion USD. In Cardano, new blocks are produced by stake pools. Any holder of Ada can delegate their stake to a pool. The underlying proof-of-stake consensus protocol is Ouroboros Praos, which divides time into a number of epochs and each epoch into a number of slots, each corresponding to one second. In each slot, leaders are randomly selected to produce and add new blocks to the blockchain, with their selection probability being proportional to their stake. Each block can contain a sequence of transactions and block production is rewarded in two ways: (i) transaction fees and (ii) monetary expansion. The producers have no control over (ii), but can optimize (i) by choosing which transactions to include in their blocks. Thus, they are incentivized to maximize the total transaction fees.

2. **Ethereum:** Currently world's largest programmable blockchain by market cap. On Ethereum, the fees depend on the execution costs of the transaction. Given that Ethereum supports arbitrary smart contracts in a Turing-complete language, the fee paid by a transaction can depend on the context in which it is executed, which is determined by the set of transactions that precede it on

the blockchain. Thus, the problem becomes considerably more challenging. An Ethereum miner who wishes to maximize her revenue has to choose not only a subset of transactions to include in her block, but also their exact order. Moreover, each permutation of the same transactions might change the amount of fee paid by each transaction. This causes a combinatorial explosion.

Our Contributions. This chapter provides algorithms and empirical results for optimal block production on Cardano and Ethereum:

1. **Cardano.** In Section 5.2, we formalized the block production problem on Cardano. Then, in Section 5.3, we show that by exploiting the sparsity of interrelations between transactions, i.e. the small treedepth of dependency-conflict graphs, it is possible to obtain a polynomial-time algorithm that outputs optimal blocks. In Section 5.4, we implemented our algorithm in a free and open-source tool called Pixiu. Using Pixiu, we provide extensive experimental results over real-world transaction data on the Cardano blockchain, demonstrating that our approach increases the block producers' revenue by almost 1,357.82 USD/day (= 495,604.3 USD/year).

2. **Ethereum.** Sections 5.5 and 5.6 study the block production problem for Ethereum and introduce a randomized framework for increasing miners' transaction-fee revenues. The framework combines testing, decision trees, integer linear programming, and localization techniques to ameliorate the combinatorial explosion. Finally, Section 5.7 reports significant empirical gains: our method outperforms real-world Ethereum miners by an average of 73.45% per block, corresponding to roughly 63 million USD per annum.

5.2 Cardano eUTXO Model

In general, block producers in Cardano have no control over the monetary expansion component of the protocol rewards. However, they can choose which transactions to include in their blocks, thereby exercising some control over transaction fees. Block producers are collectively incentivized to maximize the total amount of transaction fees in their epoch and thus in their block. See Section 2.3 for a comprehensive overview of Cardano's architecture. In this section, we focus on solving the problem of forming an optimal block on the Cardano blockchain. Specifically, given a set of unmined transactions, i.e. transactions that are not yet added to the blockchain, our goal is to create a valid Cardano block with the maximum possible amount of transaction fees, thus maximizing the total revenue of block producers.

Optimal Block Production. Given a block size limit $k \in \mathbb{N}$, a finite set TX of n unmined Cardano transactions in which every transaction $t \in \mathsf{TX}$ has a set of inputs, a set of outputs, a fee $\phi(t) \in [0, \infty)$ and a size $\sigma(t) \in \mathbb{N}$, find a subset $\mathsf{TX}^* \subseteq \mathsf{TX}$ of transactions such that:

- TX^* satisfies all the dependency and conflict requirements as above;
- $\sum_{t \in \mathsf{TX}^*} \sigma(t) \leq k$, i.e. all the chosen transactions fit into the block size limit; and
- $\sum_{t \in \mathsf{TX}^*} \phi(t)$ is maximized, i.e. the block consisting of the transactions in TX^* yields the maximum possible total transaction fee.

In practice, we always have $k = 90112$.

Dependency-Conflict Graphs (DCGs) [22]. Given an instance of the optimal block production problem above, we create a graph $G = (\mathsf{TX}, E_C \cup E_D)$ in which there is a vertex corresponding to every transaction in TX and an undirected edge $\{t_1, t_2\} \in E_C$ whenever the transactions t_1 and t_2 are in conflict, as well as a directed edge $(t_1, t_2) \in E_D$ when transaction t_2 depends on transaction t_1. See Figure 5.1.

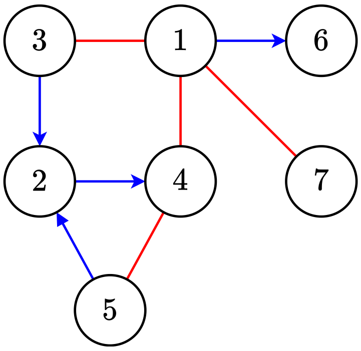


Figure 5.1: An example dependency-conflict graph. Dependency edges are shown in blue and conflict edges in red.

The work [22] (IEEE Blockchain 2022) considered DCGs in the context of Bitcoin mining. It showed that if the DCGs are sparse and path-like, then the optimal block production problem for Bitcoin is efficiently solvable. The approach in [22] depends on the concepts of pathwidth and path decompositions. Intuitively, it first computes a decomposition of the DCG into a path and then uses a dynamic programming algorithm to obtain the optimal block. Unfortunately, such an approach is too slow and not applicable to our use-case in Cardano. This is because in Cardano, a new block has to be produced every second, whereas computing the decomposition notion used in [22] takes minutes. This was not a problem in Bitcoin, where a new block is added every 10 minutes, but is not scalable enough for Cardano. Therefore, in this problem, we consider a stronger notion of decomposition, namely the treedepth decomposition, and provide an algorithm based on treedepth to solve the optimal block production problem in Cardano. As we will see in Section 5.4, our algorithm is highly scalable in practice and produces optimal blocks in less than a second, hence enabling its application in Cardano.

5.2.1 Treedepth

Treedepth Decompositions [46, 47, 48]. For a graph $G = (V, E)$, a *treedepth decomposition* is a rooted tree $T = (V, E_T)$ on the same set of vertices as G that satisfies the following requirement:

- For every undirected edge $\{u, v\} \in E$ or directed edge $(u, v) \in E$ of the original graph, either u is an ancestor of v in T or v is an ancestor of u in T.

We say that a treedepth decomposition T is optimal if it has the smallest possible depth among all decompositions of G. This smallest depth is called the *treedepth* of G. Intuitively, treedepth is a measure of graph sparsity that captures how much a graph resembles a shallow tree. Throughout this problem, we always consider the treedepth d of a dependency-conflict graph G. See Figure 5.2.

Computing Treedepth. For any small fixed d, there is an algorithm that decides whether an input graph has treedepth d in linear time and if so, outputs an optimal treedepth decomposition [49]. There are also well-optimized tools and libraries for computing treedepth decompositions [50]. As we will see in Section 5.4, DCGs in Cardano have small treedepth. Thus, in the remainder of this chapter we assume, without loss of generality, that we have access to an optimal treedepth decomposition of every DCG. In practice, we use [50] to find such decompositions.

Vertex Numbering. Recall that the vertices in our DCGs are unmined Cardano transactions. We number the vertices by a pre-order (left-to-right) traversal of our treedepth decomposition. See Figure 5.2 as an example. This is also without loss of generality, but allows us to present our algorithm more concisely.

Ancestor Sets. Suppose that our treedepth decomposition T is rooted at vertex 1. For every vertex $v \in V$, we denote by A_v the set of ancestors of v, i.e. the set of vertices that are on the path from the root 1 to v in T. For two vertices $u, v \in V$, we define $A_{u,v} := A_u \cap A_v$ as the set of their common ancestors, i.e. vertices that are ancestors of both u and v. Since we numbered the vertices in pre-order, we have $A_{i,i-1} = A_i \setminus \{i\}$ for every vertex i.

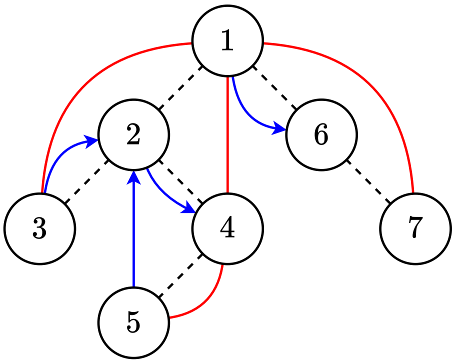


Figure 5.2: A treedepth decomposition of the DCG graph of Figure 5.1. The edges of the decomposition are dashed. Every edge of the original graph (shown in red and blue) goes between a vertex and one of its ancestors. The vertices are numbered in pre-order. This decomposition has a depth of 3, since the path from the root 1 to the farthest leaf 5 has three edges.

5.3 Block Optimization via Treedepth

In this section, we present our algorithm for finding an optimal Cardano block that maximizes the total transaction fee revenue of the block producers. Our algorithm is a dynamic programming approach based on the treedepth decomposition of the conflict-dependency graph of our unmined transactions.

Input. Suppose that we are given an optimal block production instance, consisting of the block size limit $k \in \mathbb{N}$ and a set TX of n unmined Cardano transactions as input. Each transaction $t \in \mathsf{TX}$ has a fee of $\phi(t)$ and a size of $\sigma(t)$. Additionally, we have the dependency-conflict graph $G = (\mathsf{TX}, E_C \cup E_D)$ and a treedepth decomposition $T = (\mathsf{TX}, E_T)$ of G with depth d. Our goal is to solve the optimal block production problem.

Canonical Subgraphs. We define n canonical subgraphs of our DCG G. The i-th canonical subgraph G_i consists of the first i transactions, as well as any conflicts and dependencies between them. Formally, we let

$$V_i = \{1, 2, \dots, i\}$$

and

$$G_i = G[V_i].$$

Recall that the transactions are numbered by a pre-order traversal of T. We consider subproblems on each $G_i = (V_i, E_i)$ and show how to combine the results on these subproblems to find an optimal block for the entire DCG G. See Figure 5.3.

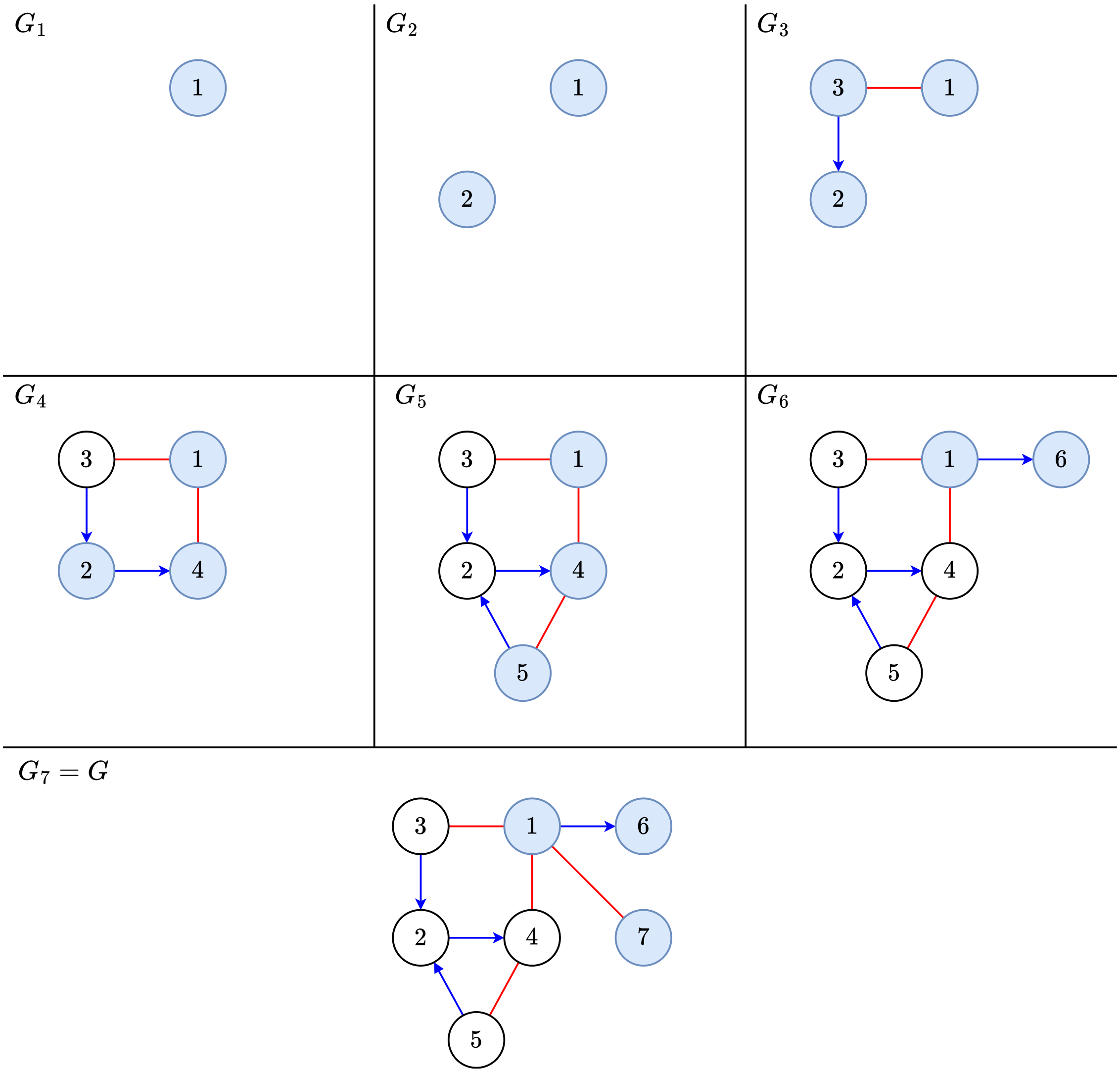


Figure 5.3: Canonical subgraphs of the graph G of Figure 5.1 based on the treedepth decomposition of Figure 5.2. In each G_i, the set A_i of ancestors of vertex i is shown in blue.

Dynamic Programming Table. For every $1 \leq i \leq n$, every partial capacity $0 \leq c \leq k$ and every subset $S \subseteq A_i$, we define a subproblem and a dynamic programming variable as follows:

$$\begin{aligned}\mathrm{dp}[i, S, c] := \quad & \text{The maximum total fees } \Sigma_{t \in \mathsf{TX}^*} \phi(t) \\ & \text{of a set } \mathsf{TX}^* \subseteq V_i \text{ of transactions in } G_i \\ & \text{such that } \mathsf{TX}^* \cap A_i = S \\ & \text{and } \Sigma_{t \in \mathsf{TX}^*} \sigma(t) \leq c \\ & \text{and } \mathsf{TX}^* \text{ satisfies dependencies/conflicts in } E_i.\end{aligned}$$

Intuitively, we are considering subproblems in which we have only the first i vertices/-transactions and their dependencies/conflicts, but we also consider the case where our capacity is $c \leq k$, i.e. part of the block is already filled and we only have c bytes of free space remaining. Finally, the set S tells us exactly which ancestors of transaction i should be taken into the solution.

Computing Values for G_1. The subgraph G_1 consists only of the root vertex 1 and has no edges. Thus, we have

$$\mathrm{dp}[1, \emptyset, c] = 0,$$

and

$$\mathrm{dp}[1, \{1\}, c] = \begin{cases} \phi(1) & \sigma(1) \leq c \\ -\infty & \sigma(1) > c \end{cases}.$$

We use $-\infty$ to show an impossible situation, i.e. when no possible set TX^* satisfying the requirements can be found.

Computing Values for Other G_i's. Suppose $i > 1$ and we intend to compute $\mathrm{dp}[i, S, c]$. Moreover, assume that the $\mathrm{dp}[j, \cdot, \cdot]$ values are already computed for all $j < i$.

- We first check if S violates any of the dependency and conflict requirements in the set A_i, i.e. between the ancestors of vertex i. Specifically, for any two vertices $u, v \in A_i$, if $\{u, v\} \in E_C$ and also $u, v \in S$, then they are in conflict but both taken in S and so we have to set $\mathrm{dp}[i, S, c] = -\infty$ since the requirements are impossible to satisfy. Similarly, if $(u, v) \in E_D$ and $v \in S$ but $u \notin S$, then the dependency requirement is violated and we set $\mathrm{dp}[i, S, c] = -\infty$. For example, in Figure 5.3 we set $\mathrm{dp}[4, \{4\}, 10]$ to $-\infty$ since 4 is included and depends on 2, which is not included. Similarly, $\mathrm{dp}[3, \{1, 2, 3\}, 10] = -\infty$ since 1 and 3 are in conflict.
- We then check if all the transactions in S can fit into c bytes, i.e. whether

$\sum_{t \in S} \sigma(t) \leq c$. If not, we set $\mathrm{dp}[i, S, c] = -\infty$.

- Let $S' \subseteq A_{i-1}$ be a subset of ancestors of vertex $i-1$. We say that S' is *compatible* with S and write $S' \leftrightarrows S$ if $\forall u \in A_{i-1,i}$ we have $u \in S \Leftrightarrow u \in S'$. In other words, compatible subsets make the same decisions about the common ancestors of i and $i-1$. If all the checks above pass, then we consider two cases:

  (1) If $i \notin S$, then we know that our solution TX^* cannot contain the transaction i. Thus, all transactions in the solution are already present in G_{i-1}. Therefore, we set

  $$\mathrm{dp}[i, S, c] = \max_{S' \leftrightarrows S} \mathrm{dp}[i-1, S', c].$$

  (1) If $i \in S$, then we must put the transaction i into our solution TX^*. This reduces the available space to $c - \sigma(i)$ but also gives us a fee of $\phi(i)$. We should then fill out our block using the previous $i-1$ transactions. Thus, we have

  $$\mathrm{dp}[i, S, c] = \phi(i) + \max_{S' \leftrightarrows S} \mathrm{dp}[i-1, S', c - \sigma(i)].$$

Final Solution. Finally, we know that $G = G_n$ by definition. In our optimal solution, we might be taking any subset S of the ancestors of vertex n. Thus, our algorithm outputs

$$\max_{S \subseteq A_n} \mathrm{dp}[n, S, k]$$

as the maximum possible amount of transaction fees that can be obtained from a subset of TX that fits into a block of size k. As is standard in dynamic programming approaches, the optimal subset TX^* of transactions can be recovered by retracing the steps of our algorithm and finding out which choices led to the maximum values.

Theorem 8. *Given a block size limit k, a set TX of n unmined Cardano transactions with dependency-conflict graph G and a treedepth decomposition T of G with depth d, our algorithm solves the Optimal Block Production problem in time $O(n \cdot k \cdot 2^d \cdot d^2)$.*

Proof. Correctness was argued in the discussion above. Note that we always find a valid solution TX^* since the dependency/conflict requirements between any vertex i and its ancestors are enforced when we are computing $\mathrm{dp}[i, \cdot, \cdot]$. To bound the runtime, note that we define a total of $n \cdot k \cdot 2^d$ dynamic programming variables. For each of them, we check dependency and conflict requirements between elements of a set A_i which has a size of at most d since they are all ancestors of a single vertex i in T. Therefore, the total runtime for the initial checks is $O(n \cdot k \cdot 2^d \cdot d^2)$. Now consider the sums computed in parts (1) and (2) above. Consider any fixed $S' \subseteq A_{i-1}$. Based on the way we numbered our vertices in pre-order, every ancestor of vertex i, except i itself, is also an ancestor of vertex $i-1$. Formally, $A_{i,i-1} = A_i \setminus \{i\}$. Thus, each S' may contribute to the sums for at most two different compatible $S \subseteq A_i$ sets. Therefore, the total runtime of all sums in (1) and (2) is $O(n \cdot k \cdot 2^d)$. The runtime is polynomial in n and k when d is a constant. $\square$

Parallelization. We remark that the computation of $\mathrm{dp}[i, \cdot, \cdot]$ in our algorithm only depends on $\mathrm{dp}[i-1, \cdot, \cdot]$ values. Thus, for every i, we can perfectly parallelize the computation of all $\mathrm{dp}[i, \cdot, \cdot]$ entries. In other words, if we have $p < k \cdot 2^d$ parallel cores, our runtime will be $O\left(\frac{n \cdot k \cdot 2^d \cdot d^2}{p}\right)$. Therefore, one can reduce the runtime of our algorithm arbitrarily by simply adding more computational power. In Section 5.4 we do not use parallelization but our runtimes are still much less than one second, enabling the direct application of our approach to Cardano.

5.4 Evaluation on Real Cardano Blocks

We implemented our algorithm in `C++` as a tool called Pixiu. Pixiu is free and open-source software donated to the public domain. We used FlowCutter [50] to find treedepth decompositions.

Machine. All experimental results were obtained on an Intel Xeon Gold 5115 CPU (2.40GHz, 16 cores) running Ubuntu 22.04 and 64 GB of RAM. We did not use parallelization in the runtimes reported below.

Benchmarks. With the help of the Cardano Foundation, we gathered the sets of unmined transactions before each of the blocks number 10,044,250 to 10,255,796 of the Cardano blockchain, corresponding to the timeframe of 2024-03-12 00:00:37 UTC to 2024-04-30 23:59:41 UTC (50 days). For each of the 211,547 blocks mined in this period, we ran our algorithm to obtain an optimal selection of transactions and compared the resulting total transaction fees with the fee revenue obtained by the block producers on Cardano.

Treedepth. We observed that the vast majority of benchmark DCGs had small treedepth. Figure 5.4 shows a histogram of the obtained treedepths. The average treedepth was 1.45. Thus, our algorithm is applicable to real-world Cardano block production and runs in polynomial time $O(n \cdot k)$ on these instances.

Increases in Revenue. Limiting the runtime to 1s, our algorithm improved the total transaction fees in 56,053 of the 211,547 blocks considered in our experiment. This suggests that many of the blocks produced on the Cardano blockchain were already optimal. This is not surprising since when the transaction load in the network

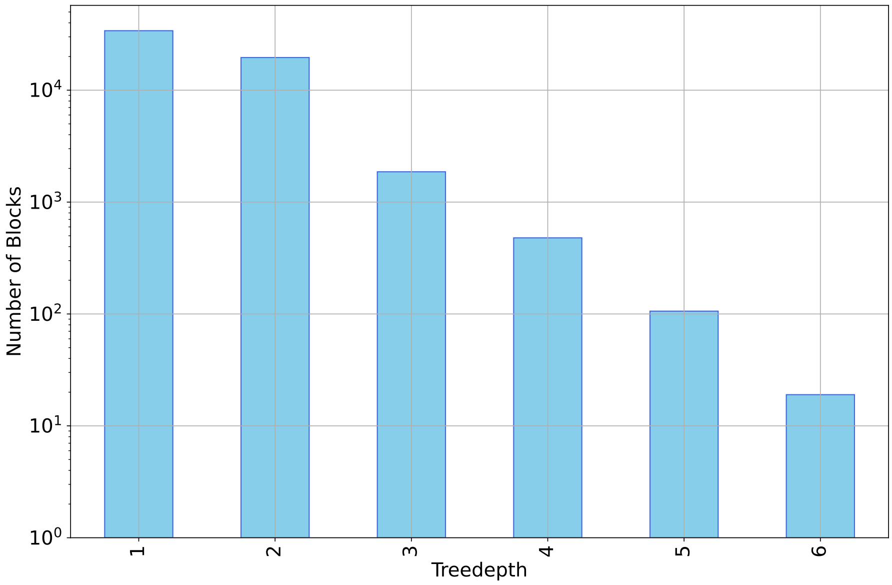


Figure 5.4: Histogram of treedepths of DCGs in real-world Cardano benchmarks. The y-axis is in logarithmic scale.

is low, the miners can often include all the available transactions in their block, which would of course be optimal. Our algorithm's advantage is most pronounced when the number of available unmined transactions is much more than the capacity of a block. Over these 56,053 blocks, the average per-block improvement was 55.68 percent, corresponding to 1.55 Ada and 1.21 USD, whereas the maximum improvement was 66.48 Ada = 51.85 USD. We used the exchange rate 1 Ada = 0.78 USD. The overall improvement over the period of the experiment was 87,040.20 Ada or 67,891.35 USD. Thus, our algorithm obtains transaction fee revenue increases of 1,357.82 USD/day = 495,604.3 USD/year. Therefore, the Cardano miners would benefit immensely from applying our algorithm and ensuring that they will always produce optimal blocks that maximize their transaction fee revenue. Figures 5.5–5.7 show the histogram of obtained improvements in percentage, Ada and USD. Our average runtime was 0.31s per block.

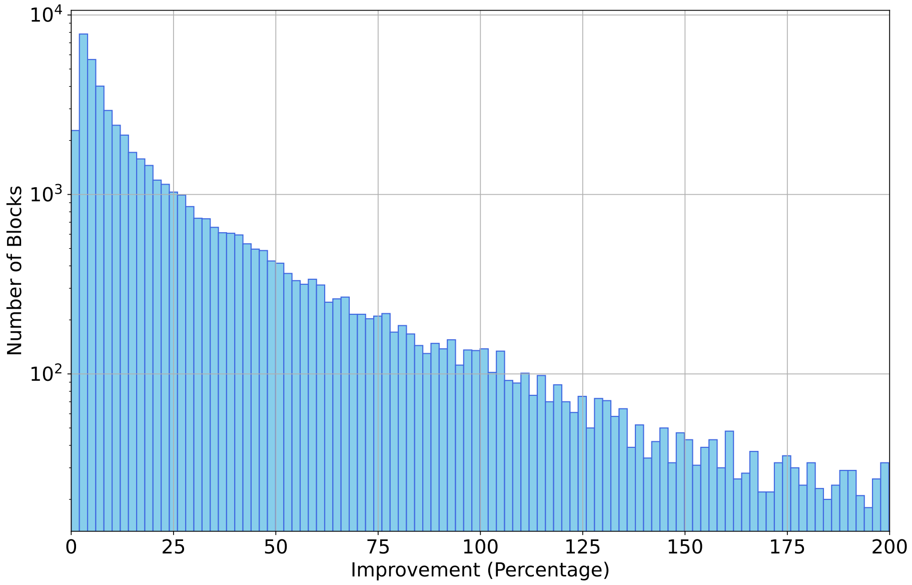


Figure 5.5: Histogram of the transaction fee improvements obtained over each block (in percentages). The y-axis is in logarithmic scale.

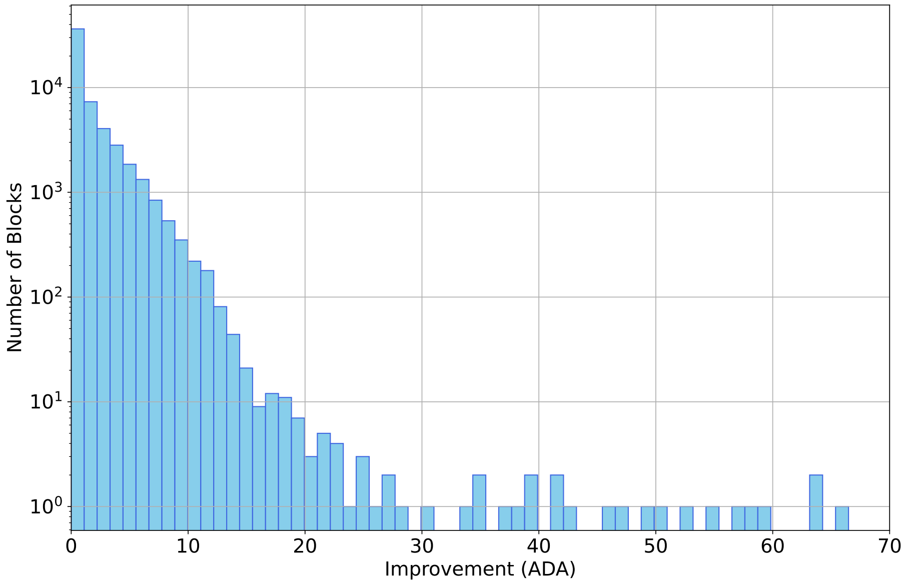


Figure 5.6: Histogram of the transaction fee improvements obtained over each block (in Ada). The y-axis is in logarithmic scale.

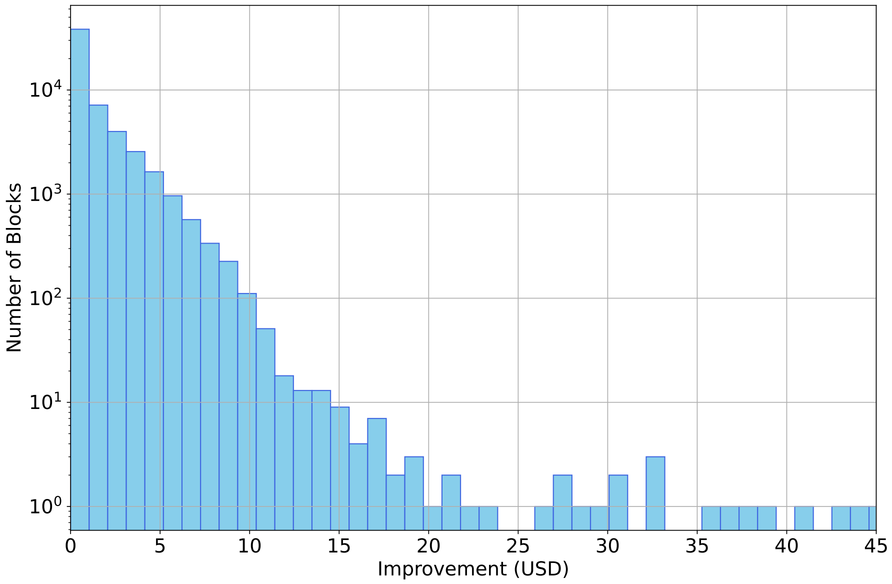


Figure 5.7: Histogram of the transaction fee improvements obtained over each block (in USD). The y-axis is in logarithmic scale.

5.5 Ethereum Account-Based Model

Our goal in this section is to develop an optimization approach to increase the transaction-fee (gas) revenues of Ethereum miners. Although we focus on Ethereum, our techniques are general in nature and can be applied to other programmable blockchains as well. There are two reasons for the emphasis on Ethereum: First, Ethereum is currently the world's second largest cryptocurrency by market cap, behind only Bitcoin. At the time of writing, it has a total value of around 250 billion USD [2]. It is also by far the largest programmable cryptocurrency, with support for arbitrary smart contracts written in a Turing-complete language [16]. Second, Ethereum has an account-based execution model which is significantly different than the UTXO (Unspent Transaction Output) model of Bitcoin. Thus, there is a research gap as the techniques developed for Bitcoin and other UTXO miners, such as Section 5.2, are not applicable in our context.

5.5.1 Problem Definition

Our Focus. In this section, we address the problem of maximizing the total transaction fee revenue of an honest miner, i.e. a miner who does not perform attacks or seek revenue from non-transaction-fee sources such as smart contract interactions. Such a miner listens for newly-broadcast transactions not yet added to the blockchain and selects an ordered subset to form her block. The goal is to maximize the sum of tips paid by the transactions.

Formal Definition of the Problem. Suppose that the miner intends to add the k-th block to the blockchain. Let Pool be the set of new valid transactions known to the miner which are not yet added on thse blockchain. A block is a sequence $B = \langle \mathrm{tx}_1, \mathrm{tx}_2, \ldots, \mathrm{tx}_n \rangle$ of transactions from Pool such that for every $i < j$, if tx_i and tx_j belong to the same user a, then tx_i's nonce is smaller than tx_j's nonce. Moreover, if there is a transaction from a with nonce $\nu \geq 1$, then there should be a previous transaction from the same user a with nonce $\nu - 1$ either in the same block or in previous blocks that are already added to the blockchain. This ensures that every transaction is added at most once and that the transactions by the same user cannot be reordered. Denote the world state at the end of the previous block $(k-1)$ by $s_0 \in S$. For all $1 \leq i \leq n$, define

$$s_i = \delta_{\mathrm{tx}_i}(s_{i-1}) \qquad \gamma_i = \mathsf{gas}_{\mathrm{tx}_i}(s_{i-1}) \qquad \tau_i = \mathrm{tip}_{\mathrm{tx}_i}(s_{i-1}).$$

Intuitively, assuming that the miner appends B to the blockchain, s_i is the world state after tx_i has finished running, γ_i is the gas usage of tx_i and τ_i is the total tip paid by tx_i to the miner (formal definitions and details can be found in Section 5.6). The miner aims to maximize the tips, while not exceeding the block gas limit L. Thus, the optimization problem is to find a valid block B so as to:

$$\mathbf{max} \sum_{i=1}^{n} \tau_i \qquad \mathbf{s.t.} \sum_{i=1}^{n} \gamma_i \leq L$$

Our Algorithmic Contribution. In Section 5.6, we design and present a randomized algorithm that combines testing, decision trees, localization techniques and integer linear programming (ILP) to handle the optimization problem above. We start by executing several test cases, i.e. random permutations of the transactions in our Pool, and profiling their gas usage. Then, using decision trees, for every transaction $\mathrm{tx} \in \mathsf{Pool}$, we find a set of other transactions that can affect its gas usage. We call this the neighborhood of tx and denote it as $\mathrm{nbhd}(\mathrm{tx})$. Intuitively, we expect the neighborhood of every vertex to be small since, in real-world instances, most transactions are completely independent of each other. For example, a pair of transactions that do not access the same smart contracts cannot affect each other's gas usage, either. We provide a probabilistic argument showing that with high probability our testing covers every possible permutation of each neighborhood. Thus, we can cut, mix and glue together parts of our test cases in order to create a block that maximizes the miner's total tip revenue. This part is modeled as an integer linear programming instance and solved by an external optimization suite. All steps of our algorithm are parallelizable and, except for the ILP solver, run in polynomial time. Due to its reliance on randomized sampling and ILP-solvers, our algorithm is not guaranteed to always produce an optimal result.

Our Practical Contribution. We implemented our algorithm and performed extensive experimental results on 50,000 Ethereum blocks. For each block, we gathered real-world transaction pool data and executed our algorithm. We then compared the tip revenues obtained by our framework and those of real-world miners. Our approach increased tip revenues by a significant margin of 73.45 percent on average per block, corresponding to roughly 24.1 USD per block and 63,357,892 USD per year at current exchange rates. We also compared our tool against the default Ethereum implementation and found that our approach increased tip revenues by 18.56 percent on average per block, corresponding to roughly 17.3 USD per block and 45,416,764 USD per year. Our tool is free, open-source and dedicated to the public domain with a perpetual copyright waiver.

5.5.2 Ethereum Architecture

The remainder of this subsection outlines the specific components of the Ethereum architecture that are prerequisites to our algorithm. Refer to Section 2.4 for a more comprehensive background.

Details of Transaction Fees [40, 41]. When creating a transaction, the user should set a maximum amount m of gas that the transaction is allowed to consume. They should also set a price p that they are willing to pay per unit of gas. Right before the transaction's execution, a deposit of $m \cdot p$ is taken from the user's account. If the transaction uses $g \leq m$ units of gas in its execution, then the transaction fee will be $g \cdot p$ and the user receives a refund of $(m - g) \cdot p$. Otherwise, if $g > m$, an out-of-gas error is raised. This sets $g = m$. When running out of gas, all effects of the transaction are reverted. However, the deposit is not refunded and the entire $m \cdot p$ is taken as a transaction fee.

Base Fees and Tips [40, 41]. The gas price $p = b + t$ consists of two parts: a base fee b which is the same for all transactions in the block and a tip t which is picked by the user starting the transaction. If the transaction uses g units of gas, the user pays a transaction fee of $g \cdot p$ as mentioned above. Of this, $g \cdot b$ units are burned and $g \cdot t$ is paid to the miner who added the transaction to the blockchain. There are two clear reasons behind the base fee: (i) it controls inflation by destroying some units of currency, and (ii) the protocol can balance the supply and demand of block space. Ethereum aims to have blocks that use roughly 15,000,000 units of gas which is almost half of the block gas limit. If a block uses more than this amount, it shows that demand for gas is high and thus the base fee goes up in future blocks. Conversely, if a block uses less than 15,000,000 units, e.g. if it is almost empty, the base fee drops.

Structure of an Ethereum Transaction. A formal specification for Ethereum transactions is given in [41]. Each transaction tx contains a nonce, a counter showing the number of transactions issued by the current user to date. It also contains the gas parameters m and $p = b + t$ above. If the transaction transfers money, it includes the recipient address and amount. Similarly, if it deploys a contract it includes its code, and if it calls a function, it includes its parameters. We ignore these details and focus on the fact that, given the world state before the execution of tx, its gas usage and the total tip paid to the miner are uniquely determined. Thus, letting S be the set of all possible world states, from the miner's point-of-view, the transaction tx induces two functions:

$$\mathsf{gas}_{\text{tx}} : S \to \mathbb{N} \qquad \text{tip}_{\text{tx}} : S \to \mathbb{N}.$$

Note that $\text{tip}_{\text{tx}} \equiv t \cdot \mathsf{gas}_{\text{tx}}$. Similarly, since executing tx changes the world state, the transaction induces a transition function $\delta_{\text{tx}} : S \to S$ between world states.

Optimal Mining on UTXO Blockchains. In the UTXO model, as discussed in Section 5.2, transactions can have dependencies and conflicts. For example, if Alice receives money in tx_1 and uses it to pay Bob in tx_2, i.e. an input of tx_2 is an output of tx_1, then tx_2 can only be added to the blockchain after tx_1. On the other hand, if Alice receives money in tx_1 and then creates distinct transactions tx_2 and tx_3 that use the same coin twice then at most one of them can be added to the consensus chain. Thus, optimizing a miner's revenues on UTXO blockchains is equal to solving a variant of the knapsack problem in which items may have pairwise dependencies and conflicts. A transaction's weight is its size and its value is the transaction fee. Crucially, the transaction fees are fixed and known apriori. Moreover, when the miner chooses the subset of transactions that are included in her block, theisr order does not matter and any topological ordering leads to the same total revenue. Despite this, the problem is shown to be strongly NP-hard and can only be solved on sparse instances [76, 77].

In contrast, our setting on Ethereum is considerably more challenging. Due to the gas model, the transaction fees are dynamic and depend on the actual resources used in runtime. The same transaction might pay vastly different fees based on the world state before its execution, which is determined by the transactions that precede it in the block. As a toy example, consider the smart contract in Figure 5.8a.

Suppose tx_1 calls `f()` and tx_2 calls `g()`. If the miner places tx_1 before tx_2 in the block, n becomes 1000, so `g()` incurs a high fee. If tx_2 is first, $n = 0$ when `g()` runs,

```
contract ToyGasDependency {
 uint n = 0;
 function f() public {n = 1000; }
 function g() public {
  for (uint i = 0; i <= n; i++) {
   /*write to storage*/
  }
 }
}
```

(a) A toy contract illustrating gas dependency.

```
contract IndirectGasDependency {
 uint n = 0, m = 0;
 function f() public {m = 1000; }
 function g() public {
  for (uint i = 0; i <= n; i++) {
   /*write to storage*/
  }
 }
 function h() public {n = m; }
}
```

(b) A contract showing indirect gas dependency.

Figure 5.8: Two example contracts illustrating gas dependency and indirect gas dependency.

resulting in a much lower gas usage and fee. Thus, transaction ordering directly affects fees. Unlike the UTXO model, one cannot determine if tx_1's gas usage depends on tx_2 by inspecting just the pair; dependencies may only arise in the presence of other transactions. For example, consider the smart contract in Figure 5.8b.

Suppose that the miner observes a transaction tx_1 calling `f()` and another transaction tx_2 calling `g()` in the mempool. These two transactions are clearly independent in terms of gas usage and each transaction's inclusion or their order cannot affect the fee paid by the other transaction. However, if a new transaction tx_3 calling `h()` is observed, then tx_1 and tx_2 suddently become dependent and the miner must use the ordering $\mathrm{tx}_1, \mathrm{tx}_3, \mathrm{tx}_2$ to maximize her payoff. Note that tx_2 and tx_3 are also independent in the absence of tx_1. Thus, we cannot talk of gas dependencies between a pair of transactions without specifying all transactions available in Pool.

MEV [78]. The term *Miner Extractable Value* or *Maximal Extractable Value* (MEV) was popularized by [79] to describe the maximum profit a dishonest miner can obtain by exploiting their knowledge of transactions. This is often called "toxic" MEV in the blockchain community. For instance, miners can engage in front-running, back-running, and sandwich attacks, leveraging their ability to manipulate, reorder, or censor transactions [80, 81, 82]. Such attacks can even threaten the security of Ethereum's consensus layer [79]. Several countermeasures against transaction reordering have been proposed [83], and formal verification tools exist to identify profitable MEV opportunities [84, 85].

MEV attacks typically involve the miner interacting with a smart contract. In most front-running attacks, for example, the miner observes a pending transaction tx_1 and uses its disclosed data to create their own transaction tx_2, placing it before tx_1 in the block to capture the payoff. Here, the miner's extra revenue comes not from mining or transaction fees, but from exploiting a vulnerable contract. In contrast, this chapter focuses on honest miners. In our formalization (Section 5.5.2), the miner seeks to maximize revenue from the available transactions, without attacking contracts, forking the blockchain, or creating new transactions. This distinction makes our approach suitable for reference implementations of programmable blockchains, which are widely used by real-world miners. Notably, all major consensus protocols, including proof of work and proof of stake, rely on the assumption that the majority of mining power is held by honest miners.

5.6 A Randomized Framework

We start this section with a high-level overview of our algorithm and then provide the details of each step. The fundamental idea behind our approach is a simple observation: in real-world instances of the problem, each transaction interacts with only a few other transactions, which we call its neighborhood. Thus, when forming a block, we do not need to search over all possible permutations of transactions, but can instead find rules that encode the gas usage of each transaction based on the portion of its neighborhood that precedes it in the block.

Overview. Our approach consists of the following steps:

1. *Sampling and Testing.* We start by sampling random permutations of the transactions available in our Pool and executing them, keeping track of the amount of gas used by each transaction in each sample permutation.

2. *Neighborhood Estimation.* For each transaction tx, we identify a neighborhood, i.e. a set nbhd(tx) of transactions whose inclusion before tx has the potential to change the amount of gas consumed by tx. To estimate each transaction's neighborhood, we rely on a decision tree generated from our samples.

3. *Extracting Gas Usage Rules.* Using the neighborhoods and samples, we extract gas usage rules. We define a simple grammar that can encode rules such as "if tx_1 appears before tx_2 which in turn appears before tx_3, then tx_3 will have a gas usage of g" for $\mathrm{tx}_1, \mathrm{tx}_2 \in \mathrm{nbhd}(\mathrm{tx}_3)$. We show that such rules can be mined from our samples, using a different decision tree.

4. *Translating Gas Usage Rules to ILP.* Our goal is to obtain a block that orders a subset of transactions in a manner that maximizes the total tip earned by the miner. To do this, we have to choose which rules from the previous step to

follow, i.e. which local orderings to include, but cannot choose rules that require conflicting orders on the transactions. Our algorithm translates this problem to integer linear programming (ILP).

5. *Solving the ILP.* Finally, we call an external off-the-shelf ILP-solver to handle the ILP instance generated in the previous step. This yields the desired block.

Step (1): Sampling and Testing. The input to our algorithm is the initial world state s_0 and a set $\mathsf{Pool} = \{\text{tx}_1, \text{tx}_2, \dots, \text{tx}_n\}$ of valid transactions which are not yet added to the blockchain. In this step, we randomly and uniformly generate several permutations $\Pi = \{\pi_1, \pi_2, \dots, \pi_k\}$ of our Pool. The number of samples, k, is a user-defined parameter. Note that the π_i's are not necessarily valid blocks as they might exceed the block gas limit.

We need to perform some simple housekeeping at this point. Recall that transactions originating from the same account should appear in the order of their nonces. We say a transaction tx is an a-transaction if it originates from an account with address a. For every i and a, we sort all the a-transactions in π_i by their nonce, thus ensuring that every sample π_i respects nonce orders. If there are several a-transactions with the same nonce in π_i, we only keep the first and remove the rest. Thus, π_i might end up containing only a subset of the transactions in Pool.

We then execute every sample π_i starting from the initial world state s_0 and keep track of the amount of gas used by each transaction tx_j in the execution of π_i. We denote this by $\mathsf{gas}(\pi_i, \text{tx}_j)$.

Example 18. *Suppose we have 6 transactions in our pool and the tip value set by the users is* $t[\text{tx}_1, \text{tx}_2, \text{tx}_3, \text{tx}_4, \text{tx}_5, \text{tx}_6] = [24, 9, 5, 29, 24, 2]$, *i.e.* tx_1 *pays a tip of* 24 *for each consumed unit of gas,* tx_2 *pays* 9, *etc. We generate* $k = 10$ *random permutations* $\pi_1, \dots, \pi_{10}$ *of the pool, execute them, and keep track of the gas used by each transaction in each sample. See Figure 5.9.*

Step (2): Neighborhood Estimation. In this step, we consider each transaction $\text{tx}_i \in \mathsf{Pool}$ separately and our goal is to identify a neighborhood for it, i.e. a set of transactions $\text{nbhd}(\text{tx}_i) \subseteq \mathsf{Pool}$ whose inclusion before tx_i can potentially change the amount of gas consumed by tx_i. We observe that the transactions usually exhibit only a few different possible gas usage values. Thus, we can use the amount of gas used by tx_i as a label in a classification problem. We introduce one binary attribute for each tx_j, which is set to 1 if tx_j precedes tx_i in the sample permutation. We then generate a decision tree. At each node of the tree, we choose the best possible attribute to branch on, based on the Gini impurity metric [86, Chapter 4.6]. After generating the decision tree, if the attribute corresponding to tx_j appears in one of the decisions, we add tx_j to the neighborhood $\mathrm{nbhd}(\mathrm{tx}_i)$.

Example 19. *Consider the samples of the previous example (Figure 5.9). In this step, our algorithm finds a neighborhood for every transaction. Let us consider* tx_3*. We can encode each permutation π_i as a vector v_i such that $v_i[j] = 1$ if* tx_j *precedes* tx_3 *in π_i and $v_i[j] = 0$ otherwise. The label associated with v_i is the gas usage of* tx_3 *in the execution of π_i. In this case, there are only two possible labels: 2 and 4. See Figure 5.11 (left). When we generate a decision tree, shown in Figure 5.11 (right), it only looks at the attribute corresponding to* tx_2*. Hence, we set* $\mathrm{nbhd}(\mathrm{tx}_3) = \{\mathrm{tx}_2\}$*. In other words, at least in our samples, the gas usage of* tx_3 *is only dependent on whether* tx_2 *precedes it. This is shown in Figure 5.10.*

Step (3): Extracting Gas Usage Rules. Now that we have estimated the neighborhood of every transaction, our next goal is to extract a set of rules that define the gas usage of each transaction tx based on the subset of neighbors that precede tx in the block and their order of appearance.

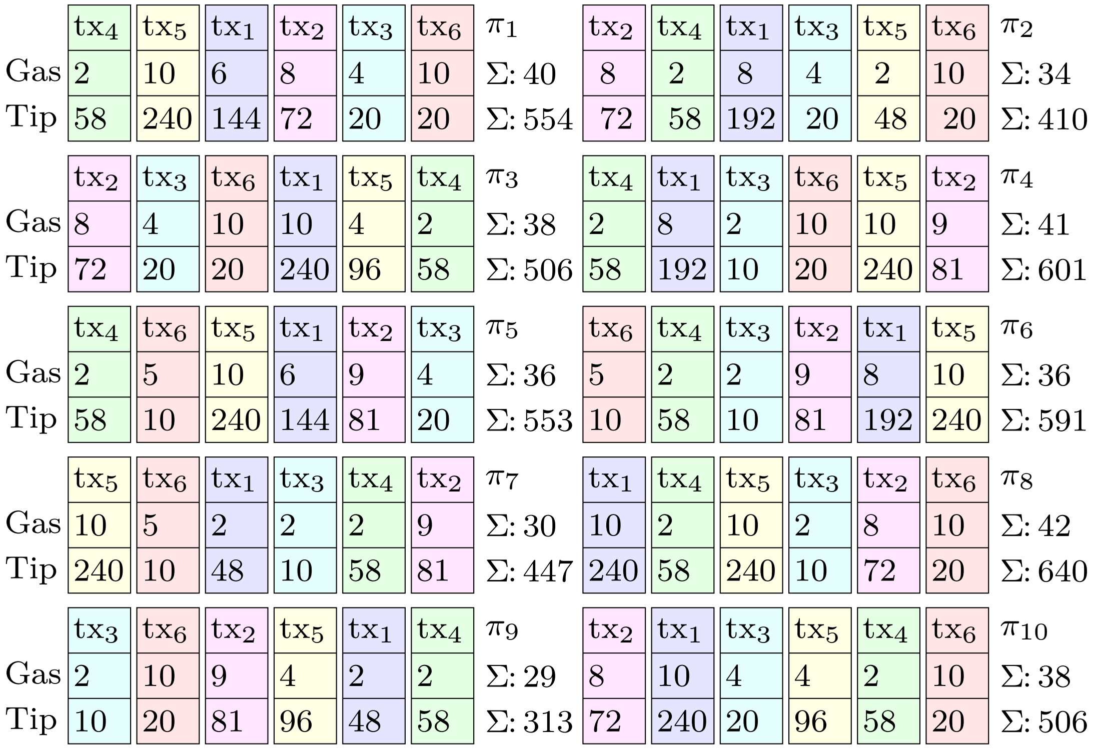
Syntax. Our rules are defined by the grammar below:

$$\begin{array}{lcl} \langle\text{predicate}\rangle & := & \Diamond\mathrm{tx} \mid \mathrm{tx} \prec \mathrm{tx} \mid \neg\langle\text{predicate}\rangle \\ \langle\text{predicate-list}\rangle & := & \langle\text{predicate}\rangle \mid \langle\text{predicate}\rangle, \langle\text{predicate-list}\rangle \\ \langle\text{clause}\rangle & := & \langle\text{predicate}\rangle \mid \mathtt{atleast}(k, \langle\text{predicate-list}\rangle) \mid \\ & & \langle\text{clause}\rangle \wedge \langle\text{clause}\rangle \\ \langle\text{rule}\rangle & := & \langle\text{clause}\rangle \Rightarrow \textsf{expected-gas}(\mathrm{tx}) = g \\ & & g, k \in \mathbb{N}, \mathrm{tx} \in \textsf{Pool} \end{array} \tag{5.1}$$

Semantics. Given a sequence of distinct transactions $\pi = \langle\pi[1], \ldots, \pi[m]\rangle$, we have:

$$\begin{array}{lcl} \pi \models \Diamond\mathrm{tx}_i & \Leftrightarrow & \exists i' \;\; (\pi[i'] = \mathrm{tx}_i) \\ \pi \models \mathrm{tx}_i \prec \mathrm{tx}_j & \Leftrightarrow & \pi \not\models \Diamond\mathrm{tx}_j \vee \exists i', j' \;\; (i' < j' \wedge \\ & & \pi[i'] = \mathrm{tx}_i \wedge \pi[j'] = \mathrm{tx}_j) \\ \pi \models \neg p & \Leftrightarrow & \pi \not\models p \\ \pi \models c_1 \wedge c_2 & \Leftrightarrow & \pi \models c_1 \wedge \pi \models c_2 \\ \pi \models \mathtt{atleast}(k, p_1, \ldots, p_m) & \Leftrightarrow & \exists i_1, \ldots, i_k \;\; (1 \leq i_1 < \ldots < i_k \leq m \wedge \\ & & \forall j \;\; \pi \models p_{i_j}) \end{array} \tag{5.2}$$

Intuitively, π satisfies the predicate $\Diamond\mathrm{tx}_i$ if it includes tx_i. It satisfies the predicate $\mathrm{tx}_i \prec \mathrm{tx}_j$ if it either excludes tx_j altogether or has tx_i preceding tx_j. A clause is simply a conjunction of predicates. We also allow clauses of the form $\mathtt{atleast}(k, p_1, \ldots, p_m)$ which require π to satisfy at least k of the predicates $p_1, \ldots, p_m$. Although the latter



	tx_4	tx_5	tx_1	tx_2	tx_3	tx_6	π_1
Gas	2	10	6	8	4	10	Σ: 40
Tip	58	240	144	72	20	20	Σ: 554

	tx_2	tx_4	tx_1	tx_3	tx_5	tx_6	π_2
Gas	8	2	8	4	2	10	Σ: 34
Tip	72	58	192	20	48	20	Σ: 410

	tx_2	tx_3	tx_6	tx_1	tx_5	tx_4	π_3
Gas	8	4	10	10	4	2	Σ: 38
Tip	72	20	20	240	96	58	Σ: 506

	tx_4	tx_1	tx_3	tx_6	tx_5	tx_2	π_4
Gas	2	8	2	10	10	9	Σ: 41
Tip	58	192	10	20	240	81	Σ: 601

	tx_4	tx_6	tx_5	tx_1	tx_2	tx_3	π_5
Gas	2	5	10	6	9	4	Σ: 36
Tip	58	10	240	144	81	20	Σ: 553

	tx_6	tx_4	tx_3	tx_2	tx_1	tx_5	π_6
Gas	5	2	2	9	8	10	Σ: 36
Tip	10	58	10	81	192	240	Σ: 591

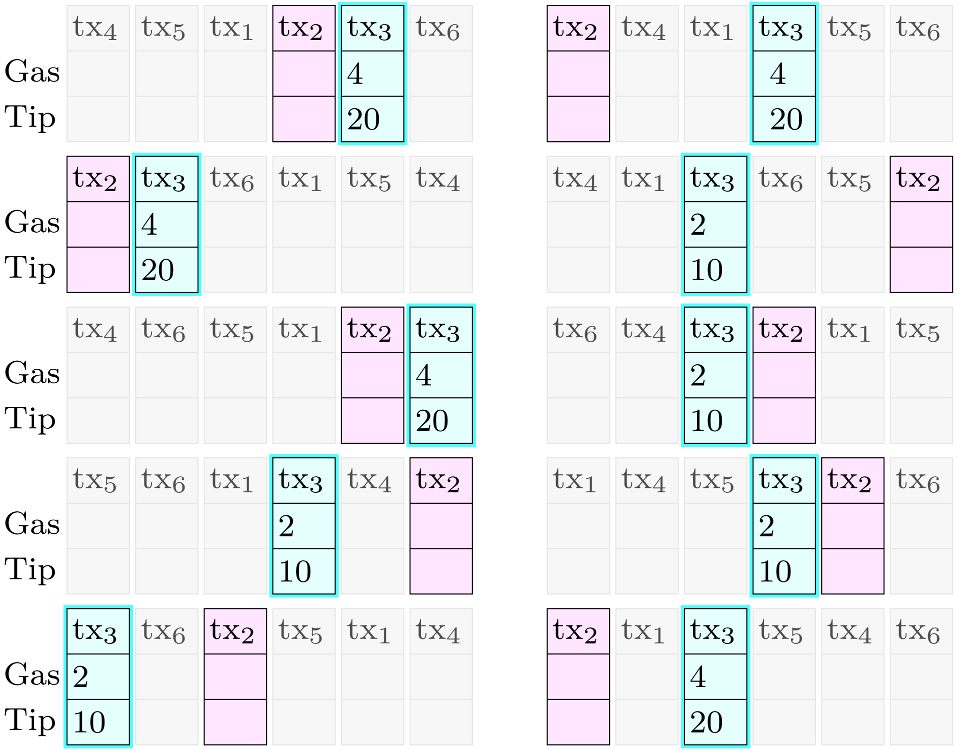
	tx_5	tx_6	tx_1	tx_3	tx_4	tx_2	π_7
Gas	10	5	2	2	2	9	Σ: 30
Tip	240	10	48	10	58	81	Σ: 447

	tx_1	tx_4	tx_5	tx_3	tx_2	tx_6	π_8
Gas	10	2	10	2	8	10	Σ: 42
Tip	240	58	240	10	72	20	Σ: 640

	tx_3	tx_6	tx_2	tx_5	tx_1	tx_4	π_9
Gas	2	10	9	4	2	2	Σ: 29
Tip	10	20	81	96	48	58	Σ: 313

	tx_2	tx_1	tx_3	tx_5	tx_4	tx_6	π_{10}
Gas	8	10	4	4	2	10	Σ: 38
Tip	72	240	20	96	58	20	Σ: 506

Figure 5.9: An example execution of 10 samples of a pool of 6 transactions, profiling the gas usage and tip revenue obtained from each transaction.



	tx_4	tx_5	tx_1	tx_2	tx_3	tx_6
Gas					4	
Tip					20	

	tx_2	tx_4	tx_1	tx_3	tx_5	tx_6
Gas				4		
Tip				20		

	tx_2	tx_3	tx_6	tx_1	tx_5	tx_4
Gas		4				
Tip		20				

	tx_4	tx_1	tx_3	tx_6	tx_5	tx_2
Gas			2			
Tip			10			

	tx_4	tx_6	tx_5	tx_1	tx_2	tx_3
Gas						4
Tip						20

	tx_6	tx_4	tx_3	tx_2	tx_1	tx_5
Gas			2			
Tip			10			

	tx_5	tx_6	tx_1	tx_3	tx_4	tx_2
Gas				2		
Tip				10		

	tx_1	tx_4	tx_5	tx_3	tx_2	tx_6
Gas				2		
Tip				10		

	tx_3	tx_6	tx_2	tx_5	tx_1	tx_4
Gas	2					
Tip	10					

	tx_2	tx_1	tx_3	tx_5	tx_4	tx_6
Gas			4			
Tip			20			

Figure 5.10: The gas usage of tx_3 only depends on whether tx_2 preceded it in the sample permutation.

Sample (π_i)	Encoding (v_i)	Label (γ_3)
π_1	$[1, 1, 0, 1, 1, 0]$	4
π_2	$[1, 1, 0, 1, 0, 0]$	4
π_3	$[0, 1, 0, 0, 0, 0]$	4
π_4	$[1, 0, 0, 1, 0, 0]$	2
π_5	$[1, 1, 0, 1, 1, 1]$	4
π_6	$[0, 0, 0, 1, 0, 1]$	2
π_7	$[1, 0, 0, 0, 1, 1]$	2
π_8	$[1, 0, 0, 1, 1, 0]$	2
π_9	$[0, 0, 0, 0, 0, 0]$	2
π_{10}	$[1, 1, 0, 0, 0, 0]$	4

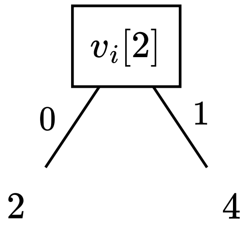


Figure 5.11: The classification problem is based on the gas used by tx_3 (left) and the resulting decision tree (right).

could be left as syntactic sugar, including it as a separate clause is helpful in practice. This is because of a common scenario in smart contracts, especially those modeling NFT or token giveaways, in which the first k transactions to call a function can receive the token. In such cases, all transactions calling this function are dependent in terms of gas, but the gas usage of each of them is only dependent on how many other transactions precede it. Finally, a rule of the form $C \Rightarrow \textsf{expected-gas}(\text{tx}_i) \geq g$ signifies a prediction: if C is satisfied, we expect the gas usage of the transaction tx_i to be at least g.

Step (3.i): Generating a Decision Tree. In this step, we generate a set of rules for each transaction $\text{tx}_i \in \textsf{Pool}$. We process each such transaction separately. When generating the rules for tx_i, we narrow our focus down to its neighborhood $\text{nbhd}(\text{tx}_i)$ since we believe these are the only transactions that can affect the gas usage of tx_i. Let π_j be one of the sample permutations. We define π_j^* as the subsequence of π_j induced by $\text{nbhd}(\text{tx}_i) \cup \{\text{tx}_i\}$, i.e. the subsequence obtained by removing every element that is not in the neighborhood or not tx_i itself. If $\pi \not\models \Diamond\text{tx}_i$, we ignore this sample as it does not include tx_i and cannot give us any information about its gas usage. We use Π^* to denote the set of neighborhood-induced samples.

As in Step 2, our goal is to form a decision tree that predicts the gas usage of tx_i based on features that encode the set of neighbors that precede it in the block and their order. Thus, for every neighboring $\text{tx}_j \in \text{nbhd}(\text{tx}_i)$ we define a corresponding feature

$$w[j] = \begin{cases} j' - i' & \text{if } \pi^* \models \text{tx}_j \prec \text{tx}_i \wedge \pi^*[i'] = \text{tx}_i \wedge \pi^*[j'] = \text{tx}_j \\ 0 & \text{otherwise} \end{cases}.$$

Informally, if π^* puts tx_j before tx_i, then $-w[j]$ is the distance between the two transactions in π^*. Otherwise, $w[j] = 0$. In other words, we care about the distance only if tx_j precedes tx_i. This is because if tx_j does not appear before tx_i, it cannot

Sample (π_i)	nbhd(tx_1)-induced Sample (π_i^*)	Encoding (w) $\lvert\pi_1, \pi_4, \pi_5, r\rvert$	Label (tx_1's gas)
π_1	$\langle \mathrm{tx}_4, \mathrm{tx}_5, \underline{\mathrm{tx}_1} \rangle$	$[0, -2, -1, 2]$	6
π_2	$\langle \mathrm{tx}_4, \underline{\mathrm{tx}_1}, \mathrm{tx}_5 \rangle$	$[0, -1, \ \ 0, 1]$	8
π_3	$\langle \underline{\mathrm{tx}_1}, \mathrm{tx}_5, \mathrm{tx}_4 \rangle$	$[0, \ \ 0, \ \ 0, 0]$	10
π_4	$\langle \mathrm{tx}_4, \underline{\mathrm{tx}_1}, \mathrm{tx}_5 \rangle$	$[0, -1, \ \ 0, 1]$	8
π_5	$\langle \mathrm{tx}_4, \mathrm{tx}_5, \underline{\mathrm{tx}_1} \rangle$	$[0, -2, -1, 2]$	6
π_6	$\langle \mathrm{tx}_4, \underline{\mathrm{tx}_1}, \mathrm{tx}_5 \rangle$	$[0, -1, \ \ 0, 1]$	8
π_7	$\langle \mathrm{tx}_5, \underline{\mathrm{tx}_1}, \mathrm{tx}_4 \rangle$	$[0, \ \ 0, -1, 1]$	2
π_8	$\langle \underline{\mathrm{tx}_1}, \mathrm{tx}_4, \mathrm{tx}_5 \rangle$	$[0, \ \ 0, \ \ 0, 0]$	10
π_9	$\langle \mathrm{tx}_5, \underline{\mathrm{tx}_1}, \mathrm{tx}_4 \rangle$	$[0, \ \ 0, -1, 1]$	2
π_{10}	$\langle \underline{\mathrm{tx}_1}, \mathrm{tx}_5, \mathrm{tx}_4 \rangle$	$[0, \ \ 0, \ \ 0, 0]$	10

Figure 5.12: The classification problem based on the gas used by tx_1

affect tx_i's gas usage. In addition to the $w[j]$'s, we also add an extra feature r which models the number of neighbors that precede tx_i, or equivalently, the index in π^* at which tx_i appears. We generate a decision tree using the same method as in Step 2, where the samples are the subsequences π^*, the features are r and the $w[j]$'s, and the labels are different possible gas usage values of tx_i. Note that our features are no longer binary.

Example 20. *For step 2 of the algorithm in our previous example, suppose we know that* $\mathrm{nbhd}(\mathrm{tx}_1) = \{\mathrm{tx}_4, \mathrm{tx}_5\}$. *Then for* $\pi_1 = \langle \mathrm{tx}_4, \mathrm{tx}_5, \mathrm{tx}_1, \mathrm{tx}_2, \mathrm{tx}_3, \mathrm{tx}_6 \rangle$ *the induced sample is* $\pi_1^* = \langle \mathrm{tx}_4, \mathrm{tx}_5, \underline{\mathrm{tx}_1} \rangle$. *The encoding* $w[\mathrm{tx}_4] = 0 - 2$ *as the position of* tx_4 *is* 0 *and the position of* tx_1 *is* 2. *Similarly,* $w[\mathrm{tx}_5] = 1 - 2 = -1$ *and* $w[\mathrm{tx}_1] = 2 - 2 = 0$. *The feature* $w[r] = 2$ *as the number of neighbors that precede* tx_1 *is* 2. *The encoding of the rest of the samples is shown in Figure 5.12.*

Step (3.ii): Extracting Rules from the Decision Tree. In the decision tree generated for tx_i in Step (3.i), every leaf is labeled by one possible gas usage value of tx_i. Let ℓ be a leaf labeled by g_ℓ. Consider a feature $w[j]$. We know that $-n \leq w[j] \leq 0$. In the path from the root of the decision tree to the leaf ℓ, some decisions bound $w[j]$ from above or below. Let m_j be the largest lower-bound and M_j the smallest upper-bound imposed on $w[j]$ in the root-to-leaf path ending at ℓ. Thus, in all samples modeled by this leaf, we have $w[j] \in [m_j, M_j]$. Similarly, we can define the tightest possible segment $[m_r, M_r]$ for values of r in these samples. For each leaf ℓ, we generate a gas usage rule containing a clause φ_ℓ as follows:

- We add the conjunct $\Diamond \mathrm{tx}_i$ to φ_ℓ.
- If for two transactions $\mathrm{tx}_j, \mathrm{tx}_{j'} \in \mathrm{nbhd}(\mathrm{tx}_i)$, we have $M_j \leq m_{j'}$, then tx_j is always preceding $\mathrm{tx}_{j'}$ in the samples corresponding to ℓ. Thus, we add the conjunct $\mathrm{tx}_j \prec \mathrm{tx}_{j'}$ to φ_ℓ.

- If for a transaction $\mathrm{tx}_j \in \mathrm{nbhd}(\mathrm{tx}_i)$, we have $m_j \geq 0$, then tx_j is never appearing before tx_i in the samples corresponding to ℓ. Thus, we add the conjunct $\mathrm{tx}_i \prec \mathrm{tx}_j$ to φ_ℓ.
- If $m_r > 0$, then in every sample corresponding to ℓ, at least m_r neighboring transactions preceded tx_i. Let the neighborhood of tx_i be $\mathrm{nbhd}(\mathrm{tx}_i) = \{\mathrm{tx}_{j,1}, \dots, \mathrm{tx}_{j,q}\}$. To capture this, we add the conjunct $\texttt{atleast}(m_r, \mathrm{tx}_{j,1} \prec \mathrm{tx}_i, \dots, \mathrm{tx}_{j,q} \prec \mathrm{tx}_i)$ to φ_ℓ.
- Similarly, if $\mathrm{nbhd}(\mathrm{tx}_i) = \{\mathrm{tx}_{j,1}, \dots, \mathrm{tx}_{j,q}\}$ and $M_r < q$, then at most M_r neighbors precede tx_i in each of the samples corresponding to ℓ. Therefore, tx_i precedes at least $q - M_r$ neighbors in each such sample. Hence, we add the conjunct $\texttt{atleast}(q - M_r, \mathrm{tx}_i \prec \mathrm{tx}_{j,1}, \dots, \mathrm{tx}_i \prec \mathrm{tx}_{j,q})$ to the clause φ_ℓ.

By definition-chasing, it is easy to prove that for every sample π^* corresponding to the leaf ℓ of the decision tree, we have $\pi^* \models \varphi_\ell$. Hence, one might assume that the expected gas usage of tx_i in a sample satisfying φ_ℓ is the same as ℓ's label g_ℓ. However, φ_ℓ might be satisfied by other samples, i.e. samples corresponding to other leaves, too. This is rare in practice, but to handle it, our algorithm takes the average gas usage of tx_i over *all* samples π^* that model φ_ℓ. Formally, let $J = \{j \mid 1 \leq j \leq k \wedge \pi^*_j \models \varphi_\ell\}$. Our algorithm computes

$$\overline{g}_\ell = \frac{\sum_{j \in J} \mathsf{gas}(\pi_j, \mathrm{tx}_i)}{|J|}.$$

In almost all real-world cases, we have $\overline{g}_\ell = g_\ell$. Finally, our algorithm generates the following gas usage rule:

$$\varphi_\ell \Rightarrow \mathsf{expected\text{-}gas}(\mathrm{tx}_i) = \overline{g}_\ell. \tag{5.3}$$

We remark that for each transaction tx_i and every leaf ℓ of the decision tree corresponding to tx_i, our algorithm generates one instance of the gas usage rule in Equation (5.3).

Example 21. *Let Figure 5.13 be the decision tree generated for* tx_1 *in our previous example. The tree has* 6 *leaves* l_1 *–* l_6 *labelled* $10, 2, 10, 8, 6, 6$ *respectively. Consider the leaf* l_5*, taking the negations into the account, the predicates along the path evaluate to:* $w[\pi_4] < 0.0$*,* $w[\pi_5] \geq -1.5$*,* $w[\pi_4] < -1.5$*, which after rounding to integers, gives us the intervals* $w[\pi_4] \in (-\infty, -2] = (m_4, M_4]$*,* $w[\pi_5] \in [-1, \infty) = [m_5, M_5)$*. Notice that* $M_4 \leq 0$ *which generates the clause* $\mathrm{tx}_4 \prec tx_1$*, and* $M_4 \leq m_5$ *which generates the clause* $\mathrm{tx}_4 \prec \mathrm{tx}_5$*. Therefore, the rule's body is* $\Diamond \mathrm{tx}_1 \wedge (\mathrm{tx}_4 \prec \mathrm{tx}_1) \wedge (\mathrm{tx}_4 \prec \mathrm{tx}_5)$*. Observe that* $\{\pi_1, \pi_2, \pi_4, \pi_5, \pi_6\} \models \Diamond \mathrm{tx}_1 \wedge (\mathrm{tx}_4 \prec \mathrm{tx}_1) \wedge (\mathrm{tx}_4 \prec \mathrm{tx}_5)$*, with average gas usage of* tx_1 *equal to* $(6 + 8 + 8 + 6 + 8)/5 = 7.2$*. The remaining rules are given in Figure 5.14.*

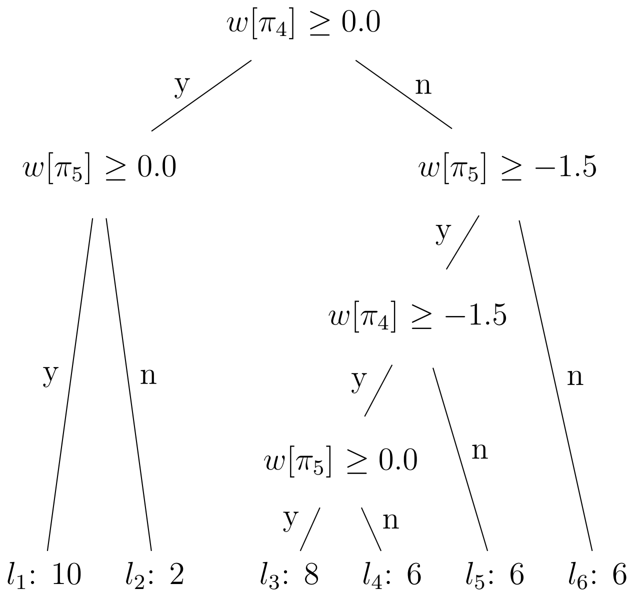


Figure 5.13: The resulting decision tree for tx_1.

	m_4, M_4	m_5, M_5	rule		expected-gas(tx_1)
l_1 :	$[0, \infty)$	$[0, \infty)$	$\Diamond\text{tx}_1 \wedge (\text{tx}_1 \prec \{\text{tx}_4, \text{tx}_5\})$	$\Rightarrow$	10
l_2 :	$[0, \infty)$	$(-\infty, -1]$	$\Diamond\text{tx}_1 \wedge (\text{tx}_1 \prec \text{tx}_4) \wedge (\text{tx}_5 \prec \text{tx}_1)$	$\Rightarrow$	2
l_3 :	$[-1, -1]$	$[0, \infty)$	$\Diamond\text{tx}_1 \wedge (\text{tx}_4 \prec \text{tx}_1) \wedge (\text{tx}_1 \prec \text{tx}_5)$	$\Rightarrow$	8
l_4 :	$[-1, -1]$	$[-1, -1]$	$\Diamond\text{tx}_1 \wedge (\{\text{tx}_4, \text{tx}_5\} \prec \text{tx}_1) \wedge$ $(\text{tx}_4 \prec \text{tx}_5) \wedge (\text{tx}_5 \prec \text{tx}_4)$	$\Rightarrow$	$\perp$
l_5 :	$(-\infty, -2]$	$[-1, \infty)$	$\Diamond\text{tx}_1 \wedge (\text{tx}_4 \prec \text{tx}_1) \wedge (\text{tx}_4 \prec \text{tx}_5)$	$\Rightarrow$	7.2
l_6 :	$(-\infty, -1]$	$(-\infty, -2]$	$\Diamond\text{tx}_1 \wedge (\{\text{tx}_4, \text{tx}_5\} \prec \text{tx}_1)$	$\Rightarrow$	6

Figure 5.14: The gas usage rules generated for tx_1.

Notice that no permutation can satisfy the rule l_4 and we assign the undefined $\perp$ average usage to it. We exclude such rules from consideration in the rest of the algorithm.

Theorem 9. *Assume there are no nonce dependencies and the neighborhood of every transaction is known. Consider the following stochastic process: We start with an empty sample set. In each step, we uniformly sample a new permutation of the transactions in* Pool *and add it to Π. The process stops at the moment T when the sample set Π becomes sufficient. Then, for any $\varepsilon \in (0, 1)$, the following holds:*

$$\mathbb{P}\left[T < \Delta! \cdot \log \frac{|\mathsf{Pool}| \cdot (\Delta + 1)!}{\varepsilon}\right] \geq 1 - \varepsilon.$$

Proof of Theorem 9. For a transaction tx, the probability that an induced permutation π^* does not appear after one step is $1 - \frac{1}{(|\text{nbhd}(\text{tx})|+1)!} \leq 1 - \frac{1}{(\Delta+1)!}$. By the independence of trials, the probability that the induced permutation remains unseen after r steps is $\left(1 - \frac{1}{(\Delta+1)!}\right)^r \leq \exp\left(-\frac{r}{(\Delta+1)!}\right)$. As for each tx there are at most $(\Delta + 1)!$ induced blocks, by the union bound, the probability that there exists a transaction tx such that the induced permutation π^* does not appear after r steps is at most $\mathbb{P}[T \geq r] \leq |\mathsf{Pool}| \cdot (\Delta + 1)! \cdot \exp\left(-\frac{r}{(\Delta+1)!}\right)$. After plugging in $r = (\Delta + 1)! \cdot \log\left(\frac{|\mathsf{Pool}|\cdot(\Delta+1)!}{\varepsilon}\right)$, and rearranging the terms, we obtain the desired result.

□

5.7 Runtime and Revenue Results on Ethereum

Implementation. We implemented our approach in Python 3 and Rust. Our tool is open-access and available online. We used Anvil [87] and Erigon [88] to execute Ethereum transactions in our samples and to obtain their gas usage. Moreover, we used Linfa [89] to generate decision trees and relied on SCIP [90] to solve Integer Linear Programming (ILP) instances.

Machine. All experimental results were obtained on an Intel Xeon Gold 5115 Machine (2.40GHz, 16 cores) with 64 GB of RAM, running Ubuntu 22.04.

Benchmarks. As our benchmark suite, we collected a dataset of real-world mempools, i.e. unmined transactions, that were available on the Ethereum network before each block. Our data spans 50,000 blocks, numbered 21800000 to 21849999, corresponding to approximately one week of activity on Ethereum in the period from February 8, 2025, 06:27:11 (UTC) to February 15, 2025, 06:18:35 (UTC). This data was gathered from Blocknative [91].

Samples and Runtime. For each of the 50,000 blocks, we generated 300 sample permutations and limited our solver's execution time to 12 seconds, i.e. Ethereum's block time. We note that the results were obtained on a relatively modest machine. The testing and decision-tree generation steps of our algorithm are parallelized, thus increasing the computational power would allow one to cover more samples for each block. We expect real-world miners to have much higher computational power than us.

Baselines. For each block, we compare the total tip revenue obtained by our approach with the real-world block that was mined and added to the Ethereum blockchain. In the experiments, we observed that some Ethereum miners routinely miss lucrative transactions when forming their blocks. This might be due to a suboptimal block formation strategy or network connectivity issues that prevented the miner from seeing the transactions, e.g. due to the censorship of the Ethereum network in some countries. Thus, it is conceivable that the miner might not have the same mempool as us. We believe this does not affect the fairness of the comparison since the onus is on the miners to ensure they have reliable connectivity to the network. However, to demonstrate that our improvements are not merely due to better connectivity, as a second baseline, we used Ethereum's reference implementation and applied it to the exact same mempools as our algorithm. The reference implementation sorts transactions greedily based on their effective gas tip value, prioritizing transactions with higher tips for miners and ordering same-sender transactions by nonce [87, 92].

Table 5.1: Improvements obtained by our approach compared to real-world miners and the Ethereum reference implementation.

Metric	Improvement compared to Real-World Miners	Improvement compared to Reference Implementation
Number of improved blocks	45,609 (91.22%)	38,505 (77.01%)
Total increase in tip revenue over 50,000 blocks	1,204,475 USD	863,385 USD
Annualized increase in tip revenue	63,357,892 USD	45,416,764 USD
Average earning increase per block	24.1 USD	17.3 USD
Average improvement percentage per block	73.45%	18.56%

Revenue Gains. The revenue gains obtained by our algorithm are summarized in Table 5.1 and Figure 5.15. Our approach outperformed real-world miners in 45,609 of the 50,000 blocks (91% of the cases), increasing the total revenue over all blocks by **1,204,475 USD**[1]. On average, this represents an increase of nearly 24.1 USD per block, which translates to an annualized improvement of around **63 million USD** . Our approach obtained significant gains compared to the Ethereum reference implementation, as well. It outperformed the reference implementation in 38,505 blocks (77% of the cases), yielding an additional revenue of **863,385 USD** over our 50,000 benchmark blocks. This translates to approximately **45 million USD** annually. On average, our algorithm earned almost 17.3 USD more per block. The data in Table 5.1 should be taken with the usual grains of salt: (i) Ether's value and its exchange rate to USD are highly volatile and unpredictable, and (ii) the reported numbers are the sums of improvements over individual blocks. Figures 5.16 and 5.17 and provide histograms of the gained revenues in USD. Figures 5.18 and 5.19 show

[1] We used the ETH/USD conversion rates provided by [93].

Table 5.2: Pairwise differences in annualized revenue across sample sizes of 50, 100, 200, and 300 in thousands of USD.

Sample Size	50	100	200	300
50	0	-1457.53	-2910.24	-3253.29
100	1457.53	0	-1452.71	-1795.76
200	2910.24	1452.71	0	-343.047
300	3253.29	1795.76	343.047	0

similar histograms based on the percentage of improvement obtained in each block.

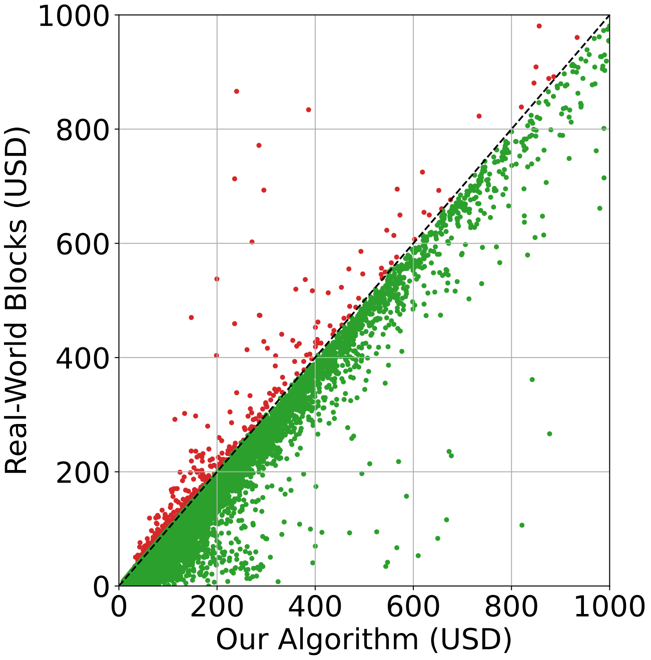


Figure 5.15: Comparison of the tip revenues obtained by our algorithm (x-axis) and the real-world blocks added to the Ethereum blockchain (y-axis). Each point corresponds to one block. Green points (91.22%) are the blocks on which our algorithm obtained a higher revenue than real-world miners. For clarity, we excluded 265 data points.

Sample Size Sensitivity. To investigate the effect of sample size on our approach, we reran our experiments with sample sizes of 50, 100, 200, and 300. The improvement in revenue slows as sample size increases. We observed that the marginal gain from 200 to 300 samples is less pronounced. See Table 5.2 for details. Moreover, the average neighborhood size is 1.21, which, according to Theorem 9, indicates that relatively small sample sizes suffice.

Runtime Performance. In practice, executing one block sample took 2.42 seconds on average. The end-to-end processing took an average of 5.7 seconds (without parallelization). Moreover, the pure ILP solving time (Step 5) was only 0.4 seconds. Refer

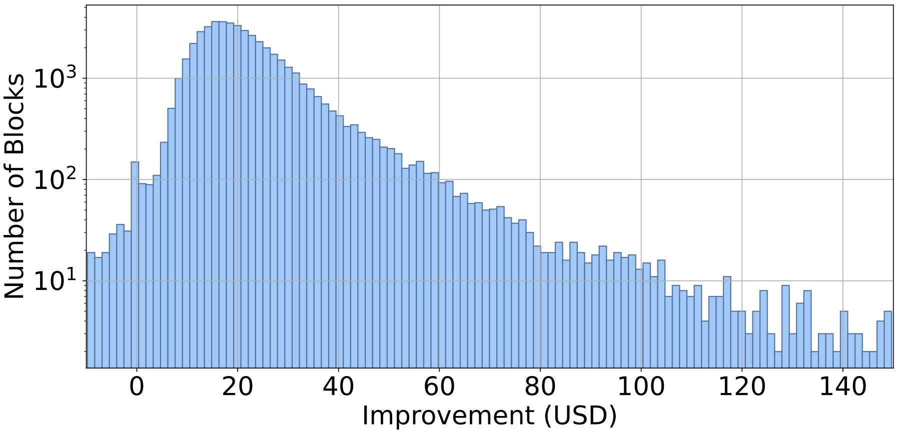


Figure 5.16: Histogram of the transaction fee improvements obtained over each block compared to real-world miners (in USD). The y-axis is in logarithmic scale. For clarity, 411 data points outside the range are omitted.

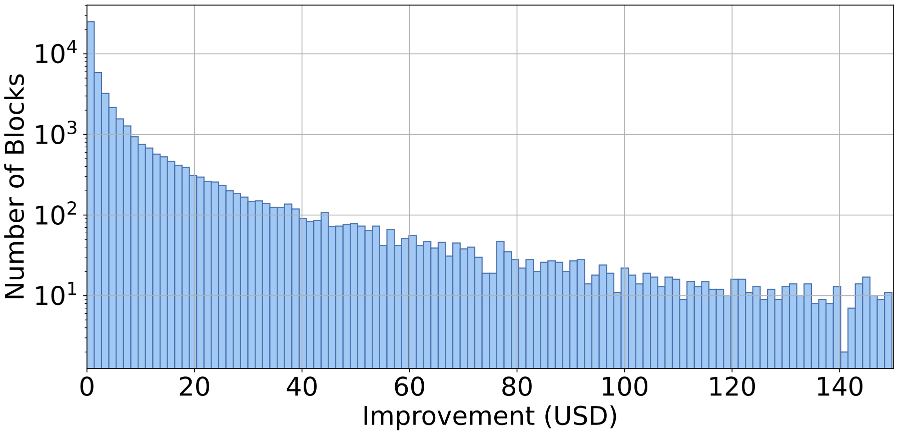


Figure 5.17: Histogram of the transaction fee improvements obtained over each block compared to the Ethereum reference implementation (in USD). The y-axis is in logarithmic scale. For readability, 868 blocks with improvements over 150 USD are omitted from the plot.

to Figures 5.20 for details. We emphasize that steps 2-4 can be parallelized, allowing for runtime reduction with additional computational power. The ILP performance depends primarily on instance sparsity rather than mempool size. We observed that increasing the number of transactions in the mempool leads to ILP instances that are easier and faster to solve. See Figure 5.21 for details.

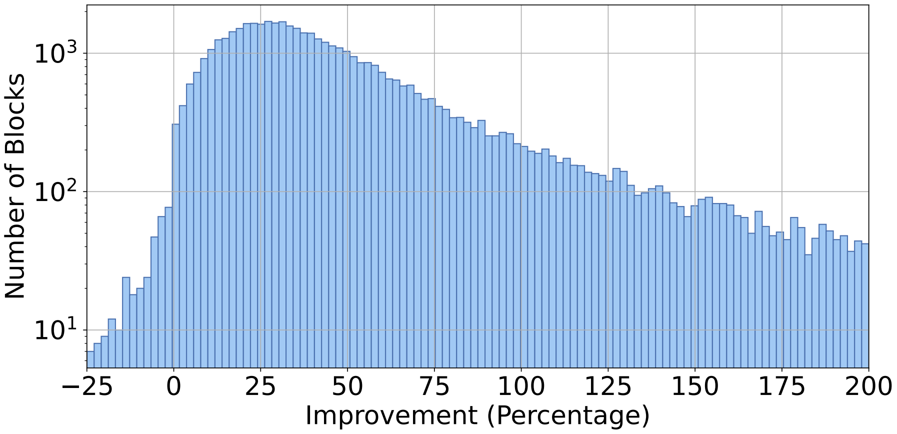


Figure 5.18: Histogram of the transaction fee improvements obtained over each block compared to real-world miners (in percentages). 608 points outside the range omitted for clarity.

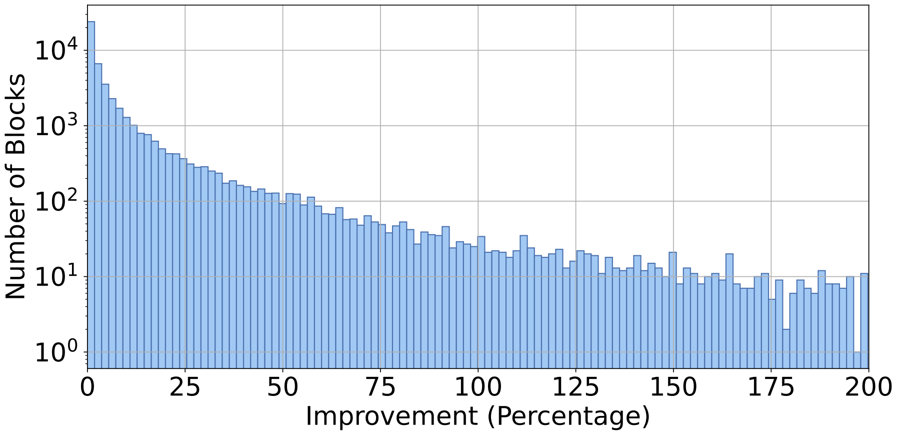


Figure 5.19: Histogram of the transaction fee improvements obtained over each block compared to the Ethereum reference implementation (in percentages). The y-axis is in logarithmic scale. 2021 points outside the range omitted for clarity.

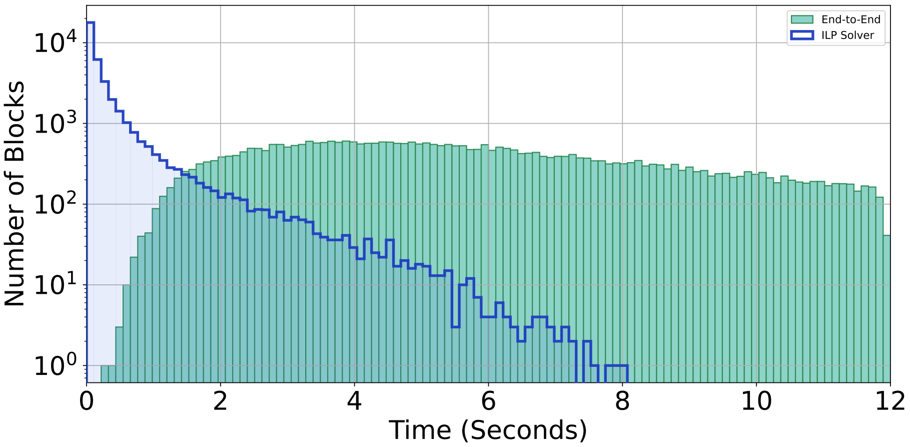


Figure 5.20: Histogram of the end-to-end (green) and ILP solver (blue) processing times per block. The y-axis is in logarithmic scale.

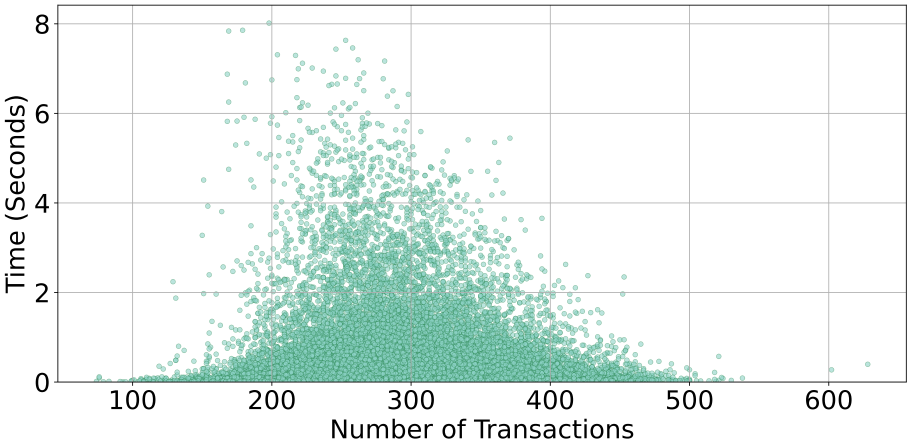


Figure 5.21: Scatter of ILP solver time (seconds) versus number of transactions per mempool, with sample size fixed at 300.

Chapter 6

Optimal Routing for Decentralized Exchanges

This chapter originally appeard in the following publication:

- S. Farokhnia, S. Novozhilov, S. Safaei, J. Shen. *Hermes: Scalable and Robust Structure-Aware Optimal Routing for Decentralized Exchanges*. IEEE International Conference on Blockchain, Blockchain 2025.

6.1 Introduction

Background. Decentralized exchanges (DEXs) have transformed the financial landscape by enabling transparent, permissionless token trading on blockchains. These platforms rely on smart contracts called liquidity pools. Each pool allows for the trading of two tokens, with the exchange price dynamically calculated by an automated algorithm based on the available liquidity. Uniswap, the leading DEX on Ethereum, features over 400,000 pools and tokens, supporting an average daily trading volume of $1.3 billion USD since 2024. However, fragmentation across DEXs and the rapid growth in the number of tokens significantly complicate the search for optimal exchange rates, particularly when no direct trading pair exists. As a result, it often requires a sequence of trades across multiple pools, a challenge known as *routing*. Routing is typically modeled as a shortest path problem on a given graph of tokens. Existing algorithms have significant drawbacks: scalable approaches often lack guarantees for route validity, while robust methods struggle with the scale and dynamic nature of modern decentralized exchanges.

Our Contributions. In this chapter, we address the problem of optimal routing on DEXs. We demonstrate that by leveraging the structural properties of this graph, in particular its *treewidth*, it is possible to reconcile scalability with robustness. On the theoretical side, we adapt a parameterized algorithm utilizing treewidth to handle the dynamic setting of DEXs, where pools frequently change. We show that our approach achieves improved time complexity over existing methods and additionally provides a formal guarantee on the quality of the computed routes. We present empirical analysis on real Uniswap data to demonstrate the suitability of a parameterized online algorithm. Furthermore, we have implemented this algorithm in a free and open-source tool called Hermes and compared it with existing methods. On small instances where both tools produced results, Hermes reduced the average runtime by four orders of magnitude, from 2.81 seconds to 0.0002 seconds. Notably, Hermes is the only tool capable of computing routes in the presence of 100,000 tokens. It achieves an average runtime of 0.19 seconds, while other approaches fail to complete within the allotted time.

6.2 Exchange Rates as a Routing Problem

Decentralized Finance (DeFi). Smart contracts have enabled the development of a new ecosystem that offers advanced, composable financial functions beyond basic token transfers on the blockchain, now collectively referred to as *decentralized* *finance* (DeFi) [94]. DeFi has grown rapidly due to its transparent, trustless, and programmable financial services. Currently, the total value locked (TVL)[1] in DeFi on Ethereum alone exceeds 65.9 billion USD [95, 96].[2]

Decentralized Exchanges (DEX). Decentralized exchanges (DEXs) are a cornerstone of the DeFi ecosystem, enabling the permissionless trading of digital assets. Among the different DEX types, *Automated Market Makers* (AMMs) have emerged as the most widely adopted by nearly every metric [97]. Central to AMMs is the concept of a *liquidity pool*: a smart contract that holds reserves of two (or more) tokens, enabling users to trade between them without intermediaries. The exchange rate between two tokens within a liquidity pool is autonomously determined by a function of the current reserves. Uniswap is the leading DEX by TVL on Ethereum. Since 2024, Uniswap has hosted an average daily trading volume of 1.3 billion USD, with an all-time cumulative trading volume reaching 2.26 trillion USD [95].

Routing in Decentralized Exchanges. A fundamental challenge within the DEX ecosystem is discovering optimal exchange rates, a problem necessitated by the decentralized and fragmented liquidity landscape. Often, a direct trading pair between two assets does not exist, requiring a trader to perform a sequence of trades across several liquidity pools, a process known as *routing*. Routing can be modeled as a graph problem on $G = (V, E, w)$, where the vertices V represent tokens, the edges E correspond to liquidity pools connecting pairs of tokens, and the weights w encode the spot exchange rates available for trading between those tokens. A route is a sequence of tokens whose cumulative edge weights determine the final exchange rate. The goal is to find the route between two tokens that yields the best price. A well-known special case of routing is *arbitrage*, where a trader earns risk-free profit by exploiting price discrepancies for the same asset across different pools. This problem is commonly modeled as finding a negative-weight cycle in the aforementioned graph [97, 94]. Arbitrages have been studied exhaustively in the literature on DEX due to their risk-free profits[96]. However, the reported arbitrages constitute only a small fraction of trades, accounting for approximately 11.71% of daily volume on Uniswap [98, 95].

Algorithmic Challenges in Routing. While financial arbitrage has been extensively studied in both traditional and decentralized markets [96, 97], research focused on optimal routing within decentralized exchanges remains comparatively limited. The existing body of work reveals two primary algorithmic challenges. First, our study shows that current routing algorithms scale poorly as the number of assets increases. This issue is particularly acute given the operational constraints of modern blockchains, such as Ethereum's 12-second block time, and the dynamic nature of

[1] Total value locked refers to the aggregate value of all assets deposited in smart contracts.

[2] All prices in this chapter are reported as of 15-06-2025.

the ecosystem, which features continuously changing prices. Second, the presence of arbitrage opportunities, which manifest as negative-weight cycles in the graph-based problem formulation, renders standard shortest-path algorithms such as Bellman-Ford incapable of finding optimal routes. This situation has led to a dichotomy in existing solutions: they are either general-purpose algorithms that guarantee optimality but are often too slow for practical on-chain execution, or they rely on heuristics that are faster but offer no formal guarantees on the quality of the route.

Our Focus. In this chapter, we bridge this gap by designing a *parameterized algorithm* tailored to the structural properties of DEX transaction graphs [99]. The design of such algorithms follows a paradigm that diverges from traditional runtime analysis. Instead of measuring performance solely against the input size, a parameterized algorithm's efficiency is also analyzed with respect to an additional structural property of the input, known as a *parameter.* For graphs, a common and powerful parameter is *treewidth*, which measures how closely a graph resembles a tree. Our central thesis is that real-world DEX transaction graphs exhibit low treewidth. By exploiting this structural property, we have developed an algorithm that is both *theoretically sound*, due to its formal complexity analysis, and *practically efficient*, a claim we substantiate through extensive experiments on real-world DEX data. This approach strikes a crucial balance, achieving provable optimality without sacrificing the performance required in the demanding DEX environment.

6.3 Graph Routing via Treewidth

6.3.1 Formulating DEX prices as a graph problem

Automated Market Makers. An *Automated Market Maker* (AMM) is a protocol for decentralized trading that replaces traditional order books with deterministic pricing algorithms [94]. In an AMM, prices are set by a mathematical function of the current token reserves in each liquidity pool. The most widely used class of AMMs is the *Constant Function Market Maker* (CFMM), of which Uniswap is a prominent example. In Uniswap V2, the CFMM uses the constant product formula, which maintains a constant product of the reserves of two tokens u and v in a liquidity pool: $R_u \cdot R_v = k$, where k is a fixed constant. Here, R_u and R_v represent the current quantities (reserves) of tokens u and v held in the pool, respectively. This means that any trade will adjust the reserves of both tokens while keeping their product constant. The spot price from token u to token v is determined by the ratio of their reserves, specifically $\text{price}(u \to v) = \frac{R_v}{R_u}$. Uniswap V3 allows liquidity providers to concentrate capital within custom price ranges, increasing efficiency. Uniswap V4 introduces *hooks* for customizable trading strategies and fee structures [100]. All three versions are active on Ethereum.

For the scope of this chapter, these versions share two pricing notions. The first is the *spot price*, which depends on the current reserves and indicates the current market price. The second is the *trade price*, which also accounts for the size of the trade. The applications of both concepts have been thoroughly studied in the literature [96]. In this chapter, we focus on spot prices.

Modeling the DEX Ecosystem as a Graph. To analyze and optimize trade routing across a decentralized exchange ecosystem, we employ the abstraction of a weighted directed graph $G = (V, E, w)$. In this model, the set of vertices V represents the universe of available tokens. A directed edge $(u, v) \in E$ exists if there is a liquidity pool capable of direct trading from token u to token v. A path in this graph, which is a sequence of connected vertices, therefore corresponds to a multi-hop trade that converts a source token into a destination token through one or more intermediaries.

Shortest Path Formulation. The objective in trade routing is to find a path that maximizes the product of exchange rates[3]. Since standard shortest path algorithms are designed to minimize a sum of weights, we align our problem by defining the weight of an edge (u, v) as the negative logarithm of the exchange rate: $w(u, v) = -\log(\text{price}(u \to v))$. In the case of multiple liquidity pools between two tokens, we choose one pool maximizing the exchange rate $\max \text{price}(u \to v)$, thus avoiding the necessity of considering the multi-edge graphs. This weight assignment converts the maximization of a product into the minimization of a sum, thereby reformulating the optimal routing task as a Single-Source Shortest Path (SSSP) problem. A critical consideration in this model is the presence of negative-weight cycles, which correspond to arbitrage opportunities: trading paths that start and end with the same token and yield a profit. Such cycles can lead to infinitely cheap paths, a complication that must be handled by the chosen algorithm.

Graph Notation and Terminology. To develop an algorithm that leverages the specific topology of the DEX graph, we must first establish a formal language for discussing its local properties. We define the *neighborhood* of a vertex v, denoted $N(v)$, as the set of all vertices directly accessible from it $N(v) := \{u \mid (v, u) \in E\}$. Furthermore, for any subset of vertices $S \subseteq V$, the *subgraph induced by* S, denoted $G[S]$, consists of the vertices in S and all edges from the original graph that connect any two vertices in S: $G[S] := (S, E \cap (S \times S))$.

Tree Decompositions and Treewidth. The efficiency of our algorithm hinges on a structural parameter known as *treewidth*, which quantifies how "tree-like" a graph

[3] We assume rates are adjusted for all transaction fees.

is. Formally, the treewidth of a graph $G = (V, E)$ is defined via a *tree decomposition*. A tree decomposition is a pair $(\mathcal{T}, \{X_i \mid i \in I\})$, where $\mathcal{T} = (I, F)$ is a tree and each X_i (called a bag) is a subset of V, satisfying:

1. The union of all bags equals V; i.e., $\bigcup_{i \in I} X_i = V$.

2. For every edge $(u, v) \in E$, there is at least one bag X_i containing both u and v.

3. For any vertex $v \in V$, the set of bags containing v forms a connected subtree in $\mathcal{T}$.

The *width* of a tree decomposition is $\max_{i \in I} |X_i| - 1$. The *treewidth* of a graph G, denoted $tw(G)$, is the minimum width over all possible tree decompositions of G. A low treewidth indicates a structure amenable to efficient dynamic programming, a property we exploit extensively. For more details, refer to Section 2.5.2.

Chordal Graphs and Chordal Completion. Our algorithm operates not on the original graph G, but on a *chordal supergraph* of it. A graph is chordal if every cycle of four or more vertices has an edge connecting two non-consecutive vertices (a chord). While most graphs are not chordal, any graph G can be transformed into one by adding a set of “fill-in” edges to create a *chordal completion*, $\widehat{G}$. A key result in graph theory states that the treewidth of a graph is intrinsically linked to its best chordal completion: $tw(G) = \min_{\widehat{G}} \omega(\widehat{G}) - 1$, where the minimum is taken over all chordal completions of G and $\omega(\widehat{G})$ is the size of the largest clique in $\widehat{G}$. This relationship is fundamental to our approach.

All-Pairs Shortest Paths via Directed Path Consistency. The core of our algorithm is an efficient method for solving the All-Pairs Shortest Paths (APSP) problem on graphs with low treewidth. The method relies on enforcing *Directed Path Consistency* (DPC) using a PEO. A weighted graph is DPC with respect to an ordering $\pi = (v_1, \ldots, v_n)$ if for every triple of vertices v_i, v_j, v_k with $i < j < k$, the path cost from v_i to v_k is no greater than the cost of the path through v_j; i.e., $w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k)$.

Enforcing this property on an arbitrary graph with an arbitrary ordering takes $O(n^3)$ time. However, by using the PEO of a chordal completion $\widehat{G}$, the process can be dramatically accelerated. The algorithm iterates backward through the PEO (from v_n to v_1), and at each step v_k, it only needs to check for path updates between pairs of neighbors of v_k that appear earlier in the ordering. Because a PEO ensures these neighbors form a clique of size at most $\omega(\widehat{G})$, the complexity of enforcing DPC is reduced to $O(n \cdot \omega(\widehat{G})^2)$, which is equivalent to $O(n \cdot tw(G)^2)$.

The advantage of the DPC property is that all shortest paths become bitonic: any shortest path from u to v consists of a segment where vertices decrease in the PEO order, followed by a segment where they increase. This structure allows SSSP queries to be answered efficiently in two passes over the PEO array, see Algorithm 9 for details.

6.3.2 The Algorithm Overview

Our proposed algorithm transforms a general graph into a structure where shortest path queries can be answered with remarkable efficiency. This transformation is achieved through a multi-phase process, illustrated in Figure 6.1. The first three phases constitute an offline preprocessing stage, which is performed only once. The final phase is the online query stage, designed for rapid, repeated execution.

The main idea is to construct a chordal supergraph leveraging graph's tree decomposition. The chordal graph admits a Perfect Elimination Ordering (PEO). This ordering is then used to enforce Directed Path Consistency (DPC). The DPC property is used to answer the Single-Source Shortest Path (SSSP) queries efficiently. The algorithm is designed to handle graphs with negative-weight cycles, ensuring that the results are correct even in the presence of such cycles. The entire process is summarized in Algorithm 6.

Algorithm 6: Unified Algorithm for Treewidth-Based Batch SSSP Queries

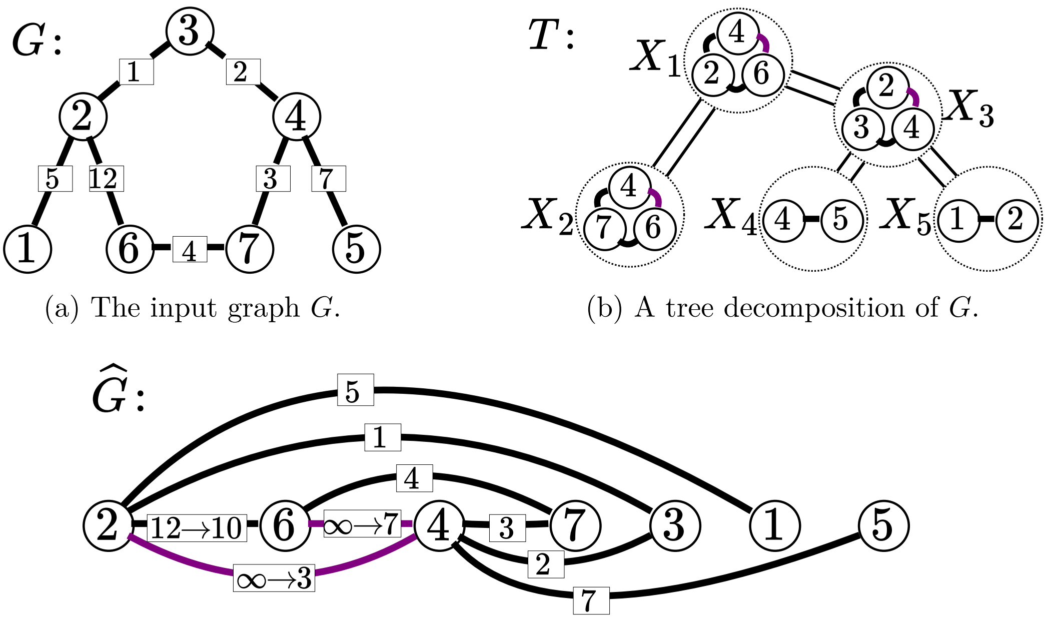
```
Input: A weighted graph $G = (V, E, d)$, a set of source vertices $Q = \{s_1, \dots, s_q\}$
Result: A list of SSSP distance arrays $(D[s_1], \dots, D[s_q])$, where each $D[s_i]$ contains the
        distances from source $s_i$ to every vertex in $V$
// Structural Preprocessing
$T \leftarrow$ ComputeTreeDecomposition($G$);
$(\widehat{G}, \pi) \leftarrow$ ComputeCompletionAndPEO($T$);
// Weights Preprocessing
$d^* \leftarrow$ EnforceDPC($\widehat{G}, \pi, d$);
// Process each SSSP query
for $i \leftarrow 1$ to $q$ do
    $D[s_i] \leftarrow$ QuerySSSP($\widehat{G}, d^*, s_i$);
end
```

The correctness and efficiency of this framework are formalized in Theorem 10.

Theorem 10 (Complexity and Correctness)**.** *Let $G = (V, E, d)$ be a weighted graph with $n = |V|$ vertices and treewidth $tw(G)$. The algorithm described in Algorithm 6 has an overall preprocessing time of $O(n \cdot tw(G)^2)$ and answers each SSSP query in $O(n \cdot tw(G)^2)$ time.*

For each source vertex $s_i \in Q$, the resulting distance array D_i has the following properties for any target vertex $t \in V$:

- *If there are no negative-weight cycles on any path from s_i to t, then $D[s_i][t]$*



(a) The input graph G.

(b) A tree decomposition of G.

(c) The chordal completion $\widehat{G}$ with fill-in edges ($\widehat{E}$) added.

Figure 6.1: An illustration of the preprocessing pipeline for the treewidth-based routing algorithm.

stores the correct shortest path distance from s_i to t in the original graph G.

- *If there is a negative-weight cycle on a path from s_i to t, the corresponding distance $D[s_i][t]$ will be less or equal to the shortest simple path (a path with no repeated vertices) distance from s_i to t in G.*

Example 22 (Input Graph)**.** *We illustrate the entire pipeline using the graph from Figure 6.1. The process begins with the graph G in Figure 6.1(a). We chose the weights to be symmetric $d(u, v) = d(v, u)$ for simplicity of presentation, generally the algorithm does not require this. The graph has 7 vertices $\{v_1, \ldots, v_7\}$, the vs are omitted in the figure for clarity. The edges has the following weights: $d(v_1, v_2) = 5$, $d(v_2, v_3) = 1$, $d(v_2, v_6) = 12$, $d(v_3, v_4) = 2$, $d(v_4, v_5) = 7$, $d(v_4, v_7) = 3$, and $d(v_6, v_7) = 4$.*

Phase 1: Tree Decomposition. This initial phase (line 1 of Algorithm 6) computes a tree decomposition for the input graph G. The goal is to find a low-width decomposition, as its width, the treewidth $tw(G)$, is the primary parameter governing the complexity of the entire preprocessing stage.

A landmark result in parameterized complexity theory by Bodlaender proves that it is Fixed-Parameter Tractable (FPT).

Lemma 1 (Complexity of Tree Decomposition)**.** *For any fixed parameter k, there exists a linear-time algorithm that can determine if a graph G has treewidth at most k and, if so, produce a corresponding tree decomposition in $O(f(k) \cdot |V(G)|)$ time. [101].*

Additionally, many heuristic approaches have been developed. These heuristics have been refined and implemented in highly optimized solvers. Modern, competitive solvers can often find low-width decompositions for very large graphs arising in practice [102].

Example 23 (Tree Decomposition)**.** *Figure 6.1(b) shows a valid tree decomposition $\mathcal{T}$ of G. The largest bag is of size 3 (e.g., $X_1 = \{2, 4, 6\}$), so the treewidth is $tw(G) = 3-1 = 2$. It is straightforward to verify that (i) for each edge (v_i, v_j) in G, there exists a bag in $\mathcal{T}$ containing both v_i and v_j, and (ii) for each vertex v_i, the bags containing v_i form a connected subtree in $\mathcal{T}$.*

Phase 2: Fill-in and PEO. In this phase (line 2 of Algorithm 6), we construct a minimal chordal completion $\widehat{G}$ and a corresponding Perfect Elimination Ordering (PEO) π from the tree decomposition. The process, detailed in Algorithm 7, iterates through the tree bags from leaves to the root. In each step, it turns the current leaf bag into a clique and prepends its unique vertices (those not in its parent bag and have not been seen before) to the PEO.

Algorithm 7: ComputeCompletionAndPEO($G = (V, E), \mathcal{T}$)

```
1 $\widehat{G} \leftarrow G$, $\pi \leftarrow ()$;
2 while $\mathcal{T}$ is not empty do
3     $X \leftarrow$ leaf($\mathcal{T}$);
4     $Y \leftarrow$ parent($\mathcal{T}, X$) or $\emptyset$ if $X$ is the root;
5     $\widehat{E} \leftarrow \widehat{E} \cup \{(u, v) \mid u, v \in X, u \neq v\}$;
6     $\pi \leftarrow [(X \setminus Y) \setminus \pi] + \pi$;
7     $\mathcal{T} \leftarrow \mathcal{T} \setminus \{X\}$;
8 end
9 return $(\widehat{G}, \pi)$;
```

We summarize the key results about the correctness and complexity below.

Lemma 2 (Properties of the Completion and PEO)**.** *Let $\widehat{G}$ and π be the outputs of Algorithm 7. The following properties hold:*

(a) The graph $\widehat{G}$ is a chordal supergraph of G.

(b) The clique number of the completion satisfies $\omega(\widehat{G}) = tw(G) + 1$.

(c) The sequence π is a valid Perfect Elimination Ordering for $\widehat{G}$.

(d) The algorithm runs in $O(n \cdot (tw(G) + 1)^2)$ time.

Proof (sketch). For *(a)-(c)*, we refer to the standard properties relating tree decompositions to chordal graphs [103]. For (d), we analyze the algorithm as follows: we can assume the given tree decomposition has $O(n)$ nodes. The algorithm iterates through each node, where the dominant operation is making the bag a clique (line 5).

This step takes $O((tw(G)+1)^2) = O(tw(G)^2)$ time. The total complexity is therefore $O(n \cdot (tw(G)+1)^2)$. □

Example 24 (Fill-in and PEO)**.** *We apply Algorithm 7 to $\mathcal{T}$ to get the chordal completion $\widehat{G}$ and a PEO π. Processing the bags from the leaves of $\mathcal{T}$ inwards yields the chordal graph in Figure 6.1(c). The required fill-in edges (shown in purple) are (v_2, v_4) and (v_4, v_6) as the pairs of vertices are present in the bags X_1 and X_3. A valid PEO $\pi = (v_2, v_6, v_4, v_7, v_3, v_1, v_5)$ that can be generated from this process by trimming the bags in the following order: $X_4 \to X_5 \to X_3 \to X_2 \to X_1$.*

Phase 3: Enforcing the DPC. This phase executes the ‘EnforceDPC‘ function (Algorithm 8), which corresponds to line 3 of the main algorithm. It takes the chordal graph $\widehat{G}$, its PEO π, and the original distances d to produce a new distance matrix d^*.

This procedure does not compute the final all-pairs shortest paths. Instead, it modifies the edge weights to satisfy the Directed Path Consistency (DPC) property relative to the PEO π. Specifically, after ‘EnforceDPC‘ terminates, for any path $v_i \to v_k \to v_j$ where v_k is a common neighbor of v_i and v_j that appears later in the PEO (i.e., $i, j < k$), the triangle inequality $d^*(v_i, v_j) \leq d^*(v_i, v_k) + d^*(v_k, v_j)$ is guaranteed to hold.

Algorithm 8: EnforceDPC($\widehat{G} = (V, \widehat{E}), \pi, d$)

```
$d^*(u,v) \leftarrow d(u,v)$ for all $(u,v) \in E$ and $d^*(u,v) \leftarrow \infty$ for all $(u,v) \in \widehat{E} \setminus E$;
$\pi = (v_1, v_2, \ldots, v_n)$ is the PEO of $\widehat{G}$;
for $k \leftarrow n$ down to 1 do
    foreach pair of neighbors $v_i, v_j$ of $v_k$ in $\widehat{G}$ with $i < k$ and $j < k$ do
        $d^*(v_i, v_j) \leftarrow \min(d^*(v_i, v_j), d^*(v_i, v_k) + d^*(v_k, v_j))$;
    end
end
return $d^*$;
```

Lemma 3 (Complexity of Enforcing DPC)**.** *The ‘EnforceDPC‘ procedure (Algorithm 8), which applies Directed Path Consistency on the chordal graph $\widehat{G}$ using its PEO, has a time complexity of $O(n \cdot tw(G)^2)$.*

Example 25 (Enforcing DPC)**.** *This is the core computational step (Algorithm 8). We initialize d^* with the weights from G, setting $d^*(2, 4) = \infty$ and $d^*(4, 6) = \infty$. The algorithm iterates backward through the PEO. The only updates that change a distance* *value occur for the following vertices:*

$$\begin{aligned}
\textit{For } v_3: \quad d^*(v_2, v_4) &\leftarrow \min(\infty, d^*(v_2, v_3) + d^*(v_3, v_4)) \\
&= \min(\infty, 1 + 2) = 3 \\
\textit{For } v_7: \quad d^*(v_4, v_6) &\leftarrow \min(\infty, d^*(v_4, v_7) + d^*(v_7, v_6)) \\
&= \min(\infty, 3 + 4) = 7 \\
\textit{For } v_4: \quad d^*(v_2, v_6) &\leftarrow \min(12, d^*(v_2, v_4) + d^*(v_4, v_6)) \\
&= \min(12, 3 + 7) = 10
\end{aligned}$$

Phase 4: SSSP Queries. The final query phase (lines 4–5) processes the batch of SSSP queries. For each source s_i in the query set Q, the algorithm calls the QuerySSSP procedure (Algorithm 9). The algorithm is an adaptation of the `Min-path` procedure [104, 105].

Algorithm 9: QuerySSSP($\widehat{G} = (V, \widehat{E}), \pi, d^*, s$)

```
Initialize distance array $D[v] \leftarrow \infty$ for all $v \in V \setminus \{s\}$, $D[s] \leftarrow 0$;
$D[s] \leftarrow 0$;
Let $\pi = (v_1, \ldots, v_n)$ be the PEO. Let $s = v_{idx}$;
for $k \leftarrow idx$ down to 1 do
    foreach neighbor $v_j$ of $v_k$ in $\widehat{G}$ such that $j < k$ do
        $D[v_j] \leftarrow \min(D[v_j], D[v_k] + d^*(v_k, v_j))$;
    end
end
for $k \leftarrow 1$ to $n$ do
    foreach neighbor $v_j$ of $v_k$ in $\widehat{G}$ such that $j > k$ do
        $D[v_j] \leftarrow \min(D[v_j], D[v_k] + d^*(v_k, v_j))$;
    end
end
return $D$;
```

Lemma 4 (Complexity of SSSP Query)**.** *The* QuerySSSP *procedure has a complexity of $O(|\widehat{E}|)$, which in a chordal graph is bounded by $O(n \cdot tw(G)^2)$.*

Example 26 (SSSP Query)**.** *Suppose we are given the query $s = v_4$. We execute Algorithm 9. The distance array D, ordered by the PEO $\pi = (v_2, v_6, v_4, v_7, v_3, v_1, v_5)$, is initialized to* $[\infty, \infty, 0, \infty, \infty, \infty, \infty]$. *The updates proceed as follows:*

Pass 1 (Backward Pass):

$$\begin{aligned}
v_6 :&[3, 7, 0, \infty, \infty, \infty, \infty] \\
v_4 :&[3, 7, 0, \infty, \infty, \infty, \infty] \\
v_2 :&[3, 7, 0, \infty, \infty, \infty, \infty]
\end{aligned}$$

Pass 2 (Forward Pass):

$$\begin{aligned}
v_2 :&[3, 7, 0, \infty, 4, 8, \infty] \\
v_6 :&[3, 7, 0, 11, 4, 8, \infty] \\
v_4 :&[3, 7, 0, 3, 2, 8, 7]
\end{aligned}$$

The remaining vertices in the PEO do not produce further updates. The final distance array from source v_4, ordered by the PEO, is $[3, 7, 0, 3, 2, 8, 7]$.

6.3.3 Handling Dynamic Edge Additions

Real-world graphs are often dynamic. In our context, new liquidity pools can be added, creating new edges in the graph. A key advantage of our framework is its ability to handle many such updates efficiently. If a new edge (u, v) already exists within the pre-computed chordal completion $\widehat{G}$, the underlying structure remains valid. In this scenario, we can bypass the expensive structural preprocessing steps (lines 1-2 of Algorithm 6). Instead, we only need to update the edge weights and re-run the weights preprocessing (line 3). If the new edge is not in $\widehat{G}$, the full pipeline must be re-executed.

6.4 Experimental Results

Implementation. We implemented our algorithm in Python 3 as a tool named Hermes. Hermes is open-source and released into the public domain. Our implementation leverages the Python library NetworkX [106] for graph operations and tree decompositions. Hermes is available at `https://github.com/SanazSafaei/Hermes-Structure-Aware-Optimal-DEX-Routing`.

Benchmarks and Experimental Setting. We collected a comprehensive dataset by capturing snapshots of all Uniswap liquidity pools across 20,000 consecutive blocks, specifically from block 22,820,000 to block 22,839,999. Data were collected using the official Uniswap APIs [98]. We evaluated Hermes in comparison with the state of the art Modified Moore Bellman Ford algorithm [97], which is the only existing graph based method. For additional comparison, we also included the standard Bellman Ford algorithm as a baseline. For each block, we constructed the corresponding token graph and queried SSSPs from 100 randomly selected tokens, recording the average query time per block. To further assess scalability, we repeated this procedure for four different token set sizes: the top 100, 1,000, 10,000, and 100,000 tokens, ranked by TVL in USD. Each query was subject to a 12-second timeout, matching the average Ethereum block interval.

Treewidth. Experimental analysis shows that the average treewidth for token sets of sizes 100, 1,000, 10,000, and 100,000 is 8, 18, 38, and 71, respectively. As illustrated in Figure 6.2, while treewidth increases with the number of tokens, its growth rate is substantially slower than that of the total number of tokens. These results demonstrate the suitability of parameterized algorithms for this problem.

Dynamics of Pool Updates. For the token set sizes studied, we observed an average of 9.7 price updates per block, about 1 new token every 12 blocks, and about

1 new pool every 33 blocks, reflecting few edge weight changes and infrequent additions of vertices and edges. Figure 6.3 shows the distribution of price updates across blocks. These results show that the proposed approach for handling dynamic edges is practical for real-world pools.

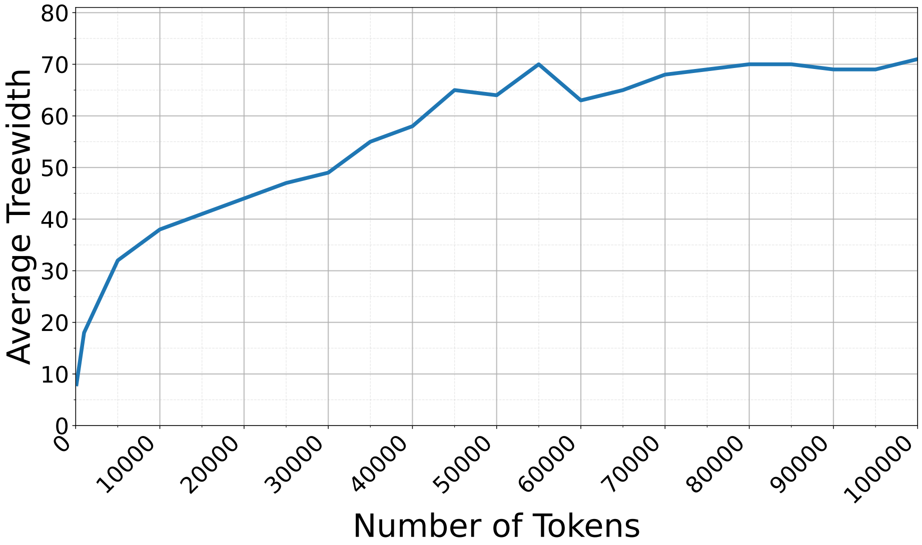


Figure 6.2: Comparison of the growth rates of the number of tokens and the treewidth.

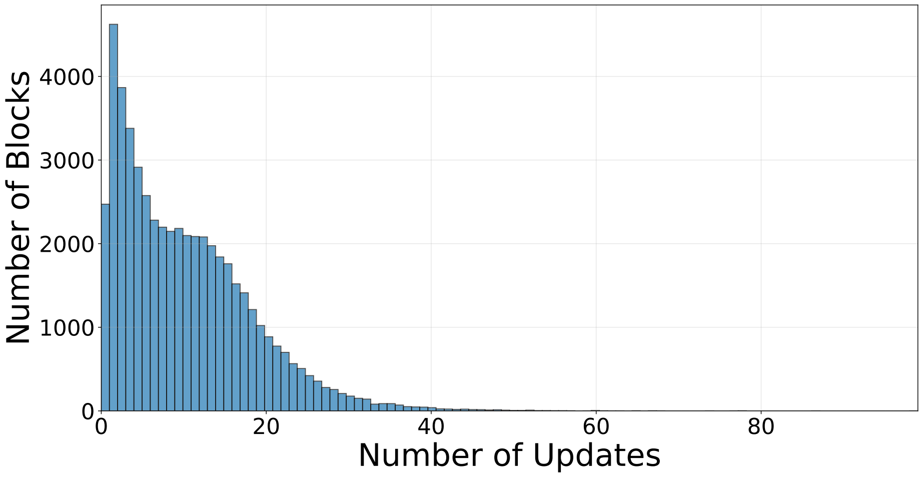


Figure 6.3: Histogram of token price updates per block.

Robustness. We measured the success rate of Hermes in finding valid routes, compared to MMBF and BF, by running 2,000,000 queries for each of the four token set sizes. Hermes successfully solved **100%** of queries, while MMBF and BF solved only **25%** and **1.4%**, respectively. The primary limitation for MMBF was frequent timeouts when the token count exceeded 100. For BF, failures on smaller token sets 100, 1,000 were mainly due to negative cycles, while timeouts dominated for larger sets 10,000 and 100,000. Tables 6.1, 6.2, and 6.3 summarize the success rates and coverage for all three algorithms.

	Hermes (Ours)					
	Solved		Failed		Timeout	
Token Set Size	#	%	#	%	#	%
100	2,000,000	100.0%	0	0.0%	0	0.0%
1,000	2,000,000	100.0%	0	0.0%	0	0.0%
10,000	2,000,000	100.0%	0	0.0%	0	0.0%
100,000	2,000,000	100.0%	0	0.0%	0	0.0%
Total	8,000,000	100.0%	0	0.0%	0	0.0%

Table 6.1: Success rates for Hermes (ours) on routing queries for four token set sizes.

	MMBF					
	Solved		Failed		Timeout	
Token Set Size	#	%	#	%	#	%
100	1,999,998	99.9%	0	0.0%	2	0.0%
1,000	0	0.0%	0	0.0%	2,000,000	100.0%
10,000	0	0.0%	0	0.0%	2,000,000	100.0%
100,000	0	0.0%	0	0.0%	2,000,000	100.0%
Total	1,999,998	25.0%	0	0.0%	6,000,002	75.0%

Table 6.2: Success rates for MMBF on routing queries for four token set sizes.

	BF					
	Solved		Failed		Timeout	
Token Set Size	#	%	#	%	#	%
100	60,000	3.0%	1,940,000	97.0%	0	0.0%
1,000	27,994	1.3%	1,972,006	98.6%	0	0.0%
10,000	10,634	0.5%	13,107	0.6%	1,976,259	98.8%
100,000	12,884	0.6%	3,627	0.1%	1,983,489	99.1%
Total	111,512	1.4%	3,928,740	49.1%	3,959,748	49.5%

Table 6.3: Success rates for BF on routing queries for four token set sizes. "Failed" indicates cases where the tool did not work due to negative cycles.

Scalability. We evaluated the runtime performance of all algorithms. MMBF completed within the time limit only for the smallest token set of 100, which precludes direct runtime comparison on larger sizes. On this benchmark, Hermes achieved an average query time of **0.0002** seconds, while MMBF required 2.8191 seconds, representing an improvement of four orders of magnitude. BF completed within the time limit for token sets of size 100 and 1,000 mainly. For queries solved by both Hermes and BF, Hermes averaged 0.054 seconds per query, compared to 0.5947 seconds for BF. It is important to note that BF frequently failed to return valid routes on 97.8% of queries owing to the presence of negative cycles, which rendered the majority of its results unusable. Hermes was the only method able to process the largest token sets, with average query times of **0.0197** seconds and **0.1942** seconds for 10,000 and 100,000 tokens, respectively. Detailed results are presented in Table 6.4.

	Hermes (Ours)		MMBF		BF	
	Runtime	Completed	Runtime	Completed	Runtime	Completed
100	0.0002s	2,000,000	2.8191s	1,999,998	0.0158s	2,000,000
1,000	0.0021s	2,000,000	TIMEOUT	0	1.1653s	2,000,000
10,000	0.0197s	2,000,000	TIMEOUT	0	1.1304s	23,741
100,000	0.1942s	2,000,000	TIMEOUT	0	0.8155s	16,511

Table 6.4: Average runtimes (in seconds) for Hermes (ours), MMBF, and BF across different token set sizes. The "Completed" column indicates the number of queries finished within the timeout; "TIMEOUT" denotes that all queries exceeded the time limit.

Limitations. We conclude this section by outlining practical limitations and considerations for deploying Hermes. Slippage arises when the execution price of a trade differs from the spot price, often as a result of market volatility or limited liquidity, and previous studies indicate that slippage can influence trading outcomes, particularly for large trades [96]. Since our algorithm relies on spot prices, similar to prior research, it encounters comparable limitations for high-volume trades; thus, this approach can benefit from preprocessing steps such as imposing additional restrictions on liquidity for a given set of pools. Executing trades also incurs transaction fees, which increase as the route includes more steps; to adjust for this, one can increase the edge prices by an upper bound estimate that reflects the cost of a single trade. Previous works have estimated such upper bounds for smart contract functions, and incorporating these estimates can prevent infeasible routes [66].

Chapter 7

Derivatives and Bitcoin Security

This chapter originally appeard in the following publication:

- S. Farokhnia, A.K. Goharshady. *Options and Futures Imperil Bitcoin's Security.* IEEE International Conference on Blockchain, Blockchain 2024.

7.1 Introduction

Background. A fundamental security assumption behind Bitcoin and all proof-of-work cryptocurrencies is that a majority of the mining power or hash rate is controlled by honest miner. It is well-known that if an adversary controls a majority of the mining power, they can perform a wide variety of malicious actions which are often called 51% attacks. Specifically, they can fork the blockchain and create a new longest chain that erases as many blocks as they wish from the current consensus chain, thus enabling double-spending. The conventional wisdom in the blockchain community is to assume that (i) having a majority of the hash power is a prerequisite for a successful attack, (ii) a majority attack is prohibitively expensive, and (iii) there is no real-world incentive for such an attack, i.e. even if a few mining pools have the majority hash rate, any such attack would effectively crash Bitcoin's price which would cause huge losses in revenue for the attacking miner. As such, it is widely believed that proof-of-work cryptocurrencies, and specifically Bitcoin, are immune to such attacks.

Our Contributions. In this chapter, we refute all three of these assumptions: (i) We show that an adversary with a small percentage of the hash power can still reliably create forks that are six blocks deep, thus fracturing the public's trust in Bitcoin and most probably causing a crash in its price; (ii) We show that a majority attack on Bitcoin would cost only 6.77 billion USD, which is much lower than the community's expectation and only 0.48 percent of Bitcoin's market cap at the time of writing. Additionally, and more worryingly, an attack with only 30% of the total hash power, which would cost a mere 2.9 billion USD, will succeed with a probability of more than 95% within 34 days. Thus, such attacks are much more affordable than expected; (iii) Finally, and most importantly, we show that it is possible for an attacker to actually benefit from the resulting crash in the value of Bitcoin, due to the vast derivatives (options and futures) market that is currently in existence. Put simply, an attacker can first short Bitcoin using widely-available derivative contracts and then intentionally perform an attack to crash the price and profit. Thus, there are real-world incentives for such attacks[1].

7.2 Overview of the Attack

Majority Attack [107, 108, 109]. If an adversary controls more than half of the hash power, she can create arbitrarily deep forks and revert as many blocks as she

[1] The prices and numerical estimates presented in this chapter are based on data from April 1, 2024. While these specific values may have changed since then, the fundamental argument and methodology remain sound.

wishes. This is often called a 51% attack[2]. See Section 2.2 for details on Bitcoin's architecture. To fork the blockchain from block B_i onwards, she can simply create new alternative blocks $B'_{i+1}, B'_{i+2}, \dots$ and continue mining on her own machine without disclosing the new alternative chain. She continues this until her alternative fork becomes longer than the network's current consensus chain, which is guaranteed to eventually happen with probability 1 since she holds a majority of the hash power. At that point, the adversary can simply publish the new chain which will immediately become the consensus chain and thus replace all the previous blocks after B_i and revert and potentially double-spend all the transactions therein. Not disclosing a valid block and instead continuing to mine to extend it is called *selfish mining* and is widely studied in the literature on game-theoretic analysis of blockchain protocols [110, 111, 112]. Specifically, it is now well-known that profiting from selfish mining is not limited to adversaries who have a majority of the hash power and can be done using a much smaller share [113, 114]. The work [113] shows that an adversary that controls a minority stake (as a mining pool) can use selfish mining to increase its mining rewards and thus attract other miner to join it until it eventually controls a majority of the computational power.

It is natural to wonder whether Bitcoin is actually vulnerable to majority attacks or any other attack that intentionally tries to revert multiple blocks. Given that the rule of thumb followed by most practitioners is to wait for 6 confirmations, a fork that goes 6 levels deep can very likely diminish the public's trust in Bitcoin and cause a crash in its market price. It is also widely accepted that a prolonged majority attack (if it happens) would be catastrophic to the cryptocurrency and can cause its downfall.

Conventional Wisdom. The conventional wisdom in the blockchain community is to assume that such block-reverting attacks are highly unlikely to happen. The reasoning goes as follows:

1. Reverting multiple blocks and specifically double-spending a transaction that has 6 confirmations requires control of a majority of the mining power;
2. Having a majority of the mining power is prohibitively expensive and requires an outlandish investment in hardware;
3. Even if a miner, mining pool or group of pools does control a majority of the mining power, they have no incentive to act dishonestly and revert the blockchain, as that would crash the price of Bitcoin, which is ultimately not in their favor, since they rely on mining rewards denominated in BTC for their income.

There is some strong yet circumstantial evidence to support these, especially the latter claim. In 2014, the Ghash.io mining pool temporarily succeeded in breaching

[2] This naming is erroneous since one does not actually need 51% of the mining power but only more than 50%. Thus, we prefer to use the term *majority attack*.

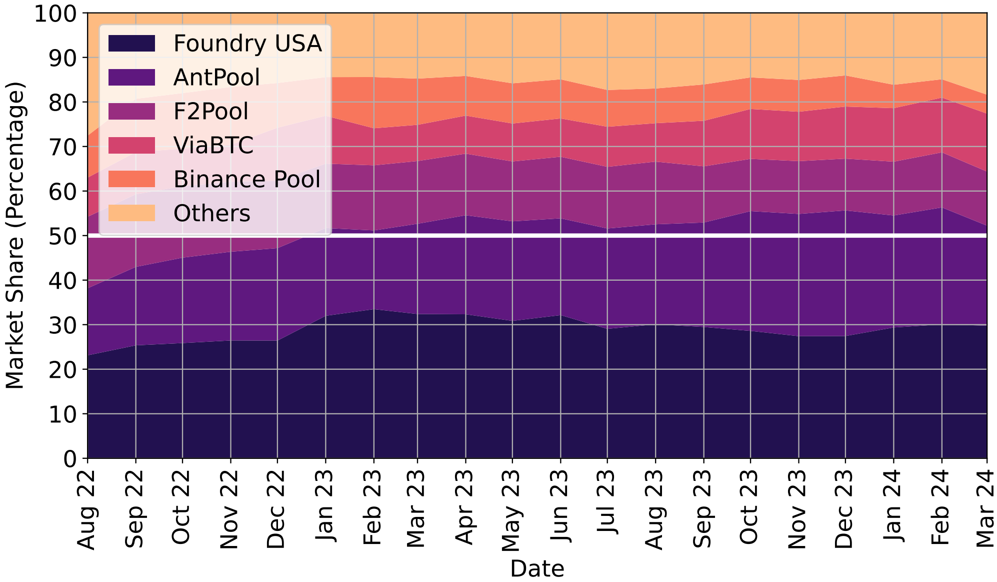


Figure 7.1: Market shares of the largest Bitcoin mining pools over time [117]

the 50% threshold and controlling more than half of the hash power [115]. However, no attack was observed. Currently, the top two mining pools, i.e. AntPool and Foundry USA, together control more than half of the computational power [116]. Indeed, centralization of hash power has been an ongoing issue and the top 2-3 mining pools have often controlled more than half of the hash rate in the past months (Figure 7.1), yet they have not attempted a majority attack. However, as we will outline in this chapter, all three assertions above are false and Bitcoin is indeed vulnerable to block reversion attacks.

Why 50% is Not Necessary. As mentioned, [113] shows that selfish mining can become profitable and help an attacker reach a majority of the mining power even when the attacker begins with a much smaller hash rate. Notwithstanding the clever attack in [113], an adversary who controls a portion q of the total hash rate, even when q is relatively small, can still attempt to perform selfish mining and create a fork that is several blocks deep. In such cases, the attacker's probability of success would be low and if she fails, she loses all the potential rewards she could have gained by honest mining. Nevertheless, if she does not care about such losses, e.g. if she has a larger incentive to act dishonestly, then she can repeat the attack until she reverts 6 blocks. Specifically, as we will see, relying on an analysis similar to that of [25] shows that an attacker who has merely 30% of the total hash rate has a success probability of more than 95% if she performs the attack for 33.7 days. An attacker with 40% of the hash rate can reach the same success probability within approximately 4.1 days and the time is reduced to only 1.9 days for an attacker with a 45% share.

The Costs of an Attack. The costs of an attack are also substantially smaller than one would expect. As we will see, a simple calculation shows that, at the time of writing, one can obtain the necessary hardware to have a majority of the hash power by spending approximately 6.77 billion USD. This is disregarding the potential discounts in bulk orders and assuming that the attacker buys the equipment rather than renting them. Crucially, although this number looks large, it is only 0.48 percent of the Bitcoin market cap at the time of writing and pales into insignificance in comparison with the trillion dollar monthly trade volume in Bitcoin derivatives. Similarly, we calculate that gaining a mining share of $20\%, 30\%$ and 40% costs only 1.69, 2.90 and 4.51 billion USD respectively. Thus, putting this together with the results mentioned in the previous paragraph, a patient attacker who is willing to wait for longer can dramatically reduce the costs of the attack[3].

Incentives for an Attack. Finally, and most importantly, the assumption that no attacker would have a financial incentive to perform such an attack is flatly wrong. By far the biggest threat is posed by the often-unregulated Bitcoin derivative contracts such as options and futures. As we will see, the monthly trade volume in Bitcoin derivatives was above 500 billion USD in 19 of the past 20 months and even reached a trillion USD in several calendar months, surpassing 2 trillion USD this past March. Thus, an attacker can first short Bitcoin and then have the incentive to intentionally crash its price.

Summary of the Attack. In short, an attacker can first use the Bitcoin derivatives market to short Bitcoin by purchasing a sufficient amount of put options or other equivalent financial instruments. She can then invest any of the amounts calculated above, depending on the timeline of the attack, to obtain the necessary hardware and hash power to perform the attack. If the attacker chooses to obtain a majority of the hash power, her success is guaranteed and she can revert the blocks as deeply as she wishes. However, she also has the option of a smaller upfront investment in hardware in exchange for longer wait times to achieve a high probability of success. In any case, as long as her earnings from shorting Bitcoin and then causing an intentional price crash outweighs her investments in hardware, there is a clear financial incentive to perform such an attack. The numbers above show that the annual trade volume in Bitcoin derivatives is more than three orders of magnitude larger than the required investment in hardware. Thus, it is possible and profitable to perform such an attack. There is also a huge derivatives market on Bitcoin with trade volumes that often exceed 1 trillion USD per calendar month and have recently even exceeded 2 trillion

[3] We are not considering electricity costs since they vary significantly in different countries. Nevertheless, we are intentionally grossly over-approximating the cost of the hardware, e.g. by considering a purchase instead of renting. Moreover, the cost of electricity would not realistically surpass the hardware costs. Additionally, one can double all of our cost estimates and the analysis and vulnerabilities still stand.

USD per month (Section 7.5).

Based on the argument above, Bitcoin options and futures imperil Bitcoin's security and its core consensus protocol. We believe there is a complete lack of awareness on the part of financial players who are issuing such derivative contracts as to their potentially destructive effect on Bitcoin. We are hoping to raise awareness about this issue so that (i) the financial institutions realize the risk that issuing Bitcoin derivatives, especially shorting vehicles such as put options, poses to their earnings, and (ii) the regulators step up to reign in the unregulated derivatives market and enforce sensible caps on these trades.

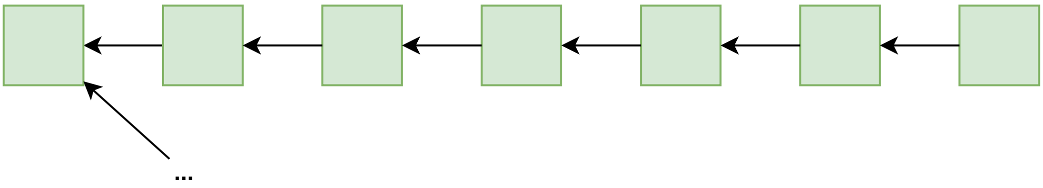
We also note that this vulnerability is, at its core, due to a disconnect between the financial players in the cryptocurrency markets on the one hand, who treat Bitcoin and other similar currencies as if they are publicly-traded stocks, and the decentralized design of proof-of-work on the other hand. If Bitcoin were a stock, the attack we are describing would be tantamount to an investor first shorting the stock using leveraged contracts for differences whose value far exceeds the company's market cap and then buying enough shares to take control of the company and intentionally crashing it. This would of course be bajpainoyearcountries due to a wide variety of insider trading regulations. However, since Bitcoin and its variants [11] use proof-of-work and do not follow a proof-of-stake protocol, they do not choose a miner randomly based on their stake as in [118, 17, 119, 120, 121, 122, 123, 124, 125]. Instead, one only needs to control a large portion of the mining power, which is much cheaper. Moreover, mining is decentralized and largely unregulated and not subject to insider trading laws.

In the following sections, we will explain the calculations that went into each part of the argument above in more detail.

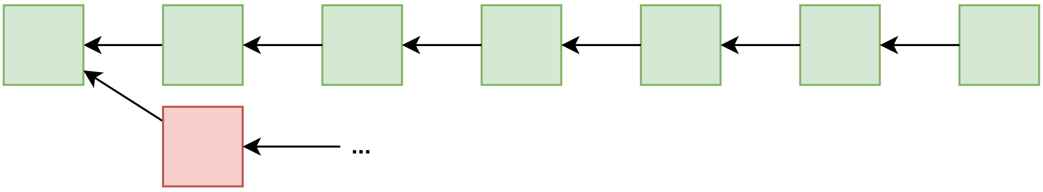
7.3 Block-reverting as a Minority Miner

Setting. In this section, we consider a malicious miner who wishes to intentionally create a fork that is 6 blocks deep, thus reverting 6 blocks of the previously-established consensus chain, in order to diminish the public's trust in Bitcoin and crash its price. Note that there is nothing special about the number 6 other than the fact that it is often used as a rule-of-thumb by the community. Our analysis below can be trivially extended to any number of blocks. We suppose that the attacker has a portion $0 < q < 1$ of the total hash power on the network. The problem is trivial if $q \geq 0.5$ since an attacker with a majority of the hash power is guaranteed to succeed in reverting any number of blocks. Thus, we focus on the case where the attacker is a minority miner, i.e. $q < 0.5$.

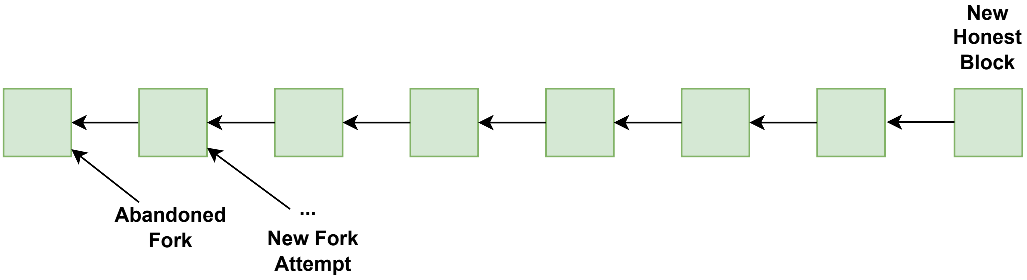
The Attack. The attack begins in the following blockchain state, where the green squares denote the current (honest) consensus chain and the dots show where the attacker is trying to add a new block to create a fork. In this state, the attacker's fork is 6 blocks behind the honest chain. Thus, we call this a -6 state.



As the attack progresses, there is a probability q that the attacker succeeds first in adding a new block, thus taking us to the -5 state below, where the attacker's chain is only 5 blocks behind the consensus chain.



On the other hand, with probability $1 - q$, the honest miner will form a new block. However, in this case, the attacker will simply abandon the previous fork and attempt a new fork that is 6 blocks deep, so we will remain at a -6 state.



Markov Chain. In general, let us model the attack by creating a Markov chain with states $\{-6, -5, -4, -3, -2, -1, 0, 1\}$, where a state $-i$ denotes that the attacker's fork is i blocks behind the consensus chain. An edge from the state i to the state j is labeled by the probability of going *directly* from i to j. The analysis above shows that the state -6 should have a transition to -5 with probability q and another transition to itself with probability $1-q$. Similarly, it is easy to see that -5 should have a transition to -4 with probability q, corresponding to the case where the attacker finds the next block and reduces the distance between the two chains, and another transition to -6 with probability $1 - q$ corresponding to the case where the honest miner mine a new block.

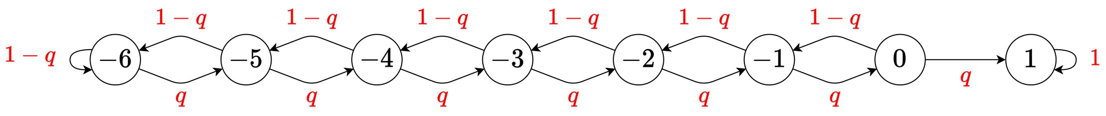


Figure 7.2: The Markov chain modeling variant A of the attack.

Variant A. We consider two variants of the attack. In variant A, the attacker only publishes her chain when it becomes strictly longer than the consensus chain, i.e. when the Markov chain reaches state 1. At this point, the attack is successful. Thus, this variant can be modeled by the Markov chain in Figure 7.2. From each state, there is a probability q that the attacker finds the new block and thus we move one step right in the Markov chain and a probability $1 - q$ that the honest miner add a new block and thus we move one step left. The only exceptions are states -6 and 1. As shown above, at state -6, we will remain at the same state even if the honest miner find a new block since the attacker would simply restart the attack at a new forking point. At state 1, the attacker succeeds and thus there is no need to continue the analysis. Hence, we assume the Markov chain never leaves state 1 after reaching it. Note that each step in the Markov chain models a single change in the state of the attack, i.e. how many blocks the attacker is behind in comparison to the honest chain, and the probability of traversing a path in the Markov chain is simply the product of probabilities assigned to its edges. Our goal is to compute the probability that the Markov chain reaches state 1 in a fixed number k of steps. This is the same as the probability of the attacker's success in reverting a continuous sequence of 6 blocks if she performs the attack for k blocks' time, i.e. in approximately $10 \cdot k$ minutes.

We note that our Markov chain is similar to the one in [25] but not identical to it. The difference is that the attacker modeled in [25] aims to perform a successful double-spending so she has to revert a particular block. In contrast, our attacker is only interested in reverting 6 consecutive blocks and does not care which 6 blocks are reverted.

Variant B. The attacker does not necessarily need to wait until her chain becomes strictly longer (state 1). She can already publish her chain when it has the same length as the other (honest) chain, which corresponds to state 0. At this point, since there are two chains of the same length, the honest miner are free to choose which chain to extend. Assuming they have no bias, half of the honest mining power will then be used in extending the attackers chain. This variant of the attack can be modeled by the Markov chain in Figure 7.3 and is slightly more likely to succeed.

Value Iteration [126]. In both variants of the attack, our goal is to find the probability that starting at vertex -6 and taking k steps, we end up in state 1. This is the same as the attacker's success probability if she continues the attack for k

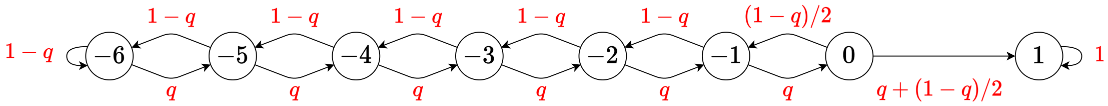


Figure 7.3: The Markov chain modeling variant B of the attack.

blocks' time. This probability can be computed exactly using a classical value iteration algorithm, with no need to perform simulations. Let $p[u, k]$ be the probability of being at state u after k steps. We have $p[-6, 0] = 1$ and $p[u, 0] = 0$ for all $u \neq -6$. Let $v_1, v_2, \dots, v_r$ be the predecessors of u in the Markov chain and the edge from v_j to u have probability $\pi(v_j, u)$. It is easy to see that for all $k \geq 1$, we have

$$p[u, k] = \sum_{j=1}^{r} p[v_j, k-1] \cdot \pi(v_j, u).$$

Intuitively, if we want to be at state u after k steps, we have to first get to one of its successors v_j in $k - 1$ steps and then take the edge from v_j to u.

Success Probabilities in Bitcoin. We consider attackers with between 10% and 45% of the hash power in 5% increments. In Bitcoin, a block is mined roughly every 10 minutes. Figure 7.4 shows the attacker's success probability if she employs variant A and Figure 7.5 provides the same results for variant B. Specifically, an attacker who has only 30% of the hash power will have a success probability of more than 95% using variant A if she persists on the attack for 33.7 days. The attack duration to obtain a 95% success rate is reduced to 4.1 days for an attacker with 40% of the hash power and 1.9 days for one with 45%.

The attack duration can be further improved with variant B. Specifically, with regards to a situation where the attack controls 30%, 40%, and 45% of the total hash power, the attack duration would be reduced to 18.7, 2.97, and 1.5 days, respectively.

7.4 Cost of the Attack

In the previous section, we showed that an attacker with a minority of the hash power can still succeed in reverting 6 blocks with high probability. Of course, an attacker who has a majority of the hash power will succeed in reverting any number of blocks with probability 1. In this section, we consider the costs of a block reverting attack. Specifically, we ask the following question: *How much does it cost to obtain a portion q of the hash power?* Our goal is not to obtain an exact number, but a ballpark estimate and upper-bound on the cost. Thus, we make the following simplifying assumptions:

- We only consider the cost of hardware at the time of writing. We assume the

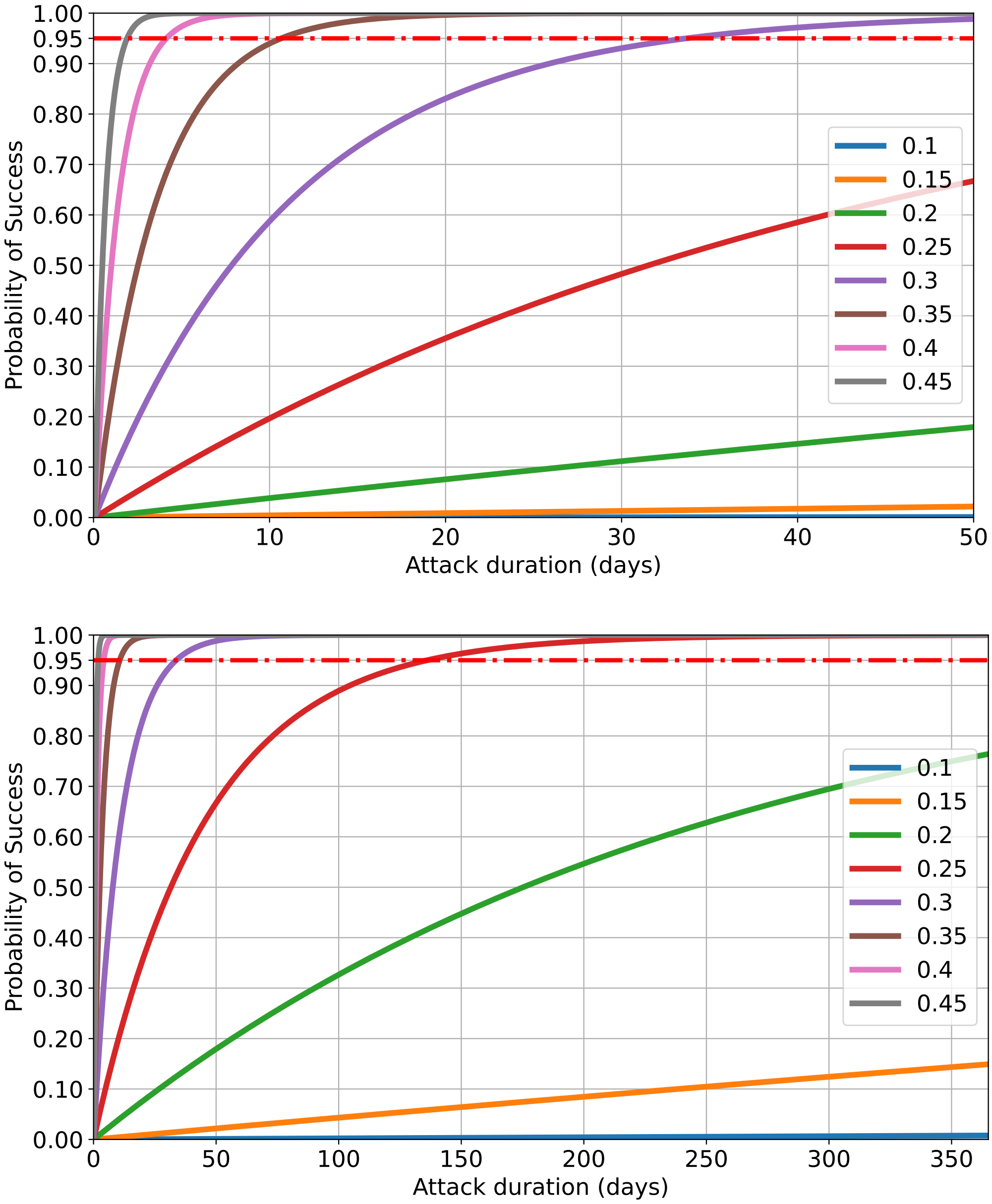
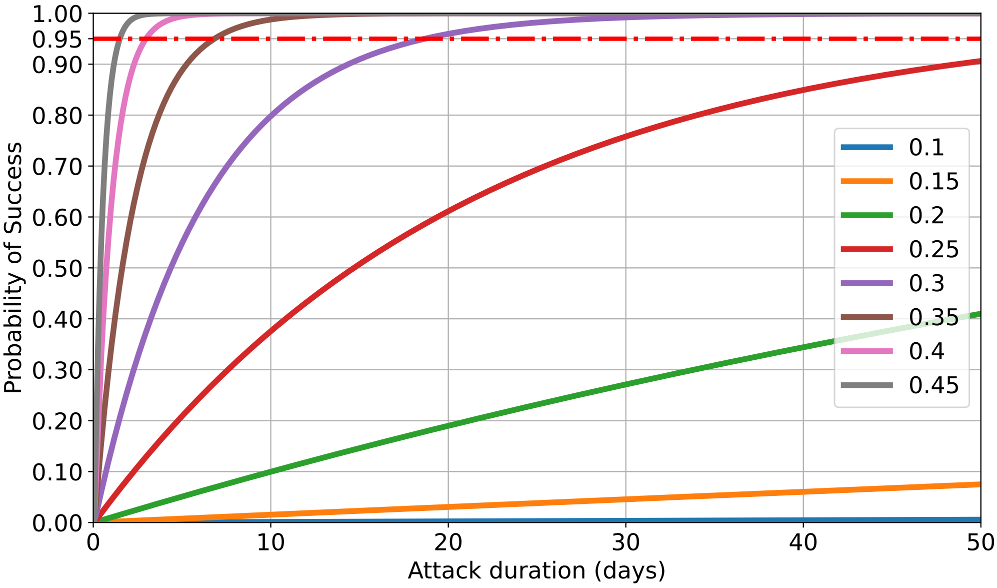


Figure 7.4: Attacker's success probability for variant A within 50 days (top) and a year (bottom). Each line corresponds to a different portion q of the hash power controlled by the attacker. The computed probabilities are exact and obtained by value iteration, not simulations.



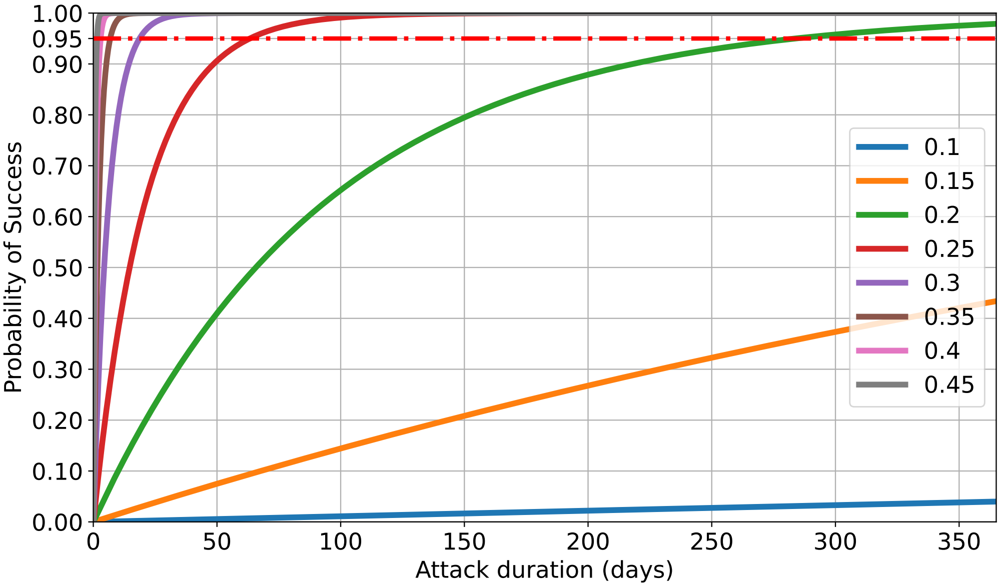


Figure 7.5: Attacker's success probability for variant B within 50 days (top) and a year (bottom). Each line corresponds to a different portion q of the hash power controlled by the attacker. The computed probabilities are exact and obtained by value iteration, not simulations.

attacker is buying the hardware, rather than renting it and do not consider potential discounts on bulk orders.

- We ignore electricity costs as they vary widely based on location.

The justification for the first assumption is that it keeps our analysis sound, i.e. we can only over-approximate the cost by making this assumption. As for the second assumption, we note that electricity costs are often negligible in comparison to hardware costs and that our main argument, i.e. the vulnerability of Bitcoin to majority attacks and block-reverting attacks, remains intact even if the estimates we obtain here are doubled. Indeed, as we will soon see, the trade volume of Bitcoin derivatives is more than three orders of magnitude larger than the numbers obtained here.

At the time of writing, the total hash rate of the Bitcoin network is 566.12 EH/s[127]. Furthermore, [128] categorized Bitcoin mining ASICs into three efficiency tiers based on their energy consumption: Under 25 J/TH, 25-38 J/TH, and Above 38 J/TH. Figure 7.6 shows the historical prices in USD per TH/s for each tier. To ensure the soundness of our analysis, we have estimated attack costs using the most expensive efficiency tier, i.e. Under 25 J/TH. This tier incorporates the most advanced generation of mining hardware that consumes the least amount of electricity. Table 7.1 summarizes the costs of obtaining various portions q of the total hash power at the time of writing. Note that if the current hash power is x, it is not enough for the attacker to purchase a hash power of $q \cdot x$, since her hash power will also be added to the network's total. Instead, she should buy y units of hash power where $\frac{y}{y+x} = q$ or equivalently $y = -\frac{q \cdot x}{q-1}$. Moreover, Figure 7.7 shows the historical price of obtaining various portions q of the total hash power based on the data from [127, 128, 2]. As Table 7.1 and Figure 7.7 show, it is easy to obtain a majority of the hash power, or a sizable minority that allows an attack as per the previous section, using an investment that is a tiny percentage of the Bitcoin's current market cap and, as we will see, three orders of magnitude smaller than the annual trade volume of Bitcoin derivatives.

7.5 Bitcoin Derivatives

In the previous section, we have already established a rough upper-bound on the costs of block-reverting attacks and argued that this cost is less than one percent of the current market cap of Bitcoin. Nevertheless, the required investment is substantial. Thus, the only remaining piece of the puzzle is to argue why anyone would be incentivized to invest such huge sums with the hope of crashing Bitcoin's price. In other words, how can an attacker profit from a price crash if she is losing her mining rewards?

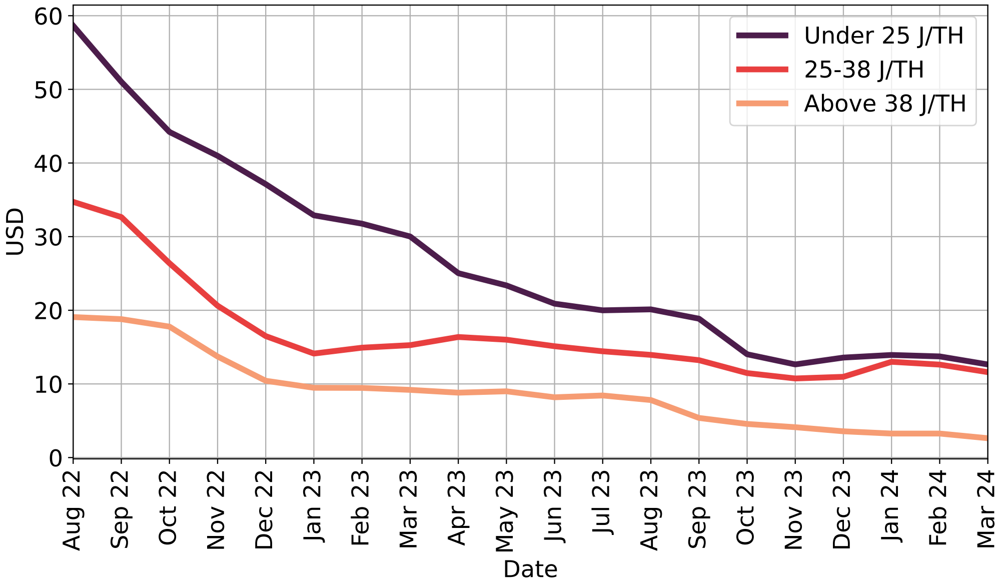


Figure 7.6: Historical prices for one TH/s of different Bitcoin mining ASICs grouped by four efficiency tiers.

q	Required EH/s	Cost (Billion USD)	$\frac{\text{Cost}}{\text{Bitcoin Market Cap}}$
0.10	62.9023	0.7523	0.0005
0.15	99.9036	1.1948	0.0009
0.20	141.5301	1.6927	0.0012
0.25	188.7068	2.2569	0.0016
0.30	242.623	2.9018	0.0021
0.35	304.834	3.6458	0.0026
0.40	377.4136	4.5139	0.0032
0.45	463.1894	5.5397	0.0039
0.50	566.1204	6.7708	0.0048

Table 7.1: The required hash power and hardware cost for an attacker who wishes to control a portion q of the total hash power.

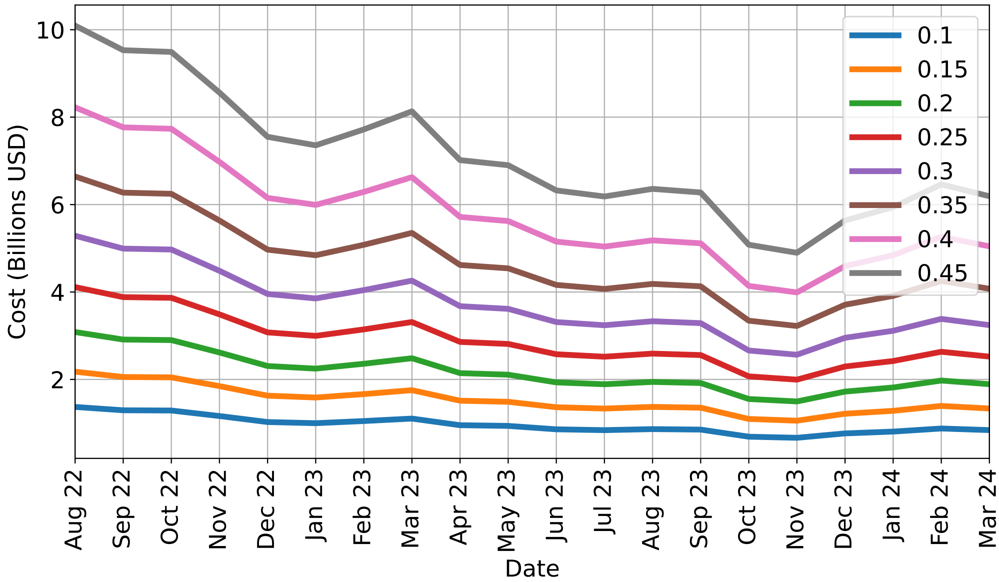


Figure 7.7: The historical cost of obtaining a portion q of the hash power for various values of q.

Futures and Options (Derivatives). A futures contract is a standard financial contract in which a seller agrees to sell something, in this case Bitcoin, to the buyer at a particular time and a preset price. For example, 1 Bitcoin is worth almost 70,000 USD at the time of writing. Alice and Bob can make a contract in which Alice promises to sell Bob 1 Bitcoin on 01-01-2025 (delivery date) at a price of 45,000 USD (delivery price). On the delivery date, Bob pays Alice 45,000 USD and Alice should either provide 1 Bitcoin or pay Bob the equivalent value in USD (actual price of Bitcoin on delivery date). In this case, we say that Alice is shorting Bitcoin, i.e. hoping that its value drops so that the 1 Bitcoin on 01-01-2025 is worth less than 45,000 USD. On the other hand, Bob is longing Bitcoin, i.e. hoping that its value increases or at least exceeds the delivery price of 45,000 USD. Usually, both sides have to provide a security deposit with a trusted third party to ensure they will honor their commitment.

Futures are often traded as options, i.e. Alice can enter a contract with Bob which, in exchange for an upfront payment, gives Alice the option but not the obligation to sell 1 Bitcoin to Bob on or before 01-01-2025 at 45,000 USD. Alice can choose whether she would like to exercise the option or not. Some options can be exercised only on their delivery date, while others allow Alice to exercise her option at any time before 01-01-2025. Please see [129] for more details on cryptocurrency and especially Bitcoin derivatives.

These contracts, both futures and futures options, are in turn traded on future exchanges, i.e. one can acquire Alice or Bob's interest in the contract in the same way as buying stocks. An attacker can acquire short positions in these contracts, i.e. bet that the price of Bitcoin will fall, and then be in a position to achieve financial gains by intentionally crashing Bitcoin's value.

Bitcoin Derivatives. There is a huge and mostly unregulated Bitcoin derivatives market whose trade volumes often exceed 750 billion or even 1 trillion USD in a calendar month. Cryptocurrency exchanges such as Binance, OKX and ByBit offer future contracts and options on a wide variety of coins. Figure 7.8 shows the monthly trade volume of these largely unregulated markets between August 2022 and March 2024 [130]. We note that there has been a recent explosion in the value of such derivative contracts, but their total value was already huge even before the recent uptick. In March 2024 alone, the volume was almost 2.5 *trillion* USD. As evident in this figure, the annual trade volume of Bitcoin derivatives is an order of magnitude larger than Bitcoin's market cap, which was itself two orders of magnitude larger than the cost of an attack. Thus, it is clearly feasible to buy enough put options, or other shorting vehicles, to benefit from a price crash that is induced by the block-reverting attacks explained above.

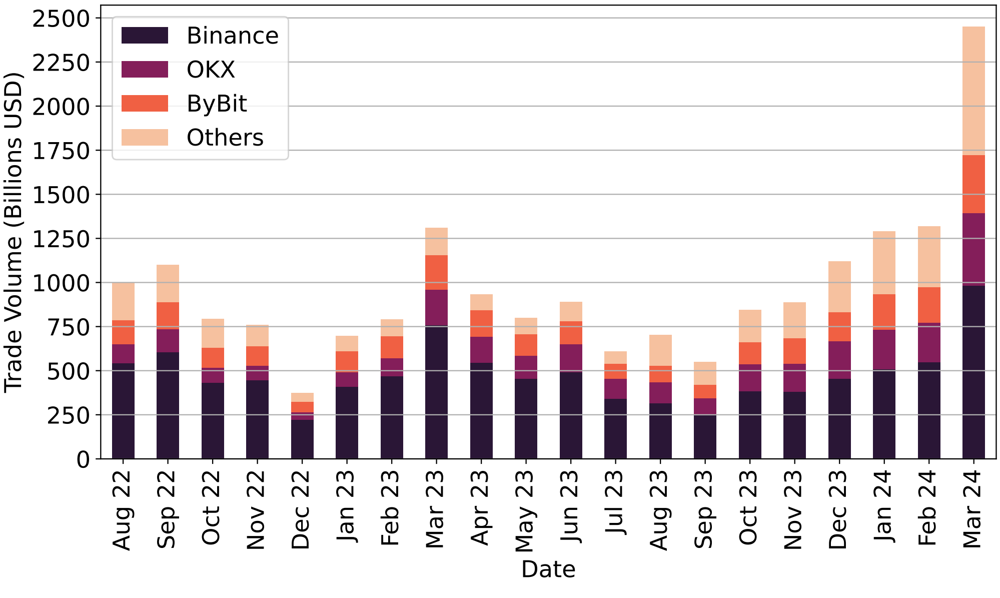


Figure 7.8: Total trade volume of unregulated Bitcoin derivatives [130].

Open Interest of Bitcoin Options. At the time of writing, the open interest in BTC options is slightly more than 20 billion USD [130]. Thus, a malicious party performing the attack mentioned in this chapter would need to obtain a considerable amount of the available put contracts. This may lead to market disruptions whose analysis is beyond the scope of this analysis. That said, if the derivatives market continues to grow and becomes much larger than it currently is, purchasing this amount of contracts might not even be detected.

s

Chapter 8

Discussion and Conclusion

8.1 Discussion

Blockchain technology stands to gain significantly from established computer science techniques, but the reverse is also true: the methods we have developed for blockchain problems offer valuable applications beyond just smart contract analysis. While the core of this thesis focuses on analyzing smart contracts and their interactions, this section outlines other related problems the author investigated during his PhD. Because this work addresses issues not directly tied to smart contracts, it is excluded from the main body of this thesis.

8.1.1 Weighted Model Integration

This section is based on the following publication:

- S. Akshay, S. Chakraborty, S. Farokhnia, A.K. Goharshady, H.J. Motwani, Đ. Žikelić. *LP-Based Weighted Model Integration over Non-Linear Real Arithmetic.* International Joint Conference on Artificial Intelligence, IJCAI 2025.

Weighted model integration (WMI) is a relatively recent formalism that has received significant interest as a technique for solving probabilistic inference tasks with complicated weight functions. Existing methods and tools are mostly focused on linear and polynomial functions and provide limited support for WMI of rational or radical functions, which naturally arise in several applications.

In this work, we present a novel method for approximate WMI, which provides more effective support for the wide class of semi-algebraic functions that includes rational and radical functions, with literals defined over non-linear real arithmetic.

Our algorithm leverages Farkas' lemma and Handelman's theorem from real algebraic geometry to reduce WMI to solving a number of linear programming (LP) instances. The algorithm provides formal guarantees on the error bound of the obtained approximation and can reduce it to any user-defined value ϵ. Furthermore, our approach is perfectly parallelizable. Finally, we present extensive experimental results, demonstrating the superior performance of our method on a range of WMI tasks for rational and radical functions when compared to state-of-the-art tools for WMI, in terms of both applicability and tightness.

Our Contributions. The main contributions of this work are:

1. We develop a novel algorithm for the approximate WMI problem. Our algorithm supports integration of general semi-algebraic weight functions over general semi-algebraic sets. This makes our method effectively applicable to a large new class of weight functions, such as rational functions and radical functions.

2. Our method provides formal guarantees on the tightness of the approximation error. In particular, for any error bound $\epsilon > 0$ provided by the user, our method produces an ϵ-tight approximation of the WMI value.

3. We implement our algorithm and integrate our tool (WMI-LP) with the state-of-the-art WMI framework of [131]. Our extensive experiments demonstrate that our approach outperforms state-of-the-art exact and approximate tools in handling a wider variety of polynomial, rational, and radical functions. It also provides consistently tighter error bounds, highlighting its robustness and accuracy across benchmarks.

8.2 Conclusion

In this thesis, we advanced principled, automated methods for smart contracts by exploiting their structural properties and transaction-level interactions. At the code level, we introduced algebro-geometric techniques to synthesize tight polynomial gas upper bounds with soundness (and semi-completeness) that outperformed prior work in both tightness and applicability. We further strengthened compiler-level gas super-optimization by integrating dynamic programming into the state of the art, more than doubling baseline gas savings on widely used contracts. At the protocol level, we designed revenue-maximizing block-construction algorithms for Cardano, using an exact parameterized approach that runs efficiently on sparse transaction graphs and achieves a 55.68% improvement, and for Ethereum, developing a randomized framework that increases miner revenue by an average of 73.45% per block. For decentralized exchange routes, we leveraged the tree-like topology of liquidity pools, enabling support for instantaneous trade sizes and delivering up to four orders of magnitude performance improvements over prior work. Beyond engineering, we analyzed Bitcoin's security under derivatives, showing that block-reversion attacks can be economically enabled with far less than a majority of hash power.

Taken together, these contributions show that exploiting the unique structure of smart contracts can yield techniques that are simultaneously grounded in formal principles and effective at scale. At the same time, the thesis opens several directions for further investigation. In particular, the compiler-level optimization results should be viewed as an initial step and can be substantially improved by applying the same foundational principles. More broadly, a range of smart-contract analysis tasks remain underexplored, and the tools developed here appear well-suited to be adapted.

BIBLIOGRAPHY

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.

[2] CoinMarketCap, “Cryptocurrency prices,” https://coinmarketcap.com/, 2024, accessed: 2025-06-25.

[3] S. Farokhnia and A. K. Goharshady, “Alleviating high gas costs by secure and trustless off-chain execution of smart contracts,” in *ACM SAC*, 2023, pp. 258–261.

[4] R. M. Polvora, A. Hakami, E. Bol, S. Hassan, J. K. Brekke, and M. Atzori, “Scanning the european ecosystem of distributed ledger technologies for social and public good,” Joint Research Centre, European Commission, Tech. Rep., 2020.

[5] M. P. D. Library, “First international blockchain for science: bloxberg,” https://www.mpg.de/13417668/first-international-blockchain-for-science-bloxberg, accessed: 2025-06-25.

[6] U. Labs, “Uniswap explore,” https://app.uniswap.org/explore, 2024, accessed: 2025-06-25.

[7] M. Thorup, “All structured programs have small tree-width and good register allocation,” *Inf. Comput.*, vol. 142, no. 2, pp. 159–181, 1998.

[8] K. Chatterjee, A. K. Goharshady, and E. K. Goharshady, “The treewidth of smart contracts,” in *ACM SAC*, 2019, pp. 400–408.

[9] J. Xu, C. Wang, and X. Jia, “A survey of blockchain consensus protocols,” *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–35, 2023.

[10] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, “Proofs of space,” in *CRYPTO*, vol. 9216, 2015, pp. 585–605.

[11] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, “Hybrid mining: exploiting blockchain’s computational power for distributed problem solving,” in *SAC*, 2019, pp. 374–381.

[12] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan, "Proofs of useful work," *IACR Cryptol. ePrint Arch.*, p. 203, 2017.

[13] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *PODC*, 2019, pp. 347–356.

[14] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," 2012.

[15] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *CRYPTO*, vol. 10401, 2017, pp. 357–388.

[16] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[17] B. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *EUROCRYPT*, vol. 10821, 2018, pp. 66–98.

[18] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theor. Comput. Sci.*, vol. 777, pp. 155–183, 2019.

[19] S. Park, A. Kwon, G. Fuchsbauer, P. Gaži, J. Alwen, and K. Pietrzak, "Spacemint: A cryptocurrency based on proofs of space," in *FC*, 2018, pp. 480–499.

[20] C. Foundation, "Plutus - cardano smart contracts," https://docs.cardano.org/developer-resources/smart-contracts/plutus, accessed: 2025-06-25.

[21] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *CCS*, 2016, pp. 3–16.

[22] M. A. Meybodi, A. K. Goharshady, M. R. Hooshmandasl, and A. Shakiba, "Optimal mining: Maximizing Bitcoin miners' revenues from transaction fees," in *Blockchain*, 2022, pp. 266–273.

[23] T. Barakbayeva, S. Farokhnia, A. Goharshady, M. Gufler, and S. Novozhilov, "Pixiu: Optimal block production revenues on cardano," in *Blockchain*, 2024.

[24] E. Blum, A. Kiayias, C. Moore, S. Quader, and A. Russell, "The combinatorics of the longest-chain rule: Linear consistency for proof-of-stake blockchains," in *SODA*, 2020.

[25] M. Rosenfeld, “Analysis of hashrate-based double spending,” *arXiv preprint arXiv:1402.2009*, 2014.

[26] G. O. Karame, E. Androulaki, and S. Capkun, “Double-spending fast payments in Bitcoin,” in *CCS*, 2012, pp. 906–917.

[27] K. C. Chaudhary, V. Chand, and A. Fehnker, “Double-spending analysis of Bitcoin,” in *PACIS*, 2020.

[28] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Čapkun, “Misbehavior in Bitcoin: A study of double-spending and accountability,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, pp. 1–32, 2015.

[29] B. Hou and F. Chen, “A study on nine years of Bitcoin transactions: Understanding real-world behaviors of Bitcoin miners and users,” in *ICDCS*, 2020, pp. 1031–1043.

[30] C. Hoskinson, “Why we are building Cardano,” 2017.

[31] CoinMarketCap, “Cryptocurrency prices, charts and market capitalizations,” 2024. [Online]. Available: https://coinmarketcap.com/

[32] Cardano Foundation, “EUTXO handbook,” 2017. [Online]. Available: https://docs.cardano.org/learn/eutxo-explainer

[33] L. Brünjes and M. J. Gabbay, “UTxO vs account-based smart contract blockchain programming paradigms,” in *ISoLA*, 2020, pp. 73–88.

[34] Cardano Documentation, “Proof of stake,” 2024. [Online]. Available: https://docs.cardano.org/new-to-cardano/proof-of-stake

[35] ——, “Slots and epochs,” 2024. [Online]. Available: https://docs.cardano.org/learn/cardano-node/#howdoesitwork

[36] ——, “Stake pools,” 2024. [Online]. Available: https://docs.cardano.org/learn/stake-pools

[37] ——, “Pledging and rewards,” 2024. [Online]. Available: https://docs.cardano.org/learn/pledging-rewards

[38] M. Dameron, “Beige paper: An Ethereum technical specification,” 2019.

[39] Ethereum Foundation, 2014. [Online]. Available: https://docs.soliditylang.org/

[40] ——, "Gas and fees," 2025. [Online]. Available: https://ethereum.org/en/developers/docs/gas/

[41] M. Dameron, "Beigepaper: An ethereum technical specification," 2018. [Online]. Available: https://github.com/chronaeon/beigepaper/blob/master/beigepaper.pdf

[42] R. G. Downey, M. R. Fellows *et al.*, *Fundamentals of parameterized complexity.* Springer, 2013, vol. 4.

[43] A. K. Goharshady and A. K. Zaher, "Efficient interprocedural data-flow analysis using treedepth and treewidth," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 13881. Springer, 2023, pp. 177–202.

[44] G. K. Conrado, A. K. Goharshady, and C. K. Lam, "The bounded pathwidth of control-flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 292–317, 2023.

[45] N. Robertson and P. D. Seymour, "Graph minors. i. excluding a forest," *Journal of Combinatorial Theory, Series B*, vol. 35, no. 1, pp. 39–61, 1983.

[46] J. Nešetřil, P. Ossona de Mendez, J. Nešetřil, and P. O. de Mendez, "Bounded height trees and tree-depth," *Sparsity: Graphs, Structures, and Algorithms*, pp. 115–144, 2012.

[47] Y. Iwata, T. Ogasawara, and N. Ohsaka, "On the power of tree-depth for fully polynomial fpt algorithms," *arXiv preprint arXiv:1710.04376*, 2017.

[48] J. Nešetřil and P. Ossona de Mendez, "On low tree-depth decompositions," *Graphs and combinatorics*, vol. 31, no. 6, pp. 1941–1963, 2015.

[49] F. Reidl, P. Rossmanith, F. S. Villaamil, and S. Sikdar, "A faster parameterized algorithm for treedepth," in *ICALP*, 2014, pp. 931–942.

[50] B. Strasser, "PACE solver description: Tree depth with FlowCutter," in *IPEC*, 2020, pp. 32:1–32:4.

[51] J. Farkas, "Theory of simple inequalities." *Journal for pure and applied mathematics (Crelles Journal)*, vol. 1902, no. 124, pp. 1–27, 1902.

[52] D. Handelman, "Representing polynomials by positive linear functions on compact convex polyhedra," *Pacific Journal of Mathematics*, vol. 132, no. 1, pp. 35–62, 1988.

[53] M. Putinar, “Positive polynomials on compact semi-algebraic sets,” *Indiana University Mathematics Journal*, vol. 42, no. 3, pp. 969–984, 1993.

[54] G. Blekherman, P. A. Parrilo, and R. R. Thomas, *Semidefinite optimization and convex algebraic geometry.* SIAM, 2012.

[55] D. S. Watkins, *Fundamentals of matrix computations.* John Wiley & Sons, 2004, vol. 64.

[56] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, “Don’t run on fumes - parametric gas bounds for smart contracts,” *J. Syst. Softw.*, vol. 176, p. 110923, 2021.

[57] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, “Safer smart contract programming with Scilla,” in *OOPSLA*, 2019, pp. 1–30.

[58] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, “Ethir: A framework for high-level analysis of ethereum bytecode,” in *ATVA*, 2018, pp. 513–520.

[59] M. Colón, S. Sankaranarayanan, and H. Sipma, “Linear invariant generation using non-linear constraint solving,” in *CAV*, 2003, pp. 420–432.

[60] K. Chatterjee, H. Fu, A. K. Goharshady, and E. K. Goharshady, “Polynomial invariant generation for non-deterministic recursive programs,” in *PLDI*, 2020, pp. 672–687.

[61] A. Goharshady, “Parameterized and algebro-geometric advances in static program analysis,” Ph.D. dissertation, Institute of Science and Technology Austria, 2020.

[62] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in *WETSEB*, 2019, pp. 8–15.

[63] L. M. de Moura and N. S. Bjørner, “Z3: an efficient SMT solver,” in *TACAS*, 2008, pp. 337–340.

[64] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, “The MathSAT5 SMT solver,” in *TACAS*, 2013, pp. 93–107.

[65] B. Nassirzadeh, H. Sun, S. Banescu, and V. Ganesh, “Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities,” in *MARBLE*, 2022.

[66] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, “*Don’t run on fumes* - parametric gas bounds for smart contracts,” *J. Syst. Softw.*, vol. 176, p. 110923, 2021.

[67] Z. Cai, S. Farokhnia, A. K. Goharshady, and S. Hitarth, “Asparagus: Automated synthesis of parametric gas upper-bounds for smart contracts,” *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, pp. 882–911, 2023.

[68] Etherscan, “Ethereum gas tracker,” 2024. [Online]. Available: https://etherscan.io/gastracker

[69] Ethereum Foundation, “Solidity: An object-oriented high-level language for implementing smart contracts,” 2024. [Online]. Available: https://docs.soliditylang.org/en/v0.8.28/

[70] L. T. Thibault, T. Sarry, and A. S. Hafid, “Blockchain scaling using rollups: A comprehensive survey,” *IEEE Access*, vol. 10, pp. 93 039–93 054, 2022.

[71] S. Farokhnia and A. K. Goharshady, “Reducing the gas usage of ethereum smart contracts without a sidechain,” in *ICBC*, 2023, pp. 1–3.

[72] E. Albert, P. Gordillo, A. Hernández-Cerezo, A. Rubio, and M. A. Schett, “Super-optimization of smart contracts,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 70:1–70:29, 2022.

[73] H. Massalin, “Superoptimizer - A look at the smallest program,” in *ASPLOS*, 1987, pp. 122–126.

[74] C. Lattner and V. S. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *CGO*, 2004, pp. 75–88.

[75] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, “Souper: A synthesizing superoptimizer,” *CoRR*, vol. abs/1711.04422, 2017.

[76] M. A. Meybodi, A. K. Goharshady, M. R. Hooshmandasl, and A. Shakiba, “Optimal mining: Maximizing Bitcoin miners’ revenues from transaction fees,” in *Blockchain.* IEEE, 2022, pp. 266–273.

[77] T. Barakbayeva, S. Farokhnia, A. K. Goharshady, M. Gufler, and S. Novozhilov, “Pixiu: Optimal block production revenues on Cardano,” in *IEEE Blockchain*, 2024, pp. 491–496.

[78] E. Foundation, “Maximal extractable value (mev),” 2025. [Online]. Available: https://ethereum.org/en/developers/docs/mev/

[79] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, “Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability,” in *SP*, 2020, pp. 910–927.

[80] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, “High-frequency trading on decentralized on-chain exchanges,” in *SP*, 2021, pp. 428–445.

[81] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *CCS*, 2016, pp. 254–269.

[82] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H. Lee, “Ethereum smart contract analysis tools: A systematic review,” *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022.

[83] L. Heimbach and R. Wattenhofer, “Sok: Preventing transaction reordering manipulations in decentralized finance,” in *AFT*, 2022, pp. 47–60.

[84] K. Babel, P. Daian, M. Kelkar, and A. Juels, “Clockwork finance: Automated analysis of economic security in smart contracts,” in *SP*, 2023, pp. 2499–2516.

[85] A. Judmayer, N. Stifter, P. Schindler, and E. R. Weippl, “Estimating (miner) extractable value is hard, let’s go shopping!” in *Financial Cryptography Workshops*, 2022, pp. 74–92.

[86] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*, 1984. [Online]. Available: https://books.google.com.hk/books?id=JwQx-WOmSyQC

[87] Foundry, “Anvil cli reference,” 2023. [Online]. Available: https://book.getfoundry.sh/reference/cli/anvil

[88] ErigonTech, “Erigon,” 2023. [Online]. Available: https://github.com/erigontech/erigon

[89] Linfa, “Linfa,” 2025. [Online]. Available: https://github.com/rust-ml/linfa/tree/master

[90] S. Bolusani, M. Besançon, K. Bestuzheva, A. Chmiela, J. Dionísio, T. Donkiewicz, J. van Doornmalen, L. Eifler, M. Ghannam, A. Gleixner, C. Graczyk, K. Halbig, I. Hedtke, A. Hoen, C. Hojny, R. van der Hulst, D. Kamp, T. Koch, K. Kofler, J. Lentz, J. Manns, G. Mexi, E. Mühmer, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, M. Turner, S. Vigerske, D. Weninger, and L. Xu, “The SCIP Optimization Suite 9.0,” Optimization Online, Technical Report, February 2024. [Online]. Available: https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/

[91] Blocknative, “Mempool archive,” 2025. [Online]. Available: https://docs.blocknative.com/data-archive/mempool-archive

[92] E. Foundation, “Go Ethereum (Geth) Client,” 2025. [Online]. Available: https://geth.ethereum.org/

[93] CoinMarketCap, “Cryptocurrency prices, charts and market capitalizations,” 2025. [Online]. Available: https://coinmarketcap.com/

[94] R. McLaughlin, C. Kruegel, and G. Vigna, “A large scale study of the ethereum arbitrage ecosystem,” in *USENIX Security Symposium*, 2023, pp. 3295–3312.

[95] “Defillama - defi dashboard,” https://defillama.com/, accessed: 2025-07-03.

[96] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, “Sok: Decentralized exchanges (DEX) with automated market maker (AMM) protocols,” *ACM Comput. Surv.*, vol. 55, no. 11, pp. 238:1–238:50, 2023.

[97] Y. Zhang, T. Yan, J. Lin, B. Kraner, and C. J. Tessone, “An improved algorithm to identify more arbitrage opportunities on decentralized exchanges,” in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2024, pp. 1–7.

[98] “The uniswap subgraph,” https://docs.uniswap.org/api/subgraph/overview, accessed: 2025-07-03.

[99] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized Algorithms*, 2015.

[100] H. Adams, M. Salem, N. Zinsmeister, S. Reynolds, A. Adams, W. Pote, M. Toda, A. Henshaw, E. Williams, and D. Robinson, “Uniswap v4 core [draft],” 2023.

[101] H. L. Bodlaender, “A linear time algorithm for finding tree-decompositions of small treewidth,” in *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, 1993, pp. 226–234.

[102] D. Delling, D. Fleischman, A. Malucelli, and R. F. Werneck, “The 2nd annual parameterized algorithms and computational experiments challenge (pace 2017),” vol. 89, 2017, pp. 32:1–32:11.

[103] P. Heggernes, “Treewidth, partial k-trees, and chordal graphs,” *Partial curriculum in INF334-Advanced algorithmical techniques, Department of Informatics, University of Bergen, Norway*, 2005.

[104] L. Planken, M. de Weerdt, and R. van der Krogt, “Computing all-pairs shortest paths by leveraging low treewidth,” *J. Artif. Int. Res.*, vol. 43, no. 1, p. 353–388, 2012.

[105] N. Chleq, “Efficient algorithms for networks of quantitative temporal constraints,” *Constraints*, vol. 95, 1995.

[106] A. Hagberg, P. J. Swart, and D. A. Schult, “Exploring network structure, dynamics, and function using networkx,” Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[107] J. Joshi and R. Mathew, “A survey on attacks of Bitcoin,” in *ICCBI*, 2020, pp. 953–959.

[108] F. A. Aponte-Novoa, A. L. S. Orozco, R. Villanueva-Polanco, and P. Wightman, “The 51% attack on blockchains: A mining behavior study,” *IEEE access*, vol. 9, pp. 140 549–140 564, 2021.

[109] S. Shanaev, A. Shuraeva, M. Vasenin, and M. Kuznetsov, “Cryptocurrency value and 51% attacks: evidence from event studies,” *The Journal of Alternative Investments*, vol. 22, no. 3, pp. 65–77, 2019.

[110] K. A. Negy, P. R. Rizun, and E. G. Sirer, “Selfish mining re-examined,” in *FC*, 2020, pp. 61–78.

[111] A. K. Goharshady, “Irrationality, extortion, or trusted third-parties: Why it is impossible to buy and sell physical goods securely on the blockchain,” in *Blockchain.* IEEE, 2021, pp. 73–81.

[112] K. Chatterjee, A. K. Goharshady, and Y. Velner, “Quantitative analysis of smart contracts,” in *ESOP*, 2018, pp. 739–767.

[113] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable,” *Commun. ACM*, vol. 61, no. 7, pp. 95–102, 2018.

[114] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, and Y. Velner, “Ergodic mean-payoff games for the analysis of attacks in crypto-currencies,” in *CONCUR*, 2018.

[115] A. Hern, “Bitcoin currency could have been destroyed by 51% attack,” 2014. [Online]. Available: https://www.theguardian.com/technology/2014/jun/16/bitcoin-currency-destroyed-51-attack-ghash-io

[116] “Bitcoin hashrate distribution,” 2023. [Online]. Available: https://www.blockchain.com/explorer/charts/pools

[117] “Bitcoin mining pools.” [Online]. Available: https://hashrateindex.com/hashrate/pools

[118] J. Chen and S. Micali, “Algorand: A secure and efficient distributed ledger,” *Theoretical Computer Science*, vol. 777, pp. 155–183, 2019.

[119] Z. Cai and A. K. Goharshady, “Game-theoretic randomness for proof-of-stake,” in *MARBLE*, 2023, pp. 28–47.

[120] V. Abidha, T. Barakbayeva, Z. Cai, and A. Goharshady, “Gas-efficient decentralized random beacons,” in *ICBC*, 2024.

[121] T. Barakbayeva, Z. Cai, and A. Goharshady, “SRNG: An efficient decentralized approach for secret random number generation,” in *ICBC*, 2024.

[122] J. Ballweg, Z. Cai, and A. Goharshady, “PureLottery: Fair leader election without decentralized random number generation,” in *Blockchain*, 2023.

[123] P. Fatemi and A. K. Goharshady, “Secure and decentralized generation of secret random numbers on the blockchain,” in *BCCA*, 2023, pp. 511–517.

[124] Z. Cai and A. K. Goharshady, “Trustless and bias-resistant game-theoretic distributed randomness,” in *ICBC*, 2023, pp. 1–3.

[125] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, “Probabilistic smart contracts: Secure randomness on the blockchain,” in *IEEE ICBC*. IEEE, 2019, pp. 403–412.

[126] R. Bellman, “A markovian decision process,” *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

[127] “Total hash rate.” [Online]. Available: https://www.blockchain.com/explorer/charts/hash-rate

[128] “Bitcoin asic price index.” [Online]. Available: https://data.hashrateindex.com/chart/asic-price-index

[129] Y. Söylemez, “Cryptocurrency derivatives: The case of Bitcoin,” *Blockchain Economics and Financial Market Innovation: Financial Innovations in the Digital Age*, pp. 515–530, 2019.

[130] The Block, “Crypto options data,” 2024. [Online]. Available: https://www.theblock.co/data/crypto-markets/options

[131] G. Spallitta, G. Masina, P. Morettin, A. Passerini, and R. Sebastiani, “Enhancing smt-based weighted model integration by structure awareness,” *Artificial Intelligence*, vol. 328, p. 104067, 2024.